Towards a common hardware/software specification and implementation approach for distributed, rel time and embedded systems, based on middlewares and object-oriented components

Gregory Gailliard

THALES

ECOLE DOCTORALE SCIENCES ET INGENIERIE
de l'université de Cergy-Pontoise

THESE

Présentée pour obtenir le grade de docteur d'université
Discipline : Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Vers une approche commune pour le logiciel et le matériel de spécification et d'implémentation des systèmes embarqués temps-réels distribués, basée sur les intergiciels et les composants orientés objet

*Application aux modèles de composants Software Communications Architecture (**SCA**) et Lightweight Corba Component Model (**LwCCM**) pour les systèmes de radio logicielle*

Towards a common hardware/software specification and implementation approach for distributed, real-time and embedded systems, based on middlewares and object-oriented components

*Application to Software Communications Architecture (**SCA**) and Lightweight Corba Component Model (**LwCCM**) component models for Software Defined Radio (SDR) systems*

par
Grégory Gailliard

Laboratoire Equipes Traitement des Images et du Signal (ETIS) - CNRS UMR 8051
Equipe Architecture, Systèmes, Technologies pour les unités Reconfigurables Embarquées (ASTRE)

Thèse soutenue le Vendredi 5 Février 2010

Devant le jury composé de :

M. Jean-Luc Dekeyser	Président
M. Michel Auguin	Examinateur
M. Christophe Dony	Examinateur
M. Laurent Pautet	Rapporteur
M. Guy Gogniat	Rapporteur
M. François Verdier	Directeur de thèse
M. Michel Sarlotte	Invité

The devil lies in the details
Proverb

Everything should be made as simple as possible, but not simpler
Albert Einstein

Acknowledgments

I would like to thank my thesis director, François Verdier, and my industrial tutor, Michel Sarlotte, for providing me with a very interesting Ph.D. topic. I appreciated the autonomy and trust they have given me throughout my thesis. I am grateful to Michel Sarlotte for having allowed me to have a trainee, Benjamin Turmel, who helped me a lot in the implementation of my ideas down to hardware. I also thank the members of the jury for having accepted to be part of my Ph.D. examinating board.

I would also like to thank Bruno Counil for discussions about the SCA, Hugues Balp and Vincent Seignole for discussions about CCM, the IDL-to-VDHL mapping and participation in the SPICES project.

I also thank Eric Nicollet for discussions about the MDE approach and the importance of an IDL-to-VHDL mapping.

I am very grateful to Bertrand Caron for its precious advices about VHDL and for the discussions about the Transceiver and the MHAL.

I also thank Bertrand Mercier for our common work on DiMITRI.

My thanks are also due to Bernard Candaele for its suggestions and the trainings.

I acknowledge my colleagues during three years for providing me a very nice work environment at Thales: Helene Came, Frédéric Dotto, Olivier Pierrelee, Eric Combot, Laurent Chaillou, François Kasperski, Jérôme Quevremont, Rémi Chau, Matthieu Paccot, Pauline Roux, Jessica Bouvier, Laurent Chmelesvki, Eric Chabod, Vincent Chiron, Eliane Carimantrant and all the others.

My gratitude also goes to Lesly Levy for its native rereading of some chapters.

Finally, I thank my close family members and relatives for their support and encouragements all through this work.

Contents

Chapter 1

Introduction

Contents

1.1 Context: Real-Time, Heterogeneous and Distributed Hardware/Software Embedded Systems

This PhD thesis took place in collaboration between Thales Communications and the ETIS laboratory of the University of Cergy-Pontoise in France. This work deals with the design of real-time, heterogeneous and distributed hardware/software embedded systems for *Software Defined Radio* (**SDR**). Within the digital hardware design service led by Bernard Candaele then Michel Sarlotte, we focused on the hardware implementation of the physical layer of radio modems (*modulator-demodulator*).

1.1.1 Modem applications

In telecommunications, a *modem* is a digital communication system that transmits and receives user application data by applying successive signal processing transformations: source coding/decoding i.e. data compression, channel coding/decoding for detection and correction of transmission errors (a.k.a. *Forward Error Correction* (**FEC**)), and modulation/demodulation of an analog carrier wave into a digital bit stream. This digital bit stream is then converted into electromagnetic signals using *Digital-to-Analog* and *Analog-to-Digital Converters* (**DAC**/**ADC**) and antenna(s).

1.1.2 Embedded Systems

Embedded systems designate almost every electronic devices commonly used in our daily life such as consumer electronics (e.g. televisions, MP3/DVD players, digital watches, hand-held game consoles), communication systems (e.g. mobile phones, ADSL modems, satellites, GPS receivers), household appliances (e.g. microwave ovens, washing machines), and many others like cars and medical devices. An embedded system is a specialized computing system dedicated to a given application domain. Typically, an embedded system consists of a set of software applications executed on a hardware platform providing limited computation, storage and communications resources. There are several classes of embedded systems from highly embedded systems based on 8-bit micro-controllers to *MultiProcessor Systems-on-chips* (**MPSoC**s).

An MPSoC is an integrated circuit that contains most hardware components of a general-purpose computing system such as processors, memories, buses, inputs-outputs (I/Os) and specialized hardware devices, but is customized for an application domain. MPSoCs are composed of heterogeneous processing elements, which are more and more complex to integrate and program. Indeed, they have different specifications and implementation languages, simulation/execution environments, interaction semantics and communication protocols.

1.1.3 Middlewares

A middleware is a software infrastructure, which supports transparent communications between heterogeneous distributed systems regardless of their physical location, hardware computing architecture, programming language, operating system, transport layer and data representation. For decades, software middlewares address these issues thanks to standard *Interface Definition Languages* (**IDL**), data encoding rules and message passing protocols. Face to the heterogeneity of MPSoCs, some works have been carried out to apply middleware concepts in distributed embedded systems.

1.1.4 Real-Time Systems

Some embedded systems are often characterized by real-time constraints. A real-time system must perform computations according to timing deadlines. Two kinds of real-time systems are typically distinguished: hard real-time systems and soft real-time systems. In hard real-time systems, a deadline miss is considered as an error and may be critical e.g. in transportation systems, while it is tolerated in soft real-time systems e.g. in multimedia applications.

1.1.5 Software Defined Radio

Our everyday environment is full of radio signals ranging from cordless and cellular phones, to AM/FM receptors, HDTV sets, garage door openers, and wireless communications between Personal Computers (PCs), WIFI routers to access the Internet, and bluetooth peripheral devices such as printer.

Face to the multiplication of wireless communication protocols, the term *Software-Defined Radio* and *Software Radio* were coined in the 90s to designate a wireless communication device, whose main functionalities including some signal processing treatments, are defined and implemented in software. The concept of SDR originates from the first US DARPA programs on military software radios such as Speakeasy [Tor92] [LU95]. To overcome the communication problems appeared during the Desert Storm operation in Irak, the Speakeasy program might allow communication interoperability among several wireless communication devices operating in different frequency bands ranging from High-Frequency (HF) band (3 to 30 MHz) to X band (8 to 12 GHz) used for *Satellite Communications* (**SATCOM**).

Software-Defined Radios are much more flexible than older "hardware-defined radios" whose functionalities were hard-wired. A Software Defined Radio is a reconfigurable radio whose functionalities are defined in software and moved closer to the antenna [Tut02] [III00]. It is able to run a set of waveform applications on the same radio platform

A *waveform application* is a component-based software application. The waveform components notably implement signal processing treatments, which process the received data from the antenna to the end user and vice versa. These treatments are defined in custom or standard waveform specifications such as military NATO (North Atlantic Treaty Organization) *Standardization Agreements* (**STANAGs**) or civil ITU(International Telecommunication Union) standards such as WIFI/WIMAX. A waveform application may be roughly seen as a radio protocol stack organized in a similar way to the OSI (Open Systems Interconnection) reference model. A *radio platform* is a set of software and hardware layers, which provide the services required by the waveform application layer through *Application Programming Interfaces* (**APIs**). A radio platform includes radio devices (e.g. a GPS receiver) and radio services (e.g. a log service).

Thanks to this software definition, a SDR fosters reuse and flexibility, since product evolutions can be easily performed via software updates instead of hardware replacements. As illustrated in figure 1.1, the two main SDR needs are the **portability** of waveform applications across different radio platforms and the **reconfigurability** of the hardware platform to execute several waveforms. Another need is that software waveform applications can be easily reused from one radio platform to another.

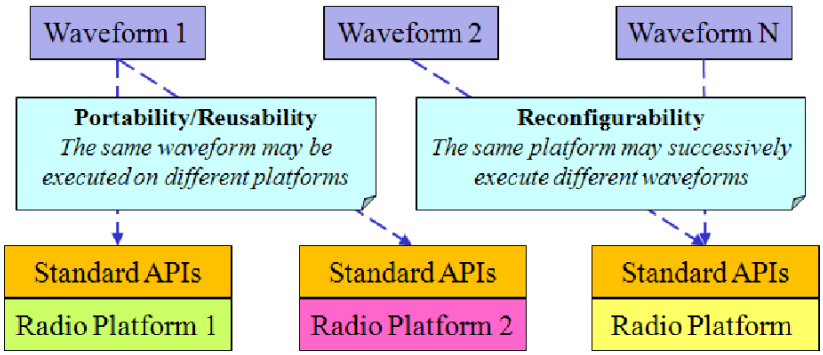


Figure 1.1: SDR needs

Multiple waveform applications may be loaded and unloaded from a radio platform according to user, operational or environmental needs.

The combination of software radio and artificial intelligence leads to the concept of *Cognitive Radio* (CR) proposed by Joseph Mitola III. A Cognitive Radio is a self-adaptive SDR system which adapts and optimizes its radio parameters according to its environment to efficiently use the available radio frequency spectrum and network resources, and better satisfy user's needs.

The *Joint Tactical Radio System* (**JTRS**) is a US Department of Defense (DoD) program aimed to create a global communication network of scalable and interoperable SDRs for US and allied terrestrial, maritime and airborne joint forces. The JTRS is built upon the *Software Communications Architecture* (**SCA**).

As depicted in figure 1.2, the Software Communications Architecture is a software architecture framework, which allows some independence between the waveform applications and the underlying radio platforms. The SCA specifies an *Operating Environment* (**OE**) in which waveform applications are executed. The OE enables the management of waveform applications and provides radio platform services. It is composed of a *Core Framework* (**CF**), a middleware compliant with *Common Object Re-*

quest Broker Architecture) (**CORBA**) for embedded (*CORBAe* previously called *Minimum CORBA*), and a *Portable Operating System Interface* **POSIX**-compliant *Operating System* (**OS**). The Core Framework is a set of interfaces and related behaviors used to load, deploy and run waveform applications.

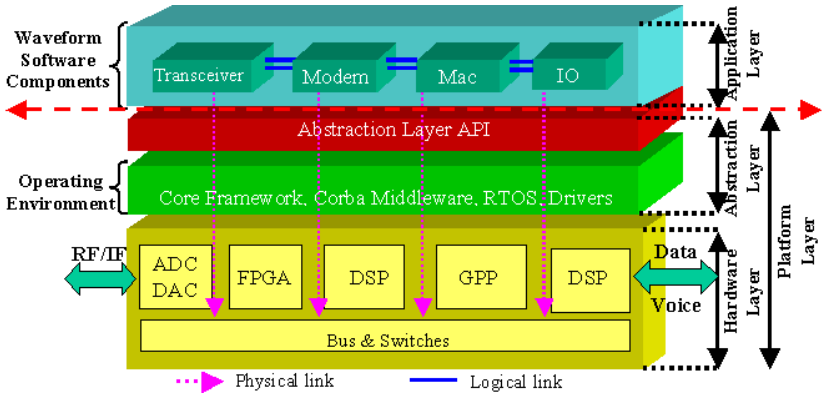


Figure 1.2: Heterogeneous and Distributed Radio Architecture

Beyond the ideal vision of software radio, some real-time signal processing treatments of waveform applications nevertheless require a huge computation power which can not be satisfied by current *General Purpose Processors* (**GPP**). They need dedicated processing units such as *Digital Signal Processors* (**DSPs**), *Field Programmable Gate Arrays* (**FPGAs**) and *Application Specific Integrated Circuits* (**ASICs**). An FPGA is a semiconductor device, which contain *Configurable Logic Blocks* (**CLBs**), I/O pins and a reconfigurable interconnection network that can be configured after manufacturing using an *Hardware Description Language* (**HDL**). Early versions of the SCA (2.0) only specified a high-level software architecture for GPPs, which is not adequate for highly-constrained real-time processing units such as DSPs, FPGAs and ASICs. The current version of the SCA (2.2) defines a *Modem Hardware Abstraction Layer* (**MHAL**) API to support the communications between GPP and non-CORBA capable devices such as DSPs, FPGAs and ASICs. However, the MHAL API is a low-level message passing API, which does not provide a high-level and common design approach for GPPs, DSPs, FPGAs and ASICs.

1.1.6 Software Communications Architecture (SCA) Requirements

In the SCA, software radio applications are broken down into object-oriented components to be deployed on radio platforms. These components provide and require abstract software interfaces specified in the SCA Core Framework as a set of operation signatures, which are described in the *Unified Modeling Language* (**UML**) profile for CORBA and in CORBA IDL. The portability and reusability of SCA components require that their abstract functional interfaces could be defined at a system-level regardless of their software or hardware implementation and can be indifferently translated in software and hardware implementation languages such as C, SystemC or **VHDL** (*Very-High-Speed Integrated Circuits (VHSIC) Hardware Description Language (HDL)*).

The interoperability between SCA components within the same heterogeneous platform require transparent communications regardless of their location and implementation.

The reconfigurability of radio platforms require standard hardware interfaces to physically connect hardware waveform components, but also standard control interfaces to software and/or hardware dynamic reconfiguration. Dynamic reconfiguration denotes the run-time modification of the architecture of component-based applications by dynamically adding/removing components. Dynamic reconfiguration may be required to dynamically switch from one waveform application to another and allows the dynamic deployment of hardware/software waveform components.

In addition to the SCA component model, we consider the general-purpose component model of the CORBA middleware called *CORBA Component Model* (**CCM**). In particular, we focus on a reduced version of CCM which is dedicated to embedded systems and called *Lightweight CORBA Component Model* (**LwCCM**). The main reason of this choice is that LwCCM is a software industry standard maintained by an open industry consortium called *Object Management Group* (**OMG**). For instance, UML and CORBA are OMG standards. Moreover, Thales is an active OMG member.

1.1.7 Problems Formulation

The ambitious goal of this PhD thesis is to apply some successful software engineering concepts used in the SCA down to FPGAs to propose a common and unified approach of specification and implementation of hardware/software component-based waveform applications. These concepts are distilled from component-based software architecture and middlewares.

To address the portability requirement for SCA hardware components, we propose some requirements and mapping rules between an operation-based abstract software interface in CORBA IDL or UML, an operation-based concrete software interface in system languages such as SystemC at transactional level, and a signal-based physical interface in HDLs such as VHDL and SystemC at Register Transfer Level (RTL).

The interoperability requirement has been addressed by prototyping a middleware core which transparently implements memory-mapping and message-passing using two protocols: the General Inter-ORB Protocol (GIOP) used by CORBA middlewares and the SCA MHAL messages. As opposed to hardware implementation of commercial CORBA middlewares, we demonstrated the overhead of a request-reply GIOP message decoder compared to a MHAL message decoder, and a classical memory-mapped address decoder with the same functionalities on an application example implemented on FPGA.

The second objective of this PhD thesis is to evaluate the use of virtual platforms in SystemC at Transaction Level to improve the design and validation of hardware/software waveform applications on virtual radio platforms. Indeed, waveforms and platforms design becomes more and more complex (high data rate antenna processing, programmable RF...) and involves new methodological needs, that we try to address in this work. Indeed, executable specifications are needed to validate waveform consistency and compliance against specifications. Hardware/software partitioning and architecture exploration have to be studied to achieve the required performances, which have to be estimated and compared to Quality of Service (QoS) and real-time constraints. We modeled part of a MPSoC platform in SystemC using *Transaction Level Modeling* **(TLM)** and propose a SystemC-based design methodology.

1.2 Thesis Organization

In chapter 2, we provide an overview of the concepts, models and methodologies used to design embedded systems. In chapter 3, we present the on-chip communication architecture of buses and network-on-chips. In chapter 4, we present the fundamental concepts of the object-oriented approach used by the SCA and their applications in hardware design. In chapter 5, we provide an overview of middlewares used by the SCA to abstract the distribution of software entities and their hardware implementations in SoCs. In chapter 6, we present the concepts of the component-oriented approach in software and hardware engineering and stress their commonalities. In chapter 7, our unified object-oriented component and middleware approach is presented for both hardware and software components. In chapter 8, some experiments are described to validate our proposition. Finally, a conclusion summarizes our contributions.

Chapter 2

Models and Methodologies for Embedded Systems Design

Contents

2.1 Introduction

Embedded systems are more and more complex to design notably due to the increasing number of application needs in terms of performances to take into account and the heterogeneity of computation, storage and communications resources which are integrated in MPSoCs. This complexity is also due to

the design trade-offs to accomplish between cost, time, performances, the inherent concurrency issues and the huge exploration space all along the design flow. The design exploration space is a multidimensional problem with some orthogonal directions such as hardware/software partitioning, hardware/-software architecture, tools availability and maturity, reusability of previous designs, etc. The design of embedded systems requires skills in various domains such as software engineering and hardware engineering, along with expertise in the targeted application domain(s) such as signal processing. In order to manage such complexity, numerous models, languages, methods and methodologies have been developed to raise the abstraction level of design. Many design methods have been developed in industry and academia to address the specification, modeling, development and validation of embedded systems. Examples include models of computation and communication, Model-Based Design and Model-Driven Engineering, Platform-Based Design, Hardware/Software Co-design, Transaction-Level modeling and High-Level Synthesis. The objective of system, software, hardware engineers consists in choosing the better combination of models, languages, methods and methodologies that satisfy the application needs and the allocated cost and time budgets.

A fundamental problem is how to unambigously specify an application with sufficient details to perform a good implementation. Industry companies may either define custom specification languages and models or use existing standards to guarantee the perenity of application specifications, tools and design flows. In the latter case, companies need to identify and select the relevant standards to build a coherent design methodology.

This chapter is organized as follows. Section 2.2 presents modeling languages such as UML and SystemC, software programming languages such as C/C++ and Java, and hardware description languages such as VHDL and Verilog. Section 2.3 describes models of computation such as Finite State Machines and Synchronous Data Flow. Section 2.4 presents models of communication such as the OSI reference network model. Section 2.5 exposes parallel programming models such as shared memory model and the message passing model. Section 2.8 presents model-based design and model-driven engineering such as the OMG Model-Driven Architecture. Section 2.9 describes platform-based design for embedded systems. Finally, section 2.10 concludes this chapter.

2.1.1 Models and Methodologies

It is commonly accepted that engineering is all about modeling. modeling allows engineers to abstract a problem regardless of the scientific domain: mathematical model, data model, biological model, social model, mechanical model, electrical/electronic model, object/component model, etc. Generally speaking, a model represents the key characteristics of a problem and provides a conceptual framework to reason about it. A model is a specification of the function, structure and/or behavior of an application or system. This specification may be formal when it is based on a strong mathematical foundation. Otherwise, the specification is informal and may be subject to interpretations e.g. in case of textual specifications. In embedded systems design, several models may be required to capture all the functional and non-functional properties of a system such as behavioral and structural models. A complete design methodology relies on a collection of models. Each model focuses on different aspects of a system like its behavior, its architecture or its real-time constraints. A typical methodology consists on a number of steps such as specification, design, developing and validating the system. *Model Refinement* consists in successively enriching a model with additional details. For instance, a high-level behavioral model of a signal-processing treatment must be refined to be implemented as an software or hardware module in an electronic circuit. Models are formalized by languages. Like any human spoken language, a language have a *Syntax* and Semantics. The syntax of a language defines its well-formed structure, while semantics define its meaning and behavior [RIJ04]. A language may have a textual and/or graphical syntax. Moreover, a language may be based on mathematically-proven formal methods or be informal.

2.1.2 General Concepts of Design

The general concepts underlying the design of any system is *abstraction*, *encapsulation*, *modularity* and *hierarchy*. They provide valuable design practices. These concepts have been presented by Booch [BME+07] in the context of the object-oriented approach, which is presented in chapter 4, but they are quite general enough to be applicable to any design approach regardless of the engineering domain. In the following, we present these concepts as the main characteristics common to the models and methodologies presented in these chapter.

Abstraction

Generally speaking, an *abstraction* is a relative distinction made between relevant and less relevant details of a problem domain to enhance understanding and reduce complexity [Arm06, BME+07]. The relevant details become explicit, whereas the other details are implicit. Abstractions help the separation or orthogonalization of concerns in a system. Raising the abstraction level is the main objective of languages, models and methodologies created for system design. However, higher the level of abstraction is, bigger the conceptual gap between the specification and their implementation is. Several abstraction levels are thus required to present a manageable view of a system and ease the refinement from one level of abstraction to another. The key point is to define the right set of abstractions for a particular domain. According to Booch [BME+07], the quality of an abstraction may be measured by its coupling, cohesion, sufficiency, completeness and primitiveness. In this chapter, we will briefly present the main abstractions used in the today models and methodologies for designing complex embedded systems.

Encapsulation

Encapsulation consists in hiding the internal view of an abstraction in order to better deal with its complexity [BME+07]. The principle of encapsulation supports the well-known separation of concerns between the definition of an interface and its implementation in a software programming language or a hardware description language as proposed in Interface-Based Design [RSV97]. Encapsulation allows one to modify the implementation of an interface without changing the way users access to it. Encapsulation is often used a a synonym for an older concept called *information hiding* [Arm06] [BME+07].

Modularity

Modularity consists in breaking down a complex system into a set of simple and more manageable modules [BME+07]. A module represents a unit of design and reuse, which allows developers to work in parallel independently of one another. The objective of a modular approach is to improve cohesion within a module and reduce coupling among modules. A highly cohesive module encapsulates a set of related abstractions, while a loosely coupled module has few dependencies on other modules. Designers have to make a trade-off on the granularity of a module and their number: too many modules result in too many interfaces and possible communication overhead, while too few modules are less manageable. In the object-oriented approach presented in chapter 4, a module is an object, while in the component-oriented approach presented in chapter 6, a module corresponds to a component. Many languages support the concept of modularity as for example through packages in Java, Ada and VHDL, or through namespaces in C++.

Hierarchy

Hierarchy establishes an ordering among a set of abstractions [BME+07]. The decomposition of a system into hierarchical layers of abstraction is a key concept to design complex systems. An upper layer is built upon a lower layer. In particular, the explicit details of the lower layer becomes implicit in the upper layer. Examples of layered systems include the *Open Systems Interconnection* (**OSI**) model, the socket-based approach for on-chip communication buses, Network-On-Chips and middlewares presented in chapter 5.

Like for modularity, designers have to trade off the abstraction level of a layer and their number. A higher abstraction level is more independent from low-level details of a platform at the expense of less control over it. Conversely, too many layers result in important performance overhead e.g. in term of memory footprint, latency, bandwidth and silicon area.

2.2 Languages

In order to better manage the ever increasing complexity of embedded systems and to increase productivity, several specification, modeling and implementation languages have been developed since 1950s to raise the abstraction level at which design is performed. These languages allows designers to reduce development time and cost as less lines of code are needed to implement the same functionality. The brief history of languages shows that each new generation of language is build upon the previous one and resides at a higher-level of abstraction The trend is applicable to both software programming languages and hardware description languages. The often cited generation of software programming languages include [BME+07] [WQ05] [MSUW04]:

- Gear-Oriented Design with Pascal's mechanical computer in 1642;
- Switch-Oriented Design with the first electronic computer ENIAC[1] in 1942;
- Instruction-Oriented Design with assembly languages;
- Procedure-Oriented Design or Structured Design based on sub-programs with Fortran, Pascal, C and RPC IDL;
- Object-Oriented Design (OOD) with Smalltalk, Java, C++, Python, CORBA IDL 2.x, UML;
- Aspect-Oriented Design (AOD) [KLM+97] with AspectJ and AspectC++;
- Component-Oriented Design (COD) with ArchJava, CORBA IDL 3.x, UML 2;
- Model-Based Design (MBD) with ArchJava, CORBA IDL 3.x;

Each term before the word *oriented* designates the main design abstraction. For instance, the Object-Oriented Design considers an *object* as the main design unit.

In the same manner, hardware design has evolved to reach higher-level of abstraction.

- Transistor-level Design;
- Gate-level Design using AND, OR and NAND gates;
- Register-Transfer Level (RTL) Design with VHDL and Verilog, SystemC RTL;

[1] Electronic Numerical Integrator And Computer

- IP-based Design with SystemC;
- Transaction Level Modeling (TLM) with SystemC;
- Object-Oriented Design with object-oriented Hardware Description Language (HDL);
- Component-Based Design with hardware modules called Intellectual Properties (IPs);
- Model-Based Design with Model-Driven Engineering generating RTL code.
- Component-Oriented Design with SCA hardware components [JTR05] and **this thesis**;

This work is an attempt to unify component-oriented design for both software and hardware components.

2.2.1 Modeling Languages

Unified Modeling Language (UML)

UML [OMG07d] is a graphical object-oriented modeling language standardized by the Open Management Group (OMG). UML allows designers to model a complex system according to multiple viewpoints such as its structural and behavioral characteristics.

UML specifies thirteen diagrams divided into two main categories: Structural modeling and behavioral modeling. The structural diagrams include package diagrams, component diagrams, deployment diagrams, class diagrams, object diagrams, composite structure diagrams, interaction overview diagrams and communication diagrams. The behavioral diagrams include use case diagrams, activity diagrams, sequence diagrams, state machine diagrams, timing diagrams. Obviously, we cannot describe all these kinds of diagrams here, nevertheless some of them are briefly described in the following [BME+07]:

- A package diagram provides the means to organize the artifacts of the development process.
- A component diagram shows the internal structure of components and their dependencies with other components. This diagram provides the representation of components, collaborating through well-defined interfaces, to provide system functionality.
- A deployment diagram shows the allocation of artifacts to nodes in the physical design of a system. During development, we use deployment diagrams to indicate the physical collection of nodes that serve as the platform for execution of our system.
- A use case diagram depicts the functionality provided by a system. Use case diagrams depict which actors interact with the system.
- An activity diagram provides the visual depiction of the flow of activities. These diagrams focus on the activities performed and who (or what) is responsible for the performance of those activities.
- A class diagram captures the structure of the classes and their relationships in the system.
- A sequence diagram represents a temporal sequence of operation calls
- An interaction overview diagram is a combination of activity diagrams and interaction diagrams to provide an overview of the flow of control between diagram elements.

- A composite structure diagram provides a way to depict a structured classifier with the definition of its internal structure. This diagram is also useful during design to decompose classes into their constituent parts and model their runtime collaborations.
- A state machine diagram expresses behavior as a progression through a series of states, triggered by events, and the related actions that may occur.
- A timing diagram shows how the states of an element or elements change over time and how events change those states.
- An object diagram shows the relationships between objects in a system
- A communication diagram focuses on how objects are linked and the messages they pass.

For further information of UML, the reader can refer to the UML specification [OMG07d] and [RIJ04].

UML profiles allow designers to define Domain-Specific Modeling Language (DSML) by extending the UML syntax with model annotations called *stereotypes*. A number of UML Profiles has also been standardized by the OMG. The most relevant UML profiles for this work include the UML Profile for System on a Chip (SoC) inspired from SystemC, the UML Profile for modeling and Analysis of Real-Time and Embedded Systems (MARTE) [OMG08h], the UML Profile for CORBA and CORBA Components (CCM) [OMG08g], the UML profile for Software Radio (a.k.a. PIM & PSM for Software Radio Components) [OMG07e] inspired from SCA [JTR01].

For instance, the SCA is specified using the UML profile for CORBA v.1.0 [OMG02] and standard UML diagrams such as class diagrams, use cases diagrams, sequence diagrams and collaboration diagrams. Furthermore, the UML Profile for Software Radio a.k.a. PIM & PSM for Software Radio Components (SDRP or P^2SRC) is based on the SCA [OMG07e]. As opposed to the SCA, the SDRP specification is not yet widely used to design software radio applications and supported by commercial tools. Another UML profile for SDR called *A3S UML profile* was defined in the scope of the A3S RNRT project [S.06] in which Thales participated.

SystemC

SystemC [GLMS02] is an object-oriented hardware/software modeling language based on a C++ class library maintained by the *Open SystemC Initiative* (**OSCI**). SystemC is based on early works from Liao et al. [LTG97] in the Scenic framework to model hardware/software systems. Thanks to its growing acceptance in the industry, SystemC has been standardized as the IEEE standard 1666 in 2005. To provide hardware modeling capabilities, SystemC extends C++ with architectural constructs such as modules, ports and signals, and behavioral constructs with a notion of time, events and concurrent sequential processes implemented as co-routines. In this language, hardware modules are declared as `sc_module` and contain hardware ports called `sc_ports`. These ports are bound to an `sc_interface`, which allows the separation between computation and communication through a mechanism called `Interface Method Call` (**IMC**). This interface is implemented by an `sc_module` that is typically an `sc_channel`.

2.2.2 Software Programming Languages

C/C++

The C language is a procedural and portable language which was initially created for system programming on the Unix operating system. Nowadays, it is widely used for the programming of embedded

systems notably thanks to the error-prone concept of *pointer* which allows developers to directly access system memory and memory-mapped hardware registers. C++ is a general-purpose language that extends C with object-oriented capability of C++ and standard high-level library such as the *Standard Template Library* (**STL**). For instance, software waveform components are mainly implemented in C/C++ on embedded radio platforms. C/C++ may also be used to build functional model of hardware waveform components.

Java

Java is a general purpose, concurrent, object-oriented and platform independent programming language. Java is compiled into abstract instructions called *bytecode* in *.class* files. This bytecode is executed by a run-time environment called the Java Virtual Machine (JVM). Java may also be compiled to a native instruction set using the GNU Compiler Collection (GCC). Due to its interpreted nature, Java is not widely used for embedded systems and notably for radio applications.

2.2.3 Hardware Description Languages (HDL)

A *Hardware Description Language* (**HDL**) is a language to formally model electronic circuits. Whereas software programming languages are compiled into a binary code executed on an instruction-set processor, HDLs are synthesized into a piece of hardware typically on Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit **ASIC**. An HDL allows hardware designers to describe at Register Transfer Level (RTL) the architecture of hardware building blocks, whose temporal and logic behavior is checked thanks to simulation tools i.e. simulators. The two main Hardware Description Languages (HDL) are the Very-High-Speed Integrated Circuits **VHSIC** Hardware Description Language (**VHDL**) and Verilog. Interestingly, HDLs were originally developed for the documentation and simulation of gate-based schematics. HDLs may model discrete events for digital hardware and continuous time for analog hardware. As hardware is inherently parallel, an HDL allows to explicitly specify concurrency through concurrent statements and time with wait statements for simulation purpose. HDLs support appropriate data types to model hardware modules like bit arrays.

Logic *synthesis* is a process by which a RTL model specified in an HDL is inferred into a set of interconnected logic gates described by a *netlist*. The hardware model have to typically respect some coding guidelines imposed by synthesis tools called *synthesizer* in order to produce the required behavior. These guidelines include the use of a synthesizable subset of HDLs and some code templates to facilitate the inference of memory units such as registers, RAM/ROM or hardware operators like multipliers. For instance, high-level HDL constructs, which are used for simulation in testbenches, are typically not synthesizable. In a similar way than code written in low-level assembly languages is more optimized, but less portable than code written in portable programming languages like C, HDL code based on vendor-specific modules is optimized for a given FPGA platform, but is not portable for another platform, while HDL code written with generic modules are portable, but not optimized. Hence, it is not always possible to write portable code for FPGAs.

2.2.4 Structured Data Description Language

Structured data description languages are used to describe user-defined data and the data used to described the user-defined data themselves called *meta-data*. The eXtensible Markup Language (XML) is a structured data description language with use markups to describe information as plain text in files with the `.xml` extension. The content and structure of XML documents are constrained by rules expressed in a XML Schema Language such as *Document Type Definition* (**DTD**) with `.dtd` extensions and XML

Scheme with `.xsd` extension. DTD files describe a type of XML documents as a structured set of elements that contain data and attributes. For instance, XML is used in software component models such as CCM and SCA, meta-data for hardware components such as SPIRIT, and storage of UML models such as XMI.

2.3 Models of Computation

A *Model of Computation* (**MoC**) is the formal abstraction of the execution in a computer. It allows one to formally describe and reason about concurrent and distributed systems. A model of computation defines the behavior and interaction semantics that govern the elements in different parts or at different levels of a system. As its name does not suggest, a model of computation describes the semantics of communication between components. It contains operational rules which drive the execution of a model in particular when components perform computation, update their state and perform communication [LN04]. Models of computations are to actor-oriented design what design patterns are to object-oriented design and what communication protocols are to signal-oriented design [LN04]. Many models of computation have been identified in the literature:

- *Synchronous Data Flow* (**SDF**) is a static data flow in which the number of data token produced or consumed by each processing node is known at design time [LM87]. Nodes communicate through queues and can be statically scheduled at compile time. Signal processing applications can be typically described as a synchronous data flow. They can be specified by a *Data-Flow Graph* (**DFG**) where each node represents a processing block and each arc a data path. This model has been used to describe both hardware and software systems.

- In the *Continuous Time* (**CT**) MoC, processing elements communicate via a continuous time signal at a given time point. Such systems are typically represented by differential equations like physical systems.

- *Communicating Sequential Processes* (**CSP**) communicate via synchronous message passing or *rendezvous*.

- *Process Networks* (**PN**) like *Kahn Process Networks* (**KPN**) communicate via asynchronous message passing. Sequential processes communicate through unbounded FIFO channels. KPNs are notably used to model signal processing applications e.g. see our experiments in Â§8.3.2.

- *Finite State Machines* (**FSM**) are represented by *Control-Flow Graph* (**CFG**) in which a node represent a state and an arc a state transition triggered by some boolean expression called guard.

- *Discrete-Events* (**DE**): processing nodes interact via an event queue. This model is used to model digital circuits, asynchronous circuits and events in physical systems.

- *Petri Nets* (**PN**): processing nodes performs asynchronous computation with explicit synchronization points. The UML2 activity model is inspired from Petri nets [RIJ04].

- *Synchronous Reactive* (SR): computation elements update the values of their output until the system stabilizes. This MoC is used in synchronous languages like Esterel, Lustre and Signal used to describe both hardware and software systems.

For instance, a domain-specific specification language called *Waveform Description Languages* (**WDL**) has been proposed by E.W. at Thales Reseach Inc. to specify waveform applications. WDL is a

component-based graphical language where components have data-flow ports and is based on a Synchronous Data Flow model of computation.

Various modeling frameworks such as Ptolemy [HLL+03], Metropolis [BWH+03] and BIP [Sif05] propose to model and simulate embedded systems through the hierarchical composition of computation models [GBA+07].

As far as SDR is concerned, waveform applications are roughly based on FSM for the control path e.g. to manage a mode switch and a synchronous data flow for the data path.

2.4 Models of Communication

2.4.1 ISO OSI Reference Model

The International Organization for Standardization (ISO) Reference Model (RM) of Open Systems Interconnection (OSI) is an abstract layered communication architecture organized as a stack of seven layers: Physical, Data Link, Network, Transport, Session, Presentation and Application [Zim80]. The OSI RM provides a common conceptual framework to describe standard protocols.

Basically, a computer network is a set of computers called *host* which communicate to one another. The seven layers are implemented on each host, while only the tree lowest layers are used in the intermediate routers.

Such a layered approach allows an independent modification of the implementation of each layer, while their interfaces and behaviors must remain the same. A system is logically divided into successive horizontal layers from the Application layer at the top to the Physical layer at the bottom.

Each layer provides and requires a set of services to the next upper and lower layer through interfaces called *Service Access Point* (**SAP**), which are implemented by groups of functions called *Entities*. Within the same horizontal layer, two entities on different hosts virtually communicate using a peer-to-peer protocol based on a standardized message format called *Protocol Data Unit* (**PDU**). In reality, the data are transferred through the protocol stack of the lower layers. Each successive protocol layer encapsulates the data received from the previous layer with a layer-specific control header to build a new message. The layers do not know the meaning of the encapsulated data, the message format or protocol used to provide a complete decoupling of each layer.

The ISO OSI reference model is structured as a hierarchy of seven layers.

- The **Physical Layer** (PHY) deals with the transfer of raw *bit* stream on a medium (cable, air, water, space, etc). This layer defines the electrical and mechanical characteristics for transmission such as wire voltages, timing, frequency, width. Examples include serial (RS232), Ethernet (10/100/1000BASE-T) and WIFI (802.11 PHY).

- The **Data Link Layer** is in charge of the reliability of *frames* used by the network layer. This layer performs physical addressing and error control by introducing additional bits in the raw bit stream to detect and correct transmission errors. It typically contains two sub-layers called *Medium Access Control* (**MAC**) and *Logical Link Control* (**LLC**). Examples include Ethernet MAC (802.3), WIFI MAC/LLC (802.11), ATM and HDLC.

- The **Network Layer** is responsible for the delivery and routing of packets used by the transport layer. This layer performs logical addressing and flow control to avoid congestion in the routers. Examples include the Internet Protocol (IP) and Internet Control Message Protocol (ICMP) used by ping and traceroute tools.

- The **Transport Layer** establishes an end-to-end communication between hosts using the network layer. Examples include TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).
- The **Session Layer** copes with setting up, managing and tearing down connection between application processes based on the transport layer between hosts. Examples include *Inter-Process Communication* (**IPC**) such as named pipes and sockets.
- The **Presentation layer** is concerned with the transformations of application data prior to transmission such as compression, encryption, conversion called *marshalling* to a common data representation. Examples include ASCII, MPEG, MIME, XDR, SSL, CORBA CDR.
- The **Application Layer** is used by the user applications for distributed communications. This layer performs synchronization between application processes. Examples include HTTP, FTP, DNS and CORBA GIOP

For instance, SDR applications are an assembly of waveform components, whose functionalities are organized as in the OSI model: digital signal processing for Modem components in the physical layer, medium acess control for MAC components, data link for Link components, routing for Network components and input/output access for I/O components.

2.5 Parallel Programming Models

A parallel computing architecture contains a set of processing elements that communicate and cooperate to execute applications according their performances requirements. Concurrent software applications executed on parallel computing architectures need to be developed according to one or several *parallel programming models*. A programming model defines how parts of an application communicate with one another and coordinate their activities [CSG98].

The two main programming models which have been traditionally identified for parallel computing architectures are shared memory and message passing. Other programming models include data parallel processing i.e. Single Program Multiple Data (SPMD), data-flow processing and systolic processing [CSG98].

A parallel programming model allows developers to abstract the heterogeneity of different hardware/software platform architectures and to abstract communication using an Application Programming Interfaces (API). This API specifies communication and synchronization operations which are used by the applications and implemented by the platform using compilers and libraries from languages, OS and middlewares. Parallel programming models must balance two contradictory objectives: reduce development time and cost using higher-level programming model, and improve system performances using lower-level programming model [JBP06]. As opposed to classical APIs for desktop and enterprise computers such as *Portable Operating System Interface* (**POSIX**) and *Message Passing API* (**MPI**), APIs for MPSoCs may be not standard and customized for a particular platform and their implementation must be optimized to satisfy the performance requirements of a specific application or application domain.

Each programming model provides different trade-offs in terms of performance, time, space and synchronization coupling [LvdADtH05]. For instance, the MultiFlex platform [PPL+06b] supports both a CORBA-like message passing model and a POSIX-like shared memory model.

2.5.1 Shared Memory Programming, Communication and Architecture Model

In the shared memory model, two or more application entities communicate through shared variables by writing and reading data at pre-defined addresses in the same shared address space. This memory

is shared and accessible by all participants in the communication. Application data do not need to be copied because they are directly accessible. The application entities reside in the same address space. The shared memory model implies a tight coupling between communicating entities due to synchronization. Since the memory locations are shared, only one writer at a time is allowed on a given address to ensure memory consistency, while several readers can read this location without harm. The writer must acquire and release a lock before and after a write. This requires special hardware support for indivisible or atomic operations for synchronization. Successive writes from several writers must be serialized to ensure ordered memory accesses and avoid race conditions. Standard APIs for shared memory include POSIX and synchronization primitives offered by programming languages. From a programming viewpoint, different processes/threads running on the same or different processors may communicate through shared variables. The basic communication primitives are *read* and *write* operations, which are implemented by the load and store instructions of processors. From a hardware architecture viewpoint, a shared memory multiprocessor called *Symmetric Multiprocessor* (**SMP**) may be used with a *Uniform Memory Architecture* (**UMA**) where the access time to the shared memory is the same for all processors.

2.5.2 Message Passing Programming, Communication and Architecture Model

In the message passing model, two or more application entities communicate with each other by sending and receiving messages. The application entities reside in different address space and have their own private memory. Application data must be copied from the sender memory to the message and from the message to the receiver memory. The message passing model allows a loose coupling between communicating entities e.g. via buffering. The basic communication primitives are send and receive operations. Standard APIs for message passing include MPI [For08]. From a programming viewpoint, processes/threads running on the same or different processors may communicate through message queues, OS or middleware calls. From a hardware architecture viewpoint, each processing unit has its own private memory and form a dedicated node which communicate with other nodes through the network e.g. a bus.

2.5.3 Combination of Shared memory and Message Passing Model

Both models may be used independently and they may be combined. For instance, a memory access may be encapsulated in a message to be sent over the network [Bje05]. Each processing unit has its own memory which has either a private and a shared part or is private the processor can access a global shared memory. This architecture has a Non-Uniform Memory Architecture (NUMA) where the memory access time depends on the memory location. Indeed, a local memory access takes less time than a remote memory access in the local memory of another processor. The local memory controller determines if a memory access on the local memory or a message passing with the remote memory controller has to be performed.

Traditionally, the design of modems relies on a shared memory model in which DSPs read/write data, commands, parameters and status to/from FPGAs using register banks. Due to the heterogeneous and distributed nature of SDR platforms, the SCA introduces the use of middlewares such as CORBA and the SCA Modem Hardware Abstraction Layer (MHAL) for modem design. Hence, SDR applications and platforms may be developped using a combination of both programming models.

2.6 Traditional hardware/software design flow

The traditional hardware/software design flow is depicted in figure 2.1. First, an executable model of the system is developed according to system requirements such as the number of operations per second,

latency, bandwidth and power consumption - regardless of implementation languages. Then, hardware/-software partitioning of the system is performed based on application performance requirements using complexity analysis and profiling, and performance offered by the target platform(s). After hardware/-software partitioning, hardware and software design are led in parallel. Hardware design consists in hardware specification using textual description, graphical block diagrams and finite state machines, hardware modeling in VHDL or Verilog, behavioral simulation, synthesis for a given RTL library, and placement and routing on a given FPGA or ASIC chip. Software design consists in (object-oriented) software analysis, design and development using a target programming languages (typically C or C++) in embedded systems. Co-simulations of the system including both software and HDL code may be performed to detect eventual errors before integration on the target platform. Finally, the last step consists in integrating hardware and software on the target platform and validating that initial system requirements are satisfied.

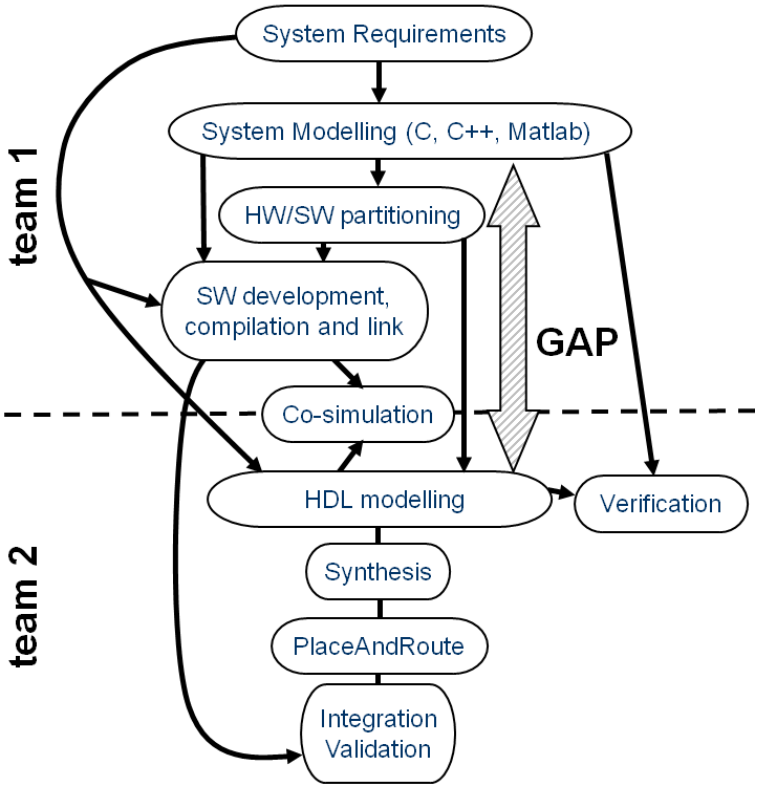


Figure 2.1: Classical HW/SW design flow

As systems are becoming more complex and implemented on multiprocessor platforms, hardware/-software partitioning is a difficult task to perform. Moreover, the partitioning study is usually performed without any tool, so only few solutions can be foreseen and studied, and significant margins have to be taken to avoid any bottleneck in the final implementation e.g. in terms of bandwidth. Since design space exploration cannot be exhaustive, sufficient margins must be taken and hardware/software partitioning is often sub-optimal.

Another drawback of this classical design flow is that software and hardware developments are separated. For example regarding timing constraints, it is necessary to allocate a timing budget at each side, while a global assignment could be more efficient. The software integration is possible only when the

hardware module are already modeled. Indeed, in the classical flow, no intermediate hardware model is available for software integration, hence hardware modeling and software integration cannot be performed in parallel. Moreover, hardware and software teams can make misleading interpretations resulting in difficult integration, rework and time loss. An important abstraction gap exists between algorithms captured by the C/C++/Matlab reference model, also called *business code*, and the final hardware implementation.

Face to the number of applications to be integrated on System-on-Chip platforms, new methodologies and tools are required to address the drawbacks of the classical hardware/software design flow. In particular, these methodologies must fill in the gap between system modeling and hardware modeling. In the next sections, we describe methodologies that may address these needs.

2.7 System-Level Design (SLD) with Transaction-Level Modeling (TLM)

System-Level Design (**SLD**) also called *Electronic System Level* (**ESL**) design consists in raising the abstraction level of design to specify a system more easily and quickly. SystemC [Ini08] is recognized as a de facto standard for System-Level Design. It allows the modeling of a system composed of hardware and software modules at different abstraction levels. System-Level Design may rely on a methodology called *Transaction-Level Modeling* (**TLM**) [GLMS02] [Ghe06]. Transaction-Level Modeling consists in modeling the communications within a system at different abstraction levels using function calls, rather than hardware signals, notably to increase simulation speed. The use of SystemC for Transaction-Level Modeling is usually referred to as SystemC TLM. Transaction-Level Modeling is based on different abstraction levels which are depicted in figure 2.2. These abstraction levels provide different views of a system that correspond to different use cases: *Functional View* (**FV**), *Programmers View* (**PV**), *Architects View* (**AV**) and *Verification View* (**VV**) Transaction duration, data granularity and timing considerations depend on the abstraction level. TLM allows designers to trade off modeling accuracy and simulation speed. The higher the abstraction level is, the faster the simulation speed is, but the less the modeling accuracy is. Designers must therefore choose the level of abstraction which is appropriate for their needs.

Transaction Level Model may help to fill in the gap between system modeling and hardware modeling. This gap implies at least two kinds of refinement: refinement of peripheral models and refinement of communication architecture. Transaction Level Modeling supports both refinements. The refinement from one abstraction level to another is validated by a functional testbench. **Transactors** are transaction adapters which allows mixing models at different levels of abstraction. Successive **Bus Functional Models** (**BFM**s), for instance a generic OCP Transaction Level (TL)2 or a cycle accurate AHB BFM, can be used for communication architecture refinement down to RTL implementation. Concrete examples of peripheral devices and bus functional models at Programmer View and will be presented in chapter 8.

Providing a high-level virtual model of the hardware platform to the system and software engineers allows developing software in an environment close to the final target and to develop the HDL code in parallel, which reduces the system implementation duration. The virtualization of the hardware platform also eases design space exploration, as the description level of the platform is sufficiently high to allow fast simulations of the architecture. This approach can mainly address the design needs of software validation and architecture exploration.

A complementary refinement approach is the automatic refinement from TLM to RTL, which address the productivity need and communication architecture refinement. An example is proposed by the TRAIN methodology [KGG06] in which abstract TLM 1.0 communication channels using `get` and `put` primitives are mapped on the on-chip communication architecture, e.g. a CoreConnect bus, via virtual channels. Another example is the synthesis of TLM 1.0 channels in Forte Design Cynthetiser [Sys]

View	Abstraction Level	Simulation Speed	Modeling Accuracy		Use Case
			Data	Timing	
	Algorithmic Level (**AL**)		Tokens	Function calls	System Functionalities
Functional View (FV)	Communicating Processes (**CP**)	> 10 MHz	No memory map, packets	Point-to-Point, No timing	Functional Partitioning
	Communicating Processes with Timing (**CP+T**)			Timing estimations	
Programmers View (PV)	Programmers View (**PV**)	1/10 MHz	Memory map, bit accurate, burst of words, no contention	Generic protocol, IRQ	Software Validation
	Programmers View with Timing (**PV+T**)			Timing estimations, concurrent tasks	
Architects View (AV)	Cycle Approximate (**CX**)	100 KHz/1 MHz	Uninterrupted burst of words	Transaction accurate, approximate timing	Architecture Exploration
Verification View (VV)	Cycle Accurate (**CA**)	10/100 KHz	Word transfers, not pin accurate	Cycle accurate	Hardware Verification
	Register Transfer (**RT**)	<10 KHz	Signal, bit vector, pin accurate	Clock Cycle accurate	Hardware Synthesis

Figure 2.2: Abstraction Levels

The second refinement approach is High-Level Synthesis (HLS) that consists in the automatic generation of HDL code using the system model with tools such as Mentor Catapult [Gra07] and Mathworks Simulink [Mat08]. As a single team is involved in the translation of system requirements, some misinterpretations may be avoided. Moreover, rework is quite simple as automatic re-generation can be performed quickly. Nevertheless, most of these tools mainly address data flow oriented digital signal processing applications, while control and scheduling functionalities are much less supported and may require third-party tools. HLS only satisfies the productivity need of coding and the translation of peripheral models. As there is no intermediate model between the high-level algorithmic model and its RTL implementation, this approach is similar to the hardware classical flow and its drawbacks: slow simulation speed and time consuming verification. A trade-off between simulation speed and modeling accuracy is no more possible and this does not provide the good level of details to address use cases such as software validation and architecture exploration.

The OSCI published a first version of TLM APIs called TLM 1.0 [RSPF05] to provide common modeling interfaces and allow portability of TLM models across academic and industry organizations. However, the format of TLM 1.0 messages was not standardized. As a result, TLM 1.0 models were not interoperable. So, the OSCI published a second version of TLM APIs called TLM 2.0 [OSC09] to better support interoperability between TLM models. TLM 2.0 notably defines a generic payload for TLM messages with ignorable extensions for advanced bus features.

In addition, working group of OCP-IP dedicated to System Level Design proposed a TLM API for OCP [OI07]. This modeling is based on three TLM layers above the RTL layer: a Message Layer (TL3), a Transaction Layer (TL2) and a Transfer Layer (TL1), while the RTL Layer is called TL0. A TLM standardization agreement was signed between OSCI and OCP-IP [OI04]. Besides, OCP TL2 and TL3 can be mapped to the OSCI TLM API [THA05], and both TLM APIs can be mixed using transactors, as presented in chapter 8. The PV API from OSCI is a standard interface enabling SystemC IP reuse, while OCP TLM layers are standard user layers, above OSCI foundation layer [THA05].

A partnership exists between OSCI and OCP-IP to guarantee some coherence between TLM interfaces.

2.8 Model-Based Design (MBD) and Model-Driven Engineering (MDE)

A model is a formal specification of the function, structure and/or behavior of an application or system. *Model Driven Engineering* (**MDE**) is a promising design methodology in which *models* are the cornerstone of system development. The objective is to extensively use models all along the design flow of a system from specification to design and implementation. A model allows designers to raise the abstraction level by hiding implementation details and to better manage complexity usually via hierarchical visual models. Models may be described at different abstraction levels and may focus on specific aspects or views of a system such as its structure or behavior.

Model-Based Design (**MBD**) aims at developing customized design environments which are tailored to a particular application domain e.g. automotive, avionic or radio domain. With MDE, designer's work is shifted from code development to system modeling. Designers describe a system using domain-specific concepts and their relationships. The most popular MDE approaches are the OMG initiatives called *Model-Driven Development*™(**MDD**™) and *Model Driven Architecture* ®(**MDA**®) [MSUW04]. These initiatives are naturally based on the UML modeling language which is standardized by the OMG. Other closely related terminologies for MDE include *Model-Based Design* (**MBD**) and *Model-Integrated Computing* (**MIC**) [SK97].

Model-Based Design places an abstract model of an algorithm at the center of the design cycle and provides a coherent view in all design phases: executable specification, simulation, test and verification,

and implementation based on automatic code generation.

An often-cited example of Model-Based Design tools is MathWorks Simulink [Mat08]. MathWorks Simulink is a graphical modeling tool for algorithms and systems whose models provide an executable specification. Designers may interconnect predefined processing blocks and configure them, or they may describe an algorithm in the Matlab language called `M` language. Code generation allows the synchronization between the model and its implementation. Designers can select particular platforms or target specific tools such as TI Code Composer, Altera DSPBuilder or Xilinx System Generator.

In addition, a design methodology for SoC/SoPC development is proposed based on MDD and the MARTE UML profile in the scope of the MOPCOM research project [AKL+09] . The MOPCOM toolset notably supports HDL code generation within the Rhapsody UML modeler.

2.8.1 OMG Model-Driven Architecture (MDA)

The Model-Driven Architecture (MDA) [OMG03] [MSUW04] is the Model-Driven Engineering approach standardized by the OMG. The MDA promotes the separation between the specification of system functionalities captured in a *Platform Independent Model* (**PIM**) from the specification of their implementation on a particular platform defined in the *Platform Specific Model* (**PSM**).

Platform Independent Model (PIM)

One or several PSMs can be associated to the same PIM. PIMs and PSMs may be specified in UML. For instance, the PIM may be defined using standard UML or a domain-specific UML profile like the UML profile for SDR [OMG07e], while the PSMs may rely on platform-specific UML profiles such as the UML profile for CORBA and CCM. A PSM results from the mapping of a PIM on the implementation technologies available on a particular hardware/software platform such as a programming language like C, C++, Java, SystemC and VHDL, a meta-data language like XML or a middleware platform like .NET, EJB and CORBA.

For instance, the UML Profile for Software Radio [OMG07e] leverages the SCA using the MDA approach in which PIMs of SDR applications are decoupled from their realization in PSMs such as CORBA.

MDA raises the level of abstraction and reusability and introduces the concept of *design-time interoperability* [MSUW04]. Indeed, MDA aims developers to build independent and reusable models of a system and to combine or glue them "only at the last minute" to build and deploy the system. These models are typically an application model and a platform model which are mapped together using a deployment model to build a system. Model transformations may also be used to generate adapters [KMD05] or interoperability bridges [Bon06]. The MDA is based on three main OMG standards: the *Unified modeling Language* (**UML**) [OMG07b] for modeling, the *Meta Object Facilities* (**MOF**) [OMG05a] [OMG06c] for meta-modeling and the *XML Metadata Interchange* (**XMI**) [OMG07c] for storing or *serializing* models and meta-models as persistent XML files.

MOF is a meta-model and modeling language to describe meta-models. A meta-model is a model of a modeling language. A meta-model is a model whose instances are the data types of another model. It specifies the concepts of the language used to define a model. A meta-model defines the structure, semantics and constraints for a family of models conforming to this meta-model. Each model is captured by a particular meta-model.

Meta-models allow the definition of model transformations from a source model conforming to a source meta-model to a target model conforming to a target meta-model. XMI is a mapping from MOF to XML. XMI is a XML schema allowing the exportation and importation of UML models from/to UML modeling tools. XMI supports the interoperability and integration of different tools along the design

flow. Any modeling language conforming to MOF can be associated to a XML schema describing its structure.

A UML profile exists for MOF allowing the description of meta-models in UML. A MOF to CORBA IDL mapping allows the manipulation of meta-models and its conforming models. Furthermore, the *Java Metadata Interface* (**JMI**) is the mapping of MOF to Java.JMI allows the generation of Java interfaces and their implementation to manipulate a meta-model and the corresponding models. The generated interfaces support the importation and exportation of models in XMI.

The objectives of the MDA are to increase the reusability of existing applications using models and to ease the integration of new technologies and platforms. The maintenance of an application does not consist in changing the application code but the high-level model of the application itself. To target another platform, developers select the corresponding target model and generate the specific implementation code.

The MDA is based on four abstraction layers:

- M0 contains the application data e.g. the name of a class instance
- M1 contains the model of the application data e.g. a class diagram
- M2 contains the meta-model of the modeling language e.g. UML concepts such as class, attribute and instance.
- M3 contains the meta-meta-model describing the properties of the meta-data e.g. MOF with the concept of class. M3 is self-descriptive and no other level is thus required.

An example of this four-layer meta-model hierarchy is depicted in Figure figure 2.3.

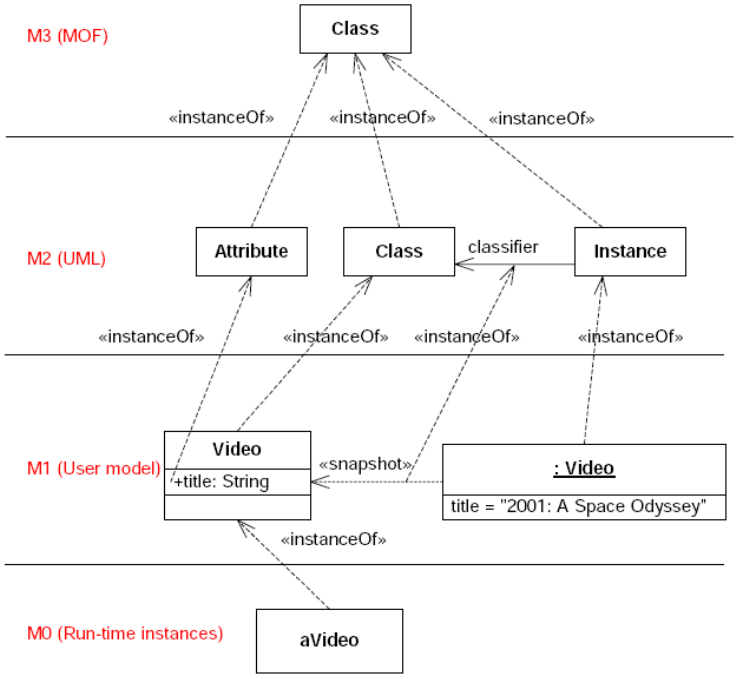


Figure 2.3: Example of this four-layer meta-model hierarchy [OMG07b]

Model Transformation

A fundamental concept of the MDE/MDA approach is the creation of models at different abstraction levels and the transformation of models to map them together and build an implementation. This concept is for instance applied by compilers and assemblers which automatically map a high abstraction level to a lower abstraction level e.g. from a high-level programming language to assembly language and machine code. A system is modeled in a PIM model using a modeling language such as UML which is described by a meta-modeling language such as MOF. The meta-model may be tailored to the particular business concepts of an application domain like SDR. The definition of a meta-model for an application domain requires the experience of a domain expert.

Transformation Rules

Model transformation consists in translating a source model M1 conforming to the meta-model MM1 into a model M2 conforming to the meta-model MM2 using mapping rules. A model transformation converts a model at a given abstraction level to another model at another abstraction level. The mapping rules are reusable and systematic design decisions as opposed to code-driven engineering where the mapping rules are implicit. Mapping rules are defined at the meta-model level and are not dedicated to a particular model. They are applicable to the set of source models conforming to the meta-model for which the mapping rules have been defined. Automatic model transformations reduce development time and cost while increasing software quality.

The principles underlying the MDE/MDA approach are not restricted to UML and may be used for *Domain-Specific Languages* (**DSL**) and *Domain-Specific modeling Languages* (**DSML**).

Face to the heterogeneity of specification, simulation, execution languages, and abstraction levels, a model-based approach may provide a separation of concerns between the applications and the hardware/-software platforms, the mapping between them and the interoperability between the different languages and abstraction levels.

The MDA approach may support the specification of application models and hardware architecture models using UML tools, the reuse of functional and physical IPs, the refinement between abstraction levels using mapping rules, the interoperability between abstraction levels during co-design and co-simulation, and the interoperability between tools using standards such as XMI [DMM+05].

As model transformation is at the heart of the MDA approach, the OMG has standardized a model transformation standard called MOF *Query / Views / Transformations* (QVT) [OMG08e]. The QVT specification is organized into a two-layer declarative part and an imperative part. The two-layer declarative part includes two declarative transformation meta-models and languages at two different abstraction levels called *Relations* and *Core* with standard mapping rules from the Relations to the Core language. The Relations language has a textual and a graphical syntax. The imperative part contains *Black Box* implementations and an imperative transformation meta-model and language called *Operational Mappings* which extends both Relations and Core languages. QVT/BlackBox allows model transformations with non-QVT languages, libraries and tools such as any programming language with a MOF binding using the MOF-to-CORBA mapping or JMI, domain-specific libraries and tools. Implementations of QVT are developed in the Model to Model Transformation (M2M) [2] subproject of the top-level Eclipse modeling Project and by France Telecom R&D with SmartQVT[3] in the IST Modelware project [4]. Three transformation engines are developed in the M2M project: Atlas Transformation Language (ATL) [5],

[2] http://www.eclipse.org/m2m
[3] http://smartqvt.elibel.tm.fr
[4] http://www.modelware-ist.org
[5] http://www.eclipse.org/m2m/atl

Operational QVT Language (QVTO) a.k.a Procedural QVT (Operational Mappings), Relational QVT Language (QVTR) a.k.a. Declarative QVT (Core and Relations). ATL is a QVT-like transformation language and engine with a large community of users and an open source library of transformations. SmartQVT is an open source model transformation tool implementing the MOF 2.0 QVT-Operational language. Other transformation languages similar in capabilities to QVT include Tefkat, Kermeta, and GREAT.

The MDA approach has been proposed for the co-design of embedded system notably by the LIFL [DBDM03] [DMM+05] and to map waveforms to radio platforms [GTS+05].

In addition, a methodology for SDR based on MDA and a dedicated UML profile were proposed in the scope of the A3S RNRT project [S.06] in which Thales participated. In the A3S design flow, the modeling and the specification of the application is performed in a PIM. In parallel, the modeling and the specification of the hardware platform is realized. Then, both models are mapped during hardware/-software partitioning to build the PSM. Non-functional properties may be verified such as scheduling and resource usage. The A3S design flow relies on a library of abstract UML components representing a library of concrete hardware and software components.

2.9 Platform-Based Design (PBD)

Platform-Based Design (**PBD**) [SV07, CSL+03, SV02] consists in mapping an application model and a platform model successively through various abstraction levels. PBD is neither a top-down or bottom-up approach, but a middle-out approach, where an application instance from the application design space and a platform instance from the platform design space are mapped together to form the desired embedded system. The application model is an abstract and formal application specification, typically an algorithm, whereas a platform model corresponds to a specific implementation. An application model describes the functionalities to be implemented and the required services that should be provided by the platform model characterized by platform constraints. This methodology is similar to the Gajski's Y-diagram, however the result of a mapping of an application to a platform model at level N is the application model at the level N+1 that will be mapped on the platform model at level N+1. A platform-based taxonomy reviews in [DPSV06] the existing methodology and tools from a platform viewpoint. Moreover, UML has been used to specify application and platform models in PBD [CSL+03].

2.10 Conclusion

In this chapter, we presented the main concepts, models and methodologies used to design hardware/-software embedded systems. These notions will be encountered all along this document.

As far as the SCA is concerned, the UML profile for CORBA v.1.0 is used to specify abstract software interfaces, which are typically implemented in C/C++ by application and platform components. The UML Profile for Software Radio a.k.a. PIM & PSM for Software Radio Components (SDRP or P^2SRC) is based on the SCA and leverages its architecture with the Model-Driven Architecture (MDA) approach. Waveform applications are roughly based on Finite State Machine for the control path e.g. to manage a mode switch and a synchronous data flow for the data path. SDR applications are an assembly of waveform components, whose functionalities are organized as in the OSI model. In this work, we focus on the functionalities of the physical layer of SDR applications, which are implemented on FPGAs. Traditionally, the design of modems relies on a shared memory model using register banks. Due to the heterogeneous and distributed nature of SDR platforms, the SCA introduces the use of middlewares such as CORBA and the SCA Modem Hardware Abstraction Layer (MHAL) for modem design. Hence, SDR applications and platforms may be developped using a combination of both programming models. Due

to the portability need of the SCA and its use of OMG standards such as UML and CORBA IDL, the SCA naturally tends to follow Model-Driven Architecture and Platform-Based Design.

As far as our work is concerned, we propose to combine the presented concepts, models and methodologies to build a common object-oriented component approach for embedded systems. Our objectives are to describe applications are a set of components whose interfaces are defined by abstract specification languages at system-level and to map these definitions to hardware, software or system implementation languages. We propose language mappings from UML/IDL to VHDL and SystemC RTL/TLM according to MDA. Application components have ports, which may be gouverned by different computation models such as FSM and synchronous data flow. These components are deployed on a component-based hardware/software platform following a platform-based design. To support transparent communications between distributed components, we propose a layered middleware architecture framework supporting both programming models and inspired from the OSI communication model.

Chapter 3

Hardware Communication Infrastructure Architecture

Contents

Numerous hardware communication interfaces and protocols exist for System-On-Chip such as ARM Advanced Microcontroller Bus Architecture (AMBA), Advanced eXtensible Interface (AXI) [ARM04], IBM CoreConnect and its Xilinx's implementation, Altera Avalon, OpenCores Wishbone [Ope02], and Open Core Protocol International Partnership (OCP-IP) OCP.

Some critical application domains such as SDR have hard-real time constraints and require Quality-of-Service such as guarantees regarding throughput, latency and arbitration policies.

In this chapter, we review some on-chip bus protocols used for communication between hardware components.

This chapter is organized as follows. Section 3.1 presents on-chip bus architecture and some examples such as AMBA and CoreConnect. Section 3.2 describes interconnection sockets such as WishBone, XI, Avalon, OCP and VCI. Section 3.3 presents network-on-chip characteristics such as topology, routing and quality of services, along with some examples of NoCs. Finally, section 3.4 concludes this chapter.

3.1 On-chip buses

In this section, we present some on-chip buses which are commonly used in System-on-Chips.

3.1.1 Shared and Hierarchical Bus Architecture

An on-chip bus is shared between all interconnected entities. These entities may have a master or slave role and associated to a bus address. A master initiates bus requests such as write or read commands, while a slave receives requests and may return read data. A bus is typically composed of a command bus, an address bus, write and read data buses. In write transfers, a master sends a write command, a write address within the slave address space and the data to be written at the given address in the slave. The slave receives the write message and writes the data at an address offset. In read transfers, a master sends a read command and a read address, while the slave returns the data at the given address. Only a single master can communicate with a single slave at each time.

A hierarchical bus is an assembly of two or more buses which are interconnected by bridges. It divides a SoC into different traffic classes such as a high-speed bus for high-performance IP cores such as processors and memories, and a low-speed bus for low peripheral IP cores like I/O devices.

A shared and hierarchical bus may have a *centralized* or *distributed* architecture. The centralized bus architecture is composed of a bus arbiter and an address decoder. The bus arbiter selects a unique master among the masters that compete for the bus according to some arbitration policies such as priority-based or round-robin schemes. The address decoder selects a unique slave among all slaves according to the address provided by the granted master. Examples of shared and hierarchical bus include ARM AMBA 2 and IBM CoreConnect.

The distributed bus architecture is based on a bus wrapper for each IP core. Wrappers are configured with TDMA slots, then each wrapper counts the clock cycles until it can transfer data. This provides a simple approach to provide guaranteed services using a traditional bus approach. An example is HIBI [Lah04] [SKH05].

3.1.2 ARM Advanced Microcontroller Bus Architecture (AMBA)

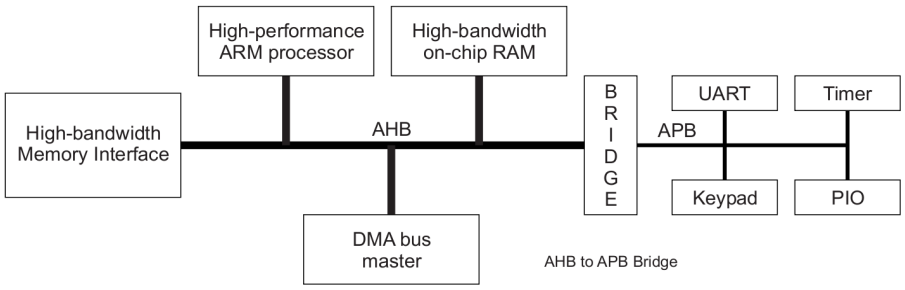


Figure 3.1: Example of AMBA-based System-On-Chip

As illustrated in figure 3.1, the Advanced Microcontroller Bus Architecture (AMBA) developed by ARM is an on-chip shared and hierarchical bus specification targeting high-performance System-On-Chips.

The AMBA specification defines three buses: the Advanced High-performance Bus (AHB), Advanced System Bus (ASB) and the Advanced Peripheral Bus (APB). This bus architecture consists of multiple AHB masters and AHB slaves, an AHB address decoder and an AHB bus arbiter. The AHB decoder serves to decode the address of a data transfer and to select the appropriate slave. This decoder is implemented as a single centralized decoder. The role of the AHB bus arbiter is to grant access to the AHB bus, the shared resource, to only a single bus master among the masters wanting to transfer data. The data bus width may be 32-, 64-, 128-, or 256-bit, while the address bus width is 32 bit. The

AMBA bus protocol pipeline supports single read/write transfer, burst transfer with 4, 8 or 16 beats and split transactions. The granularity of a data transfer may be a byte, half-word or a word. The AHB is used for high-performance communications between modules such as processors, memories and DMA controllers. The ASB is a subset of the AHB when all AHB features are not required by a SoC. The ASB is deprecated in favor of the most widely used AHB.

The APB is designed for on-chip peripheral devices that can be slow and require less high-performance communications and power consumption. The APB can be indifferently used with the AHB or the ASB. The address bus width is 32 bit. Its architecture supports a single master i.e. a AHB-APB bridge and multiple slaves. The APB does not support Pipelined and burst transfers. The APB protocol requires two cycles to perform a single read/write transactions. All these bus are interconnected to each other by a bridge.

For instance in the DiMITRI MPSoC for Digital Radio Mondiale (DRM) [QSC04] presented in section 8.2, an ARM processor and a Viterbi decoder IP core are interconnected by an AHB bus, while an APB bus is used for I/O IP cores such as analog RF frontends.

3.1.3 IBM CoreConnect

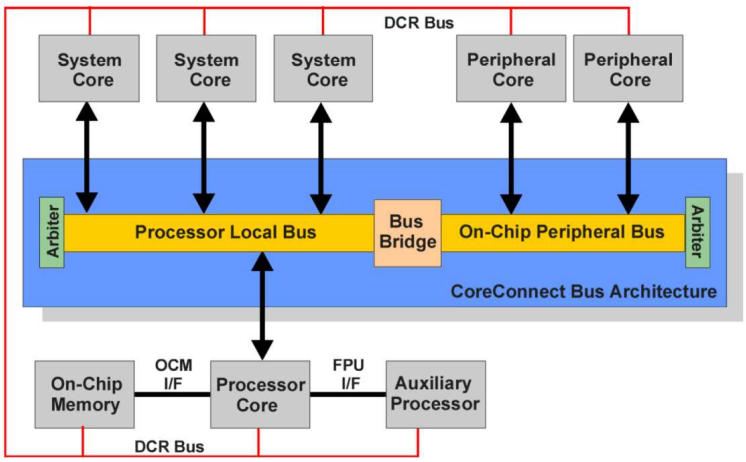


Figure 3.2: CoreConnect Bus Architecture

CoreConnect is a high-performance on-chip bus architecture developed by IBM and presented in figure 3.2. The CoreConnect bus architecture contains buses: the Processor Local Bus (PLB), the on-chip peripheral bus (OPB) and the Device Control Register (DCR) bus. The PLB bus is a high-performance and low latency bus for hardware modules such as processor core, memories and hardware accelerators. The OPB bus is designed for low speed and low power peripheral devices. The DCR bus is used to transfer data between the general purpose registers of a processor and the device control registers. DCR bus removes configuration registers from the memory address map, which reduces loading and thus improves bandwidth of the PLB.

As an example, the hardware implementation of CoreConnect in Xilinx FPGAs was used in the dynamic partial reconfiguration of FPGAs under the control of the SCA Core Framework [SMC+07] presented in paragraph 6.7.6.0.0 and in the hardware middleware prototype presented in section 8.3.3.

3.2 Interconnection Sockets

A socket is a point-to-point hardware interface used to connect an IP core to an on-chip communication bus. A socket allows the separation of concerns between computation and communication. The objective of socket should be independent from any interconnection network to improve IP cores portability, interoperability and reusability regardless of the bus/network protocol. In the following, we present five interface sockets: WishBone, AXI, Avalon, OCP, and VCI.

3.2.1 Wishbone

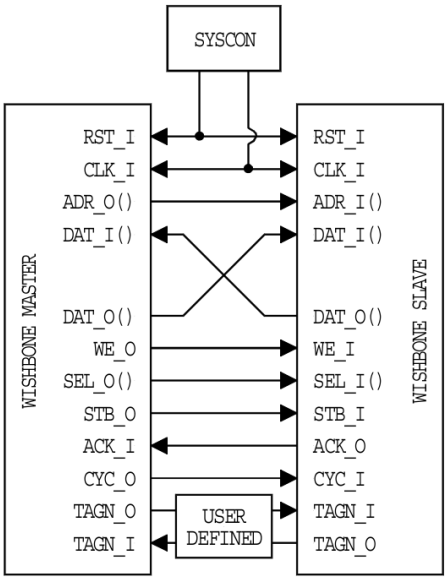


Figure 3.3: Wishbone socket interface [Ope02]

Wishbone was developed by Silicore and is in the public domain. The Wishbone specification is maintained by OpenCores.org [6], which is a web site hosting open source IP cores compliant with Wishbone. According to its specification [Ope02], the purpose of Wishbone is foster design reuse by creating a common, logical interface between IP cores.

As represented in figure 3.3, Wishbone specifies a simple master-slave interface for high-speed data transfers. This interface is configurable in terms of signal width. The address bus can support an addressing space up to 64 bit. The data bus width can be configured from 8 to 64 bits. Wishbone supports four types of data transfers: single or block read/write, Read/Modify/Write (RMW) e.g. to support atomic transactions and constant and incrementing burst. The Wishbone interface is based on a two-phase handshaking protocol and can support retry and error.

Wishbone provides flexibility using *tag* signals, which are user-defined signals attached to an existing group of signals i.e. the address bus, the data bus and the cycle bus. For instance, a parity bit can be added to the data bus as a data tag. However, these tags do not guaranty the interoperability of WishBone-compliant IP cores. Such extensions must thus be well-documented to minimize integration problems.

The Wishbone interface supports streaming and incremental burst transfers to transfer a large amount of data. Streaming bursts allow successive transfers at the same address like for instance to read/write a FIFO. Incremental bursts allow successive transfers at increasing addresses like for instance to read/write

[6] http://www.opencores.org/

a RAM. In this last case, the transfer address is incremented with a linear or wrapped increment. In linear bursts, the address is incremented by one according to the data width e.g. 4 for 32-bit transfers. In wrapped burst, the address is also incremented by one, but the address is modulo the size of wrap, which is 4, 8 or 16 in Wishbone.

Wishbone supports various topologies such as point-to-point, shared bus, data flow or crossbar switch. However, no arbitration schemes are specified and arbitration policies must be defined by the users. To improve design reuse, the Wishbone specification requires that Wishbone-compliant IP cores are described in a template document similar to a datasheet. Examples of Wishbone compliant IP cores can be freely downloaded from the OpenCores web site along with a Wishbone interconnection fabric generator and a Wishbone-to-AHB bridge.

As more advanced bus features such as access control are not defined by the WishBone specification, they have to be developed for more complex SoCs.

3.2.2 Advanced eXchange Interface (AXI)

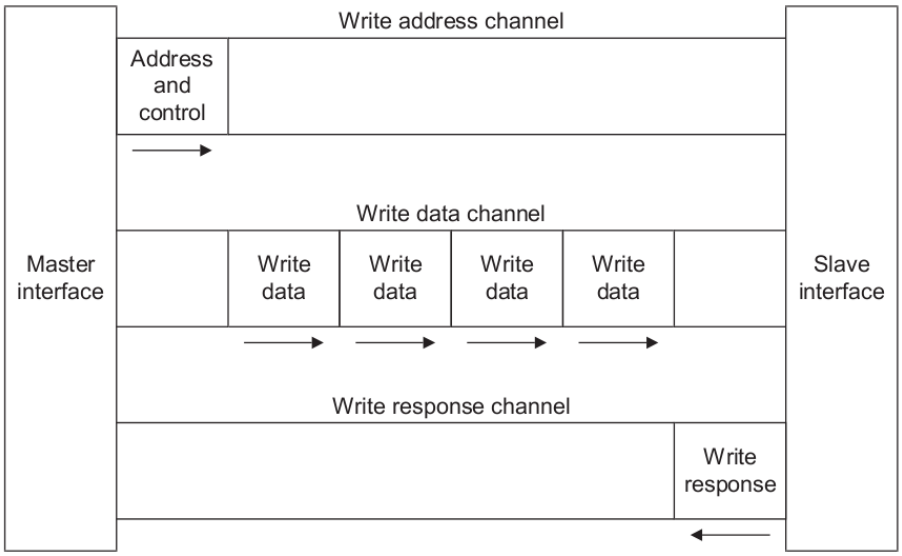


Figure 3.4: AXI write interface

The *Advanced eXchange Interface* (**AXI**) supports low-latency and high-speed data transfers. AXI is part of AMBA 3.0 and is backward compatible with the AHB and APB interfaces from AMBA 2.0. AXI specifies five communication channels: two read channels and three write channels shown in 3.4. The read channels consist of a read address channel for address and control information and a read data channel to carry the read data. The write channels is composed of a write address channel for address and control information, a write data channel for the written data and a write response channel to retrieve write responses.

These five channels are unidirectional and independent. In each channel, a two-phase handshaking protocol is used to acknowledge transfers of control and data between the source and the destination. The source asserts the `VALID` signal to indicate that the data are valid or that the control information are available, while the destination activates the `READY` to indicate that it can accept the data. The address and data bus are configurable and their width may be between 8 and 1024-bit wide. The AXI protocol is packet-oriented. Each AXI transaction is a burst from one to 16 data. The size of a transfer is from 8 to 1024 bits. Burst may be at constant, incrementing or wrapped addresses. A `LAST` signal indicates

the end of a transaction. The response to a request may be out-of-order. Atomic operations guarantee data integrity through exclusive locked accesses. Non-aligned transfers enable burst to start at unaligned addresses.

AXI defines various optional advanced features. A protection unit supports three types of access: normal or privileged, secured or not, instruction or data. A system cache proposes optional bit attributes: the *bufferable bit* allows to delay a transaction for an arbitrary number of cycles, the *cacheable bit* allows multiple write or read fetches to be merged, a read (resp. write) *allocation bit* enables to allocate a read (resp. write) transfer in case of cache miss. AXI also supports a *low-power* interface which allows peripheral devices to request the activation or deactivation of their clock.

3.2.3 Altera Avalon

The Altera Avalon protocol contains two open but proprietary interface protocols: the Avalon Memory-Map (MM) Interface and the Avalon Streaming (ST) Interface.

Avalon Memory-Map (MM) Interface

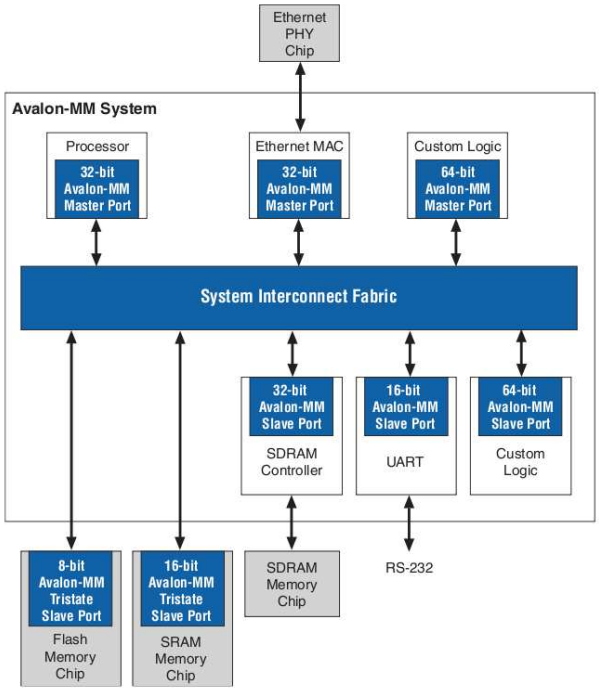


Figure 3.5: Example of System-On-Chip based on Avalon Memory-Map interface [Alt07a]

The Avalon Memory-Mapped [Alt07a] interface specification defines address-based read/write interfaces typically used by master and slave IP cores such as microprocessors, SRAM memory, UART, timer, etc. As presented in figure 3.5, the Avalon MM interface is a lightweight and high-performance socket interface, which resides between application or platform IP cores and an interconnect switch fabric. This fabric supports the interconnection of any type of master and slave IP cores e.g. regardless of their data width. The number and type of interface signals used by an IP core are limited to those required to support its data transfers e.g. without burst signals. The Avalon-MM interface has dedicated control, address, data signals. The data bus may have an arbitrary data width up to 1024-bit and is not

mandatorily an even power of two. The Avalon-MM interface is an open standard and does not require a license to be used.

Avalon Streaming (ST) Interface

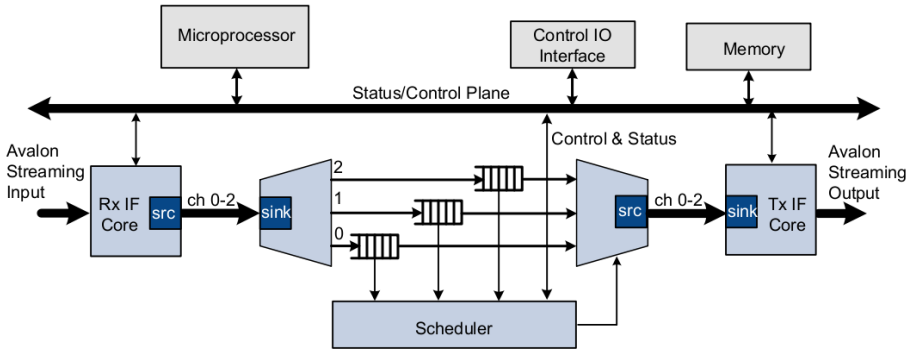


Figure 3.6: Example of System-On-Chip based on Avalon Streaming interface

The Avalon Streaming Interface allows component designers to build interfaces for unidirectional flow of data, including multiplexed streams, packets, and DSP data. The Avalon-ST interface defines a protocol for data transfers from a source interface to a sink interface. The Avalon-ST interface signals can describe traditional streaming interfaces supporting a single stream of data without knowledge of channels or packet boundaries. The interface also supports burst and packet transfers with packets interleaved across multiple channels.

3.2.4 OCP-IP Open Core Protocol (OCP)

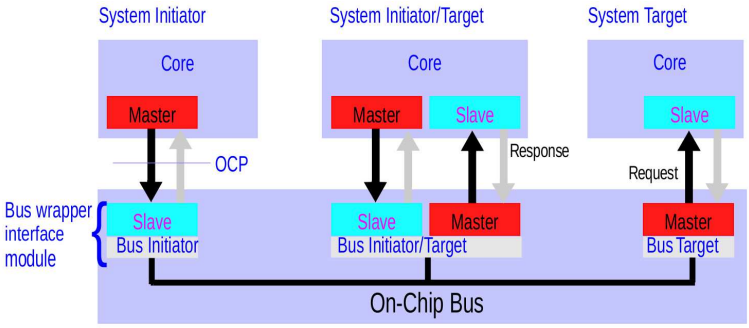


Figure 3.7: Open Core Protocol (OCP) socket interface

The *Open Core Protocol* (**OCP**) [OI06] is a hardware interface specification maintained by the OCP-IP organization created in 2001 by members such as TI, MIPS, Sonics, Nokia and ST. As illustrated in figure 3.7, OCP aims at providing an interconnection standard independent of any bus to enable true reuse. OCP offers configurable and rich protocol semantics, which can be mapped on any protocol. As stated in the specification, "the OCP is the only standard that defines protocols to unify all of the inter-core communication". OCP is highly configurable thanks to profiles which captures IP cores communication needs notably interruption or error signaling. OCP supports communication protocols ranging

from simple acknowledge to multiple out-of-order pipelined concurrent block transfers. The OCP interface acts as a contract between two communicating entities, a master and a slave. Both entities are directly connected via point-to-point signals. Each OCP-compliant IP core is connected to a wrapped interconnection network via a complementary master, slave or both OCP interface. Two IP cores can thus have a pair of master/slave entities for peer-to-peer communications. The OCP promotes a socket based layered architecture [OIb].

This communication architecture reminds the one of distributed systems like CORBA. When two OCP compliant IPs core communicate through an on-chip bus, the initiator or client sends a request in the form of command, control and data signals to a kind of stub called bus wrapper interface. It consists of a slave entity and a bus initiator which converts an OCP request into a bus access according to the underlying bus protocol. The slave entity acts as a proxy and the bus initiator as an adapter. A kind of skeleton receives the request and achieves the dual work. It builds up again a valid OCP request from the on-chip bus transfers and an OCP master entity applies it on the point-to-point link towards the slave side of the target IP core. This socket approach provides communication transparency.

OCP supports advanced features such as threads identifiers to support interleaved requests and replies and out-of-order transaction completion, connexion identifiers for end-to-end identification from one IP core to another one through the underlying interconnection network and extensions for test (JTAG). Address and data width are not necessarily a power of two. This is for instance useful for DSP with 12-bit bus

OCP have well-defined protocol semantics. The layered hierarchical protocol is made of transactions, transfers and phases. A transaction is composed of a set read/write transfers. A transfer may have three phases: request from master to slave, response from slave to master and optional data handshake for write operations. Each phase has a two-way handshaking protocol. The decoupling of these phases allows high performance through pipelining. At each phase is associated of group of signals. All OCP signals are point-to-point, unidirectional and synchronous to the OCP clock. There are three groups of signals: dataflow, sideband and test signals. The data flow signals are composed of basic signals and simple, burst, tag and thread extensions. Only the `MCmd` and `Clk` signals of the basic signals are mandatory, while all the others signals are optional and selected by profiles. The basic signals provide signals for address and data bus, request and data acknowledge. `MCmd` signals encodes eight commands in MCmd: basic write, basic read, exclusive read, linked read, non posted write, conditional write and broadcast.

OCP defines posting semantics for writes. In non posted writes, a response is send after a basic, conditional or broadcast write commands (*writeresp_enable* = 0). `SResp` encodes four responses: no response (*NULL*), data valid/accept (*DVA*), request failed (*FAIL*) or response error (*ERR*). The simple extension add support to signal multiple address spaces, byte enable and in-band information in any of the three OCP phases. This extension can be used for error correction.

The burst extension enables to regroup several independent transfers associated by an address relationship. Burst transfers use more efficiently available bandwidth by sending only the starting address, the address relationship and the length of the burst. OCP supports incrementing, wrapping, user-defined with or without packing of data and streamed burst. Burst address sequences are packing or not. With packing, data width conversion between different OCP interfaces is achieved by aggregating words from narrow-to-wide and splitting from wide-to-narrow. Without packing, data width conversion is achieved by padding words from narrow-to-wide and using byte enable, and stripping from wide-to-narrow. The OCP support precise (exact) and imprecise (approximate) burst length, and packets thanks to single request/multiple data (SRMD) bursts.

The tag extension enables concurrent transfers within a single shared flow of control. Without tags, the order of slave responses shall match the master request one. Tags acts as request identifiers and allow out-of-order responses. The thread extension enables concurrent transfers within independent flows of control. Whereas transfers within a single thread shall be ordered unless tags are used, transfers within

different thread are independent and have no ordering constraints. Threads are identified by *thread ID*. Flow control or split transactions may use thread busy signals. Sideband or out-of-band signals add support for control information: reset, interrupt, error and core-specific flags between master and slave; control and status between an IP core and the rest of the system. Test signals support scan, clock control and JTAG.

OCP enables partial words transfers thanks to configurable byte enables. The OCP defines a word as the natural transfer unit of a block. The OCP word size corresponds the configurable data width of power-of-two or not. Byte enable specify which octets are to be transferred. Transfers can be aligned within precise incrementing burst with *burst_aligned* parameter : the number of transfers is a power-of-two and the starting address is aligned with the burst length. The parameter *force_aligned* force the byte enable patterns to be aligned to a power-of-two size. The endianness of OCP interfaces can be little, big, both or neutral and is specified thanks to the parameter endian. Byte enable allows to be endian neutral.

Connections establish a end-to-end logical link between a system initiator and a system target in a global scope. A contrario, threads have a local scope in the point-to-point logical link between a master and a slave entity. Connections are identified by *connection ID*. The OCP specification proposes a networking analogy: thread ID refers to the data link layer (level 2), whereas connection ID refers to the transport/session layer (level 4/5). A connection ID can be allocated by the system and associated to an initiator for identification by the target (access control) or a QoS.

Others strength of OCP are documentation, configuration, verification and compliance. Interface configuration file written in TCL provides a description of non OCP interface signals: width, direction and type. Core RTL configuration file is required to describe a core and its interfaces: version, memory map, matching between custom signals names and OCP ones. Core synthesis configuration file documents clock, area and timing. The OCP specify native profiles to describe native OCP interfaces for block data flow, register access, AHB like bus and AXI like read/write channels. Layered profiles provide additional features to existing profiles like security.

As an example, we use OCP to encapsulate custom IP core interfaces as detailed in paragraph 8.3.2.0 and for the hardware middleware prototype presented in section 8.3.3.

3.2.5 VSIA Virtual Component Interface (VCI)

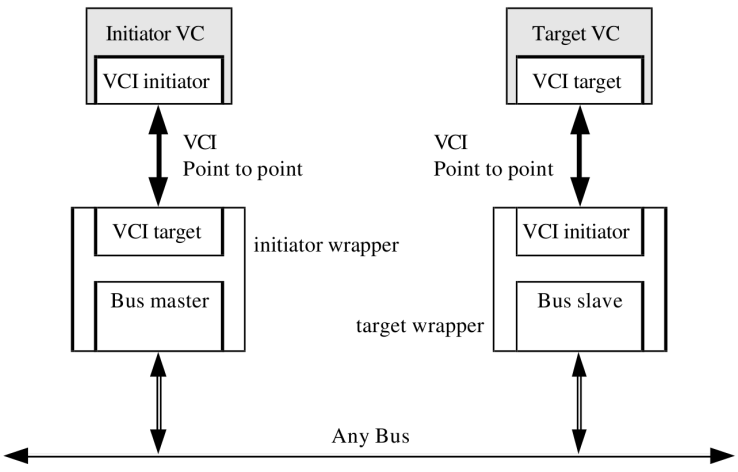


Figure 3.8: VSIA Virtual Component Interface (VCI)

The *Virtual Component Interface* (**VCI**) [Gro01] was standardized by the Virtual Socket Interface[7] Alliance (VSIA). The VSI Alliance was an open international organization founded in 1996 by tool vendors such as Mentor Graphics, Cadence Design Systems and Synopsys and others companies from the SoC industry such as ARM, ST, NXP, Intel and IBM. It aimed at defining standards to improve the productivity, development, integration (see figure 3.8), and reuse of IP cores. The VSI Alliance was dissolved in 2008 and the maintenance and ongoing development of its documents were transferred to other international organizations. For instance, future works related to the *Hard Intellectual IP Tagging Standard* are moved to the SPIRIT consortium[8], while the Virtual Component Interface Standard (On-Chip Bus - OCB 2 2.0) [Gro01] are archived by OCP-IP. The VSI Alliance dedicates all copyright of its documents notably the Virtual Component Interface Standard to the public domain.

The differences between VCI and OCP are described in the FAQs of the OCP-IP web site [9]. VCI is considered as outdated as it has not evolved for many years and has been superseded by OCP. Indeed, VCI can be viewed as a subset of OCP that goes further with not only data flows signals, but also standardizes control and test signals.

As an example, VCI is used in the SoCLib[10] virtual platform for multi-processors system on chips.

3.3 Network-On-Chip (NoC) Architecture

Traditional bus architectures are not scalable and provide poor performance when the number of IP cores to integrate in SoC platforms significantly increases. Network-On-Chips are recognized in academia and industry as the next generation of on-chip communication infrastructure for SoCs [BM02]. Some surveys on network-on-chips have been published like [BM06] [SKH07] where 60 NoCs are briefly compared on criteria such as topology, routing and switching scheme.

Distributed Layered Architecture

As illustrated in figure 3.9, the basic building blocks of a network-on-chip are the following [BM06]:

- Nodes contain the computation, storages and I/O IP cores such as processors, memories, hardware accelerators and RF front-ends.
- *Network Adapter*s (**NA**) provides the Network Interface through which IP cores communicate across the network. They support the separation of concerns between computation in the nodes from communication in the on-chip interconnect. Network Adapters are in charge of end-to-end flow control and encapsulation of core messages or transaction into packets with routing information such as X-Y coordinates for packet-switch networks or connection identifier for circuit-switched networks. A Network Adapter provides two interfaces: a *Core Interface* (CI) at the core side and a *Network Interface* (NI) at the network side. It offers high-level communication services to the node built upon the primitive read/write services implemented by the network. The Core Interface of the NA may implement a standard interconnection socket such as OCP [OI06], AXI [ARM04], VCI [Gro01] or Philips DTL (*Device Transaction Level*). The Network Interface may be specific for each NoC thanks to the decoupling offered by the Network Adapter.
- Switches or routers route the packets from the nodes to other routers and vice versa according to a static or dynamic adaptive routing policy.

[7] http://vsi.org
[8] http://www.spiritconsortium.org
[9] http://www.ocpip.org/about/faqs/vsia
[10] http://www.soclib.fr

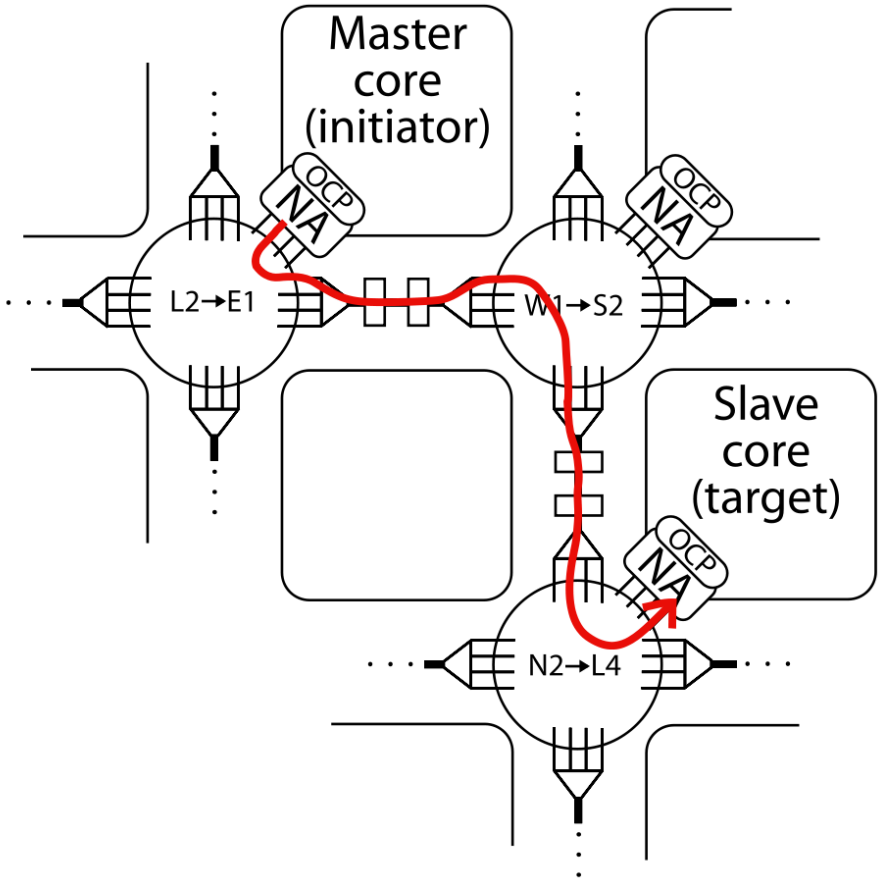


Figure 3.9: Network-On-Chip main components [Bje05]

- Links corresponds to the point-to-point buses that interconnect the routers. They provide physical or logical channels to transfer atomic unit of data called flits (flow control units) or phits (physical units) into which packets are fragmented.

Topology

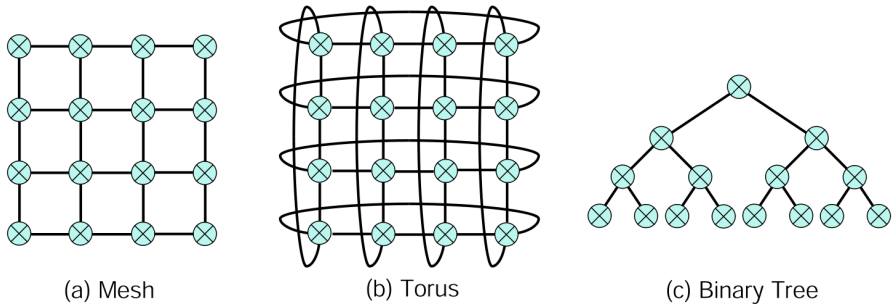


Figure 3.10: Example of network topology [Bje05]

Like in the networking domain for desktop computers, a network *topology* for NoC designates the arrangement of nodes, routers and links. As illustrated in figure 3.10, the typical types of network topology include point-to-point, bus and hierarchical bus, crossbar, star, ring, mesh, tree and fat-tree, torus and hybrid topology. Each topology provides different trade-off between scalability, area consumption, throughput and latency. Most NoCs use a 2D mesh and torus notably to ease hardware placement and routing on the regular resources of FPGAs and ASICs. NoCs may be customized to meet the specific requirements of a given application domain e.g. with additional links.

Switching policy

Switching refers to the basic multiplexing of data among switch ports, whereas *routing* performs smarter processing such as find a path. In **circuit switching**, a path or *circuit* is allocated from a source to a destination node through a set of routers and links whose resources (e.g. time slot, buffer) are reserved before transferring data. The path is set-up then torn down for each data transfer. The routing information only consists of a circuit or connection identifier. Circuit switched networks provide connection-oriented communication Circuit switching necessitates less buffering since the data can be directly forwarded. This mechanism may be used to guarantee the QoS of continuous data streams at the expense of the latency required to establish the circuit.

In **packet switching**, a packet is routed at each router depending on the routing information included in the packet such as the X-Y coordinate in a 2D mesh. Packet switching requires more buffering than circuit switching and may introduce additional transfer jitter which are harmful to provide applications with performance guarantees. Packet switched network may support both connection-less and connection-oriented communication. This switching policy is the most used, for instance in 80% of the NoCs compared in [SKH07]. Some NoCs can support both switching techniques like Octagon and Slim-Spider (see [SKH07]). Three usual strategies may be employed to transfer packets through a router: *store-and-forward*, *cut-through* and *wormhole*. In store-and-forward switching, the packet is entirely stored in the router before being forwarded to another router according to the routing information in the packet header, while the packet is forwarded in cut-through switching as soon as the routing information are available. Both schemes require the router to have a buffer size of at least one packet. In wormhole

switching, a packet is fragmented into several flits, which are routed as soon as possible like the cut-through switching. The packet may be spread among several routers along the data path like a worm. In this case, the router needs less buffering resources and may require at least one flit. Wormhole switching is widely used in NoCs to reduce silicon area while providing the same switching functionality.

Routing

Routing consists in forwarding a packet to the right router or node according to a routing strategy. Routing may be *deterministic* or *adaptive*. In deterministic routing, all packets are routed along the same path from a source to a destination node as opposed to adaptive routing. Deterministic routing schemes include source routing and X-Y routing. In source routing, the source node provides the route e.g. a list of router included in the packet header up to the destination node. Each router may remove itself from the list before forwarding the packet to the next router and gradually reduce the size of the packet header. In X-Y routing, packets are forwarded in one direction along the columns (X) or the rows (Y) then along the other direction up to the destination node. In adaptive routing, the path is re-determined at each hop from one router to another according to some criteria such as link congestion. Adaptive routing may thus be used to dynamically find a better path to reduce congestion, bypass faulty routers and support load balancing [BM06]. However, additional hardware resources are required to avoid livelocks/deadlocks and re-order out-of-order packets.

NoCs may also support routing schemes for group communications between one source node and several destination nodes such as one-to-all i.e. broadcast communications, one-to-many i.e. multicast communications [BM06].

Flow Control

NoCs may control the flow of data using *virtual channels* (**VC**). A virtual channel is a logical channel within a router or network adapter. It has its own buffering resources, but shares a common physical channel i.e. a link with other virtual channels. For instance, a router may have between 2 and 16 virtual channels per physical channel [BM06]. Obviously, virtual channels consume more area and may increase power consumption and transfer latency. However, they also provide some additional benefits such as decreasing latency, better wire efficiency, better performance, avoiding deadlock and providing different levels of QoS.

Quality of Services

Some high-performance and real-time applications may require stringent guarantees on the Quality of Services (QoS) provided by NoCs. Each IP core in a SoC may have different communication needs and the level of services proposed by NoCs should be independent of the number and nature of IPs to be interconnected. The QoS guarantees may concern properties such as timing (throughput, latency and jitter), integrity (error rate, packet loss) and packet delivery (in-order or out-of-order) [SKH07].Two classes of QoS are commonly distinguished: *Best-Effort* (**BE**) services and *Guaranteed Services* (**GS**). Best Effort services provide no guarantees on packet routing and delivery. Guaranteed Services mostly provide predictable throughput, latency and jitter using Time-Multiplexing Division (TDM) like in Nostrum [MNTJ04], AEthereal [RGR$^+$03] and HIBI [SKH05] or Spatial-Multiplexing Division (SDM) like in [MVS$^+$05]. To provide hard-real time guarantees and avoid congestion, each data stream must be assigned to a dedicated connection. A connection is established by creating a virtual circuit with logically independent resources such as virtual channels, temporal slots (TDM) or individual wires in links (SDM). Each one of these resources may be associated with a priority to allow high-priority packets to be routed before lower-priority packets.

Some router implementations implement both Best-Effort and Guaranteed Services like AEthereal [RGR+03] and Mango [Bje05].

NoC services for platform-based design

To support the portability of applications from one NoC platform to another and reuse a NoC platform in different applications, the interfaces of the services required by the application layer and provided by the platform layer should be standardized. Obviously, computer network protocols [Tan96] cannot be directly applied to NoCs and must be reduced to limit the necessary overhead in terms of memory/area footprint, latency and throughput.

Some works have been carried out to propose a layered communication model for NoCs [BM06] [MNT+04] [SSM+01].

Millberg et al. propose a communication protocol stack for the Nostrum NoC [MNT+04]. The Nostrum protocol stack provides a layered communication architecture based on four layers: physical layer, data link layer, network layer and transport layer.

Each layer process different granularity of data. The Physical Layer deals with the physical transfer of words from the output port of a switch to the input port of another. The Data Link Layer is in charge of the reliability of frames transmitted on top of the physical layer using synchronization, error and flow control. The Network Layer copes with the delivery and routing of packets from the source network adapter to the destination network adapter and the mapping between a node address and an application process identifier. This layer may provide differentiated services such as best effort delivery and guaranteed bandwidth using virtual circuits. The Transport Layer establishes an end-to-end communication between nodes with flow-control e.g. to avoid congestion.

The NoC from Arteris is based on three layers: transaction, transport, and physical layers [Mar05]. Each layer is independent of the other layers. The transaction layer provides IP cores with communication primitives. A *Network Interface Unit* **(NIU**) transforms IP core interface protocols such as AHB, AXI, VCI and OCP to the interface protocol of the transaction layer. The transport layer routes prioritized packets among NIUs according to a given quality of service. The physical layer transfers the raw packets on the physical wires.

Examples of NoCs

Numerous examples of NoCs have been proposed in the literature including AEthereal [RDP+05] from NXP, MANGO [Bje05] from DTU and Nostrum from KTH, SPIN from LIP6, μSpider from LESTER.

Nostrum is a packet-switch NoC with a 2D-mesh topology to simplify layout and switches and provide high-bandwidth and scalability. Nostrum contains resources and switches which are interconnected by channels. Each resource is a computation or storage unit. A switch routes packet and may include queues to store messages and avoid congestion. A channel comprises two one-directional point-to-point buses between a pair of switches or between a resource and a switch. Nostrum supports both best effort (BE) and guaranteed latency (GL) Quality-of-Service [Jan03]. In BE, packets are independently routed accorded to a long header including the whole address information. They may travel over different paths depending on the network congestion and thus can not provide latency guarantees. In GL communications, an independent virtual circuit is first opened by allocating dedicated time slots in every switch along the path from the source to the destination node and provide a bounded latency. GL packets contains short headers without address information which are configured in switches. The Nostrum architecture is organized in four communication layers: a physical, link, network and session layer [MNT+04]. Communication primitives have been defined to support both shared memory and message passing. These primitives allows the definition of QoS properties such as direction, burstiness, latency,

bandwidth, reliability for message passing channels. The NoC Assembler Language (NoC-AL) [Jan03] provides an interface contract between applications and NoC platforms that specifies application functionality, communication semantics and performance, and the mapping of tasks to resources. NoC-AL describes the architecture of a NoC in terms of the topology, number, type and location of its resources and the mapping of application processes on these resources. These resources communicate through explicit channels via predefined communication primitives similar to the Berkeley socket API.

3.4 Conclusion

In this chapter, we presented the architecture and interface of on-chip buses and network-on-chips. A shared and hierarchical bus may have a centralized architecture like ARM AHB and IBM CoreConnect, or a *distributed* architecture like HIBI. The centralized bus architecture is composed of a bus arbiter and an address decoder. A bus arbiter selects a unique master among several that compete for the bus, while an address decoder selects a unique slave among all slaves according to the address provided by the granted master. The distributed bus architecture is based on a bus wrapper for each IP core. Wrappers are configured with TDMA slots, then each wrapper counts the clock cycles until it can transfer data.

A socket is a point-to-point hardware interface used to connect an IP core to an on-chip communication bus. A socket allows the separation of concerns between computation and communication. The objective of a socket interface should to be independent from any interconnection network to improve IP cores portability, interoperability and reusability regardless of the underlying bus/network protocol. To provide communication transparency to IP cores, a socket based layered architecture converts socket transactions into bus transactions.

Unfortunately, there is not one but several open socket interfaces such as AXI, VCI and OCP. This multitude of socket interfaces do not favour the adoption of a socket approach. Socket standards define low level data transfer interfaces and protocols at Register Transfer Level. As modeling and verification at this level are very slow, new modeling methodologies have been developed like Transaction Level Modeling (TLM) to raise the abstraction level and the simulation speed. We will present TLM in chapter 2 section 2.7.

Traditional shared bus architectures are not scalable and provide poor performance when the number of IP cores to integrate in SoC platforms significantly increases. Network-On-Chips (NoCs) are recognized in academia and industry as the next generation of on-chip communication infrastructure for SoCs.

NoCs have a distributed layered architecture composed of nodes, network adapters, switches/routers and links. Nodes may be computation, storage or I/O nodes. Network adapters adapt IP cores interfaces and network interfaces, and en/de-capsulate transactions into/from packets. Switches route packets in the NoC. Links are point-to-point buses between switches that carry packets.

The two main switching policies are circuit switching and packet switching. In circuit switching, a path or circuit is allocated from a source to a destination node through a set of routers and links whose resources (e.g. time slot, buffer) are reserved before transferring data. In packet switching, a packet is routed at each router depending on the routing information included in the packet.

A network topology designates the arrangement of nodes, routers and links. The typical types of network topology include point-to-point, bus and hierarchical bus, crossbar, star, ring, mesh, tree and fat-tree, torus and hybrid.

NoCs may control the flow of data using virtual channels. A virtual channel is a logical channel within a router or network adapter. It has its own buffering resources, but shares a common physical channel i.e. a link with other virtual channels.

The choice between a bus and a NoC depends on the number of IP cores to be interconnected and

the required performances in terms of latency and throughput. As a rule of thumb, a bus is preferred below 15/20 IPs in a SoC to minimize area and latency, otherwise a NoC is better for its scalability and throughput [Bje05].

NoCs may typically provide two classes of Qualities of Service (QoS): Best Effort (BE) services provide no guarantees on packet routing and delivery, whereas Guaranteed Services (GS) mostly provide predictable throughput, latency and jitter using Time-Multiplexing Division (TDM) or Spatial-Multiplexing Division (SDM).

An important point is that custom or legacy military applications such as NATO standards and commercial applications such as 3G/4G do not have the same performance requirements, and therefore do not need the same hardware/software architectures. In traditional hardware/software architectures of modems, digital signal processing IP cores in the data path do not need to be shared and can be typically interconnected via point-to-point links to maximixe throughput and reduce latency. For the control path, off-chip or on-chip shared buses commonly support communication between DSPs and FPGAs. As the number of IPs in military waveform applications are typically inferior to 20, the main on-chip communication architectures usable in a SCA context are classical on-chip buses. However, future high data rate modems may require NoC architecture.

We will see in our contribution presented in chapter 7 how we used some of these concepts for our hardware middleware prototype. In the next chapter, we will present the concepts of the object-oriented approach and their applications for hardware/software design.

Chapter 4

Object-Oriented Design (OOD)

Contents

In this chapter, the terminology and main concepts of the concurrent and distributed object-oriented model are introduced notably the principles of encapsulation, inheritance and polymorphism. The subtle differences of terminology, semantics and implementation of these concepts are also presented when relevant in the UML object-oriented modeling language and in object-oriented programming languages such as SmallTalk, C++, Java and Ada.

A state of the art is then presented on the applications of these concepts in hardware/software co-design of embedded systems. Finally, the intrinsic strengths and weaknesses of the object paradigm are identified to justify the emergence of the more evolved *component-oriented model* presented in the next section.

This chapter is organized as follows. Section 4.1 presents the fundamental concepts underlying the object model such as the equivalence between method invocation and message passing, encapsulation,

inheritance and polymorphism. Section 4.2 describes Object-Oriented Hardware Design such as Object-Oriented Hardware Description Languages (OO-HDL), Object-Oriented Languages for Hardware Design like SystemC, Object-Oriented Hardware Architecture and Model Driven Engineering (MDE) for Hardware Design. Section 3.3 presents network-on-chip characteristics such as topology, routing and quality of services, along with some examples of NoCs. Section 4.3 proposes a synthesis on the works presented in this chapter. Finally, section 4.4 concludes this chapter.

4.1 Fundamental Concepts

Surprisingly, no real consensus seems to exist about the concepts characterizing the object model and their definition [Arm06]. Armstrong reviewed the OO literature from 1966 to 2005 to identify the fundamental concepts underlying the object model. She identified 39 concepts and classified them by their number of occurrences in the literature. The eight first retained concepts mentioned by more than 50% of the reviewed papers were inheritance, object, class, encapsulation, method, message passing, polymorphism and abstraction. Instead of selecting only eight of these concepts, we will attempt to put together all these concepts and stress their relationships to build the whole concurrent and distributed object paradigm.

4.1.1 Origin

The object-oriented approach evolved from previous works achieved during the sixties in various research domains such as data structure abstraction, system simulation, operation systems, database models, graphical user interface (GUI) and artificial intelligence [Cap03] [Kay93]. The concept of "object" emerged almost independently and simultaneously in these research domains in the early 1970s to designate a set of common concepts [BME+07] [Kay93]. This concept was introduced in the Simula 67 programming language by Nygaard and Dahl. Its designers observed that Simula I "processes often shared a number of common properties, both in data attributes and actions, when writing simulation programs" [ND81]. The term "object" was considered more neutral compared to the word "process" of Simula I. Interestingly, the first application of Simula I was the simulation of logic circuits. The term "object-oriented" was chosen by Alan Kay to refer to this new design paradigm during the development of the Smalltalk programming language from 1972. "Smalltalk's design - and existence - is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages" [Kay93].

4.1.2 Identifiable Objects as Instances of Classes

The primary abstraction of the object-oriented design is *classes of objects* to represent a system. An object is a self-contained and identifiable entity, which encloses its own state and behavior and communicates only through message passing [Arm06] [BME+07]. In the client-server model of interaction, a client object is any object that requires the services provided by another object called the server object. An object typically represents a category or *class* of real or imaginary things. In object terminology, an object is said to be an *instance* of a class. A class describes the common characteristics shared by a set of objects in term of structure and behavior [BME+07]. In programming languages, a class defines the *type* of an object. The state of an object corresponds to the set of values of all static and dynamic properties or *attributes* of an object [BME+07]. An object behavior is provided by the implementation of class functionalities.

4.1.3 Method Invocation and Message Passing

The behavior of an object takes the form of a set of methods in the object class. A method describes the actions that an object must perform depending on the type of invocation message received by the object. The process of selecting a method based on a request message is called *method dispatching*. The term *method* originates from Smalltalk, whereas the terms *operation* and *member function* are respectively employed in Ada and C++ [BME+07]. In such programming languages, the terms "method" and "operation" can be considered as synonyms. However, OMG standards such as UML and CORBA make a clear distinction between method and operation. In UML, a method is an operation implementation that specifies the algorithm associated with an operation, while an operation declares a service that can be provided by class instances [OMG07d]. An operation is invoked by a call. A method may have additional constraints compared to its operation. For instance, a sequential operation may be implemented as a concurrent method [RIJ04]. Generally speaking, a class provides services to clients through its methods. Methods typically allows one to get and set object attributes and process the data passed as method parameters. A method may raise an *exception* to indicate that an user-defined or system error has occurred. The concepts of method and message are often used as synonyms in the OO literature [Arm06].

The concept of message undeniably makes sense in distributed systems, where it provides the same abstraction to unify both local and remote communications between distributed objects. Indeed, the local and implicit transfer of control and data of a method invocation can be formally translated into the remote and explicit transfer of a message and vice versa. In distributed systems, the systematic translation between method invocation and message passing is the cornerstone of *Remote Method Invocation* (**RMI**) used in object-oriented middlewares such as Sun Java RMI, Microsoft DCOM or OMG CORBA. RMI results from the straightforward transposition of *Remote Procedure Call* (**RPC**) [BN84] in the object-oriented model. Typically, **proxies** or *surrogates* [GHJV95] are automatically generated to make local and remote communications indistinguishable from the viewpoint of client and server objects thanks to *request-reply* message protocol. A client-side proxy called **stub** encodes the input and input-output parameters of a remote method invocation into a request message payload and conversely decodes a reply message payload into the output and input-output parameters and the return value. A server-side proxy called **skeleton** decodes a request message payload into the input and input-output parameters of a local method invocation to the object implementation and conversely encodes the output and input-output parameters and the return value into a reply message payload.

In the UML object model, objects communicate through two types of messages. A message is either an *operation call* or a *signal*. A signal may represent, for instance, a hardware interruption e.g. mouse button click. A message has argument values, which have to be compliant with the direction and types of the corresponding operation parameters or signal attributes. The parameter direction may be *in*, *inout*, *out*, or *return* for input, output or return parameters.

An operation call may be asynchronous or synchronous i.e. blocking as the caller waits until it receives a response with any return values.

A signal is sent from one sender object to one or several receiver object i.e. broadcast. The sending of a signal is oneway and asynchronous i.e. non-blocking as the sender continues immediately its execution after signal sending. The receipt of a signal may trigger a state machine transition.

Figure 4.1 represents the graphical notation used for all these types of messages within an UML sequence diagram. Real-time constraints may be specified by *timing constraints* and *duration constraints*.

In hardware, these interaction styles will depend on the hardware invocation protocol used by hardware objects to communicate. We will analyze how they have been implemented in hardware in section 4.2 p. 63.

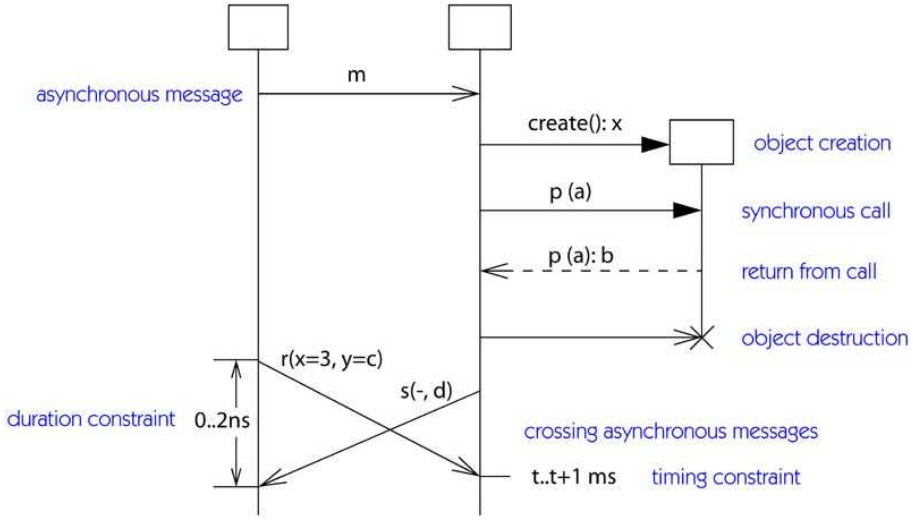


Figure 4.1: Message notation in UML sequence diagrams [RIJ04]

4.1.4 Encapsulation: Interface and Implementation(s)

An *interface* is the boundary between the inside view of an object, which should be hidden or private, and its outside view, which should be visible or public. A class interface captures the outside view, whereas the class implementation performing the required behavior constitutes the inside view [BME+07]. In Smalltalk, the interface of an object is the set of messages to which an object can respond [GR83]. This interface is the only way to interact with an object and may be viewed as a "service access point (SAP)". The separation of concerns between class interface and class implementation is known as the *encapsulation* or *information hiding* principle. The concept of information hiding was used in structured design and programming, whereas the object paradigm refers to the term encapsulation.

The declaration of a class corresponds to the specification of the access right, the type and the name of class attributes and to the specification of a set of operation *signature*s or *prototype*s. An operation signature contains the name of the operation, the type and name of its input, output, input-output parameters, the type of the return value and the name of errors or exceptions which may be raised. The set of operations signatures specifies an **interface** between the outside view of a class provided by its declaration and the inside view incarnated by the object implementation. This programming interface serves as a *functional contract* between the user and the developer of an object. As an object is seen as a black box from the user's point of view, the object implementation can be transparently substituted by the developer as long as it respects the interface contract. Thus the user code is independent from object implementation and can be reused.

In the object-oriented approach, the principle of **encapsulation** means at least two things. First, the state of an object should only be modified from the outside by method invocations. Dedicated object methods are typically used to modify the object state. They are named getter/setter or accessor/mutator methods to respectively read and write private and protected attributes. Second, the implementation of an object does not need to be visible to the user, who can only interact with the object through its public and published interface. The implementation of an object can be change without modifying its clients as long as the object interface remains the same.

In mainstream object-oriented languages such as Java or C++, an interface is typically a *syntactical contract* used by compilers to check the conformance between the declaration of operations and their use through type checking. The deployment of critical applications onto embedded platforms may require

more guarantees in terms of behavior, performance or origin e.g. trusted computing.

Beugnard et al. define four levels of contract [BJPW99] of increasing negotiability: basic or syntactical, behavioral, synchronization and quantitative.

Syntactical contracts refer to static or dynamic type checking in any language. This include modeling languages like UML, interface definition languages (IDLs), usual software programming languages e.g. Java, C/C++/C#, SystemC and hardware description languages e.g. VHDL and Verilog.

Behavioral contracts rely on boolean assertions called pre- and post-conditions [Hoa69] to constraint operation behavior and enforce class invariant properties. An interface may be defined by its behavior protocol [PV02] that describes the ordered set of operation invocations that a client is allowed to perform. Such a protocol can be represented by a so-called protocol state machine [OMG07d] in UML. The term *Design by Contract*™ (DbC) [Mey92] was coined by Meyer to refer to this design methodology also know as *Programming by Contract* . This approach has been supported either natively or via libraries in various programming languages such as Eiffel designed by Meyer, Java and C/C++ and in UML through behavioral and protocol state machines and the Object Constraint Language (OCL) [OMG06d]. In hardware, Programming by Contract corresponds to Assertion-based Verification with Hardware Verification Languages (HVL) such as *Property Specification Language* (PSL) [Acc04a], Cadence Specman ***e***, SystemC [Ini08] and SystemVerilog [Acc04b].

Synchronization contracts specify synchronization constraints between concurrent method calls. Such constraints may take the form of synchronization policies such as mutual exclusion or single writer/multiple readers as defined by McHale [McH94]. For instance, the mutual execution of method executions is supported in Java with the *synchronized* keyword. In hardware, synchronization between hardware modules relies on explicit arbiters in classical HDLs or rules in BlueSpec that produce implicit arbiters.

Quantitative contracts allow one to define quality of service constraints also known as non-functional properties to ensure execution guarantees such as performance (latency, throughput), security or availability [Kra08]. Examples are the Real-Time (RT) CORBA policies. In hardware, quality of services typically refers to latency and throughput of data transfers between hardware master and slave modules in on-chip buses or NoCs.

This work primary aims at defining an interface mapping from IDL to VHDL therefore at a syntactical level. However, we will attempt in our proposition to include other aspects related to behavioral and concurrency semantics and quality of services within on-chip interconnection networks.

4.1.5 Inheritance and White-Box Reuse

Inheritance permits to gather common functionalities between classes and to reuse interface definitions and implementations. This mechanism avoids duplication and redundancy. It also provides an elegant way to extend functionality and to more easily manage complexity by using a hierarchy of classes from a general parent to specialized derived classes. The definition of interfaces is therefore modular and reusable. The **inheritance** principle consists in deriving the declaration and possibly implementation of a *derived* or *child* class from a *base* or *parent* class. The child class includes by default the data and behavior of the parent class. It may also declare additional attributes and methods. The behavior of the derived class may be either based on or replace (*override*) the behavior of its base class. Inheritance supports the substitution principle of Liskov: if an interface I2 is derived from an interface I1, then I2 is a subtype of I1 and a reference to an object of type I1 can be substituted by any object of type I2 [OMG08b]. Reuse based on inheritance is called white-box reuse [GHJV95]. Indeed, the subclass may access the private implementation of its parent, actually breaking the encapsulation principle.

Multiple inheritance allows a child class to inherit from more than one parent class. For instance, Java only supports single inheritance, whereas C++ supports multiple inheritance.

An *abstract* method does not provide an implementation. A class is said to be abstract, when at least one of its methods is abstract. An abstract class typically cannot be instantiated and delegates the implementation of methods to concrete classes. If a derived class inherits from an abstract class, the derived class has to implement the abstract methods. An abstract class specifies thus an interface offering a *framework*, which is customized by derived classes [GHJV95]. In C++, an abstract interface is declared using pure virtual methods with the *virtual* keyword, whereas Java includes the *abstract* keyword.

Inheritance is often associated with white-box reuse, because the parent class implementation is often visible to the subclasses.

An alternative approach to inheritance is an implementation mechanism called delegation [BME+07], in which an object forwards or delegates a request to another object. For instance, this feature is used in languages which do not natively support inheritance.

4.1.6 Polymorphism

Polymorphism refers "the ability to substitute objects of matching interface for one another at run-time" [GHJV95]. Methods may be invoked on objects with a common parent class in the same way whatever the object specific subclass. The type of an object is dynamically determined at runtime. Polymorphism implies late binding i.e. the binding of a method to a name is not known until execution. *Dynamic dispatching* refers to the process of determining at run-time which method to invoke.

Polymorphism may be mainly implemented either thanks to memory pointer like in C++ or thanks to tagged references like in Ada. The first approach uses dynamic memory allocation to maintain a table of pointers to methods implementations called *Virtual Method Table* (**VMT**) in C++. In the second one, the object data structure have an implicit field, which contains a tag corresponding to the object type.

As an example, an abstract ALU [11] class may define an abstract *compute* method inherited by concrete adder and multiplier classes. Both derived classes implement appropriately this operation as shown in figure 4.2 and listing 4.1. Thanks to polymorphism, a pointer designating an abstract ALU class may transparently indicate any objects, whose class derives from this abstract class. When the *compute* method is invoked, the type of the pointed object is dynamically determined and the corresponding implementation is executed. Polymorphism supports the addition of new implementations without changing the invocation code.

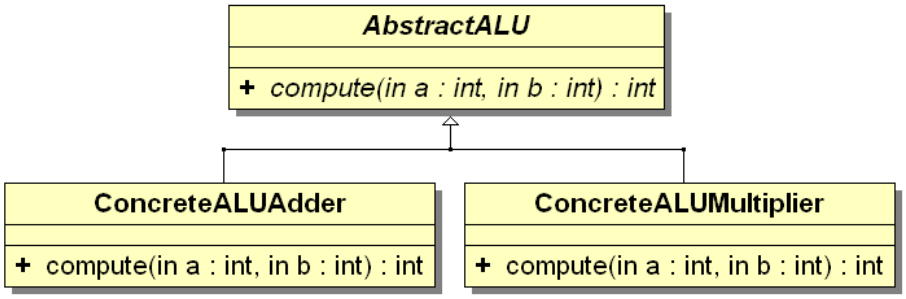


Figure 4.2: UML class diagram of the abstract ALU and concrete adder and multiplier classes

```
// Abstract Base Class Declaration
class AbstractALU {
  public: virtual int compute(int a, int b) = 0; };
// Two Concrete Derived Classes Declaration
```

[11] Arithmetic Logic Unit

```
5     class ConcreteAdder: public AbstractALU {
        public: int compute(int a, int b){return a+b;} };
      class ConcreteMultiplier: public AbstractALU {
        public: int compute(int a, int b){return a*b;} };
      // The method to invoke is determined at runtime
10    // depending on the type of the pointed object
      AbstractALU* theALU;                    // pointer declaration
      theALU = new ConcreteAdder();           // creation of a new class
          instance
      cout<<"a+b="<<theALU->compute(1, 2)<<endl;
      theALU = new ConcreteMultiplier();
15    cout<<"a*b="<<theALU->compute(1, 2)<<endl; // reuse of static
          invocation code
```

Listing 4.1: Illustration of the polymorphism principle in C++

As illustrated in listing 4.1, polymorphism enables transparent method invocations, extensibility and reuse. Because derived classes implement the same interface, the code to invoke methods on objects remains the same. This allows updating or extending object functionality without breaking the code and recompiling thanks to dynamic loadable libraries. Without polymorphism, the developer would have to write switch-case statements to explicitly determine the type of an object and invoke the appropriate methods.

4.1.7 Relationship

As objects collaborate with each other to fulfill the functionalities of a system, the concept of relationship among classes is an integral part of the object-oriented approach. A relationship is a semantic link, which associates classes of objects [RIJ04].

According to [Cap03] and [BME+07], the object model was notably inspired by the Entity-Relationship (ER) model of Chen [Che76], who proposed to represent the real world as consisting of entities, their attributes and the relationships among them.

Rumbaugh proposed to combine both models into the *object-relation* model [Rum87], where relations exist between the objects themselves and not between their attributes. Rumbaugh's work helped to create the object-oriented modeling language UML [RIJ04] and refine the definition of the object model itself.

Relationships may be naturally expressed during object-oriented design and are notably incarnated by annotated lines in UML. Traditional OO languages provide syntactic or semantic constructs to express relations such as instantiation ("instance of") and generalization ("kind of") with the *new* and *extends* keywords in Java. However, no construct exists to make explicit association relationships. As a consequence, the traceability of these relations between design and implementation may be lost during code development [Rum87].

Besides, in the recent Armstrong's survey [Arm06], the concept of relationship appears at the 13th place of the most cited OO concepts with only 14% of the OO literature considering this notion. Consequently, the relationship concept is lacking in the taxonomies studied and the one proposed by Armstrong.

Hence, the importance of relationships is often underestimated in the OO approach [Rum87]. Some works proposed to treat relationships as first-class citizens with OO languages such as the Data Structure Manager (DSM) [Rum87] or Relationship Java (RelJ) [BW05]. We argue that this notion is incarnated by the concept of connector [BP04] in the component-oriented design, which goes further by encapsulating different interaction patterns and communication styles.

Three main kinds of relationships may be distinguished among objects: association, generalization/specialization and containment including aggregation and composition [BME+07].

An association denotes an "exchange messages with" relationship. In OO languages this relation takes the form of a method invocation between the associated objects. Association relations are typically simulated in programming languages by relative references or pointers from one object to another as instance variables [Rum87]. As we will see in chapter 6, this militates in favor of a component-oriented approach in which references are automatically managed by component frameworks, and the association and containment relationships are explicit thanks to the connector and component concepts.

A generalization/specialization designates an "is a" or "kind of" relationship. This relationship is represented in OO languages by inheritance.

A containment denotes "has a", "part of" or "whole/part" relationship. A distinction may be made as in UML between the weak containment by reference called *composition* and the strong containment by value called *aggregation*.

In the aggregation relationship, part and whole objects do not exist independently. They are the same lifecyle and are tightly coupled. Aggregation is typically implemented in OO languages by including the contained objects as instance variables inside the whole object.

When a whole object is composed of part objects, their instances are loosely coupled and may be created and destroy independently.

Such conceptual relationships are typically translated in OO implementation code by academic and commercial modeling tools according to translation rules or language mappings. For instance, skeletons of code may be generated from UML diagrams in various OO languages such as C++ or Java.

Reuse based on object composition is called black-box reuse [GHJV95], because the only way to interact with composed objects is through their object interface and there are less implementation dependencies compared to class inheritance. Composition thus allows less coupling and is often preferable over inheritance [GHJV95]. Object composition may be as powerful as inheritance in terms of reuse thanks to delegation. The combination of inheritance from abstract interfaces, object composition and delegation is a powerful mechanism notably used in design patterns [GHJV95] and also in the component-oriented model.

For instance, consider a Mac class, which needs an ALU class to multiply and accumulate two integers. The unidirectional association from the Mac to the ALU class is represented in UML by an arrow as depicted in figure 4.3. This relation is translated into a pointer called *alu_ptr* of ALU type as shown in listing reflst:relation. The ALU interface contains the *add* and *multiply* abstract methods. Two different architectures are proposed to implement this interface through an inheritance relation. The *CompositeALU* class is composed of an Adder and Multiplier object appearing as instance variables, whereas two pointers on these object types are aggregated as instance variables in the *AggregateALU* class. The composition relation is a containment by value, which implies that both objects are unique to the ALU class and have the same lifecycle as the composite. As contrario, the participants in the aggregation relation are shared and may be created or destroyed independently of the ALU class leading to possible errors.

4.1.8 Concurrency, Synchronization and Coordination

For Booch et al. [BME+07], concurrency is a useful, but not essential concept of the object model. Besides, this concept was only cited by a single work among 88 papers reviewed in Armstrong's survey on OO concepts. However, concurrency is a fundamental characteristic to take into account from the specification, design and implementation of embedded systems.

Embedded systems are inherently concurrent to achieve several things at the same time and react simultaneously to various external events such as interruptions.

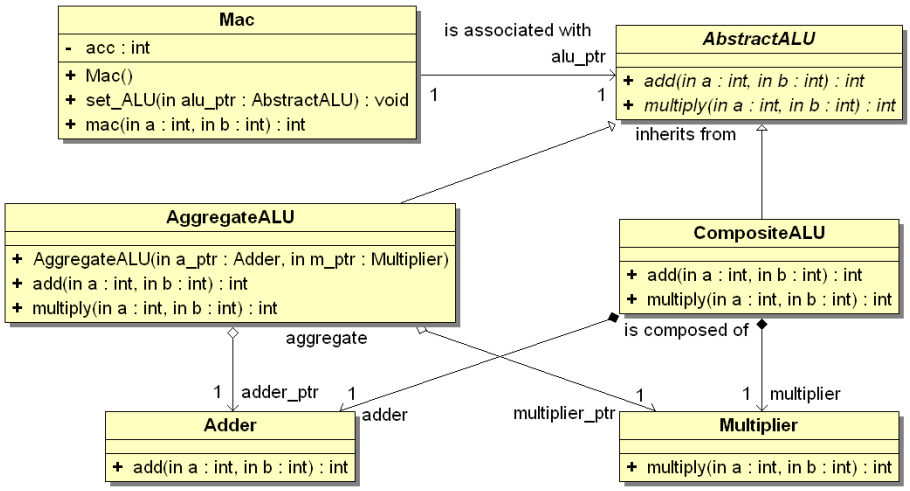


Figure 4.3: Illustration of the main relationships in UML

A flow of control designates the number of things that can take place concurrently. In SW, a flow of control may designate either a SW process or a SW thread in an operating system. A SW process is an heightweight and independent flow of control that has its own state information and address space. A SW thread is a lightweight flow of control executed concurrently with other threads within the same process.

Concurrent programming uses two main types of synchronization: mutual exclusion and condition synchronization [And00]. Mutual exclusion guarantees that when one process is accessing a shared resource e.g. a global variable no other process can gain access [BW07]. The sequence of statements that manipulates the shared resource is called a *critical section* of code. Mutual exclusion is typically implemented with mutual exclusion locks called *mutex* that protect critical sections of code. Condition synchronization ensures that a process only accesses a shared resource when this resource is in a particular state e.g. "buffer is not full". This is typically implemented with condition variables as provided by the Pthread library.

A semaphore is a non-negative counter decremented to acquire a lock and incremented to release a lock. If the semaphore value becomes zero, then the calling thread will wait until it acquires the semaphore.

A monitor protects a shared resource by guaranteeing that the execution of all operations accessing the resource are mutually exclusive. Instead of managing explicitly critical sections of code using locks, monitor declarations are used by compiler to transparently introduce synchronization code. For instance, Java and Ada 2005 use the `synchronized` keyword to declare monitors.

As an object is conceptually similar to a logical execution unit communicating through messages, the concept of object has been used to abstract concurrency. The combination of concurrent programming and object-oriented programming led to the creation of concurrent object models and concurrent object-oriented programming (COOP) languages [GT03] [Pap92].

A taxonomy of concurrent object models is proposed by Papathomas in [Pap92] according to three aspects: *object models*, *object threads* and *object interactions*. Object models are classified into three approaches: *orthogonal* if the concurrent execution of threads is independent of objects, *homogeneous* if objects are only active and *heterogeneous* if objects may be active or passive. Object threads refer to the number and scheduling of threads of control in an object: *sequential* with a single thread of control, *quasi-concurrent* with only one active thread among several possible threads and explicit thread switching, *concurrent* with several active threads. Object interactions are divided into the sending of

requests by clients and the acceptance of requests by servers. The sending of requests can be either *asynchronous* via one-way messages without reply or *synchronous* via request/reply messages. The acceptance of requests can be *explicit* through a statement, *conditional* through a guard or *reflective* through a meta-object. We note that the acceptance of reply messages is not considered in the studied software concurrent object models and languages.

In many concurrent programming languages, the concept of *active object* has been usually used to introduce concurrency between objects [GT03] [GMTB04]. For Booch et al., "Concurrency is the property that distinguishes an active object from one that is not active" [BME+07]. The term *actor* is sometimes employed to refer to active objects [Lea99]. The concept of active object relies on the support of concurrency by OO languages that is either built-in as in Java, Smalltalk and Ada or provided by external libraries such as POSIX OS libraries for C++.

In VHDL, parallelism is modeled with hardware semantics according to 1 writer-N readers protocol: only one sequential or concurrent process is allowed to write onto an entity port or an architecture signal to avoid conflicts, whereas any number of processes can read them in the same clock cycle. Otherwise, concurrency between several writers has to be explicitly managed by the hardware designer. An access arbiter controlling a multiplexer is typically used to implement an via an ad-hoc N writers-N readers protocol.

As concurrency inherently exists in embedded systems and have to be implemented in HW and SW languages by designers, object-oriented system specifications should make explicit concurrency among objects and among methods within an object.

In the UML object model, concurrency may be modeled by activity, state, sequence and class diagrams. Two granularity levels of concurrency may be specified in class diagram: at the class level with the *active class* concept or at the operation level with a dedicated concurrency attribute [GT03].

At the class level, a class is either *active* or *passive*. An active (resp. passive) object is an instance of an active (resp. passive) class.

According to [OMG07d] [RIJ04] [BRJ05] [BME+07], an active object has "its own thread of control", whereas a passive object is executed in the context of another thread of control. However, an attentive reading of [BRJ05] shows that "an active class is a class whose objects own one or more processes or threads". In this work, we consider that active object can own multiple flows of control to allow the most general object model as supported by object-oriented programming languages.

To model class behavior, two kinds of state machines exist in UML: *behavioral state machine* and *protocol state machine* [OMG07d]. A behavioral state machine specifies an executable behavior of a class or method through states and transitions, which are triggered by events. A transition is associated with a sequence of actions. In contrast, a protocol state machine specifies the legal order of operation invocations and signals reception that an object expects.

As the independent execution of an operation may access shared resources e.g. attributes in a passive object, the concurrent execution of multiple operations may require some execution control to prevent data corruption or inconsistent behavior. At the operation level, UML defines three semantics for concurrent invocations to the same passive object i.e. with the *isActive* attribute set to false. The meta-class *Operation* owns an inherited attribute called *concurrency*, whose value is defined by the enumeration *CallConcurrencyKind* [OMG07d] [BRJ05]:

- *sequential*: the operation assumes sequential invocations. Concurrent invocations have to be explicitly serialized to allow only one invocation at a time. Object semantics and integrity cannot be guaranteed with multiple flows of control. Hence, some coordination may be required between calling objects through synchronization, scheduling or arbitration.

- *guarded*: the operation supports simultaneous invocations from multiple flows of control, but only

a single execution. Only one invocation is executed, while the others are blocked until the first one completes. Thus, invocations are implicitly serialized.

- *concurrent*: the object supports simultaneous invocations and executions of any concurrent operation requested from multiple flows of control. For instance, these flows may read data or write to independent set of data.

Concurrency must be defined separately for each operation and for the entire object [BRJ05]. Operation-level concurrency specifies that multiple invocations of an operation can execute concurrently, while object-level concurrency means that invocations of different operations can execute concurrently.

In OO programming languages, the UML *concurrency* attribute is supported either natively like in Java with the equivalent *synchronized* keyword [GJSB05] or by construction using mutex, semaphores and monitors.

Active objects can control the simultaneous invocations of their behavior as they have their own thread(s) of control. Each instance of an active class controls its own concurrency. Their behavior is typically specified by state machines, whose semantics have to be clarified to support unambiguously concurrent invocations [GMTB04].

However, even after several decades of concurrent and object-oriented technologies, UML semantics have still ambiguities and inconsistencies concerning concurrency. Numerous works emphasize their limitations and propose to clarify UML semantics about active and passive object model [OS99] [GO01] [SS01] [GMTB04] [Sel04] [JS07].

As opposed to software objects, hardware modules have always one or several independent threads of control, which are modeled in VHDL by sequential processes and concurrent statements. From a conceptual viewpoint, VHDL processes correspond to SW threads and are modeled by SC_THREAD in SystemC. These processes are simulated concurrently, but truly run in parallel in hardware and coordinate their execution through signals within a VHDL architecture. Hardware entities can be considered as *active* objects, which are internally parallel and communicate with each other through signals. Synchronous or asynchronous interaction styles depend on the hardware invocation protocol. We will analyze how concurrency and synchronization have been implemented in hardware objects in section 4.2 p. 63.

4.1.9 Persistence

Persistence refers to the capability for an object to exist, while its creator e.g. process or thread has ceased to exist. In practice, persistence is performed by saving the state of an object from a volatile memory e.g. RAM or registers to a non-volatile or persistent storage e.g. file, database or flash memory. The persistent object is saved before it is destroyed and restored in another class instance at its next creation or "reincarnation". Objects that are not persistent are called *transient* objects.

4.1.10 Distribution

The metaphor of objects as little computers that interact only through messages [Kay93] provides a suitable model to abstract distributed systems and let to the emerging of the distributed object model. Object oriented-middleware platforms such as CORBA have been designed to offer an infrastructure supporting the development, deployment and execution of distributed object-oriented applications. One of the main goals of object-oriented middlewares is to make invisible or *transparent* the distribution of objects from the viewpoint of the objects themselves, and the application designers and users. The *Reference Model of Open Distributed Processing* (**RM-ODP**) [Org98] standardizes an object modeling approach to system

specification along with numerous transparency criteria. Some of these criteria directly originate from a previous initiative called the Advanced Networked Systems Architecture (ANSA) project [Man89].

RM-ODP defines the following distribution transparencies [Org98]:

- **Access transparency** masks differences in data representation and invocation mechanisms to enable interworking between objects.
- **Failure transparency** masks from an object the failure and possible recovery of other objects (or itself) to enable fault tolerance. Failure transparency depends on replication and concurrency transparencies [PPL+06a].
- **Location transparency** masks the use of information about location in space when identifying and binding to interfaces. This distribution transparency provides a logical view of naming, independent of actual physical location.
- **Migration transparency** masks from an object the ability of a system to change the location of that object. Migration is often used to achieve load balancing and reduce latency. Migration transparency depends on access and location transparencies [PPL+06a].
- **Relocation transparency** masks relocation of an interface from other interfaces bound to it.
- **Replication transparency** masks the use of a group of mutually behaviorally compatible objects to support an interface. Replication is often used to enhance performance and availability. Replication transparency depends on access and location transparencies [PPL+06a].
- **Persistence transparency** masks from an object the deactivation and reactivation of other objects (or itself).
- **Transaction transparency** masks coordination of activities among objects to guarantee consistency.

The ANSA project also defines *technology transparency* as the fact that different technologies such as programming languages or operating systems are hidden from the user [PRP05]. Other forms of transparency include [PPL+06a]:

- **Scalability transparency**: the system can continue to operate as more resources and services are added and more concurrent requests have to be dispatched and processed. Scalability transparency depends on migration and replication transparencies.
- **Performance transparency** means it is transparent to users and programmers how performance is actually achieved. Performance transparency depends on migration and replication transparencies.
- **Concurrency transparency** allows multiple clients to request transparently the same service simultaneously. Each client does not need to know the existence of other clients or how concurrency and consistency are managed by the service.

However, a variable number of these distribution criteria may be not relevant for some class of embedded systems, which are static by nature. In these cases, objects are statically assigned to execution nodes and do not require location, migration or replication transparencies.

In a unified approach, an application of middleware concepts into hardware necessitates defining an architecture supporting part or all of the distribution transparencies. In the following we will present the works performed towards this goal.

4.2 Object-Oriented Hardware Design

Although it originates from software engineering, the object model appears to be sufficiently general and technology independent to be applicable to the hardware domain in an attempt to provide a common and unified high-level approach to embedded systems design.

As studied in a recent survey performed in 2004 [Dv04a], many researchers have proposed to apply the object-oriented concepts for hardware design. The main questions that cross our mind are how many object concepts and how these concepts have been interpreted and implemented in hardware.

In the following, the various forms through which the object-oriented model has been applied down to hardware will be presented. On the one hand, hardware description languages like VHDL and Verilog have been extended to support object-oriented features like OO-VHDLs and SystemVerilog. On the other hand, mainstream object-oriented languages such as Java and C++ have been extended with hardware-specific libraries like SystemC. Moreover, hardware object architectures have been proposed using JavaBean, Finite-State Machine (FSM) and distributed objects. Other works have presented object-oriented hardware platforms with OO-ASIP and hardware middlewares like StepNP, OOCE (Object-Oriented Communication Engine) and hardware CORBA, implementations. Finally, model-driven engineering has been used to model hardware objects in UML and generate VHDL and Verilog. The presentation of the related work and the associated discussions will be focused on the interpretations of objects in hardware and particularly on the hardware interfaces used for method invocations.

4.2.1 Object-Oriented Extensions to Hardware Description Languages

In the 1990s, numerous works were devoted to the proposition of object-oriented extensions to HDLs [AWM99]. VHDL was widely selected since it natively separates the declaration of component interface in *entity* from their implementation(s) in *architecture(s)*. These extensions took the form of new OO language constructs, which were translated by preprocessors into synthesizable VHDL. According to [Rad00], two main approaches have been distinguished to provide object-orientation to VHDL.

In the *entity-based* approach, a VHDL entity is considered as a class declaration, whereas a VHDL architecture is viewed as a class implementation. Hence, hardware objects are interpreted as VHDL component instances.

The *type-based* approach extends HDLs with a novel class data type, whose instances are equivalent to VHDL variables and signals. If one consider that VHDL was inspired by Ada, this approach is similar to consider the object-oriented data types added to Ada83 to build Ada95. Examples of this category include OO-VHDL [Sch99], SUAVE [AWM98] and Objective VHDL [Rad00]. Interestingly, an IEEE study group called OO-VHDL [12] was created in late 1993 to establish requirements for an object-oriented VHDL and to propose language extensions. A merger of the SUAVE and Objective VHDL proposals was investigated [AR98]. However, many advanced concepts of SUAVE were not synthesizable and no simulator was developed, whereas Objective VHDL was designed to be synthesizable, but the translation tool to VHDL was not mature enough [Opp05]. Both works stopped and these efforts unfortunately failed to be standardized. Afterwards, the community preferred to model hardware design using high-level object-oriented languages as presented below. Recent revisions of the VHDL standard called VHDL-200x should include object-orientation with inheritance and interface modeling based on Ashenden's proposal according to [BMB+04] [Lew07].

Furthermore, Verilog has been enhanced with object-oriented extensions to build SystemVerilog 3.x [Acc04b] to facilitate the verification of abstract hardware models. As Verilog syntax is inspired by the C language, SystemVerilog naturally provides OO extensions close to C++. A *class* construct was intro-

[12] http://www.eda.org/oovhdl

duced to declare instance variables called *properties* and member functions named *methods*. SystemVerilog supports class inheritance, polymorphism and dynamic allocation of classes, but the synthesizable semantics of these advanced features are not defined.

Although our work could benefit from OO extensions to HDLs, we do not consider such custom extensions and we restrict our proposition to the synthesizable subset of standard HDLs such as VHDL and Verilog.

4.2.2 Object-Oriented Languages for Hardware Design

In order to raise the level of hardware abstraction and increase simulation speed, mainstream OO languages such as Java and C++ have been intensively used to model hardware design. The main objective was to replace the use of two languages such as C/C++ and VHDL/Verilog to specify software and hardware with different programming models and semantics by a common language for system specification [HO97, YMS+98, KSKR00]. Obviously, usual OO languages do not natively provide adequate constructs to model hardware characteristics like ports, wires, time, event or specific data types such as bitvectors. OO libraries have thus been developed to provide new language syntax and semantics close to those found in HDLs. The main advantage of this approach is the possibility to model both hardware and software thanks to widely available software tools like the free and open-source GNU tools. This trend gained so importance in industry and academia that a de facto C++ library called *SystemC* became an IEEE standard in 2005.

Java

The Java software programming language has been proposed in the 2000s to design, specify and simulate hardware/software embedded systems. Java is a high-level and platform independent language with real-time extension. Java provides an object-oriented framework with standard APIs allowing to explicitly specify concurrency with threads and monitors, and reactivity with events and timing. However, dynamic loading and linking of Java objects at run-time complicate the analysis and optimization of system specification [HO97] [YMS+98].

In [HO97], Java is used as an executable system specification language to support functional validation and hardware/software co-design. In [YMS+98], Java serves as a general purpose language to capture a first system specification, which is successively refined by an user-guided tool into a formal model that is deterministic in term of behavior, bounded execution time and memory consumption.

Whereas both previous approaches are only focused on behavioral specifications of hardware/software systems from an algorithmic viewpoint, Java is considered as a behavioral and structural description at system, algorithmic and register transfer levels in [KR98], [KRK99] and [KSKR00]. These later works will be described in more details in the next section.

From our viewpoint, Java cannot be considered as a system-level language as Java is basically not used for real-time signal-processing applications on embedded GPPs and DSPs. In contrast, C and C++ are quasi-exclusively used in such a context. Hence, system-level languages derived from C/C++ such as SystemC are more relevant notably to refine a behavioral specification down to implementations on GPPs and DSPs.

SystemC

SystemC [GLMS02] is an OO modeling language based on a C++ class library. It allows the modeling of hardware/software embedded systems at various levels of abstraction. However, there is still no consensus on the number and purpose of these abstraction levels. Although the OSCI Transaction Level

Modeling (TLM) [Ghe06] standards have defined some APIs (TLM 1.0 and TLM 2.0), the main difficulty remains the refinement from one level of abstraction to another. The transformation from a high-level functional specification down to a synthesizable hardware description is still tedious and error-prone. For instance, SystemC FIFOs have to be manually translated into signals. Moreover, the draft of a synthesizable subset [oOSI04] started in 2004 is still under public review. As a consequence, commercial and academic tools such as Cynthesizer [Sys, CM08] and FOSSY [GGON07] may propose their own abstraction levels and refinement methodology. Furthermore, OO concepts like inheritance or polymorphism are not supported by such synthesis tools. In conclusion, SystemC allows OO HW modeling, but not OO HW design [SVL01] [SKWS+04].

From SystemC-Plus to OSSS+R at the University of Oldenburg and OFFIS Institute In the scope of the project ICODES [ICO08], a synthesizable subset of SystemC called *OSSS* [GGON07] [BGG+07] for **O**ldenburg **S**ystem **S**ynthesis **S**ubset was developed by the OFFIS research institute and the University of Oldenburg to improve the refinement of SystemC models down to HW implementation. OSSS is a C++ class library on top of SystemC with well-defined synthesizable semantics to model HW/SW embedded systems based on OO principles. OSSS is based on the SystemC-Plus library [ICE01] [GTF+02] [GO03]. A *Remote Method Invocation* (**RMI**) mechanism has been designed to transform method-based communications between HW/SW objects to signal-based communications through buses and point-to-point channels.

The concept of *Shared Object* is used to homogeneously model HW/SW as well as HW/HW communications through method invocations. Shared objects arbitrate concurrent invocations on shared resources like a buffer using provided or user-defined scheduling algorithms.

In the scope of the ANDRES [AND08] project, the OSSS+R [13] [SON06] library is developed as an extension of the existing OSSS library to target the modeling and simulation of dynamic and partial reconfiguration in digital hardware systems. OSSS+R aims to offer new language features to SystemC to manage the reconfiguration process transparently from the designer's point of view. OSSS+R approach is based on an analogy between polymorphism and hardware run-time reconfiguration. Polymorphism allows one to dynamically change the behavior of an object, while the code to call it remains static. The same remains valid for reconfigurable hardware where the hardware interface between the static and dynamic part is fixed, while the implementation of the dynamic part may be changed at run-time by loading partial bitstreams. In OSSS+R, polymorphic objects act as reconfigurable containers to model reconfigurable areas. A container is responsible for arbitrating the access to the contained objects and to request a unique reconfiguration controler.

As the analogy between polymorphism and routing in [GZH07], we consider that the analogy between polymorphism and reconfiguration is interesting since the high-level operation-based interface specified in a class hierarchy is directly translated into a low-level signal-based hardware interface specified in VHDL. Moreover, polymorphic objects cope with non functional services such as object life cycle and resource sharing in a similar way to *containers* and component factories in the component approach described in the next section.

The OASE project at the University of Tübingen The OASE project [UoT08] at the University of Tübingen was concerned with the object-oriented design of hardware/software systems. This project studied the modeling [SKWS+04], specification [KSKR00], co-simulation [KR98, KRK99] partitioning [SKKR01], synthesis and verification [KOSK+01, KOW+01] of embedded systems based on various object-oriented languages such as Java [KR98, KRK99, KSKR00], **e** [KOSK+01, KOW+01], C++ and SystemC [SKWS+04].

[13] OSSS+Reconfiguration

OOAS proposes a unified object-oriented design flow from high-level OO specifications down to HW/SW implementations. Classical synthesis approaches such as CDFG extraction and data dependencies analysis have been adapted to take into account OO features like polymorphism with a multiplexer. OOAS uses inheritance for HW/SW partitioning, which is invasive as the user class diagram has to be changed at each new partitioning. An alternative consists in using deployment diagrams like [SFB05]. We also observe that the proxy design pattern was applied to support transparent blocking communications between HW/SW objects and the delegation pattern to migrate part of an object state and implementation. Both patterns are used by the IDL compilers of object-oriented middlewares to generate communication adapters.

Instead of relying on implementation-specific object-oriented languages such as C++, Java or even SystemC, we consider interface-centric and component-oriented specification at system-level in CORBA IDL3 or UML2. As the mapping of such specifications to software languages already exists, we focus on mappings to SystemC and HDLs.

4.2.3 Object-Oriented Hardware Architecture

The object-orientation of hardware allows one to raise the abstraction level of interactions between hardware blocks as the low-level transfer of control and data of the communication protocol is interpreted as a high-level method invocation. According to Kuhn et al. [KRK99, KSKR00], two interpretations of hardware objects can be distinguished: the behavioral and the structural interpretation. As we focus on hardware object interfaces, both interpretations provide two kind of hardware interfaces which are presented below.

Behavioral Interpretation

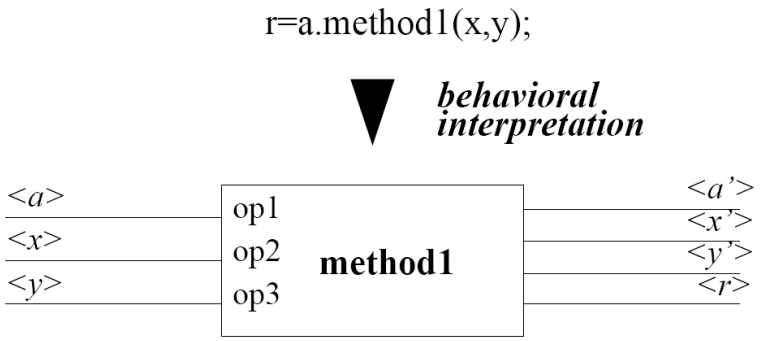


Figure 4.4: behavioral Interpretation [KRK99][KSKR00]

Description In the behavioral interpretation shown in figure 4.4, an object method corresponds to a hardware module that takes as inputs an object environment and its input method parameters and reply by a return value along with its inout parameters.

This interpretation is similar to the way object-oriented extensions to programming languages such as C++ are built on top of procedural languages like C. Indeed, a method invocation is equivalent to calling a function using as additional parameter an object environment that contains the object state. In C++ and Java, this object environment is represented by the *this* pointer or reference. In software languages, this environment is typically passed by reference, whereas Kuhn et al. propose to pass it by value in HDLs. As opposed to [Dv04a] and [KRK99], we consider that the behavioral interpretation is inappropriate for the external interface of hardware object, since it breaks the encapsulation concept of the object

model. Indeed this interpretation does support encapsulation of behavior inside a hardware black box, but no encapsulation of state as public and *private* object attributes would be clearly transferred outside hardware modules.

Structural Interpretation

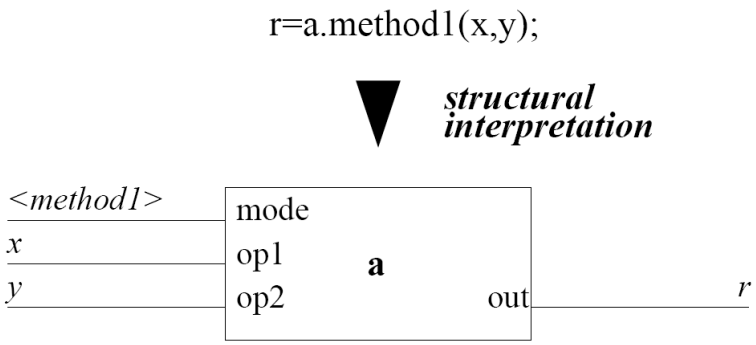


Figure 4.5: Structural Interpretation

The structural interpretation shown in figure 4.5 considers objects as hardware modules. Object methods are invoked by writing to the input ports of the hardware object a request message consisting of a method identifier and of the input method parameters. The result and the output parameters are then read, thanks to the output ports. The hardware invocation protocol may rely on additional signals to indicate the validity of parameters data and to signal the method completion. The type of method parameters must be of primitive types to be synthesizable as bitvector [KRK99] [Rad00]. This interpretation is quite natural for hardware designers and supports both encapsulation of state and behavior. Moreover, hardware objects can be used like any other legacy hardware modules with a structural description [KRK99].

Damacevicius et al. suggest a third interpretation [Dv04a] of hardware objects based on Radetzki's implementation [Rad00]. However, we will notice that this work follows in fact both approaches: a structural interpretation according to the public and external interface of Radetzki's hardware objects and a behavioral interpretation according to their private internal interface. Both interpretations are thus complementary. In virtue of the encapsulation principle, only the public object interface really matters for client objects, while the private interface and their implementation are hidden. Hence, we consider that Radetzki's approach belongs to a structural interpretation.

As a conclusion, we argue that the only valid interpretation of hardware objects is structural as it guarantees both encapsulation of state and behavior. In the remaining parts of this work, we will consider the structural approach and focus on a fundamental object concept neglected in the hardware object literature: the *hardware object interface*. Indeed, three structural approaches will be now presented each with its own hardware interface.

FSM-based Implementation of Radetzki

Radetzki proposed in his PhD thesis [Rad00] a hardware implementation of the main OO concepts such as encapsulation, message passing, inheritance and polymorphism. The mapping from an object-oriented specification in UML to a hardware model was mathematically formalized. Concretely, object instance variables of primitive types are mapped to a synthesizable bitvector and the object interface to a hardware module interface. Hardware objects are modeled in an OO-HDL called Objective VHDL and implemented based on the well-known and synthesizable FSM model as shown in figure 4.6. Indeed,

this object architecture typically separates the control path with state transitions and the output logic from the data path with state storage in memory.

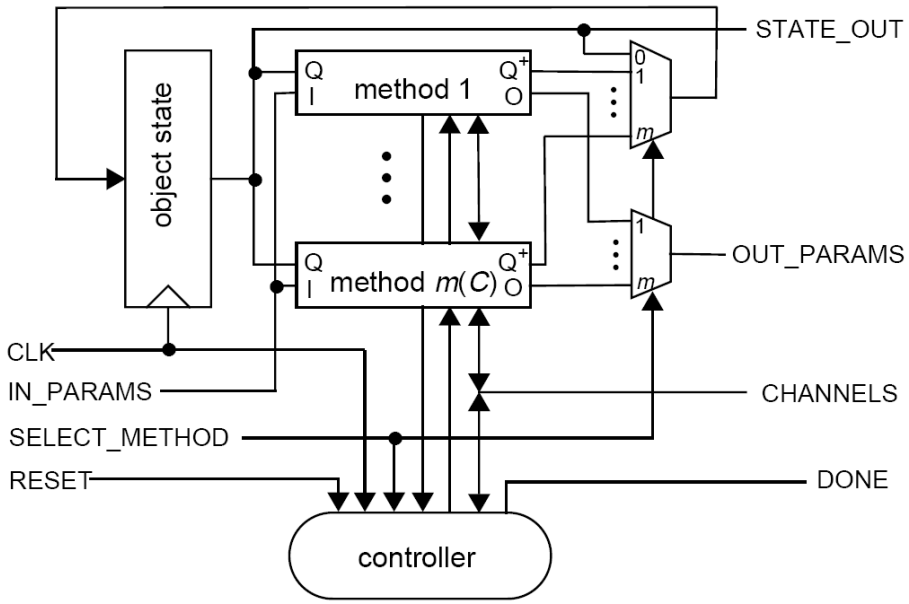


Figure 4.6: FSM-based implementation of hardware object [Rad00]

The hardware object interface consists of the following signals:

- CLK and RESET signals are the clock and reset signals to synchronize and initialize the object implementation.
- SELECT_METHOD signals carry a binary encoded method identifier. This bitvector controls a multiplexer, which selects the output parameters of the desired method implementation. The value '0' is reserved to mean 'no invocation request'. The signal width is the binary encoding of the total number of object methods plus one.
- IN_PARAMS signals contain the concatenation of input parameters of one method at a time. The width of this bitvector is the maximum number of bits required to encode the largest aggregation of inputs among all object methods.
- STATE_OUT signals transport the current object state monitored by guards to forward an invocation request if a so-called guard expression is verified.
- OUT_PARAMS signals carry the largest concatenation of output parameters among all methods. As input parameters, output parameters are binary encoded.
- DONE signal is used both to accept an invocation request and to signal that output parameters are available when a method execution is completed.
- CHANNELS signals are used for the communication between a client to a server method.

We notice that this public and external hardware interface corresponds to the *structural interpretation* of Kuhn et al. [KRK99].

The object state is stored in memory elements such as registers or SRAMs depending on instance variable size. An object method is implemented as a hardware sub-component.

Inheritance is supported via expansion by adding new method components and extending the object state memory, the controler and the multiplexers shown in figure 4.6. The object state memory may include additional bits to encode a class identifier or tag like Ada to support polymorphism by value. The implementation of an inherited method is dynamically dispatched by a multiplexer according to the current object class type in the tag, which may change at run-time.

Message passing is performed by sending a method invocation on a point-to-point channel from a client object to a server object. As concurrent client objects may send an invocation request to a server object, the access to the server interface is granted by an arbiter that is composed of a scheduler and a guard. The guard accepts an invocation request only if a corresponding execution condition is true. The client object is blocked until the condition becomes true. The guard indicates to the scheduler the accepted client requests. The scheduler selects a client request according to a classical scheduling policy such as a round-robin policy or a priority-based policy.

As a conclusion, the proposed architecture for hardware objects addresses all mentioned OO concepts. The combination of both Kuhn et al.'s interpretations demonstrates their complementarity for the implementation of objects. The hardware implementation of arbiters, schedulers and guards permits to guarantee the concurrency and synchronization constraints defined in object-oriented specifications. However, we observe some drawbacks in this architecture. This implementation may require a lot of silicon area. Even if hardware resources may be shared with classical high-level synthesis methods, static class instance variables and method implementations are not shared among class instances, since a method implementation is duplicated in each class instance. Moreover, all method parameters are sent in parallel simultaneously. This is not desirable for routing purpose. Furthermore, the invocation and execution of object methods are sequential, since the hardware object interface is shared by all method invocations and the hardware object implementation is based on a FSM. This is regrettable, as hardware is inherently parallel. The important requirement of parallelism between configuration and computation cannot be satisfied for real-time signal processing applications. Indeed, the parallel invocation of configuration and computation methods is intrinsically impossible with a single hardware interface.

As typical shared bus architecture, a single channel arbiter implemented as a multiplexer is not scalable for a great number of clients and constitutes a communication bottleneck. Communications between distributed and heterogeneous HW/SW objects is not considered through communication infrastructures like buses. Moreover, the only supported interaction mechanism is blocking, whereas non-blocking interactions are required to hide communication latency.

In the following ICODES and ANDRES projects, object-oriented specifications have been captured in SystemC instead of OO-VHDL with the OSSS(+R) library. Radetzki's hardware object architecture has evolved into the previously presented concept of shared object. Heterogeneous communications between HW/SW objects have been supported by a RMI protocol through OSSS channels.

Figure 4.7 represents the hardware architecture of an OSSS shared object, which is connected to clients via two communication interfaces IF1 and IF2. For instance, these interfaces may be an OPB bus used by a SW client and a point-to-point connection used by a HW client. A *Scheduler* determines which client request is granted according to the Client ID and Method ID of the RMI request header, a *Guard Evaluator* and a pre-defined or user-defined scheduling algorithm. The Scheduler returns the granted Client ID and MethodID to a *controler*. This controler selects via a multiplexer the invocation parameters from the RMI request message to be written into an Argument RAM (Arg-RAM). It then asserts the MID signal to activate the Shared Object method, which can directly read and write operation parameters from/to the Arg-RAM. After operation completion, the controler must indicate to the granted interface module that it can build a reply RMI message from the ArgRAM and send it on the OSSS channel.

We notice that OSSS application and platform models provide an executable specification, which encompasses explicit behavioral informations required for high-level synthesis. In contrast, an abstract

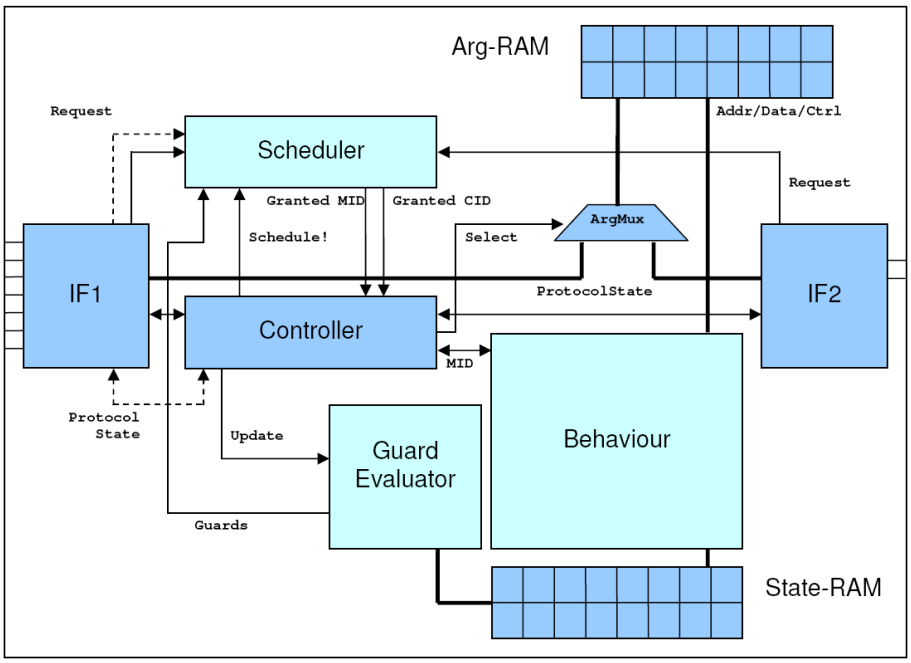


Figure 4.7: OSSS shared object architecture [BGG+07]

interface specification e.g. in CORBA IDL does not inherently provide such information to guide interface synthesis. All necessary behavioral properties have thus to be incorporated into a complementary specification e.g. in UML or in a mapping configuration file.

Distributed Objects from Barba et al.

Barba et al. [BRM+06] propose to apply the distributed object model to SoCs that are considered as a distributed system in which hardware/software objects communicate through the on-chip interconnection network. The proposed hardware object model is based on a standardized method invocation interface and a synchronous method invocation protocol. One of their goals is to define a direct mapping from the specified abstract object interface to the concrete hardware module interface, which allows it to automate integration. The abstract object interface is specified in UML and in an interface definition language called SLICE (Specification Language for ICE). ICE (Internet Communications Engine) is a proprietary object-oriented middleware inspired by CORBA, which was notably created by Michi Henning [Hen04] [HM07] at ZeroC Inc. [HS08].

The hardware object interface shown in figure 4.8 is composed of:

- common signals such as clock and reset
- an input signal to invoke a method and indicates that input method parameters are valid.
- a logical group of input and output signals called *data port* for one or several input and output method parameters, which may be transferred in parallel or serial to save area.
- an output signal called *done* to inform that the method is completed and that output parameters are available.

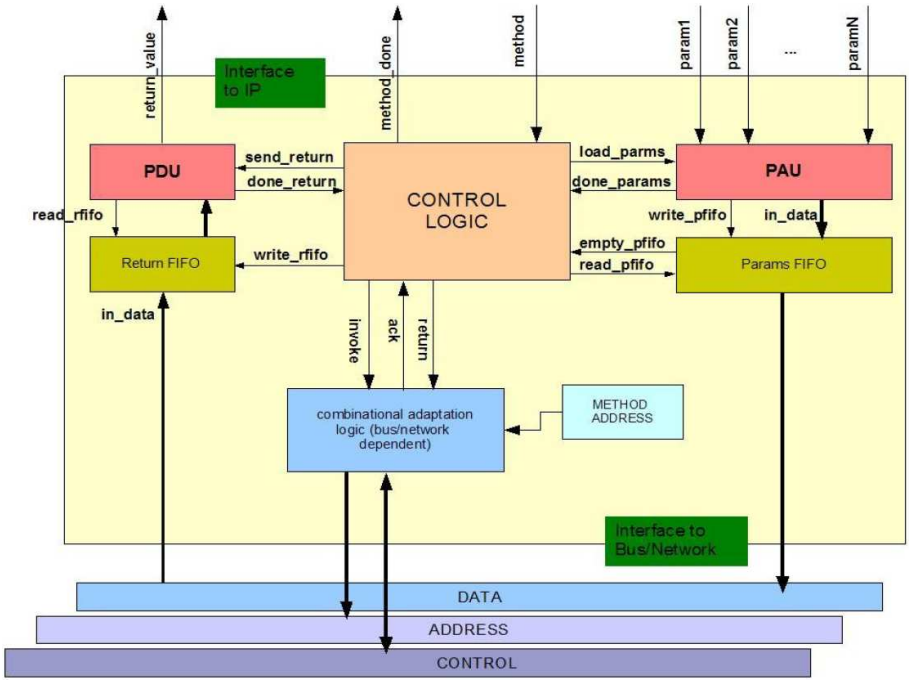


Figure 4.8: Hardware Proxy Architecture [BRM+06]

The proxy design pattern [GHJV95] is used to encapsulate the network protocol and to support location transparency. The proxy architecture is depicted in figure 4.8. As its software counterpart, the role of the hardware proxy is to handle the RMI protocol. It translates a local method invocation into an invocation request message, which is transferred on the available interconnection network e.g. bus and NoC. A method invocation is interpreted as a sequence of reads and writes on the communication network. Some data encoding/decoding rules are required to standardize the data layout "on the wires". A hardware object has an unique identifier, which corresponds to the base address of the skeleton. The method signal is associated with a skeleton address offset in the proxy that is transferred on the network address bus, while packed input and output method parameters are respectively written and read on the network data bus. The proxy architecture template is composed of a *Port Acquisition Unit* (**PAU**) and the *Port Delivery Unit* (**PDU**) modules responsible for the packing and unpacking of method parameters. The *Combinational Adaptation Logic* (**CAL**) module establishes a mapping between the RMI protocol and bus/network protocols. This mapping is described for each network protocol in configuration files and allows one to be independent of protocols. The object-oriented interface synthesis flow is based on Object and Collaboration diagrams [RIJ04] to represent a system in terms of its objects, their relationships and the ordered sequence of messages they exchange. These UML diagrams guides the wrapper generation process, which consists in customizing the proxy architecture template for a given interconnection network (bus, NoC).

The proposed hardware object model follows a structural approach, in which objects are considered as IPs. Compared to Radetzki's works, there is a clear separation between local method invocations based on the signal-level protocol and remote method invocation based on message passing. The proposed invocation protocol and hardware interface are simple, custom and an adaptation is required with a standard bus/NoC protocol. As soon as the interface protocol is concerned, a binary encoding of methods similar to Radetzki's approach is preferable over the linear encoding used to save area and ease routing

as the number of method activation signals is equal to the number of methods. From a hardware protocol design point of view, a single activation signal is better than the bitvector proposed by Radetzki and method parameters may be transferred in parallel or in serial to save area.

One weakness of this proposition is that method invocations may be potentially parallel, but no mechanism is considered to guarantee state coherency. Moreover, the arbitration and scheduling of concurrent accesses to hardware objects are entirely delegated to the underlying communication network.

HardwareBean from Kuhn et al.

Kuhn et al. proposed to use Java to model HW/SW systems at behavioral, algorithmic and Register-Transfer Level (RTL). A Java class library was developed to model and simulate hardware at RTL level in Java [KR98]. Kuhn et al. specified hardware objects as JavaBeans [Mic97], which are called *HardwareBeans*.

A JavaBean is an ordinary Java class that must conform to a set of standard coding and naming conventions. A JavaBean class is mainly characterized by attributes called *properties* that can be read and written thanks to getters and setters, *methods* that are public Java methods and *events* that are fired by event source bean and received by event listener bean. Events notify interested listener objects that a state change has occurred [Mic97].

The *java2struct* tool [KRK99] transforms the behavioral description of a hardware object based on method invocations into the structural description of a HardwareBean. Each port of a hardware module is represented by a JavaBean property.

The hardware module interface contains:

- an input *method* port to indicate a method identifier;
- an input *start* port to invoke a method;
- an input port for each input method parameter;
- an output *ready* port to signal that the result is available;
- an output port for each return/output method parameter.

Getter and setter methods are respectively created for each input and output ports composing the hardware object interface. The width of a hardware port is determined during synthesis by the associated property data type. Events are used to notify that the value of a property i.e. a port has changed. This fires an event that may be captured by a listening JavaBean that may retrieved the computed values by invoking getter methods on the skeleton. The stub translates a method invocation towards a hardware object into a set of methods invocation on the skeleton.

Thanks to inheritance, a HardwareBean may be used both as a behavioral and structural description. A stub and skeleton JavaBean are generated from the hardware class. Another tool called *JavaPart* allows one to simulate and prototype hardware/software systems. A proxy object is a HardwareBean that may forward a method invocation from a software object to ever the software bean for simulation or the synthesized hardware bean for prototyping. The *JavaSynth* tool translates a hardwareBean into a VHDL component according to a given object interpretation and may include proxy objects. Communication from hardware to software is also supported.

Kuhn et al propose the JavaBean object model to provide a common framework for HW/SW objects. This work demonstrates the need to go beyond the general principles of the object model by adopting a more constrained model with common rules. Here, the JavaBean model only provides common coding and naming conventions, but the next section will show that component models impose common

interfaces to automatically manage component creation/destruction, configuration and communication. Distributed communications are besides mediated through stubs/skeletons. Like Radetzki's work, the invocation interface only permits sequential invocations and parallel transfers of parameters. However, the HardwareBean interface has a single activation signal and cannot control the flow of requests as in Barba's proposal. Moreover, concurrent accesses to a HardwareBean are not taken into account for instance using arbiters, schedulers or guarded methods as in Radetzki.

OO-ASIP

In the scope of the ODYSSEY (Object-oriented Design and sYntheSiS of Embedded sYstems) project, an Object-Oriented Application Specific Instruction set Processor (OO-ASIP) [Gou05] has been proposed to provide another implementation alternative to the OO concepts. An OO-ASIP is a dedicated processor, whose custom instructions encode object methods, while instruction operands are considered as method parameters.

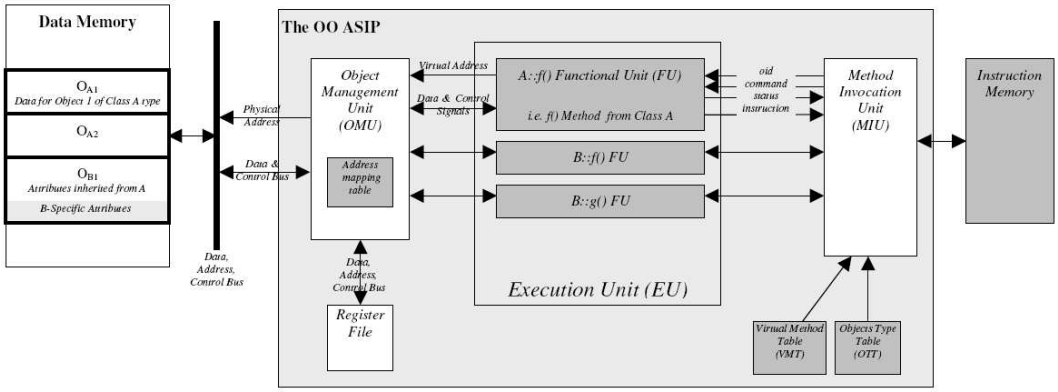


Figure 4.9: OO-ASIP Architecture Template [GHM03] (dark grey indicates dedicated parts)

A first architecture of the OO-ASIP [GHM03] is depicted in figure 4.9. This architecture is similar to a von Neumann processor with Harvard architecture [PH07], which is extended to support object-orientation in hardware. The traditional Control Unit, ALU and memory controler are respectively transformed into a *Method Invocation Unit* (**MIU**), an *Execution Unit* (**EU**) and an *Object Management Unit* (**OMU**) in the OO-ASIP [Gou05]. The MIU implements a subset of the Java Virtual Machine (JVM) instruction set and particularly its *invokevirtual* instruction, which is used to dispatch a method invocation on an instance method according to the runtime type of the object [LY99]. This instruction is used as OO-ASIP instructions to indicate hardware method implementations. A Functional Unit implements a method of the "hardware class library" and is part of the Execution Unit. The OMU manages the object data stored in registers or in data memory and the access from the FUs to these shared resources. A HW method may invoke another HW or SW method.

In [GZH07], the processor and FUs have been interconnected through a Network of Chip (NoC) and the OO-ASIP architecture has evolved to exploit the routing capabilities of NoCs. Goudarzi et al. propose to use packet-switched networks to dispatch virtual methods. Method invocations are seen as network packets and a network address is assigned to each HW or SW implementations of methods. Thanks to this well-chosen addressing scheme, virtual method dispatching is equivalent to packet routing.

A method implementation i.e. FU has two interfaces: an invocation interface with the MIU and an object data access interface with the OMU [Gou05]. The invocation interface consists of a *command*

signal to activate the FU, an *oid* signal to indicate the object on which the operation is invoked, signals for input method parameters, a *status* signal to indicate operation completion and signals for output method parameters. Operation parameters of primitive data types like integers are passed by value, whereas only objects in global memory can be passed by reference. An optional *instruction* signal allows a FU to invoke another operation through the MIU i.e. when the method implementation acts as a client. The object data access interface consists of a *virtual address* signal, which carries an oid and an index to an object memory element, and data/control signals to read and write data. This virtual address is then translated to a physical address by the OMU, which performs the desired memory accesses.

In conclusion, an object is separated between its state, which is stored in OO-ASIP shared memory, and its behavior, which is either implemented in software as Java instructions or in hardware as custom instruction accelerators. As each hardware method is associated with a hardware module with an object data access interface, this architecture may correspond to the behavioral interpretation of Kuhn et al. [KRK99]. As in [BRM+06], a single activation signal thus is required instead of a method identifier as in [Rad00]. A single method implementation per class reduces area and power consumption, but implies the sequential execution of a method for all objects as opposed to one method implementation per object as in [Rad00]. As noticed in [GHM03], both approaches represent different trade-offs between concurrency and area/power consumption in the object-oriented design space We observe that the duality between method invocation and message passing through a network is similar to how object-oriented middlewares work.

NXP Interface-based and Platform-based approach: Task Transaction Level (TTL)

Van der Wolf et al. propose an interface-based [RSV97] and platform-based [SV02] approach for MPSoC design [vdWdKH+04][vdWdKH+06]. The authors argue for an integration of HW/SW modules based on the mapping of *abstract interfaces* to systematically refine the specification of application models into its implementation on MPSoC platforms and thus to improve design productivity and quality.

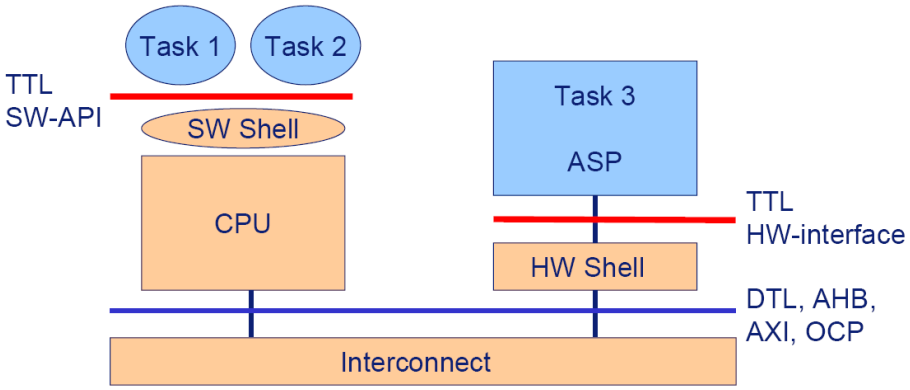


Figure 4.10: HW/SW mapping of the abstract TTL interface [vdWdKH+04]

Multimedia applications are designed as a set of concurrent *task*s which exchange streams of tokens through communication channels. Tasks communicate by calling functions on their ports, which implement an abstract interface called *Task Transaction Level* (**TTL**). Each channel supports uni-directional, reliable, ordered and multicast communications between one producer task and several consumer tasks. The TTL interface supports different communication styles through a set of interface types for each task port. The seven TTL interface types represent different tradeoffs between abstraction and efficiency in terms of synchronization, data transfer granularity and ordering of data accesses and re-

leases. The platform infrastructure provides services such as inter-task communication, multi-tasking and (re)configuration to application tasks through the abstract TTL interface. Both shared memory and message passing models are supported.

The abstract TTL interface has been implemented as a SW API for the SW shell and as a HW interface for an Application-Specific Processor (ASP) as illustrated in figure 4.10. The HW shell implements the platform services on top of a lower interconnect interface such as AHB, AXI, DTL or OCP. The abstract TTL interface has been mapped onto a HW interface that resides between the application co-processor and the TTL HW shell connected to the interconnection network via a proprietary DTL interface. The TTL interface consists of the following signals:

- the *request* and *acknowledge* signals are used to initiate and return from a function call, so only sequential function calls are supported by a shell. Concurrent function calls could be implemented with multiple pairs of shell and DTL interface instead of more complex multiple DTL interfaces per shell.
- the *port_type* signal indicates an input or output task port with the respective value 0 and 1. The shell implementation includes a mapping table between port_ids and the channel location to which they are logically connected.
- the *prim_req_type* signal should refer to the 2^2 TTL interface classes: Combined Blocking, Relative, Direct In-Order and Direct Out-Of-Order, since no description is available in [vdWdKH$^+$04].
- the *is_non_blocking* and *is_granted* signals are used to designate a non-blocking function and return the boolean result value.
- the *port_id* signals indicate the logical task port on which a sequential call is performed.
- the *offset* signals should correspond to the offset parameter of the store and load operations of the Relative TTL interface class and maybe the count parameter of acquire and release functions, since no description is available in [vdWdKH$^+$04].
- the *esize* signal represents the size parameter of the store and load operations.
- the *wr_** and *rd_** signal are used to handshake the transfer of data between the coprocessor and the shell.

We consider this API at the same abstraction level than communication APIs such as MPI. This API has been mapped onto hardware interfaces according to a custom mapping. As a result, this manual mapping has to be performed at each new definition of an abstract business interface. We argue that arbitrary user-defined application interfaces should be defined and manually or automatically mapped into hardware interfaces according to strict and explicit mapping rules to enable portable definitions of abstract business interfaces and to ease integration independently of any proprietary platform. These domain-specific interfaces could be mapped on top of available communication APIs such as TTL in the same way that CORBA IDL application interfaces could be mapped on top of MPI.

Synthesis and Contribution

We argued that the behavioral interpretation of hardware object breaks the encapsulation principle of the object model. Moreover, the structural interpretation is coherent with the component-based model supported by HDLs. A synthesizable implementation of main OO concepts such as inheritance, message passing and polymorphism is undeniably feasible with a structural interpretation of objects. Such

an application of OO principles may potentially provide the same benefits as obtained in software to hardware design such as a higher level of abstraction and the reuse of specification and implementation. However, these works suffer from some limitations notably due to the object model itself. The proposed hardware interfaces and protocols do not provide enough flexibility to address the complex communication requirements of modern IPs. For instance, the previous hardware objects cannot control the flow of requests and replies. Moreover, they provide a single hardware interface, which only supports sequential invocation and execution of methods. Both sequential and concurrent invocations of methods are made possible using several hardware interfaces.

4.2.4 Model Driven Engineering (MDE) for Hardware Design

Model Driven Engineering (**MDE**) [Sch06] is a promising design methodology, which shifts designer work from code implementation to system modeling. A model allows designers to better cope with complexity. It represents a system through domain-specific concepts and their relationships. One of the most popular MDE approaches is *Model-Driven Development*™(**MDD**™) and *Model Driven Architecture* ®(**MDA**®) [MSUW04], which are OMG trademarks. These initiatives are naturally based on the OMG UML modeling language. Another closely related terminologies for MDE are *Model-Based Design* (**MBD**) and *Model-Integrated Computing* (**MIC**) [SK97].

The Unified Modeling Language (UML) has been used to specify embedded systems and notably hardware modules. UML provides a semi-formal graphical object-oriented specification language, which is intensively used in software engineering. A number of works [Dv04b] [SFB05] [RSB+06] have proposed UML to offer an *unified* specification language for hardware and software. The main challenge to target the hardware domain is to define the *mapping* between the high-level object-oriented concepts of UML and the low-level hardware constructs of HDLs such as VHDL. We consider that defining a mapping from an object-oriented IDL such as CORBA IDL to VHDL is similar to the definition of an UML-to-VHDL mapping from class diagrams. Indeed, UML-to-IDL mapping already enables UML modeling tools such as Rational [Rat] to generate IDL definition files from UML class diagrams. In the scope of this work, we focus on mapping object interface and structural specification, and not behavioral specification, that is why we present existing mappings from UML class diagrams and not from other diagrams such as state diagrams e.g as in [CT05] and [AHMMB07].

McUmber and Cheng

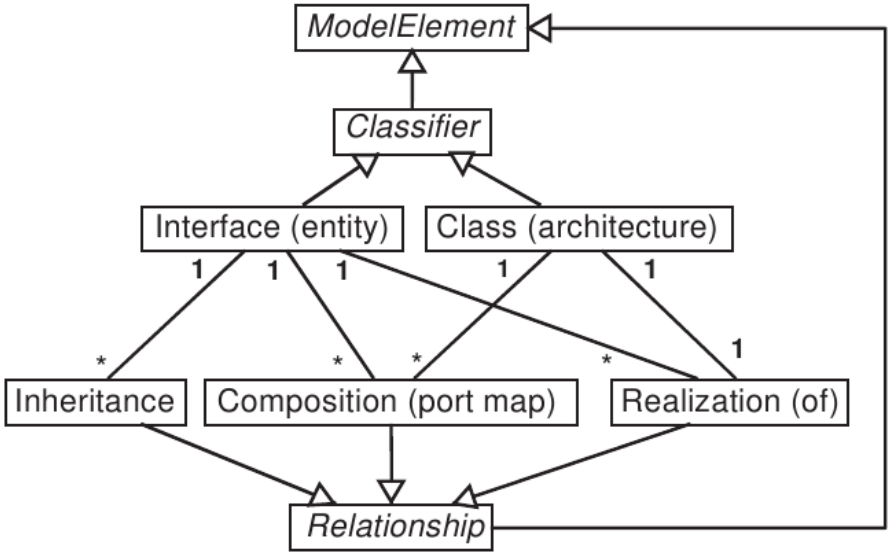
McUmber and Cheng [MC99] propose a UML-to-VHDL mapping to validate the behavior of embedded systems specified by UML class and state diagrams. The mapping rules from class diagrams to VHDL are the following. A UML class is mapped to a pair of VHDL entity/architecture constructs. Associations between UML classes i.e. message passing correspond to VHDL signal declarations in the entity. Class instance variables are represented with VHDL shared variables declared in the entity. A package and a package body are declared for each UML class.

This work follows an "hardware unaware" top-down approach from UML-to-VHDL and targets a simulable rather than a synthesizable specification in VHDL. For an efficient mapping, a "hardware aware" top-down approach is required. This implies that the mapping should be configurable and flexible enough to allow different trade-offs between area and performance and that the result of the mapping should be synthesizable.

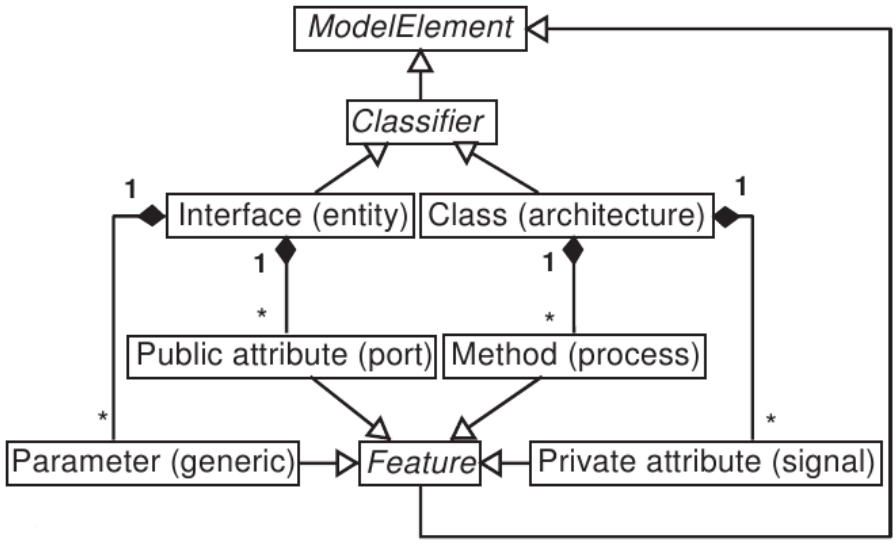
Damaševičius and Štuikys

Damaševičius and Štuikys present a mapping of VHDL language constructs on a simplified UML meta-model in [Dv04b] [Dv04a]. This UML meta-model contains the basic modeling elements of an UML

class diagrams notably classifiers, relationships and features [RIJ04] as depicted in figure 4.11.



(a) Relationships



(b) Features

Figure 4.11: Structural mapping between UML class diagrams and VHDL language constructs [Dv04a]

Kinds of classifiers that describe UML models include class and interface. As previously described in the object model, the relationships between classes include inheritance, composition and realization, as shown in figure 4.11a. The features depicted in figure 4.11b represent classifier properties such as operation, attribute or parameters. An *interface* i.e. an abstract class is mapped to a VHDL *entity*. Class template parameters are translated into VHDL *generic* statements. Public class attributes correspond to VHDL *port*s, while private class attributes are converted into VHDL signals. Class methods are mapped to VHDL *processes*. The inheritance relationship implies that the VHDL *port*s of the parent entity is added to the entity declaration of a derived entity. The composition relationship is translated to a *port*

map statement. The realization relationship that exists when a class implements an interface is mapped to a VHDL *architecture*.

Compared to the previous work, this approach follows a bottom-up approach, which targets the reuse of third-party IPs. As shown by the application of the wrapper design pattern presented in [Dv04b], the proposed mapping does not raise the abstraction level at which hardware modules are specified and only allows an UML *representation* of hardware modules. The mapping of public class attributes to VHDL ports implies that the same UML specification cannot be used independently of the implementation languages. Indeed, the UML specification inferred from VHDL modules provides a signal-based interface, which is incompatible with a traditional high-level method-based interface used in software. Moreover, the mapping of a method-based interface to a signal-based interface is inefficient and is incomplete as this mapping does not specify an invocation protocol, which is sadly lacking in hardware.

MOCCA from the University of Mittweida, Germany

The Embedded Control Laboratory (LEC) of the Institute of Automation Technology (IfA) at the University of Mittweida in Germany has developed a HW/SW co-design methodology for reconfigurable computing based on Model-Driven Architecture (MDA) and Platform-Based Design (PBD). A tool called **MOCCA** for *Model Compiler for reConfigurable Architectures* [Moc08] supports the object-oriented system-level specification, design and implementation of applications for Run-Time Reconfigurable (RTR) architectures [Frö06] [SFB05] [BFS04].

The proposed approach uses UML and a dedicated action language [MB02] called *MOCCA Action Language* (**MAL**) [SBF04] for object-oriented specification and design at system level. An action language is a platform independent and domain-specific language, which allows designers to specify behavior in UML. The objectives are to build executable specifications also known as eXecuTable UML (xUML or xtUML) [MB02] and to map such languages into platform-specific programming languages such as C, C++. MAL is an action language targeting distributed embedded systems and is compliant to the UML action semantics. Its syntax is inspired by Java and allows explicit data organization and access.

From the design, implementation and deployment models of applications and platforms, the MOCCA compiler supports design space exploration, estimation, model transformations and HW/SW implementation languages generation such as C++, Java, VHDL and partly Verilog. The main object-oriented concepts such as inheritance, polymorphism and encapsulation are supported down to the final implementation.

MDA based design methodology MOCCA design methodology is depicted in figure 4.12. Applications and target platforms are described by separate models with UML and MAL. An application is modeled as objects communicating through messages. The structure and behavior of these objects are captured in an executable and platform-independent design model (PIM). The target platform model (TPM) specifies the architecture of the target hardware and the necessary information for platform mapping and synthesis. The application objects of the PIM are mapped to the target architecture described by a TPM to build a Platform Specific Model (PSM). The PIM elements are bound to the target platform devices resources. The application PSM is then transformed into an implementation model, which is synthesized into software and hardware modules. This synthesis is parameterized by implementation language specific patterns and rules, which are included into the target platform models and the synthesis tools.

An UML component is implemented as a hardware component template depicted in figure 4.13. This template includes a memory-mapped communication interface to access internal hardware objects

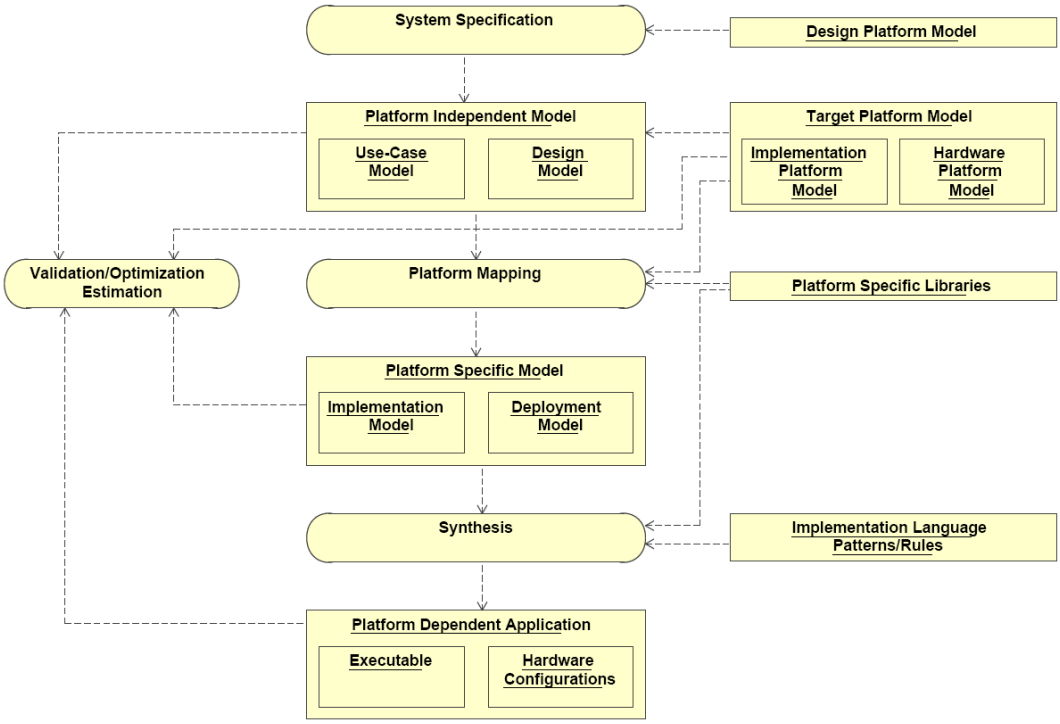


Figure 4.12: MOCCA Design Methodology [BFS04]

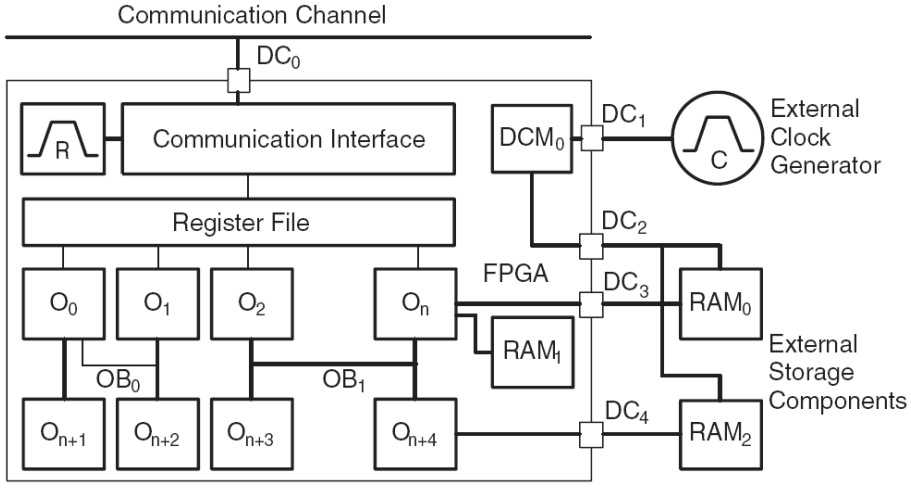


Figure 4.13: Hardware Component Template [SFB05]

through a register file and common building blocks to generate clock and reset signals. This communication interface is the external interface of each hardware component. Hardware objects may communicate through point-to-point links or via multiple single-master shared buses called *MOCCA Object Bus* (**MOB**) with a Wishbone-like interface within a group of collaborating objects.

Model compilers perform the memory address layout of the object interface register file according to given alignment constraints and automatically generate the appropriate address decoders.

Mapping of UML class The behavior of classes is implemented as a Finite State Machine with Datapath (FSMD). The implementation of class behavior contains a controller, a datapath and a synchronization process. For each hardware object, the state and behavior of the object class and its parent classes are replicated. This offers better performance at the expense of extra hardware resources.

The logical hardware object interface designates a memory-mapped HW/SW interface. It consists of a control interface, a data interface and an exception interface, which are physically implemented in a register file. Each HW method invocation interface includes the following signals:

- Control interface
  - the *GO* signal activates the execution of the FSM implementing the method behavior. If this signal is deasserted, the FSM state is reset to its initial state. Before activation, all input parameters have to be written in the register file through the data interface.
  - the *DONE* signal indicates the completion of the method execution when the FSM has reached its final state. It signals the availability of output parameters in the register file.
- Data interface
  - inout signals for attributes of scalar data types
  - MOB bus interface for attributes of array data types
- Exception interface
  - This interface is not described in details and its automated implementation was not supported in [Frö06].

The type register stores the current type of polymorphic objects and controls a multiplexer to select the appropriate method implementation. Object attributes and parameters are mapped to a storage component of the necessary width. Storage components are either individual registers or dedicated memory blocks and must be dual-ported.

Software proxy objects synchronize concurrent accesses to hardware objects by serializing them.

Application deployment on reconfigurable platform and SW-to-HW object communication A *RTR Manager* manages the dynamic creation and destruction of hardware objects deployed in reconfigurable hardware. It serves as hardware component factory and a hardware object broker. Each hardware object is accessed from software through a dedicated proxy, which encapsulates communication protocols. The proxy is explicitly modeled in the implementation platform model thanks to the *remote* type. An application requests a hardware object from the RTR Manager by its type. If no object of the required type was previously deployed, the RTR Manager loads the appropriate bitstream and returns a SW proxy to the application. If a reconfigurable fabrics (RF) is not mapped into the local address space of the host processor, the RTR-manager may handle data conversion, marshalling, and unmarshalling for remote communication [Frö06].

In conclusion, the MOCCA project comprehensively illustrates the application of the OMG MDA approach from object-oriented specifications of embedded systems down to the final implementation of HW/SW objects. The use of a domain-specific action language to describe application behavior allows a truly system-level executable specification independent from HW/SW partitioning, target platforms and final implementation languages. The implementation of state-of-the-art compilation and synthesis techniques [Frö06] allows MOCCA to generate optimized implementations in C++, Java and VHDL based on the precise information incorporated with application, platform and deployment models.

The proposed hardware invocation interface is similar to Barba's proposal and shares the same drawbacks. There is one control interface i.e. GO and DONE signals for each operation. This is not scalable with the number of operations. As opposed to previous proposals, the mapping of arrays is supported and may be mapped to a data transfer interface with control flow thanks to the ACK signal. The MOB is restricted to a simple shared bus interface. A socket based-approach would allow one to be independent of any bus/network protocol and topology, while offering advanced features such as request/response pipelining.

GASPARD2 from the University of Lille (LIFL), France

GASPARD2 (Graphical Array Specification for PARallel and Distributed computing) [WT08] is a model transformation and code generation tool for intensive and systematic multidimensional signal processing on MPSoCs. The targeted signal processing applications perform iterative, massively parallel and systematic computations. Gaspard2 uses the Array-OL (Array-Oriented Language) [GRE95] specification language, whose concepts are used in the application, architecture and mapping meta-models to describe repetitive structures such as array of application tasks, hardware modules (SIMD units, multi-bank memories, NoCs) and task-to-computation unit mapping. Array-OL allows one to specify both task and data parallelism within intensive signal processing applications.

GASPARD2 is based on a UML profile [ABC$^+$07] to model applications and MPSoC architectures by means of components and the mapping between both models. The Gaspard2 UML profile is a subset of the UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) [OMG08h]. Gaspard2 revisits the Y chart of Gajski and Kuhn [GK83], which is based on structural, functional and geometrical representations with an MDE approach based on application (PIM), architecture (PDM) and deployment (PSM) models. Gaspard2 uses a meta-model called *Deployed* that combines four meta-models: application, architecture, association and deployment meta-models depicted in figure 4.14.

The application, architecture and association meta-models are platform independent. The application meta-model supports the modeling of an application, while the architecture meta-model is used to model the physical components of a hardware platform. The association meta-model describes the mapping of tasks and data from the application model to the hardware components in the architecture model. The deployment meta-model defines the mapping of an application or architecture model elements on SW or HW modules both called *Intellectual Property* (**IP**) from a library. A Deployed model describes the modeling of an application, an architecture and their association model in compliance with the Deployed meta-model. Based on this model, the Gaspard2 tool can automatically generate application descriptions in synchronous languages such as Lustre or Signal, in procedural languages such as Fortran/OpenMP, in SystemC at PVT and CABA abstraction levels [Bon06] and in VHDL [Beu07].

Le Beux proposes a RTL meta-model independent from HDLs syntax [Beu07]. This meta-model allows one to describe HW components, their interface, their recursive and repetitive composition in a factorized form. The execution model of data parallelism within HW accelerators is either parallel or sequential depending on the mapping or "placement" of application components into a HW accelerator in the target architecture.

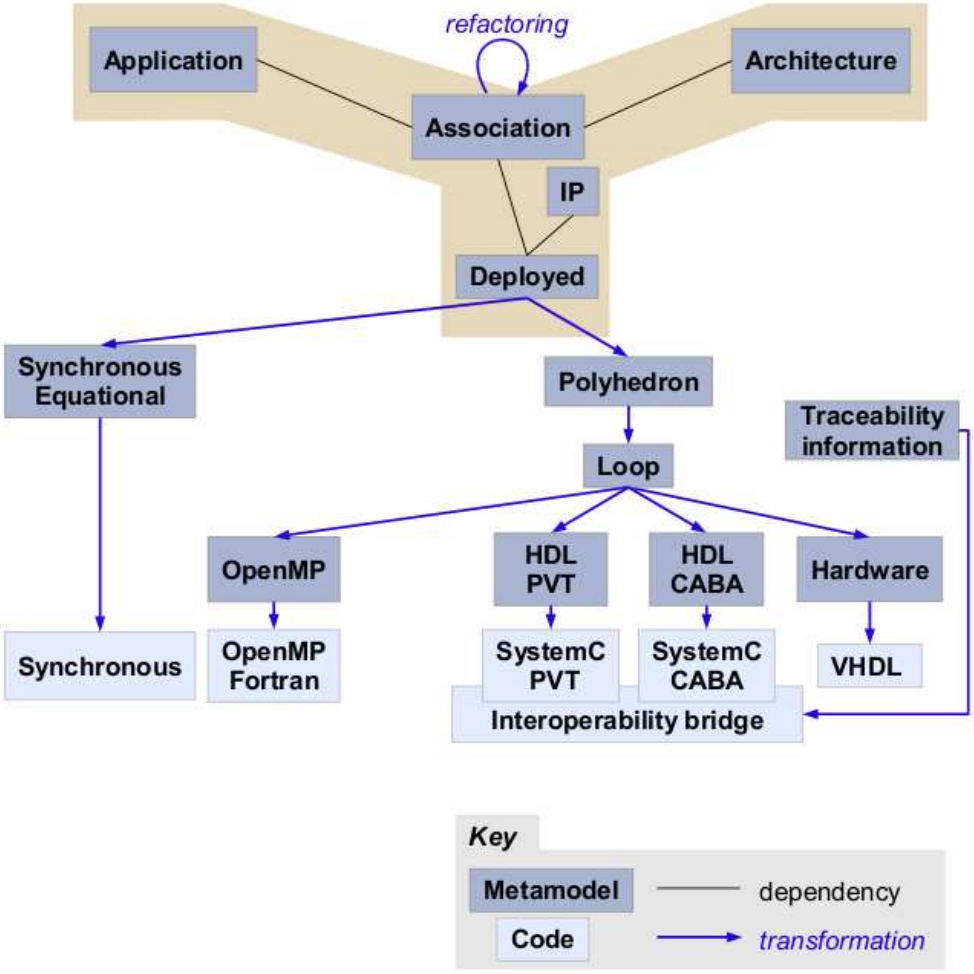


Figure 4.14: Gaspard2 MDE based design flow

In the proposed RTL meta-model, a HW ***Component*** is either ***Elementary*** or ***Hierarchical***. An elementary component called **ElementaryTask** denotes an atomic computation unit associated with an IP **Implementation**. A hierarchical component is either **Compound** to represent task parallelism or **Repetitive** to model data parallelism with a sequential or parallel execution. The concepts of *controller* and *(de)multiplexer* have been modeled to support the sequential and repetitive execution of HW accelerators. HW component interface in the RTL meta-model is represented by the concept of ***Port***, whose meta-model is depicted in figure 4.15.

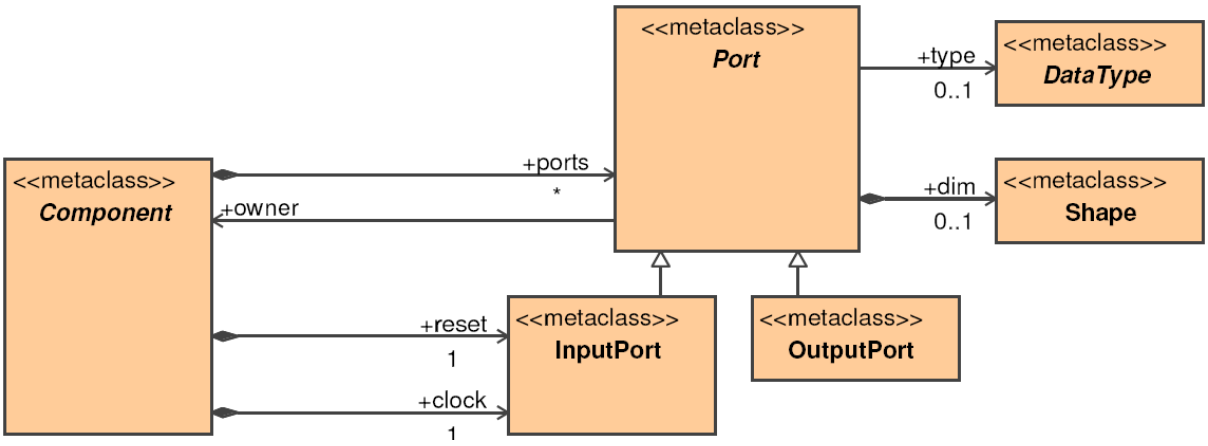


Figure 4.15: Component interface meta-model

A ***Port*** is either an **InputPort** or an **OutputPort**, whose dimension and size is defined by **Shape** and its type by **DataType**. Semantics is added to a port through a named composition relationship. For instance, each component consists of one **reset** port and one **clock** port. The concept of **Connector** represents the signals of data, control, clock, etc. that bind port instances , which are incarnated by the **PortInstance** concept. The RTL meta-model extends the **DataType** concept of Gaspard2 meta-models with a *nb_bits* attribute to estimate resources consumed by a data type.

Based on this RTL meta-model, a high-level application model compliant with the Deployed meta-model is translated into a RTL model through a model transformation called *Deployed2RTL*. Deployed2-RTL is composed of a set of transformation rules.

During the Deployed2RTL model transformation, ArrayOL tilers expliciting data parallelism are transformed into wires when there is no data dependency and into shift registers when data overlap in time.

As the RTL meta-model represents generic hardware concepts existing in HDLs, VHDL code is directly generated from the RTL meta-model by a model to text transformation called *RTL2VHDL*. Code generation is based on Eclipse *Java Emitter Templates* (**JET**) [Fou08]. JET allows one to write code templates in an implementation code e.g. Java or VHDL, which are customized by Java code.

In conclusion, the Gaspard2 tool supports a MDE approach in which model transformations allow one to refine a deployment model from an application model on an architecture model into an RTL model from which VHDL code can be automatically generated.

Due to the underlying ArrayOL MSDF model and intensive signal processing domain, only data paths are specified in the Gaspard2 meta-models. The RTL meta-model represents hardware component interface in a generic way with input and output ports. No semantics are attached to RTL ports apart from clock and reset ports introduced by the Deployed2RTL model transformation. The generated "business" interface of hardware accelerators only consists of data ports and does not require a communication

protocol. This data interface is the simplest member of the family of interfaces, which is required for the communication between HW components. Our goal is to be able to specify this interface family and not only simple data wire interface. Furthermore, HW/SW communication through some buses or NoCs is not supported in Gaspard2 as the application is entirely deployed either in SW or in HW.

Other approaches

Some approaches define a mapping from UML to C-like hardware description languages such as Handel-C in [SR04] or ImpulseC [PT05] in [WX07]. In our proposition, we do not target such proprietary languages and focus on the standard HDLs such as VHDL.

4.3 Synthesis

Tables 4.1 and 4.2 summarizes the state of the art concerning the application of the object-oriented concepts down to hardware. This table particularly shows the mapping space from object-oriented specifications into the interface and architecture of hardware objects. A flexible UML/IDL-to-VHDL mapping should be able to address all these mapping solutions with standardized hardware interfaces and architecture templates/patterns while only consuming the required hardware resources.

4.4 Conclusion

The object-oriented design methodology is a widely teached and used methodology to develop software applications. It is based on high-level concepts such as encapsulation, inheritance and polymorphism, and favors reuse. In the object-oriented approach, an object is a self-contained and identifiable entity, which encloses its own state and behavior in a method-based interface, and communicates only through message passing. A method describes the actions to perform depending on the type of message received by the object. A method invocation requests the execution of a method. Method invocation is equivalent to message passing. The local and implicit transfer of control and data of a method invocation can be formally translated into the remote and explicit transfer of a message over a network and vice versa. A client object is any object that requires the services provided by another object called the server object. Proxies or surrogates make local and remote communications indistinguishable from the viewpoint of client and server objects by translating local method invocations into request-reply messages. A client-side proxy called **stub** encodes the input parameters of a remote method invocation into a request message and conversely decodes a reply message into the output parameters and the return value. A server-side proxy called **skeleton** decodes a request message into the input parameters of a local method invocation to the object implementation and conversely encodes the output parameters and the return value into a reply message payload.

Usually, the integration of hardware modules relies on low-level hardware/software interfaces, which consists of proprietary bus interfaces, memory-mapped registers and interrupts to synchronize hardware/-software modules. As a result, this integration is tedious and error-prone, and the reuse of HW/SW modules is limited. Co-design methodologies have been developed to bridge the conceptual gap between hardware design and software design. To provide a common hardware/software design approach, the concepts of the object model has been applied to hardware design through four main directions: object-oriented extensions to hardware description languages such as OO-VHDL, object-oriented languages for hardware design such as Java and SystemC, object-oriented hardware architecture with hardware objects, and model-driven engineering for hardware design using UML.

University/Company	Tübingen U.	Oldenburg U.-OFFIS	UCLM
Project	OASE	ODETTE - ICODES, ANDRES	OOCE
Language	UML, Java, SystemC, C++, e	Objective VHDL, SystemC (SystemC Plus, OSSS+R)	UML, ICE SLICE IDL
Attribute	local	local, registers or SRAMs	registers or SRAMs
Method	inlining, replication	sub-component, entity, impl. per object, FSM	N.C.
Polymorphism	method inlining, multiplexing	method replication and multiplexing	N.C.
Inheritance	yes	yes, expansion	N.C.
Composition	yes	yes	N.C.
Concurrency	active class, SW thread mapped on process, data dependency analysis	arbiter, scheduler with user defined policies, guard	N.C.
Distribution	stub/skeleton, delegation, inheritance based partitioning	no - RMI stub/skeleton	RMI stub/skeleton
Comm.	non-blocking, synchr.	request blocking, synchr	non-blocking, a/synchr
Op. Invocation	sequential, fixed, custom	sequential, fixed, custom	parallel, fixed, custom
Op. Execution	sequential?	sequential (FSM)	N.C.
Flow control	no	request	no
Local Transfer	parallel	parallel	parallel or serial
Message	N.D.	client_id, method_id	method_id mapped to a network address
Deployment	N.C.	N.C.	N.C.
Reconf.	N.C.	OSSS+R container	reconfiguration interface
Interface			
request	start	no	method
request accept	N.C.	DONE, guards	N.C.
method id	method	SELECT_METHOD - MID	no
in method param	param1...N	IN_PARAMS - bus	param1...N
invocation reply	ready	DONE - ?	method_done
out method param	return	OUT_PARAMS - bus	return_value

Table 4.1: Synthesis of hardware implementation of object concepts (part 1)

University/Company	**Sharif U.**	**ST**	**Prismtech**	**MOCCA**
Project	OO-ASIP	MultiFlex	ICO	MOCCA
Language	C++	ST System IDL (SIDL)	CORBA IDL	MDE: MOCCA UML2 profile, MOCCA Action Language (MAL) then C++, Java, VHDL and partly Verilog
Attribute	central data memory	local	local	local
Method	shared, SW instructions, HW entity, impl. per class	N.D.	N.D.	impl. per object
Polymorphism	tables then routing messages as NoC packets	NC	NC	method replication and multiplexing
Inheritance	yes	N.C.	N.C.	yes
Composition	N.C.	N.C.	N.C.	N.C.
Concurrency	inside method implementations	HW multithreading, Concurrency Engine, ORB FIFO scheduler	N.C.	active class, guarded operation.
Distribution	SW on ASIPs using NoC	HW message passing engine, HW custom ORB	HW CORBA ORB	memory mapped stub/skeleton
Comm.	synchr. ?	Both blocking and non-blocking	non-blocking	blocking
Op. Invocation	sequential, fixed, custom	N.D.	parallel?	parallel
Op. Execution	sequential	N.D.	can be parallel	parallel
Flow control	no	N.D.	no	no
Local Transfer	data bus	N.D.	parallel	parallel
Message	mid, oid, method impl. mapped on a network address	N.D.	GIOP	N.D.
Deployment	N.C.	N.C.	yes	yes
Reconf.	N.C.	N.C.	yes	yes
Interface				
request	command	N.D.	opName_write	GO_*opname*
request accept	N.C.	N.D.	N.C.	N.C.
method id	no	N.D.	no	no
in method param	param1...N, bus	N.D.	inParam1...N	param1...N or bus
invocation reply	status	N.D.	opName_read	DONE_*opname*
out method param	signals or bus	N.D.	outParam1...N	signals or bus

Table 4.2: Synthesis of hardware implementation of object concepts (part 2)

To apply the object model down to synthesizable hardware, the previous works propose a mapping between a method-based interface to a signal-based interface. Hardware object-oriented design allows to raise the abstraction level of hardware module interfaces, to reduce the conceptual and abstraction gap between interface specification and implementation, and to enforce a clean separation between computation and communication through remote method invocation.

However, these works suffer from some limitations notably due to the object model itself. The proposed hardware interfaces and protocols do not provide enough flexibility to address the complex communication requirements of modern IPs. Indeed, they provide a single hardware interface, which only supports sequential invocation and execution of methods. Each presented work proposes a mapping, which represents one solution in the mapping exploration space.

We propose to generalize and leverage the existing mapping propositions with an explicit and fine mapping configuration using a named family of hardware interfaces with well-defined semantics. Our mapping approach gives the user a level of control over the mapped hardware interface never reached by all other fixed mapping approaches and is similar to what is available for hardware designers in high-level synthesis e.g. from C-like languages. To go further to the object-oriented approach, we propose a component-oriented approach in which component ports may be independently mapped to hardware interfaces and which allows both sequential and concurrent invocations of methods. In the next chapter, we will present middlewares. In particular, we will focus on the application of the distributed object model in object-oriented middlewares like CORBA.

Chapter 5

Middlewares

Contents

This chapter is divided into three parts: first we present in a general way the definition, the main requirements and a classification of middlewares. We then focus on distributed object middlewares and in particular on the OMG CORBA middleware required by the SCA. Finally, we present a state of the art on software implementations of real-time and embedded CORBA middlewares and on hardware implementations of object middlewares including CORBA.

This chapter is organized as follows. Section 5.1 defines what are middlewares. Section 5.2 describes the main requirements of middlewares, portability and interoperability. Section 5.3 presents a middleware classification. Section 5.4 presents the Object Management Architecture (OMA) from the Object Management Group (OMG). Section 5.5 presents the CORBA Object Model and its software implementations notably for embedded and real-time systems. Section 5.6 describes a state of the art on hardware implementations of object-oriented middlewares including CORBA. Finally, section 5.7 concludes this chapter.

5.1 Middleware Definition

A **middleware platform** is a software communication infrastructure that is used for the development, deployment and execution of applications that are distributed across different computing environments ranging from enterprise to embedded computers. From an application designer viewpoint, a middleware is a software layer between applications, operating systems and network protocol stacks. It is basically implemented as a software library with which applications are linked. From an OSI model viewpoint, a middleware is traditionally a set of layers from the application layer down to the network layer. A middleware aims at supporting transparent communications between heterogeneous systems regardless of their distribution, hardware computing architecture, programming language, operating system, transport layer and data representation. The objective is to provide a uniform view of a distributed system to application developers.

A middleware is often referred to as a software or logical bus between communicating entities, although in reality a middleware instance typically resides at each communication node. A middleware raises the abstraction level at which distributed applications are developed. Developers no longer need to cope with low-level, tedious and error-prone aspects of distribution such as network programming e.g. using Internet sockets to locate objects, marshal application data, demultiplex and dispatch requests [SLM98]. This complexity is shifted from application to middleware designers. Developers can thus focus on business application logic.

5.2 Middleware Requirements

The two key requirements of middlewares are to provide *portability* and *interoperability* to distributed applications [PRP05].

5.2.1 Portability

Portability denotes the property of an application, which can be migrated or ported from one operating environment to another one with little or no efforts in terms of time and cost. Two kinds of portability can be distinguished: portability at *source code* level via *Application Programming Interface* (**API**) and portability at *binary code* level via *Application Binary Interface* (**ABI**) [PH07]. Concretely, an API is a set of operations which are directly called by developers according to some use rules. Examples of APIs include the Portable Operating System Interface (POSIX) for system programming and the Berkeley Socket API for network programming. An API is traditionally redefined for each specific procedural or object-oriented implementation language and operating environment. For instance, the Socket API has been redefined in C for Linux and Windows, in Java for Sun Java Virtual Machines (JVMs) and in .NET languages (C++.NET, VB.NET, C#) for Microsoft Virtual Machines called Common Language Runtime (CLR). The porting of an application from one tuple (language, operating system, processor) to another one may require its source code to be adapted to the target API interface and to be recompiled and linked to its implementation library.

In contrast, an ABI corresponds to the instructions set and calling convention of real and virtual machines. It is indirectly used by developers via the binary code executed by real machines from the compilation of native programming languages such as C/C++ or by virtual machines from the interpretation or run-time/design-time compilation of languages such as Java and Microsoft .NET languages. The porting of an application from one tuple (language, operating system, processor) may require no extra work as the binary code of the application compiled for one machine can be directly executed onto another machine without recompilation.

Middlewares support the portability of distributed applications through standard middleware APIs and abstraction of underlying language, operating system, processor and network. The middleware API is used by distributed applications and provided by each middleware platform implementation. It allows applications to communicate with each other and use middleware services e.g. to locate application entities. Each middleware platform like CORBA, Java RMI or .NET provides its unique, yet closely related, API in terms of functionalities [SSC98] [VKZ04].

A standard middleware API guarantees that the porting of a distributed application from one compliant middleware implementation e.g. the TAO CORBA middleware [SLM98] to another one e.g. the omniORB CORBA middleware [GLR06] only requires some minor source code modifications, recompilation and linking with the target middleware library.

In addition to using middleware APIs, distributed applications specify user-defined APIs. These application APIs are provided by distributed object implementations and required by client objects to request application services through the middleware platform. Whatever the implementation technology, middleware platforms allow client and server objects to be portable so long as the application interface on which they depend remains the same. An application API is either specified in a specific programming language like Java for Java RMI or .NET languages for .NET Remoting or an implementation-independent *Interface Definition Language* (**IDL**) such as CORBA IDL, Microsoft COM IDL and XML-based WSDL (Web Service Definition Language).

IDLs allow the specification of abstract APIs that are translated into concrete APIs. IDL compilers automatically perform this translation in target implementation languages according to language mappings. Developers no longer need to manually redefine APIs in each implementation language and operating environment. The automatic mapping of APIs guarantees a systematic consistency across languages and enforces common coding rules. Hence, the combination of IDLs and standard mapping rules enables the portability of APIs themselves regardless of their implementation and operating environment.

In summary, IDLs allow portability at source code level using API, while programming languages and virtual machines provide portability at binary code level using ABI. The main differences between

these interface and implementation-centric approaches is 1) an interface is less likely to change than its implementation and 2) the integration of legacy systems is achieved through interface adaptation rather than source code reimplementation [Vin03]. In this work, we follow an interface-centric approach using IDLs since this approach is technology independent and more applicable to hardware application modules i.e. IPs.

5.2.2 Interoperability

The second objective of middleware is interoperability. Interoperability is the ability of applications to interoperate (i.e. understand each other) and perform system functionalities although they are executed using different operating environments. Middlewares provide interoperability to distributed applications thanks to standard message protocols. A message protocol specifies the ordered set of messages, which are sent and received between middleware implementations through the communication network. These messages are formatted according to message building blocks called *Protocol Data Units* (**PDUs**). The associated message format defines a standard data structure with an ordered set of message fields for message header and payload along with standard data encoding and alignment rules. These encoding rules indicate how the common IDL data types defined and used by an application are encoded in a neutral data representation for network transmission in messages. Examples of message protocols include CORBA GIOP (General Inter-ORB Protocol), Sun JRMP (Java Remote Method Protocol) and SOAP (Simple Object Access Protocol) used in Web Services.

5.3 Middleware Classification

Middlewares may implement one or several distributed communication models. These communication models belong to different abstraction levels and are not obvious to classify and compare [EFGK03]. The main communication models include: Message Passing, Message Queuing, Remote Procedure Calls (RPC) and Remote Method Invocations (RMI), Publish-Subscribe, Transaction, Distributed Shared Memory (DSM) [Tho97] [EFGK03] [LvdADtH05].

Based on these communication paradigms, middlewares can be divided into the following main classes [Hur98] [Rit98] [Emm00]: Procedural middlewares, Object middlewares, Message Queuing middlewares, Messaging middlewares, Publish-Subscribe middlewares, Transaction middlewares, Database middlewares, Shared Spaces middlewares, and Web middlewares.

The separation between these different kinds of middlewares is rather fuzzy [Hur98]. Indeed, database middlewares may have object-orientation. Object middlewares like CORBA provide services for transaction, queuing, messaging and publish-subscribe. Finally, message queuing middlewares may also support transaction, DBMS and publish-subscribe. For instance, the Java Message Service (JMS) [Mic02] supports both message queuing and publish-subscribe models.

5.3.1 Message Passing

In Message Passing, a sender directly sends messages to a receiver using low-level send/receive primitives. Communications are mainly point-to-point and unidirectional from the sender to the receiver. Senders may send messages asynchronously and continue their processing, while receivers wait for the messages and receive them synchronously. The send/receive operations may be blocking, non-blocking or a combination of both. Higher-level interactions can be built on top of these primitives like remote invocations. Senders and receivers are time and space coupled as they need to be running at the same time and the sender needs to know the receiver address. Message passing does not generally provide access and location transparency as data marshalling and addressing are explicit at the application level.

Examples of message passing API are Internet sockets and Message Passing Interface (MPI) [For08]. The socket API explicitly requires the network address endpoint (IP address and port number). Read-/write operations include as parameters a data buffer and its size. Sockets may also be used for group communication. In MPI, instead of using an interface definition language, developers explicitly describe the data types of user-defined data structures programmatically at low-level in order that the MPI runtime library automatically performs data alignment (MPI_GET_ADDRESS) and marshalling.

5.3.2 Procedural middlewares

Procedural middlewares rely on Remote Procedure Calls (RPCs) [BN84]. Examples include Distributed Computing Environment (DCE) [14] RPC from the Open Software Foundation (OSF), the Open Network Computing (ONC) [15] RPC from Sun Microsystems, and Microsoft RPC (MSRPC) [16]. A distributed entity is viewed as a loosely cohesive set of user-defined procedures typically in C. A caller acting as client calls an user-defined callee procedure at server side. The signatures of remotely accessible procedures are defined in a language-independent Interface Definition Language (IDL) such as DCE/RPC IDL or Microsoft IDL. The call statement is exactly the same regardless of the local or remote location of the server. This location transparency is allowed by a proxy representing the server called stub that performs data marshalling and unmarshalling to and from a neutral network data representation. The marshalling and unmarshalling routines are automatically generated by an IDL compiler and provides access transparency to client and server entities. The IDL compiler performs the mapping or binding of language-independent interfaces defined in IDL to language-specific procedures e.g. in C. When the client performs a remote call, the input procedure parameters are marshalled by a stub into a byte stream, which is sent in a request message to the server using network protocols such as TCP or UDP. Then the message is received by the server, the parameters are unmarshalled and the procedure is called. Finally the results of the remote procedure are marshalled into a reply message and returned to the stub, which unmarhalls the return value and delivers it to the local procedure. In synchronous RPCs, the caller thread of control is blocked until a reply is returned, while asynchronous RPCs do not block the caller [ATK92]. RPCs are space and time coupled with partial synchronization decoupling [LvdADtH05]. RPC request is a blocking send and non-blocking receive, while RPC reply is a non-blocking send and blocking receive. Asynchronous RPCs allow a caller to initiate an invocation that immediately returns and continues its processing, whereas the invocation is concurrently performed. Caller execution and callee invocation are thus decoupled, while the latency of communication and callee computation are masked. Asynchronous RPCs may be without a reply (unreliable oneway called fire-and-forget [VKZ04]) or with a reply (twoway). A twoway synchronous invocation can be transformed into two request-reply asynchronous invocations for the request and the reply. Communications between the caller and the callee are primarily point-to-point, while group communications could be built on top of the basic RPC mechanism.

5.3.3 Distributed Object Middlewares

Distributed Object Middlewares, also known as Object-Oriented Middlewares, are based on Remote Method Invocation (RMI). Examples include OMG CORBA [OMG08b], Sun Java RMI [Mic08], ZeroC ICE [HM07], Microsoft *Distributed Component Object Model* (DCOM) [17] and Microsoft .NET Remoting [18]. RMI is to object-oriented programming languages what RPC is to procedural languages. RPCs

[14] http://www.opengroup.org/dce
[15] http://www.ietf.org/rfc/rfc1831.txt
[16] http://msdn.microsoft.com/en-us/library/ms691207(VS.85).aspx
[17] http://www.microsoft.com/com/default.mspx
[18] http://msdn.microsoft.com/en-us/library/72x4h507(VS.85).aspx

have been basically applied to object-oriented languages and extended with object-oriented concepts such as interface, object references and exceptions to build RMI. Object reference can be passed by value and used to call remote operations. The object paradigm is used as the primary abstraction in RMI to encapsulate the state and behavior of a distributed entity. A distributed object represents typically an atomic distribution unit. After obtaining an object reference typically from a naming service, a client object invokes user-defined remote methods on a server object as if it were local. As RPC, an IDL such as CORBA IDL or Java for Java RMI is used to describe the interfaces provided by objects, but are more naturally mapped to the native concept of interfaces in object-oriented languages. The automatic generation of client and server stubs is also supported by IDL compilers to marshal method parameters into requests and replies. Object Request Brokers (ORB) convey and dispatch requests from client to server objects and return replies in the opposite direction. RMI may be synchronous or asynchronous e.g. using CORBA reliable oneway operations and Asynchronous Method Invocation (AMI) with explicit polling or callback model. Both RPC and RMI are based on a request-reply message protocol, which is inherently asynchronous and characteristic of client-server interaction. However, request-reply messages are more implicit than in message passing and message-oriented middleware. As opposed to RPCs, Java RMI is space decoupled, but also time coupled with partial synchronization decoupling [LvdADtH05]. Java RMI request is blocking send and non-blocking receive, while Java RMI reply is non-blocking send and blocking receive. CORBA is time coupled, space coupled or decoupled with blocking send and non-blocking receive [LvdADtH05]. Based on the basic mechanism of RMI, Distributed Object Middlewares can provide the same functionalities as RPCs, Transaction middlewares and Message-Oriented Middlewares and replace them [Pin04].

5.3.4 Component middlewares

Component middlewares [SDG+07] are an extension to distributed object middlewares. A distributed entity is no longer a single object, but an aggregation of objects with explicit dependencies called a component. Examples are as OMG *CORBA Component Model* **CCM** [OMG06a], Sun *Enterprise JavaBeans* (**EJB**) [Mic06], Microsoft *Distributed Component Object Model* **DCOM** [19]. The concept of component will be studied deeper in the next section.

5.3.5 Message-Oriented Middlewares

Message-Oriented Middlewares (MOM) include message passing, Message Queuing [Kra08] and Publish-Subscribe middlewares. Many MOM supports both Message Queuing and Publish-Subscribe models like the Java Message Service (JMS).

Message Queuing (MQ) middlewares provide a fixed set of predefined asynchronous messaging primitives to send and receive messages anonymously from one sender to one or multiple receivers through indirect, explicit and named communication channels called *message queues*. A message queue is basically a FIFO message buffer and is managed by a queue manager. The sender send a request message to a message queue, then the receiver receives the request message from the queue and may return a reply message via another queue. Typically, a message consists of a header including control and routing information and a payload with application-specific data. Message Queuing may provide Quality of Services such as reliable and ordered message delivery. Message Queuing systems may store/forward and route messages. Message queues may be persistent for fault-tolerance or mobile receivers, which are sometimes disconnected from the network. They may be also transactional to guarantee atomic i.e. all or nothing delivery of messages to multiple receivers. Messages can be reordered by priority or deadlines or converted from one sender native format to another receiver format. Compared to RPCs, messaging

[19] http://technet.microsoft.com/en-us/library/cc722927.aspx

systems support more network protocols [Emm00]. Message Queuing provides location transparency as senders and receivers do not know the physical address of each other. Access transparency is limited since message queues are used for remote (but not for local) communications and marshalling is not automated, but handled by application developers [Emm00]. Messaging products do not guaranty portability and interoperability of applications due to the lack of platform-independent standards. Indeed, JMS appears as a de-facto standard API, but only for Java platforms.

Examples of MOM include Java Message Service (JMS) [Mic02], IBM WebSphere MQ[20], Microsoft Message Queuing (MSMQ)[21], and Apache ActiveMQ [22]

Message-Oriented Middlewares such as JMS, MSMQ, WebSphere-MQ are time and space decoupled with blocking send and blocking receive (pull) or non-blocking receive (notification) for synchronization decoupling [LvdADtH05]. Indeed, senders and receivers do not know each other and can use the message queues at different time. Group communication from one sender to several receivers may be supported using multicast delivery. Besides, the concept of message queues is also used for inter-process, task and thread communication in RTOS such as VxWorks and μC/OS. The SCA MHAL [JTR07] specification defines a standard MOM API and message header for GPP, DSP and FPGAs/ASICs. Hence, the MHAL can be considered as a hardware/software message broker for SDR platforms. The MHAL Logical Destination (LD) serves as Message Endpoint [BHS07].

5.3.6 Publish-Subscribe middlewares

In Publish-Subscribe middlewares, message receivers called *subscribers* register their interest in the middleware to receive a particular kind of information typically called *event*. When a message producer called *publisher* generates this information, the middleware notifies i.e. deliver the desired event to all registered subscribers. Examples of such middlewares include OMG Data-Distribution Service for Real-Time Systems (DDS) [OMG07a], TIBCO Rendezvous, Tuxedo, Apache ActiveMQ , CORBA Event and Notification services and JMS. Publication and subscription may be based on the topic, content or type of events [EFGK03]. Publish/subscribe messaging allows one-to-many, non-blocking and anonymous communications between a publisher and its subscribers [McH07]. Publish-Subscribe provides thus decoupling in terms of time, space and synchronization [EFGK03]. Publish-Subscribe supports location transparency as senders and receivers do not know the physical address of each others. For instance, DDS is a topic and content-based data-centric publish-subscribe middleware.

5.3.7 Transaction middleware

Transaction middleware or Transaction Processing (TP) Monitors are based on the concept of transaction. A transaction is a sequence of operations, which is either totally executed or not at all. It represents a single logical operation and unit of distributed processing. If an operation in a transaction fails, all the previous operations are cancelled or *rolled backed* using transaction logs. Otherwise, the transaction is totally executed or *commited*. A widely cited example of transaction is a money transfer operation, which requires two operations i.e. to withdraw money from one account and deposit it to another [CDK01]. Examples of transaction middlewares include IBM CICS [23] (Customer Information Control System), Oracle Tuxedo [24], CORBA Object Transaction Service (OTS), Java Transaction API (JTA) and X/Open Transaction Architecture (XA).Transaction provides a powerful abstraction to manage concur-

[20] www.ibm.com/software/mqseries
[21] http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx
[22] http://activemq.apache.org
[23] www.ibm.com/cics
[24] http://www.oracle.com/products/middleware/tuxedo/index.html

rent accesses to shared resources such as data from multiple distributed clients. Transaction middlewares support both synchronous and asynchronous interactions, fail over and recovery capabilities that provide fault-tolerant and reliability, load balancing and replication. However, they do not automate marshalling and unmarshalling of complex data structures [Emm00].

The fundamental properties of reliable transactions are contained in the "ACID" mnemonic: Atomicity (transactions are indivisible - all or nothing), Consistency (the state of a system remains consistent before and after a transaction), Isolation (transactions are independent from each other) and Durability (the commitment of a transaction is persistent and resistant to failures). The concept of atomic transaction has been used in database systems, distributed systems, file systems like Transactional NTFS, Software Transactional Memory (STM) [ST95] and in BlueSpec for High-Level Synthesis (HLS) [CM08]. Note that the term transaction aforementioned has nothing to do with "Transaction Level Modeling" (TLM) [Ghe06], which will be described later in this work. The link between both terms is studied in [NH07].

5.3.8 Database Middleware

Database Middleware mediate access to Database Management Systems (DBMS) like Microsoft Access, Oracle databases or MySQL using standard APIs like the Open Database Connectivity (ODBC), which is independent of programming languages, operating systems and DBMS. SQL (Structured Query Language) is a declarative query language with procedural programming extensions that is used to create, insert, update, retrieve and delete data in relational DBMS.

5.3.9 Distributed Shared Memory

Distributed Shared Memory (DSM) provides a global logical address space, which is shared by producers and consumers in distributed address spaces.An example is tuple spaces. Tuple spaces can be viewed as software Distributed Shared Memory, Blackboard [BHS07], content-addressable memory or associative memory. In mathematics, a n-tuple is a bounded sequence i.e. ordered list of n elements, which may be of various types and present multiple times. A tuple space is a repository of tuples, which are accessed concurrently by producers and consumers working in parallel. Producers write data as tuples to the tuple space, while consumers read or take (i.e. read and remove) data from the tuple space according to some search criteria e.g. content or type. Tuple spaces are time and space decoupled with total or partial synchronization coupling thanks to asynchronous notifications [LvdADtH05]. Indeed, producers and consumers are anonymous and do not know each other. The tuple spaces paradigm originates from early works on the Linda coordination language [GC92]..

5.3.10 Web middlewares

Web middlewares are based on Internet technologies and protocols such as XML and HTTP to offer an uniform view of Web applications independently of languages, operating systems or databases. Examples are Web Services, Web portals and application servers. Web Services are based on XML for both service definition using the Web Service Definition Language (WSDL) and message protocol using the Simple Object Access Protocol (SOAP).

Web Services are application services, which are remotely accessible using Internet technologies and protocols. Web Services target the integration of distributed enterprise applications that may rely on heterogeneous middleware platforms such as RPC, CORBA, Java RMI, .NET, JMS or MQSeries to build an independent Service-Oriented Architecture (SOA).

Web Services use XML for both service interface definition and message protocol using respectively WSDL and SOAP. The eXtensible Markup Language (XML) allows storing any data in .xml files under

Middleware Type	Communication Model	Participation	Space Decoupling	Time Decoupling	Synchronization Decoupling
Message Passing	oneway messaging	1-1, 1-N	No	No	from blocking to non-blocking
RPC/RMI	Request/reply	1-1	No	No	partial
Async. RPC/RMI	Request/reply	1-1	No	No	Yes
Notification	Event	1-N	No	No	Yes
Database	distributed shared memory	N-M	Yes	Yes	Yes
Message queuing	Message queuing	N-M	Yes	Yes	Producer-side
Tuple spaces	distributed shared memory	N-M	Yes	Yes	Producer-side
Publish/subscribe	event	N-M	Yes	Yes	Yes

Table 5.1: Comparison of middleware communication models [Tho97] [EFGK03] [LvdADtH05]

a structured form using markups. The functionalities provided by a Web Service are specified in the Web Services Definition Language (WSDL). However, no APIs for Web Services are standardized.

5.3.11 Position

In this work, our target application domain is SCA-compliant Software Defined Radio based on real-time and embedded radio platforms. Communications are primarily point-to-point between application objects and platforms or services objects.

We will first focus on object-oriented middlewares, then on component middlewares using CORBA middleware platform as example.

Table 5.2 presents a comparison of some middleware platforms.

After this general introduction to middlewares, we focus on the particular CORBA middleware platform to provide more details on typical middleware architecture and services that collaborate to offer portability, interoperability and distribution transparency to distributed objects. To give an overview of middleware architectures, we will briefly describe the main elements and services of the Common Object Request Broker Architecture (CORBA) that collaborates to offer portability, interoperability and transparency of access, location and technology to distributed objects.

5.4 OMG Object Management Architecture (OMA)

The *Common Object Request Broker Architecture* (**CORBA**) is a freely available specification published by the *Object Management Group* (**OMG**).

CORBA is the specification of a concrete middleware platform based on an abstract **Core Object Model** and reference architecture called the *Object Management Architecture* (**OMA**) [OMG95] depicted in figure 5.1. The abstract Core Object Model cannot be directly implemented as nothing is specified in terms of data types, signature and semantics of operations provided by the object interface, invocation message format etc. as opposed to the concrete CORBA object model. The OMA captures the conceptual OMG vision on object technologies. This vision evolves to Model Driven Architecture (MDA) with the multiplication of middleware technologies such as CCM, .NET, J2EE and WebServices and the growing need for higher level of abstraction and reuse through modeling e.g. using UML.

Criteria	CORBA	Java RMI	.NET	Web Services
Interface Definition	IDL	Java	.NET languages	WSDL
Interface	language mappings, CORBA API	Java RMI API	.NET Remoting API	*not supported*
Representation	CDR - specific	Object Serialization	Object Serialization (Binary Format or SOAP mapping)	XML serialization
Message	GIOP - specific	GIOP, JRMP	.NET Remoting Core Protocol	SOAP, GIOP
Transport	TCP, UDP - ATM, VME, SHM - CAN	TCP/UDP	TCP, HTTP, SOAP on TCP or HTTP	HTTP, SMTP, FTP, JMS, TCP (IIOP)

Table 5.2: Comparison of some middleware platforms

The OMA is structured around:

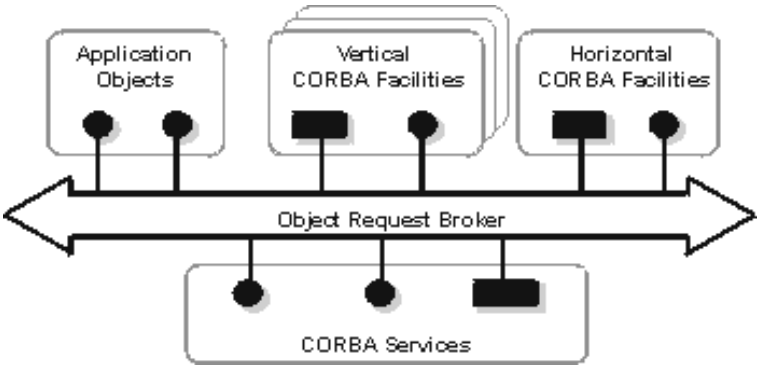


Figure 5.1: Object Management Architecture (OMA) Reference Architecture

- an **Object Request Broker** (**ORB**), which provides a communication bus between distributed objects. It allows transparent communications independently of the location and implementations of objects. These communications take the form of method invocations on objects. An ORB isolates objects from heterogeneity of programming languages, operating systems, hardware platform and networks;

- **Object services** denote a set of common system services independent of any application domain such as naming, life cycle, event services. The services are accessed through standard interfaces by application objects;

- **Common facilities** or Horizontal Facilities represent services used by different application domains such as printing service;

- **Domain services** or Vertical Facilities designate domain specific services e.g. medical, telecommunications and finance;

- **Application objects** provide application-specific services for which no standard interfaces are specified by the OMG as opposed to the previous ones.

5.5 CORBA Object Model

The CORBA object model is a concrete object model derived from the abstract Core Object Model defined in the Object Management Architecture. However, this object model does not specify implementation details such as if one or more threads should be used in service user (i.e. client), or service provider (i.e. server). An object is defined as "an identifiable, encapsulated entity that provides one or more services that can be requested by a client" [OMG08b]. Service users are independent from the implementation of services through a well-defined interface. An interface is a set of operations that a client can invoke on an object. An object is identified by one or several *object references* used to request a service. An object reference designates an *object type*, which is represented by an *interface type*. The CORBA object model is a typical object model, in which a client sends request messages to an object implementation, which may return a reply message. A client request consists of an operation, a target object, zero or more parameters and an optional request context. A parameter may be an input, output or inout i.e. input-output parameter. A method designates the code to be executed to perform a service. The execution of a method is referred as a *method activation*. When a client sends a request, a method is selected according to the operation identifier either by the object, but more generally by the ORB. After method dispatch, a method is called on the target object. Operation results, if any, are sent back to the client by a *return*, output or inout parameter. If a problem occurs during operation execution, an exception may be returned with exception-specific parameters.

5.5.1 CORBA Overview

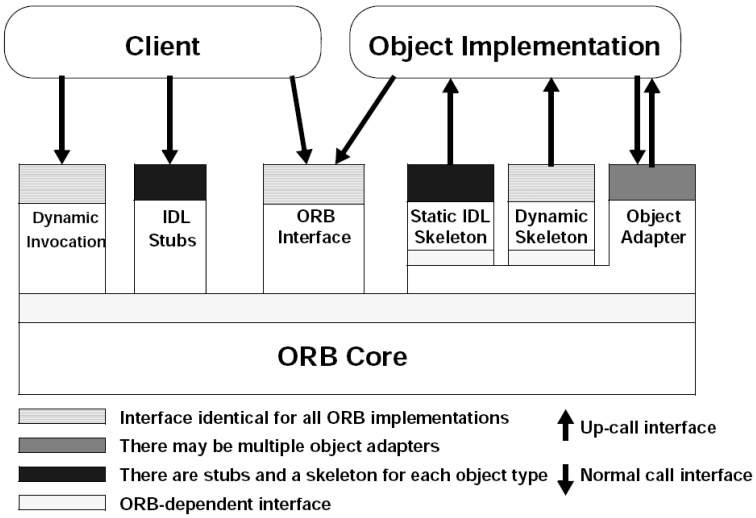


Figure 5.2: CORBA Overview

In CORBA terminology, a server is a process that includes object implementations called *servant*s, whereas a client is a process that calls object operations [McH07]. Before any communication with an object through method invocations, a client must retrieve their object references.

Object Reference

Each CORBA **Object** is represented by a universally unique identifier called *Interoperable Object Reference* (**IOR**) in CORBA and a servant. The IOR is indistinguishably used by clients for both local and remote access to servants. This provides access transparency, since object methods are invoked in the same way regardless of object locations. An IOR contains an IDL interface identifier called *RepositoryId*, an addressing information and an Object Key. An ObjectKey identifies an object within an ORB. An Object Key includes the Object Identifier (ObjectId) and an unique identifier for the POA of the object. An ObjectId is unique within an object adapter and is assigned by an object adapter or the designer. An ObjectId maps an object to its servant implementation(s). An object can be associated with one or several servants over the duration of its existence. A client can obtain an IOR though various means: a file, an URL [25] address called *corbalocation* (*corbaloc*) or a lookup service called *Naming Service*. An IOR corresponds to the *Absolute Object Reference* pattern [VKZ04]. The association between a CORBA Object and a Servant is called *Incarnation*, whereas the de-association is called *Etherealization*.

Remote Object Implementation

A **Servant** implements the operations of IDL interfaces in a target programming language. In OO languages such as C++ and Java, servants are implemented as classes. In procedural languages like C, they are implemented as a collection of functions and data structures. A Servant is activated by an object adapter or through a **Servant manager**. A Servant is referred as a *Remote Object* in [VKZ04].

Client-side invocation interface adapter

A **Stub** adapts the user-defined interface used by a client and the *Static Invocation Interface* (**SSI**) of the ORB. It acts as *Client Proxy* [VKZ04] to the servant. A stub encodes or marshals the input parameters of the operation into a request message payload according to common data encoding rules. It then deserializes or unpacks the reply message to present operation results to the client. Hence, it plays the role of a *Marshaller* [VKZ04]. A stub is also considered as an *Invocation Interceptor* [VKZ04]. A dynamic stub allows one to send CORBA requests and receive CORBA responses to an object unknown at compile-time thanks to a reflection service called *InterFace Repository* (**IFR**).

Server-side invocation interface adapter

A **Skeleton** is a glue between the user-defined interface implemented by a servant and the generic interface provided by an ORB. It demarshals binary request packets into native data types and calls the operation on the servant as an *Invoker* [VKZ04]. Then it packs results and errors or exceptions in a response returned to the client. A skeleton is also considered as an *Invocation Interceptor* [VKZ04]. A dynamic skeleton allows one to receive CORBA requests and send CORBA responses at run-time by dynamically discovering the IDL of an object through the Interface Repository.

IDL Compiler

An **IDL compiler** generates automatically stubs and skeletons from the interface definitions in IDL according to standard language mappings. An IDL compiler offers the portability of IDL interfaces, since client and server-side applications can be independently written in any language supported by language mappings. IDL compilers automate the tedious and error-prone tasks of marshalling/demarshalling application data to and from a binary stream.

[25] Uniform Resource Locator

Object Adapter

A *Portable Object Adapter* (**POA**) is responsible for generating the Object Key, associating objects and servants, demultiplexing and dispatching invocations to the right servant using skeletons. It conforms to the Object Adapter design pattern [GHJV95]. Typically, a POA maps Object IDs to servants via an *Active Object Map* (**AOM**). The association between an object to an implementation may be one-to-one, one-to-many for load balancing or fault-tolerance through replication, or many-to-one to shared resource. This latter mapping is referred to the *default servant*. POAs are organized as a hierarchy from the top-level default **RootPOA** to several levels of user-defined child POAs.

ORB Core

An ORB core is in charge of delivering the client request to the target servant and return its reply if any through *Client Request Handler* and *Server Request Handler*. An **ORB** core is a *Local Object* [VKZ04] that implements the standard ORB interface. This interface notably serves to start/stop the ORB, to convert hexadecimal IOR to human readable strings and vice versa, and to build at run-time requests from a list of parameters for the Dynamic Invocation Interface (DII). An ORB core can be designed as a Micro-Kernel [BHS07] with well-defined interfaces to plug in ORB modules such as schedulers, invocation adapters, object adapters, *Protocol Plug-In*s [VKZ04] and transport modules [PRP05]. An ORB Core may implement several strategies for network event demultiplexing like *Reactor* [BHS07] and for concurrency such as Thread-per-Request, Thread-per-Connection, Thread-per-Object and Thread pool like Worker and *Leader-Followers* [BHS07] architectures [Sch98].

ORB message protocol

An abstract message protocol called **GIOP** defines the format of service request and response messages and how data types are encoded for network transmission using data encoding rules defined in the *Common Data Representation* (**CDR**). GIOP supports communication interoperability.

ORB transport protocol

ORB messages are transferred on a standard or proprietary transport layer such as TCP/UDP-IP, ATM or a CAN bus.

Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI)

The DII allows clients to build requests at run-time while they do not know object interfaces at compile-time. Conversely, the DSI allows an ORB to dispatch requests to servants, whose interfaces are unknown at compiler-time. Clients and servers do not need an a priori knowledge of the static or dynamic interfaces used at each side. This is totally transparent and provides access transparency.

Interface Repository (IFR)

The InterFace Repository is a reflection service, which allows one to dynamically discover the service definition of objects unknown at compile time. The IFR is used to dynamically create requests and invoke the associated methods through the DII.

The *Implementation Repository* (IMR)

The Implementation Repository maps an object reference to another one. The IMR is typically used to dynamically and transparently start servers or perform load balancing in case of failure thanks to the location forwarding capabilities of CORBA ORBs.

The OMG specifies a multitude of common object services for Event, Log, Security or Naming Service and domain specific services for Real-Time, QoS, Streams or Asynchronous Messaging. We only introduce the main services:

Naming Service

The Naming Service is a lookup service [BHS07] [VKZ04], which maps a hierarchical object name to an object reference. A server `bind`s a name to an object within a naming context to register the service it provides. Each client `resolve`s this name to retrieve a proxy for this object on which it invokes its methods. The **Lightweight Naming Service** specifies a PIM in UML and a PSM for the CORBA platform. It is limited to the most commonly used subset of the Naming Service.

Event Service

The Event Service is a type-based publish-subscribe service [EFGK03]. An event source or **supplier** signals the occurrence of an event to an *event channel*, which notifies all *registered* event sinks or **consumers**. The **Lightweight Event Service** specifies a PIM in UML and a PSM for the CORBA platform.

Notification Service

The Notification Service is a content-based publish-subscribe service [EFGK03], which overcomes the limitations of the Event Service. The Notification Service extends the Event Service interfaces with the capability to define structured event data, notifies not all registered but all interested consumers using *event filtering* and a filter constraint language, obtains the event types required by consumers through *event discovery*, retrieves the event types provided by suppliers through *event subscription* and configures the *QoS* of event delivery.

Invocation path

As a summary, we describe the different steps involved during an invocation. The client-side ORB builds the GIOP and request message headers, marshals operation input parameters in a request message body and sends the message across the network. Then the server-side ORB parses the message header, demarshals the parameters, finds the servant, invokes the method, marshals return and output parameters, builds the reply message and send it. Finally, the client-side ORB receives the reply message, parse the reply header, demarshals the parameters and returns them to the application.

5.5.2 CORBA Implementation

The CORBA specification defines a reference model and architecture, its IDL interfaces and functionalities. However, CORBA does not specify the implementation of these functionalities and their quality. There is no official reference implementation, although the TAO ORB [TAO08] can be considered as such an implementation.

Middleware architecture and services have been extensively formalized in the literature under the form of numerous *design patterns* [GHJV95] [VKZ04] [BHS07] [Kra08] [SC99]. Middleware design

patterns capture key elements and services of middlewares architecture. Instead of continuously reinventing the wheel, a design pattern names and describes a common solution to a recurring problem. Even if software design patterns deal with software architecture, some concepts behind design patterns are fundamentally technology-agnostic and have been applied to hardware design [DMv03] [RMBL05] [RMB+05].

In the following, an overview of middleware architecture and functionalities is presented from a CORBA viewpoint. The suggestive names of the underlying design patterns are indicated as an illustration.

The main design patterns used to implement middlewares include:

1. The PROXY or Surrogate pattern [GHJV95] allows one to give the illusion to a client object that it directly interacts with a server object, while it may be remote. A proxy encapsulates various services such as object location, un/marshalling, network transfers or access control [Sha86].

2. The ADAPTER or *Wrapper* pattern [GHJV95] allows one to adapt two incompatible interfaces, this is referred to an *interface adapter* or to an interface to an implementation, this is referred to an *object adapter*.

3. The BROKER pattern [GHJV95] is used to define the general architecture of an ORB. A broker provides a communication infrastructure that enables distributed objects to communicate transparently in a heterogeneous environment regardless of location or languages. It also provides decoupling between the applications and the middlewares themselves.

4. The LAYER pattern allows one to decouple middleware services, whose implementation may evolve without impacting each other.

5. The WRAPPER FACADE pattern encapsulates low-level services with portable object-oriented interfaces.

6. The REACTOR pattern serves to demultiplex network events such as incoming connections and data transfers that are triggered by multiple concurrent clients.

7. The PROACTOR pattern extends the Reactor pattern via efficient asynchronous I/Os, which permits concurrent request demultiplexing and processing.

8. The STRATEGY pattern is used to transparently change the implementation of middleware services via dynamic loading/unloading of libraries.

Obviously, these patterns are connoted by software abstractions like thread, network programming with connection-oriented transport layers like TCP-IP and typical software source of overheads such as dynamic memory allocation. Design patterns capture different solutions in middleware design space. A similar approach is required for the design of hardware middleware implementations. In this work, we will discuss the design trade-off of such implementations.

5.5.3 Illustration of CORBA Middleware Design Flow

To illustrate the development of distributed applications using the CORBA middleware, we present a small example based on the CORBA compliant MICO ORB [PRP05].

IDL Definition

First, an object interface is defined in CORBA IDL. As presented in listing 5.1, the interface `Example` contains a single operation called `send`. This operation takes as input parameter a short called `s` and a sequence of bytes called `o`.

```
interface Example {
  typedef sequence<octet> OctetSeq;
  void send(in short s, in OctetSeq o);
};
```

Listing 5.1: example.idl

Then, this IDL interface is translated by an IDL compiler in accordance to a language mapping. Here, we chose the C++ mapping. The IDL compiler generates an header file, `Example.h`, which contains the glue code (invocation adapters) for the ORB of this IDL compiler.

```
idl example.idl
```

The server and client part of the applications are then developed.

Server Implementation

The server application implements the IDL interface. As shown in listing 5.2, the servant class `Example_impl` inherits from the skeleton class called `POA_Example`. The `send` operation simply prints the parameters received from the client application.

```
1  #include "example.h"
   using namespace std;
   class Example_impl : virtual public POA_Example {
     public:
       void send(CORBA::Short s, const Example::OctetSeq& o) {
6        cout << "Received: \n0x" << hex << uppercase << s << endl;
         for (CORBA::ULong i = 0; i < o.length(); i++)
           cout << "o[" << i << "] = " << (int)o[i] <<  endl;
   }
};
11 int main (int argc, char *argv[]) {
   // Initialize the ORB
   CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
   // Obtain a reference to the RootPOA and its Manager
   CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
16 PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
   PortableServer::POAManager_var mgr = poa->the_POAManager();
   // Create an object Example
   Example_impl ex;
   // Activate the object
21 PortableServer::ObjectId_var oid = poa->activate_object (&ex);
   // Obtain the object reference
   CORBA::Object_var ref = poa->id_to_reference (oid.in());
   CORBA::String_var str = orb->object_to_string (ref.in());
   cout << str.in() << endl;
26 // Activate the POA and start serving requests
   mgr->activate ();
```

```
  orb->run();
  // Shutdown (never reached)
  poa->destroy (TRUE, TRUE);
  return 0;
}
```

Listing 5.2: Server Implementation

Client Implementation

As presented in listing 5.3, the client application invokes the `send` operation on the stub.

```
#include "example.h"
int main (int argc, char *argv[]) {
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
  CORBA::Object_var obj = orb->string_to_object (argv[1]);
  Example_var ex = Example::_narrow (obj);
  Example::OctetSeq o;
  o.length(3);
  o[0]=1;o[1]=2;o[2]=3;
  ex->send(0x1234, o);
  return 0;
}
```

Listing 5.3: Client Implementation

After compilation of the client and server applications, the server and client processes are launched e.g. in a shell. The server process waits on a network connection from the client application. The following command line parameter are specific to the MICO ORB.

```
./server -ORBNoResolve -ORBIIOPAddr inet:127.0.0.1:5000 -ORBNoCodeSets
```

The client application takes as parameter the object reference of the servant hosted in the server process.

```
client IOR:010000001000000049444c3a4578616d706c653a312e30000100000000
0000002c000000010100000a0000003132372e302e302e3100881314000002f31393
634392f313232383934373434302f5f30
```

Table 5.3 describes the content of an object reference.

5.5.4 Overview of CORBA Interface Definition Language (IDL)

In CORBA, an object interface is specified independently of any particular programming language thanks to the OMG IDL. An IDL interface defines the public *Application Programming Interface* (**API**) provided by an application object. The type of a CORBA object is called an interface, which is similar in concept to a C++ class or a Java interface. In contrast to imperative languages that describe a sequence of instructions to be executed, CORBA IDL is a declarative language only used to formally describe a set of operation signatures i.e. the set of messages that can be sent to an object. In CORBA IDL, an operation signature is typically defined as:

[oneway] <op_return> <op_name>(param1, ... paramL) [raises(except1,... exceptM)] [context(name1,...,... nameN)]

which consists of:

Index	Hex Data								Comments
0	01	00	00	00					Endianness: Little endian + Padding
4	10	00	00	00					Length of the *type_id* string = 16
8	49	44	4c	3a	45	78	61	6d	Repo Id: "IDL:Example:1.0\0"
16	70	6c	65	3a	31	2e	30	00	
24	01	00	00	00					A tagged profile
28	00	00	00	00					TAG_INTERNET_IOP
32	2c	00	00	00					Length profile_data = 44 octets
36	01	01	00	00					Little Endian + IIOP Version 1.0 + Padding
40	0a	00	00	00					Length host = 10
44	0a	31	32	37	2e	30	2e	30	"127.0.0.1\0"
50	2e	31	00						
54	88	13							TCP Port = 5000
52	14	00	00	00					Length object_key = 20 octets
56	2f	31	39	36	34	39	2f	31	object_key = /PID/timestamp/ObjectID
64	32	32	38	39	34	37	34	34	/19649/1228947440/_0
72	2f	5f	30						

Table 5.3: Example of IOR in little endian

- **Invocation semantics**: an operation is executed *exactly-once* when an operation returns normally, *at-most-once* when an exception is raised or *maybe* when the optional **oneway** keyword is used to denote *best-effort* or *Fire and Forget* [VKZ04] semantics. Best-effort semantics mean that the delivery of an operation call is not guaranteed. A best-effort operation is a request-only operation for which no results i.e. output/inout parameters and user-defined exceptions can be returned. The return value is thus declared as a *void* type. Oneway operations are often referred as non-blocking or "asynchronous" invocations, as the caller cannot synchronize with the operation completion. To avoid synchronization side effects, reliable oneway operations have been introduced in Real-Time CORBA by specifying four synchronization scopes: none, when the transport layer accepts the message, before and after invocation on the servant [OMG05b]. Operation invocations may raise user-defined exceptions or standard system exceptions.

- **Return type**: specifies the type of the return value. A *void* type indicates that no value is returned. A return result is equivalent to an output parameter.

- **Operation name**: defines an unique operation identifier within an interface.

- **Parameter list**: each operation parameter has a mode, a type and a name. The mode may be **in**, **out** or **inout** to respectively indicate whether the parameter is transferred from the client to the server, from the server to the client or in both directions.

- **Exception list**: the optional **raises** keyword is followed by a list of user-defined error types that can be returned to a client. An exception type may have exception-specific information defined as a data structure.

- **Context list**: the optional **context** keyword precedes a list of request context information. This feature is rarely used and won't be addressed in this work.

CORBA IDL is a specification language currently only employed to describe software interfaces. In this work, we investigate the use of CORBA IDL at system level independently of hardware or software implementations.

In the following, we briefly introduce the common object-oriented type system of CORBA IDL version 2.x. Words in bold denote IDL keywords. IDL provides basic and constructed types, whose syntax is closely inspired by C++. The basic data types of CORBA IDL are listed below. A constant data type is preceded by the `const` keyword.

- `boolean` is a data type taking TRUE or FALSE.
- `char` is a character encoded in ISO latin-1.
- `octet` is an opaque 8-bit data type, which is not encoded for network transmission.
- `short`, `long`, `long long` and their `unsigned` versions are respectively 16, 32 and 64-bit integer types;
- `fixed`-point decimal numbers;
- `float`, `double` and `long double` are respectively single-precision (32-bit), double-precision (64-bit) and double-extended IEEE floating-point types;
- `any`, which can contain any basic or constructed IDL data type. An `any` includes a value and a `TypeCode` to indicate the type of this value. A TypeCode is an enumeration (see below), which identifies an IDL data type like basic and constructed types, object references and interfaces. TypeCodes are for instance internally used by ORBs to pass by reference an object in operation calls.

The constructed data types of CORBA IDL are listed below.

- `enum`, which is an ordered set of identifiers like in C/C++
- `struct`, which is similar to a C/C++ struct or a Java class with public members
- `array`, which is a multi-dimensional fixed-size array of a single type like in C/C++. A `typedef` defines a new name or alias for an existing type. A `typedef` is often used for array and sequence
- `sequence`, which is a one-dimensional variable-length array of a single type similar to a vector in the C++ **STL** (*Standard Template Library*) or a list in Java. A sequence has either a maximum length fixed at compile time for bounded sequence or a dynamic length for unbounded sequence available at run-time.

```
typedef sequence<short>    ShortBSeq; // unbounded sequence
typedef sequence<short, 16>ShortUSeq; // bounded sequence
ShortUSeq.length();  // Return 16;
```

- `union`, which contains a value determined at runtime by the *discriminator* of a switch-case statement.
- `string`, which is a sequence of characters;

- `interface`, which is a set of operations that an object of this type should implement. An interface implies a pass-by-reference semantics. CORBA IDL supports multiple inheritance of `interface`s like in C++. IDL interfaces can be constrained by the *local* keyword. A `local` interface is implemented by local objects, which can only be used within a local node. Local objects correspond to usual objects in the target programming languages. It does not have an IOR reference, but it is locally identified in the target programming language e.g. as a C++ pointer or a Java reference. Invocations on local objects correspond to usual local invocations, which are not mediated by an ORB. Examples of CORBA entities defined as local interfaces include the ORB and the POA.

- `attribute`, which may be contained in an interface definition. An attribute is not translated in an implementation language as a class member, but as a pair of getter/setter operations. A `readonly` attribute is similar to a single get operation.

- `exception`, which is either an user-defined or system exception.

- `module`, which allows one to group together related user-defined data types like `interface`s and is similar to a namespace in C++.

The OMG has standardized an UML profile for CORBA [OMG08g], which allows developers to graphically represent user-defined IDL data types like interface, sequence and struct. This graphical representation can then be translated into textual IDL definitions.

As we target hardware, real-time and embedded systems, we do not consider in the following sophisticated IDL data types such as wide characters, wide character strings, value type and abstract interface. However, `string` and `any` are still considered for "backward compatibility" with existing specifications as they are used in SCA interfaces.

5.5.5 Portability using CORBA IDL mapping to implementation languages

IDL specifications can be automatically translated by *IDL compilers* into target implementation languages according to standard *language mappings*, which have been specified for various software programming languages such as C [OMG99], C++ [OMG08c], Java [OMG08d], Ada 95 [OMG01], COBOL, Lisp, Python and Smalltalk. The term *binding* is also sometimes employed as synonym for the term mapping.

A language mapping establishes a link between the abstract abstractions offered by CORBA IDL and the concrete abstractions found in implementation languages. A mapping between two languages addresses the intersection, not the union of the concepts, functionalities and data types of both languages and removes the disjoint characteristics [Vin03]. As Vinoski said, we consider that a 100 percent mapping is not desirable or even possible and should restrict to the language features really required. OMG language mappings have inherent limitations, but they are standardized and thus assure long-term support and no vendor lock-in for enterprise applications.

One important point about IDL language mappings is that they are not specified by formal mapping rules, but by textual descriptions in which each IDL concept is expressed in a target language concept along with some basic examples. One goal of this work is to investigate the specification of such a language mapping for hardware description languages such as VHDL and system-level languages such as SystemC.

IDL compilers generate for each IDL interface a pair of stub and skeleton proxies. These glue codes act both as invocation interface adapters and proxies. Both proxies give the illusion to the client and the server that they directly interact even when they may reside on different network nodes. The

stub presents to the client object the same user-defined interface as the server object, while the skeleton invokes uses by respectively presenting and calling the same user-defined interface. On the one hand, these glue codes adapt user-defined IDL interfaces e.g. with a *void add(in long a, inout long b, out long c)* operation with the generic and ORB-specific interface with an operation like *void invoke(in string operationName, in any inArgs, inout any inoutArgs, out any outArgs*. Another important point is that IDL language mappings only standardize glue code interfaces, but not their implementation. This allows ORB implementers to provide additional features like smart proxies as in TAO ORB or optimizations e.g. when the client and the server are colocalized in the same address space. Application developers have to implement their business logic within the code generated by the IDL compiler.

User-defined IDL interfaces implicitly inherit from the pseudo `Object` interface. An `Object` must provide a set of predefined operations, which are part of the CORBA Object model. The implementation of these operations is not provided by the application, but by the ORB or by the glue code generated by the IDL compilers. The `Object` interface does not represent an user object implementation, but its object reference.

Tables 5.4 and 5.5 summarize language mappings for C, C++, Java and Ada 95. Obviously, we do not aim at describing in detail these mappings, but only the interesting points for this work. We chose to present the mappings to the C language and its object-oriented extension, C++, since these languages are widely used in the embedded domain, whereas Java was designed as a portable object-oriented language from the outset. The mapping to Ada 95 is also presented as VHDL syntax was inspired from Ada syntax.

What is interesting to note from the table 5.4 is that each mapping is closely adapted to the syntax and semantics of the target language.

As the C and Ada languages are not object-oriented programming languages, IDL object-oriented constructs had to be "emulated" in both IDL language mappings. The lack of code modularization constructs like C++ namespace and class is overcome by naming conventions, in which IDL data types and operation names are respectively prefixed with *CORBA_* and the name of IDL modules and interfaces in C. An IDL operation is mapped to a C function using as arguments: an opaque data type called *CORBA_Object* similar to the self-object reference *this* in C++ and Java that includes the object state and identity, the operation parameters mapped to function arguments, an optional opaque data type called CORBA_Context that provides a request invocation context, and a partially opaque data type CORBA_Environment that allows developers to set and get an exception identifier when an exception is raised.

The C++ mapping is reputed to be complicated and to require an important learning curve [Hen04] [KKSC04] [SV01]. As the size of C++ data types are not standardized and are thus not portable, CORBA data types were defined in C++. The mapping of multiple inheritance in IDL and C++ is direct. As Java data types are portable, a direct mapping of type has been proposed between CORBA IDL and Java.

Each CORBA middleware implementation includes its own IDL compiler. As any compiler, the main objective of IDL compilers is to generate optimized proxy code in terms of footprint and performance. IDL compilers improve local communications between objects colocalized in the same address space thanks to *colocalization* proxies [PRP05] [SLM98]. First IDL compilers had an monolithic implementation with inextensible and hardwired IDL language mappings. Some works have been carried out to study the design and implementation of flexible IDL compilers such as Flick (*Flexible IDL Compiler Kit*) [ESSL99], IDLFlex [RSH01] and USC (Universal Stub Compiler) [OPM94]. Basically, there are two marshalling approaches for stub and skeleton proxies: compiled and interpreted approaches. In the compiled approach, the marshalling code is hardwired and dedicated to the user-data types. In the interpreted approach, the marshalling code dynamically encodes and decodes operation parameters based on the types of user-data that are either transferred with the messages or read from the IDL meta-data stored in the Interface Repository. A natural trade-off exists between the size and speed of marshalling code

IDL	C	C++
module M	M_ operation name prefix	namespace M
interface I	typedef CORBA_Object I and I_ operation name prefix	class I, smart pointer class I_var, skeleton class POA_I
const	#define	static const
short	CORBA_short	CORBA::Short
unsigned short	CORBA_unsigned_short	CORBA::UShort
long	CORBA_long	CORBA::Long
unsigned long	CORBA_unsigned_long	CORBA::ULong
long long	CORBA_long_long	CORBA::LongLong
unsigned long long	CORBA_unsigned_long_long	CORBA::ULongLong
float	CORBA_float	CORBA::Float
double	CORBA_double	CORBA::Double
long double	CORBA_long_double	CORBA::LongDouble
char	CORBA_char	CORBA::Char
wchar	CORBA_wchar	CORBA::WChar
boolean	CORBA_boolean	CORBA::Boolean
octet	CORBA_octet	CORBA::Octet
string	0-byte terminated character arrays	String_var class
enum E	#define	enum E
struct S	struct S	struct S
union U	struct with union	class U
array A	array A and A_slice	class A_var and A_slice
sequence S	struct S	class S
fixed<X,Y>	CORBA_fixed_X_Y struct	Fixed class or implementation-defined typedef
typedef X Y	typedef X Y	typedef X Y
any	CORBA_any struct	Any class
TypeCode	CORBA_TypeCode	TypeCode class
exception E	typedef struct E, #defined ex_E, CORBA_Environment with CORBA_exception_set	class E derived from CORBA::UserException class
operation	function with CORBA_Object, optional CORBA_Context, CORBA_Environment	member function
attribute T A	T _get_A(...) and _set_A(T,...)	A(), A(T)
local interface l	N.A.	derived both from the mapped class and from CORBA::LocalObject
servant	void*	inherited from ServantBase class
single inheritance	interface expansion	single inheritance
multiple inheritance	interface expansion	multiple inheritance

Table 5.4: IDL Language mappings to C++, Java, Ada95 and C - N.A.: Not Available (part 1)

IDL	Java	Ada95
module M	package M	package M, child package M if nested
interface I	interface I and IOperations, abstract class IHelper, IHolder, IPOA and optional IPOATie	package with Tagged Type, child Package if nested)
const	interface field	constant
short	short	CORBA.Short
unsigned short	short	CORBA.Unsigned_Short
long	int	CORBA.Long
unsigned long	int	CORBA.Unsigned_Long
long long	long	CORBA.Long_Long
unsigned long long	long	CORBA.Unsigned_Long_Long
float	float	CORBA.Float
double	double	CORBA.Double
long double	N.A.	CORBA.Long_Double
char	char	CORBA.Char
wchar	char	CORBA.Wchar
boolean	boolean	CORBA.Boolean
octet	byte	CORBA.Octet
string	java.lang.String	CORBA.String and CORBA.Bounded_Strings
enum E	class E, EHelper, EHolder	type E is (...)
struct S	class S, SHelper, SHolder	record S
union U	class U, UHelper, UHolder	type E(...) is record *with case/when*
array A	class AHelper, AHolder	type A_Array is array ...
sequence S	class SHelper, SHolder	type S is new *sequence package*
fixed<X,Y>	java.math.BigDecimal	Fixed_X_Y decimal type
typedef X Y	Helper classes	type/subtype Y is
any	class org.omg.CORBA.Any	helpers
TypeCode	class TypeCode and TypeCodeHolder	child package CORBA.TypeCode.Object
exception E	class E extends org.omg.CORBA.UserException	Exception identifier and record type
operation	method	Primitive Subprogram
attribute T A	A(), A(T)	Set_A and Get_A subprograms
local interface l	interface l, lOperations, _lLocalbase	based on CORBA.Local package
servant	interface l, lOperations, _lLocalbase	based on CORBA.Local package
single inheritance	single inheritance	Tagged Type Inheritance
multiple inheritance	delegation	Tagged Type Inheritance for first parent, interface expansion for subsequent parents

Table 5.5: IDL Language mappings to C++, Java, Ada95 and C - N.A.: Not Available (part 2)

[Hos98]. Compiled marshalling code is faster, but requires more memory, while interpreted marshalling code is slower, but more compact [SLM98]. After presenting how CORBA ORB provide portability to business applications through language mappoibngs, we will now see how they guarantee interoperability between ORBs.

5.5.6 Interoperability with CORBA Inter-ORB Protocol (IOP)

GIOP is the native protocol of CORBA ORBs. GIOP is an abstract protocol which has to be mapped onto concrete transport protocols. For instance, *Internet Inter-ORB Protocol* (**IIOP**) is a mapping of GIOP on TCP-IP. An *Environment-Specific Inter-ORB Protocol* (**ESIOP**) provides interoperability between CORBA ORBs and other middleware through legacy or custom protocols. GIOP specification describes a *Common Data Representation* (**CDR**), messages format and transport layer requirements. CDR specifies the bijective mapping from IDL data types to a byte or octet stream and vice versa. In CDR, primitive data types are aligned on their natural boundaries. Any primitive of size n octets must start at an octet stream index that is a multiple of n. This data alignment implies padding. In constructed data types of fixed length like struct, union or array, constituents are marshalled by their respective type. The marshalling of variable length types like string or sequence consists in encoding their length, then their constituents. To minimize the overhead of encoding integers as octets, *Compact CDR* (**CCDR**) [KJH+00a] was porposed as an optimization of CDR. GIOP defines eight messages to send requests, receive replies, locate objects and manage logical connections. Four versions of GIOP messages exist from 1.0 to 1.3. To be conformed to the 1.n version, the ORB shall implement the 1.m versions with $m \leq n$.

Like a visiting card, an object reference may list all the ways to contact an object using several transport profiles. Each profile contains a profile Id and addressing information to locate a communication endpoint for a given transport protocol. A profile may include information about QoS, network, transport and message protocols, data encoding rules, addresses or routes. For instance, IIOP profiles comprise IIOP version, host address and port number to locate the server process and an Object Key. This latter contains typically a POA Id an Object Id to locate a servant.

GIOP requires that the transport layer is connection and byte stream oriented, reliable and notifies failures. Multiple independent requests for different objects, or a single object, may share the same connection. GIOP requests can be ordered or not thanks to request_id. Clients may have multiple pending requests. A client need not wait for a reply from a previous request before sending another request.

For instance, the GIOP request message for the invocation of the *send* operation presented in listing 5.3 is depicted in table 5.6.

5.5.7 Real-Time CORBA

Real-Time systems require strict QoS guarantees in terms of execution time, bandwidth, latency and jitter. However, CORBA does not offer natively IDL interfaces to configure QoS properties. **Real-Time (RT) CORBA** [OMG05b] [KKSC04] is an extension to the CORBA and CORBA Messaging specifications to support *end-to-end predictability* from client to server applications across the network. Middleware elements have been extended with RT-POA, RT-ORB and RT-policies. To support QoS constraints, ORBs must schedule CPUs, memory and network resources. RT-CORBA allows the explicit control and configuration of processor, communication and memory resources [KKSC04]:

- **Processor resources** with priorities, mutexes, threadpools and a static scheduling service.
- **Communication resources** with priority banded connections, private connections and protocol properties and asynchronous invocations.

Index	Hex Data								Comments
0	47	49	4F	50					'G' 'I' 'O' 'P'
	01	00	01	00					GIOP '1'.'0' + little endian + Request message
	24	00	00	00					message size
12	00	00	00	00					service context
	00	00	00	00					request_id
20	01	00	00	00					response expected + padding
24	14	00	00	00					Length object_key = 20 octets
32	2F	31	39	36	34	39	2F	31	object_key = /PID/timestamp/ObjectID
40	32	32	38	39	34	37	34	34	/19649/1228947440/_0
48	2F	5F	30	00					
52	05	00	00	00					operation name length = 5
	73	65	6E	64	00				operation name = "send\0"
64	00	00	00	00					principal
68	34	12	00	00					s short value = 0x1234 + padding
72	03	00	00	00					o octet sequence length = 3
76	01	02	03						o sequence values = [1, 2, 3]

Table 5.6: Example of GIOP request message in little endian in MICO ORB

- **Memory resources** with request buffering and fixed thread pools size.

5.5.8 CORBA for embedded systems

Many embedded devices including SDR platforms have strict constraints regarding memory footprint, performance, size, weight and power (SWAP). To address these constraints, the **CORBA for embedded** specification alias CORBA/e™defines two lightweight CORBA profiles for DRE systems: the **Compact** and **Micro** profiles. The objectives are to reduce the footprint and overhead of embedded middlewares and to improve the predictability of resource-constrained distributed applications. Hence, CORBA/e eliminates many dynamic features of CORBA such as the Interface Repository, dynamic invocations (DII/DSI) and implicit activations. Both profiles support Real-Time CORBA with static scheduling and interoperability with legacy CORBA applications by supporting all versions of GIOP and IIOP. The **Compact profile** targets real-time applications like signal and image processing that are executed on board-based systems with 32-bit embedded processors supporting Real-Time Operating Systems (RTOS). This profile replaces the Minimum CORBA specification and includes most of the POA features and Naming, Event and Lightweight Logging services. The **Micro profile** targets chip-based mobile devices with deeply embedded processors like low-power microprocessor or high-end Digital Signal Processor (DSP). This profile removes `Any` and common services. It only supports a single fixed POA and Mutex interfaces from RT-CORBA. CORBA/e compact profile would only reduce memory footprint by about a half [MDD+03].

5.5.9 OMG Extensible Transport Framework (ETF)

The *Extensible Transport Framework* (**ETF**) [OMG04] specifies a set of **local** IDL interfaces implemented by *transport plug-ins* to allow CORBA ORBs to support new transport protocols other than the default TCP transport layer.

The ETF specification suffers from a number of issues, which have been notably presented in [FAM05]. As its name suggests, ETF specifies a transport framework and not a protocol framework. Hence, it is not clear from [OMG04] if alternative ORB message protocols other than GIOP can be used to build new transport protocols using the same transport layer [FAM05]. Furthermore, this specification seems to be not widely implemented by software middlewares.

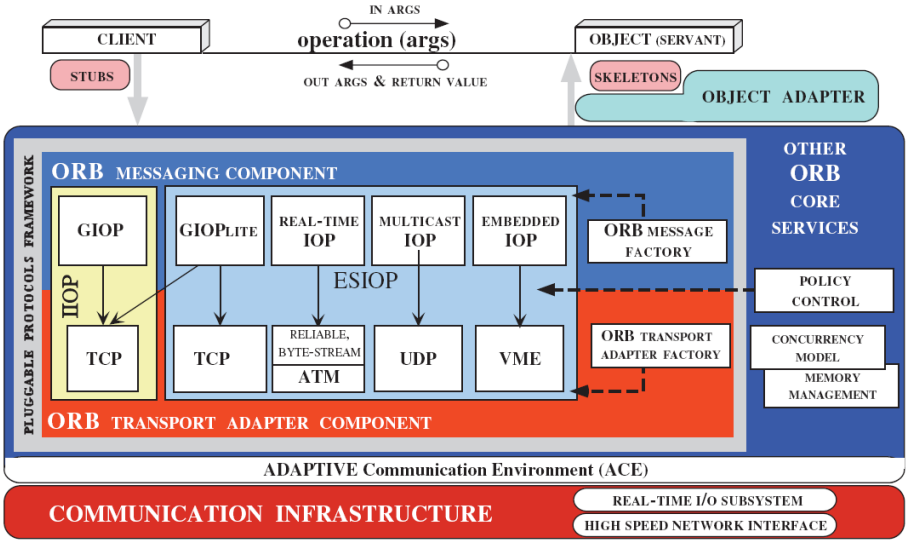


Figure 5.3: TAO Pluggable Protocols Framework [OKS+00]

TAO does not implement ETF, but proposes a similar approach called "pluggable protocol framework" depicted in figure 5.3 [OKS+00]. TAO provides transport protocols such as UIOP (GIOP over local IPC i.e UNIX domain sockets), SHMIOP (GIOP over shared memory), DIOP/MIOP (GIOP over UDP-IP uni/multi-cast) and SSLIOP (GIOP over SSL (Secure Socket Layer)). Compared to ETF, TAO pluggable protocol framework aims at using not only transport protocols such as ATM or PCI, but also other lightweight pluggable ORB protocols e.g. ESIOPs as an alternative to GIOP. Moreover, TAO ORB designers have applied the same approach to the real-time Java ORB ZEN that contains a dynamically configurable messaging protocols framework [KRS03].

Beyond a standard transport framework like ETF, embedded middlewares require frameworks for message formats other than GIOP like MHAL messages for SDR, custom data encoding rules other than CDR e.g. CCDR (Compact CDR) [KJH+00a] or ASN.1 Packed Encoding Rules (PER) [IT02], etc.

5.5.10 State of the art on real-time and embedded CORBA middlewares

General-purpose CORBA implementation

TAO [TAO08] TAO [SLM98] is an open-source Real-Time CORBA middleware written in C++. TAO targets static hard real-time applications such as avionics or medical systems. TAO architecture is based on an object-oriented framework for concurrent and network programming called The ADAPTIVE Communication Environment (ACE) framework [Sch94] [SSRB00] (TAO means *The ACE ORB*) and numerous design patterns [BHS07] [SB03]. TAO supports RTCORBA 1.0, most of CORBA 3.0 and parts of RT-CORBA 2.0 [KKSC04]. The design and performance of CORBA middlewares and its services have been deeply studied through the implementation of TAO: optimized IDL compiler [GS99], POA [PS98], request demultiplexing strategies [PSG+99] such as perfect hashing and active demultiplexing with constant time lookup of servants regardless of the number of objects, operations or nested POAs,

multi-threaded CORBA ORB architecture [Sch98], pluggable protocols framework [OKS+00], [GS98], A configuration file allows the definition of the strategies to be loaded in the TAO ORB to cope with aspects like concurrency, request demultiplexing, scheduling and connection management. TAO is a highly configurable and efficient middleware for real-time systems, however its footprint between 1 and 2Mo is a limitation for some memory-constrained embedded systems. An inherent limitation cited in [KKSC04] is the complexity and the associated learning curve of the IDL-to-C++ mapping.

RTZen [RTZ] RTZen is an open-source Real-Time CORBA middleware [RZP+05] developed according to the Real-Time Specification for Java (RTSJ) to enhance predictability. Java provides interesting features such as dynamic class loading, introspection, and language-level support for concurrency and synchronization. RTZen is based on the experience gained in the TAO design and evaluation. RTZen has also a pattern-oriented software architecture. The primary design goal are predictability with end-to-end priority preservation, bounded latency and jitter, bandwidth guarantees, and 2) a micro-kernel [BHS07] architecture to dynamically plug in the needed middleware services and to reduce the ORB footprint. Eight modular ORB services have been identified: object adapters [KKS02], message buffer allocators, GIOP message handling, CDR Stream readers/writers, protocol transports [KRS03], object resolvers, IOR parsers and `Any` handlers. Java is both an advantage and a drawback as real-time virtual machines are not widely available and employed in radio platforms with embedded processors and DSPs. For instance, Java is typically used for portable GUI on workstations and not for signal processing applications.

ZeroC Internet Communication Engine (ICE) According to Henning [Hen04], the CORBA Object model suffers from some limitations such as opaque object reference, complex type system and lack of multiple interfaces. ICE provides interface inheritance and interface aggregation like COM. It has an IDL called *Specification Language for ICE* (**SLICE**). SLICE does not include character, unsigned, fixed-point, unicode string, typedef, any and unions, array and bounded sequences, attributes, inout, context. The ICE C++ mapping is thread-safe and uses ISO C++. The only standardized transport layer for CORBA is TCP-IP, while ICE supports TCP-IP, SSL and UDP. Developers can implement new transport plug-ins. In ICE, all data are always in little endian byte order and with byte-alignment (no padding) for simplicity, compactness and to minimize bandwidth consumption. The protocol state machine consists of five messages types: Request, BatchRequest, Reply, ValidateConnection and CloseConnection. The ICE protocol supports compression. ICE compresses messages larger than 100 bytes in bzip2 which is useful for low-bandwidth links. ICE-E is an implementation of ICE for embedded systems, which removes the dynamic features of ICE.

Special-purpose CORBA implementation

PicoObjects [MVV+07] PicoCORBA and PicoICE are minimalist implementations of CORBA and ZeroC ICE [HM07] middlewares to provide interoperability with embedded objects called *picoObjects* in Wireless Sensor Networks (WSN). Instead of generating one skeleton per object interface as in conventional object-oriented ORBs, the picoObject compiler generates a single skeleton for all the interfaces implemented on a server node. This skeleton contains an ad-hoc Finite State Machine (FSM) that parses requests and builds replies according to CORBA and ICE message protocols. Instead of byte-level decoding, the FSM compares the request signature with the set of expected signatures. PicoCORBA objects implement the bare minimum to ensure CORBA interoperability. They only implement the `non_existent` and `is_a` operations from the CORBA::Object interface and the Request and Reply GIOP 1.0 messages. All non-supported messages are simply ignored. The FSM is generated in a implementation independent description, which is translated in supported target languages such as C,

Microchip PIC assembler, Java and VHDL. The footprint of minimal software servers in PicoCORBA is significantly lower than in various middleware implementations. However, this ad-hoc approach does not target portability using standard language mappings.

CAN-CORBA CAN-CORBA [KJH+00b] is a specialized CORBA implementation dedicated to distributed embedded control systems using the CAN[26] bus. The CAN bus is a standard embedded bus originally designed for the automotive domain that supports up to 8 bytes communication at 1Mbps between microcontrollers and devices e.g. within a car. CAN-CORBA supports both connection-oriented point-to-point and publish-subscribe communication over CAN. Kim et al. [KJH+00a] [HK05] propose an Environment Specific Inter-ORB Protocol (ESIOP) called *Embedded Inter-ORB Protocol* (**EIOP**) for the CAN bus. EIOP is based on the optimization of CORBA CDR encoding rules called *Compact Common Data Representation* (**CCDR**). To remove padding bytes, integers are not aligned on 32-bit boundaries, use variable-length encoding and big-endian ordering. The first two MSBs are used to indicate the integer size in bytes. An integer is encoded using one to five bytes according to its value. Moreover, EIOP supports only the `Request` and `CancelRequest` messages from the eight messages of GIOP. The packed encoding rules may require more processing, but use the network bandwidth more efficiently. EIOP drastically reduces the size of invocation messages by optimizing GIOP and CDR. Even if CORBA interoperability is sacrified, application portability is preserved using the same CORBA IDL and APIs.

ROFES ROFES (Real-Time CORBA For Embedded Systems)[27] [LJB05] is a RT-CORBA middleware, which extends the Embedded Inter-ORB Protocol (EIOP) from Kim et al. [KJH+00a] to support RT-CORBA on the CAN bus. Lankes et al. propose another ESIOP called the CAN-based Inter-ORB Protocol (CANIOP), which reuse the same CCDR encoding rules [KJH+00a]. CAN priorities are mapped on RT-CORBA priority bands. The CANIOP message header only contains a single octet to indicate the message type and some flags like protocol version and endianness, and two octets for the message size.

nORB nORB[28] [SXG+04] [SCG05] is a customized middleware for distributed real-time and embedded systems. It targets the networks of memory-constrained microcontrollers. Instead of using a top-down approach by subsetting an existing middleware, nORB follows a bottom-up approach by reusing only the needed concepts and features from existing frameworks like ACE portable infrastructure, TAO ORB core, Kokyo real-time scheduler and dispatcher, UCI-Core canonical middleware and CORBA object model. From CORBA IDL, nORB supports only the data types required by the application domain such as primitive types, structures and sequences. From CORBA GIOP, nORB uses CORBA CDR encoding rules. A subset of GIOP messages has been selected and customized like such as Request, Reply, Locate Request and Locate Reply [SCG05]. Some message fields have been moved like the `operation` name in Request and `object_key` in object references. Others fields have been removed like `requesting_principal` and `service_context` in Request. A `Priority` field has been added to Request messages and object reference to support similar concepts as RT-CORBA. nORB only provides a single object adapter per ORB. Object registration operations have been moved from the object adapter interface to the ORB interface. In conclusion, nORB sacrifices both CORBA portability and interoperability to reduce the middleware footprint and presents the trade-offs needed to adapt

[26] Controller Area Network
[27] http://www.rofes.de
[28] http://deuce.doc.wustl.edu/nORB

general-purpose CORBA middleware concepts to the requirements, memory and real-time constraints of the deeply embedded domain.

MicroQoSCORBA MicroQoSCORBA [MDD+03] is a CORBA middleware framework and design environment for small embedded devices including a GUI-based fine-grained configuration tool and an IDL compiler. The IDL compiler generates optimized stub and skeleton code for a customized ORB and POAs. The generated stubs use a dedicated protocol and transport layer. Interoperability may be maintain or removed. Unneeded functionality like exception, large data types and some messages can be removed. MicroQoSCORBA supports fault tolerance, security and real-time QoS constraints. Different implementations allows the trading off of QoS and resource consumption. MicroQoSCORBA has been developed in Java 2 Standard and Micro Edition. MicroQoSCORBA notably supports IIOP and its own protocol called MQCIOP, optional inclusion of CORBA system and user exceptions and fixed length messages.

Reflective middlewares

Reflection middlewares are based on the separation of concerns proposed by Kiczales in the Meta-Object Protocol (MOP) [RKC01]. The meta-object protocol separates base-level objects implementing functional aspects from the meta-level objects implementing non-functional aspects such as policies, mechanisms and strategies for communication, security or resources allocation. Reflection middlewares use meta-level objects to observe and modify base-level objects. Reflection middlewares enable self-introspection and reconfiguration of the ORB core and services at run-time. In the following, we present DynamicTAO and its successors: LegORB and UIC-CORBA. This family of reflective middlewares targets ubiquitous computing that requires dynamic adaptation to the environment and relies on heterogeneous and resource-constraint devices such as PDAs, sensors and phones.

Dynamic TAO Dynamic TAO [29] [RKC01] [KCBC02] is a reflective middleware built on top of TAO. It provides interfaces to retrieve SW modules from the network, to load and unload these modules into the ORB and to examine and change the configuration of the ORB. Application servants can also be dynamically reconfigured. ORB modules implement different ORB strategies relative to concurrency, scheduling, security or monitoring. A *Persistent repository* contains the software modules organized by category.

LegORB LegORB [30] [RMKC00] is a dynamically reconfigurable and component-based micro-ORB kernel for ubiquitous computing. It targets communications between heterogeneous devices e.g. between the PDA acting as the client and the desktop computer acting as server. Component configurators at ORB, client and server-level allows the assembly of only the components required by an application from a set of pre-defined services: de/marshaller, GIOP reader/writer, Connector/Acceptor and Invocation/Object Adapter. In contrast to monolithic ORB cores, this modular approach allows the dramatic reduction of the middleware footprint. LegORB provides a simplified dynamic invocation interface and CORBA interoperability e.g. IIOP on top of which additional interfaces may be implemented e.g. CORBA ORB interface. Hence, LegORB addresses interoperability of embedded applications, rather than portability, which still remains at the charge of the designer.

[29] http://srg.cs.uiuc.edu/2k/dynamicTAO
[30] http://srg.cs.uiuc.edu/2k/LegORB

UIC-CORBA Universally Interoperable Core (UCI) [RKC01] is a reflective middleware for ubiquitous computing and a modular middleware framework that captures within abstract modules key middleware services such as network and transport protocols, connection establishment, object registration, object reference generation and parsing, marshalling and demarshalling strategies, method invocation, method dispatching, client and server interface, object interface and attributes, scheduling, memory management and concurrency strategies. These abstract modules are extended by concrete modules, which are dynamically loadable. A *personality* refers to the specialization of the UIC framework for a given execution environment (middleware platforms, devices, networks) to satisfy specific application requirements. Three kinds of personalities have been distinguished: client-side, server-side or both. UIC can support both single and multiple personalities. A single personality UCI is dedicated to a certain middleware platform like CORBA, JavaRMI or DCOM. The difference between these middleware platforms is the implementation of the abstract middleware services. In contrast, a multi-personality UIC allows applications to use the same invocation interface independently of the target middleware platform, while the UIC can transparently dynamically loads and/or selects the required personality. As a result, applications directly interact with native UIC interfaces and are portable regardless of the underlying middleware platforms. For instance, a servant can be registered with different personalities at the same time. Personalities can be built in different configurations: from fully static to purely dynamic or hybrid configurations to trade off size and flexibility. Personalities are entirely interoperable with standard middleware platforms. The footprint of client-side CORBA personalities can range from 18KB for PalmOS to a 48.5KB for a Windows CE. The CORBA personality implements a custom dynamic invocation interface, a subset of data types like basic types, struct and sequences.

PolyORB PolyORB[31] [VHPK04] [Hug05] is a *schizophrenic* middleware in Ada 95 that supports different distribution models such as CORBA, SOAP, Ada 95 Distributed System Annex (DSA), JMS (Java Message Passing) adapted for Ada 95 and Ada Web Server (AWS) [32].

PolyORB proposes a *Neutral Core Middleware*(NCM) sandwiched between *application-level* and *protocol-level* personalities. Application personalities make the adaptation between applications and the middleware, register applications in the NCM and support communications between different applications. Protocol personalities translates neutral application requests into messages. The NCM is a decoupling layer, which enables interoperability between all combinations of local, remote, application and protocol personalities. The NCM and personalities implement the following primitive middleware services: addressing for global identification e.g. CORBA IOR, binding to associate entities with the resources they use to communicate e.g. CORBA stubs, representation for marshalling e.g. CORBA CDR, protocol for interoperability e.g. CORBA GIOP, transport for data transmission e.g. CORBA ETF, activation to associate a request to a concrete implementation e.g. CORBA POA with servants and execution to allocate resources for request processing e.g. RT-CORBA pooled thread. PolyORB supports the following application personalities: RMI with CORBA and DSA, RPC with DSA, message passing with MOMA (Message-Oriented Middleware for Ada) and Web applications with AWS (Ada Web Server). The supported protocol personalities include GIOP 1.0 to 1.2 with CDR, Internet IOP, Multicast IOP and Datagram IOP) and SOAP [Rec07] based on XML. Moreover, PolyORB has been formally modeled with PetriNets to verify its behavior. [HVP+05] PolyORB-HI [33] is a subset of PolyORB for High-Integrity Systems written in Ada2005. It is modeled in (**AADL**) *Architecture Analysis & Design Language* for analysis, verifications and automatic code generation with an AADL tool suite called Ocarina.

In summary, the following optimization principles can be used to reduce the footprint of embedded

[31] https://libre.adacore.com/polyorb/
[32] https://libre.adacore.com/aws
[33] http://aadl.enst.fr/polyorb-hi

middlewares [PSG+99] [KKSC04] [MVV+07]: elimination of dynamic invocation and instantiation, reduced interface definition language without complex or variable length data types, bypass of optional fields in messages, simplification or complete elimination of optional protocol features like exceptions, common services and indirect references are not supported, modular design with only needed modules.

The difference with our work is that research WSNs and networked devices use custom and deeply embedded environments i.e. microcontrollers. Some of these optimization principles cannot be used in application domains where backward compatibility with stringent standards like the SCA is required. Instead of reducing the embedded unaware GIOP message format, our approach is based on the SCA MHAL push-only protocol originally designed for embedded radio systems.

After presenting the state on the art on software middleware implementations, we will present the state on the art on hardware middleware implementations.

5.6 State of the art on hardware implementations of object middlewares

5.6.1 ST StepNP MPSoC Platform and MultiFlex Programming Models

StepNP [PPB02] [PPB+04] [PPL+06c] is a simulation environment to explore network applications and Network Processor Units (NPU) architectures.

StepNP is based on a NPU virtual platform with multi-threaded RISC processor models, a NoC channel model and dedicated coprocessors. These models communicate with each other using a SystemC Open Core Protocol (SOCP) communication channel. As in object-oriented middlewares, the SOPC interface is defined in a *SystemC Interface Definition Language* (**SIDL**), which corresponds to pure virtual C++ class declarations like the SystemC *sc_interface*. An SIDL compiler generates glue code between the model implementations acting as server and the client StepNP tools written in various languages (TCL, C++, Java). The generated stubs and skeletons allow distributed simulation among multiple workstations, virtual platform introspection, instrumentation and control of SystemC models.

The *MultiFlex* [PPL+06b] [PPL+06c] mapping tool automates the integration of parallel and distributed application objects onto HW/SW heterogeneous MPSoC platforms. This integration relies on two high-level parallel programming models as used in Java and C#: a loosely coupled CORBA-like object-oriented message passing model called *Distributed System Object Component* (**DSOC**) and a tightly coupled POSIX-like shared memory model called *Symmetric Multi-Processing* (**SMP**). As both programming models provide different trade-offs in terms of performance and coupling, Multiflex allows one to combine them in a unified approach.

The DSOC programming model supports interoperable communication between heterogeneous processing elements and is inspired by the distributed object model used in CORBA and DCOM. The DSOC model is based on three hardware services:

- a hardware *Message Passing Engine* (**MPE**) encodes service requests into messages according to a neutral data format, encapsulates them into NoC packets and performs the reverse work at the reception node.

- a hardware *Object Request Broker* (**ORB**) engine coordinates and dispatches communications between distributed client and server objects. It also synchronizes and performs the mapping between client and server threads. The HW ORB maps client service requests to servers and allows run-time load balancing. The least loaded server is currently selected.

- a hardware thread manager by processor schedules hardware threads with a round-robin policy, while a priority based policy is explored.

Each DSOC object is described by a *System IDL* (**SIDL**) interface. An SIDL compiler generates software MPE drivers for processors, while for coprocessors the hardware data encoding logic is produced and bound to the NoC interface. Both blocking and non-blocking invocations are supported.

The SMP programming model supports threads, monitors, conditions and semaphores. Its implementation relies on a memory-mapped hardware *Concurrency Engine* (**CE**). The CE is controlled by a POSIX-like C++ class library, which is implemented by in-lining read/writes operations.

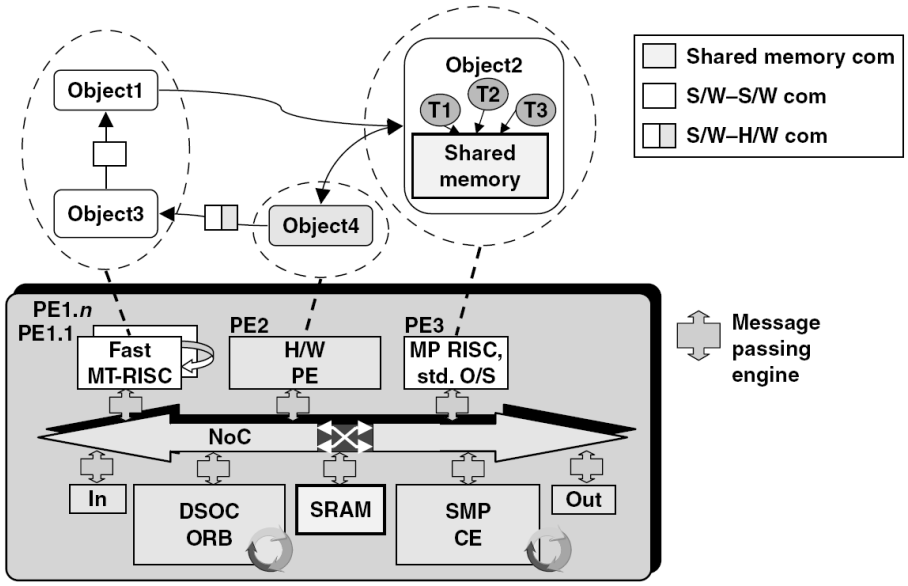


Figure 5.4: Mapping of OO applications to MPSoC platform based on a mixed DSOC-SMP model

In MultiFlex, both shared memory and message passing programming models are supported through hardware ORBs, message passing and concurrency engines. No operating system or virtual machine is required, yet the same high-level programming models are supported. The use of well-defined interfaces either in IDL or through POSIX-like APIs allows a system-level application to be independent of the underlying platform implementation. Horever, a single HW ORB represents a communication bottleneck and constitutes a central point of failure.

Instead of the custom SIDL, we propose to use an industry standard, CORBA IDL, which supports a language-independent type system and standard software language mappings. This allows application interfaces to be portable and independent of platform providers. What is required for a unified approach is to propose standard IDL-to-HDLs mappings. We aim at proposing such a mapping for VHDL in this work. Besides, we consider that POSIX-like APIs could be written as local IDL interfaces like the mutex interface in Real-Time CORBA.

Moreover, a MultiFlex component-oriented programming model inspired by the Fractal component model [Fra08] is under development to enable the composition and (re)configuration of DSOC objects. From our point-of-view, this initiative demonstrates the need of a component model to address the limitations of the object model presented in chapter 6.

5.6.2 UCLM Object-Oriented Communication Engine (OOCE)

The Object-Oriented Communication Engine (OOCE) [BRM+07] is an HW/SW platform based on object-oriented model, which provides remote method invocation semantics between HW/SW objects

on top of which different synchronization, concurrency and programming models can be implemented. OOCE is based on the non-standard ICE middleware, which proposes to resolve CORBA limitations with custom and improved implementations of CORBA architecture and services. OOCE uses the ICE SLICE IDL, the ICE message protocol called ICE Protocol (ICEP) and its data encoding rules. This choice enables OOCE to provide interoperability with off-chip objects communicating with ICEP. OOCE supports blocking and non blocking communications.

Communications between HW/SW objects are handled in OOCE by two elements: the *Local Network Interface* (**LNI**) and the *Local Object Adapter* (**LOA**). The LNI is a hardware coprocessor, which monitors the bus/network traffic to detect the address of software objects based on a mapping table between object identifiers and bus/network addresses. The LOA acts as a skeleton implemented as an Interrupt Service Routine (ISR).

For communications from off-chip client objects to on-chip server objects, a *Remote Object Adapter* (**ROA**) translates a TCP-IP message encapsulating an ICEP message into a local message for the object skeleton. For on-chip to off-chip communications, the ROA has a mapping table between server object identifiers and their remote network addresses.

OOCE provides a reconfiguration service [DRB+07] consisting of four logical layers. In the layer 1, dynamically reconfigurable objects implement an additional interface to start, stop and reconfigure its processing and to load/store their state. In the layer 2, the entire reconfiguration process is performed - notably object persistence. The layer 3 includes high-level services such as the scheduling, location and migration of dynamically reconfigurable objects. The last layer corresponds to the application or OS, which directly or indirectly uses the underlying services.

In OOCE, synchronization and concurrency are the concern of the designer, who is not limited by a fixed and specialized set of APIs as in other interface-centric approaches like Philips TTL [vdWdKH+06], but his implementation can use a HW/SW library of basic building blocks such as mutexes, locks and semaphores.

The same middleware architectural building blocks were applied to NoC-based platforms to support location transparency between off-chip and on-chip objects [RMB+05] with a minimum overhead. In this later work, an ORB is called a communication engine or *communicator*, which is a hardware core that contains middleware services such as objects adapters, remote servants, indirect servants and message passing engines.

As their software counterpart, the object adapter maps object identities to object implementations i.e. servants through an Active Object Map (AOM). An object adapter contains a finite state machine to implement the message protocol (e.g. GIOP for CORBA, ICEP for ICE etc.) and to decode/encode data types.

Hardware objects are implemented as servants, which are registered in an Object Adapter (OA) inside the communicator. The OA forwards messages received from the remote network to the local network towards the desired servant. Each servant implements all class methods and manages the attributes of each object along with their consistency. These attributes may be stored in a local memory, a shared memory, a register file or flash memory to support persistence. Each method invocation message has a unique server object identifier, a unique method identifier and the associated parameters. Proxies know the location of servants and translate signal-based invocation into bus transactions, while skeletons convert the received invocation request into writes on the hardware object interface. Local objects may directly interact by method invocation without the communicator.

Remote servants translate local invocations for remote objects into remote messages through the off-chip network using the necessary encoding rules and message protocol. Each remote servant contains a *Remote Object Map*, which maps local bus addresses to global object identifiers.

Run-time location of objects through location forwarding is implemented by *indirect servants*.

In conclusion, key architectural elements of the object-oriented middlewares have been adapted to

embedded SoC platforms. Local network transparency is based on stubs and skeletons, remote network transparency is provided by remote servants and objects adapters using look-up tables, and location transparency relies on indirect servants via location forwarding. High-level services may be implemented on top of these middleware building blocks such as load balancing, fault tolerance based on replication, object persistence and migration and reconfiguration transparency.

In [RMB+05], Rincon et al. consider that OCP provides a common syntax between hardware modules, but not a common semantics, and propose the remote method invocation model to offer these lacking semantics. We argued in [GBSV08] that OCP does provide common syntax and semantics but only at transport level instead of at application level and we investigate a mapping of semantics between RMI and OCP.

Moreover, we follow a component-oriented approach, in which connectors enable to explicitly express synchronization and concurrency, while their deployments are useful to infer local or remote communications through stubs and skeletons.

5.6.3 Commercial Hardware CORBA ORBs

In the SDR context, the SCA software architecture relies on CORBA middlewares, up to its version 2.2.1, to support transparent communications between waveform application objects based on the minimum CORBA profile. However, digital signal processing performed in modems from the physical layer or in a transceiver [NP08] is typically implemented using resource-constrained computation units such as DSPs, FPGAs and ASICs on which commercial CORBA middlewares were not available. Until recently, three main approaches based on the Adapter [GHJV95] design pattern [JTR06] have been used to abstract the communications between the application modules on non CORBA-capable devices (FPGAs/ASICs and DSPs) and application objects on CORBA-capable devices (GPPs and DSPs) [PB07]. These adapters translates a CORBA IDL interface into a function-based software interface on Instruction-Set Architectures (ISAs) i.e. GPPs/DSPs or a signal-based hardware interface on FPGAs/ASICs. These three approaches are described below.

Application-level Adapter a.k.a. *Resource Adapter*: an application object on GPPs represents an application module on FPGA or DSP. It directly uses a custom low-level HAL API to communicate with these modules. This HAL API encapsulates software device drivers to send data to the application modules via proprietary transport layers. Such transport layers may be based on memory-mapping or message-passing such as off/on-chip serial or parallel buses. At the receiving side, a decoding logic dispatches the transferred data to the requested application module based on a physical memory address or a logical address e.g. an identifier within a message. This decoding logic is dedicated to a given transport layer and thus not portable. Two kinds of application-level proxy can be distinguish: generic and custom proxies. A *generic proxy* a.k.a. Resource proxy implements the fixed and generic SCA Resource interface to control and configure application modules e.g. via calls to *start()* or *configure(sequence(struct(param_id, value)))*. A *custom* proxy is dedicated to an user-defined business application interface e.g. with *doFFT()* or *setFrequency* operations. As proposed in [JTR05], a custom proxy may use a generic proxy to translate some user-defined operations e.g. *setFrequency* into standard operations from the Resource interface e.g. *configure*. A custom proxy may also directly use a custom HAL. Application-level proxies satisfy the SCA requirement of portability across SDR platforms only for software application objects on GPPs, but not for application modules on DSPs or FPGAs. This approach is also called "Proxy Component Level Adapter" approach in [PB07].

Platform-level Adapter a.k.a. *Custom Device*: when the existing standard SCA APIs like GpsDevice or EthernetDevice used to abstract hardware devices like Audio and Ethernet chipsets do not cover all needed devices typically a modem device, a platform SCA Device object implements a custom business interface e.g. ModemDevice to abstract the required devices. This approach is not compliant with

the SCA specification as 1) platform SCA Devices should not provide user-defined interfaces and 2) application SCA Resource objects uses custom Devices and are thus not portable. This approach is called "Device Component Adapter" in [PB07].

Modem Hardware Abstraction Layer **(MHAL**) [JTR07] API: to avoid the two previous approaches, the MHAL API has been recently standardized by the JPEO JTRS in May 2007. The MHAL API abstracts modem devices by platform *MhalDevice* objects from application software on GPPs. This API defines standard message-passing communication and routing interfaces combined with a common message format to support homogeneous communications between application modules among *Computational Elements* (CEs) i.e GPPs, DSPs and FPGAs. The MHAL interfaces are specified in CORBA IDL for GPPs, C for DSPs and VHDL for FPGAs. A MhalDevice on GPP is a CORBA object used to route MHAL messages to application modules on DSPs and FPGAs. The FPGA MHAL API defines VHDL interfaces and timing diagrams of Transmit and Receive nodes used to respectively write and read MHAL messages. The MHAL API is a low-level message passing API used at application level, instead of being used as a message protocol plug-in at middleware level. Indeed, waveform application developers on GPPs and DSPs has directly to build and manipulate MHAL messages. Standard messages are only defined for the RF Chain for instance used to configure Power Amplifiers (PA). For other application messages, no common data type system and encoding rules are specified to automatically build messages as in CORBA from high-level and (semi-)formal system interfaces described in UML or IDL(s). As in traditional approaches, application messages must be defined in textual specifications prone to interpretations. The application modules portability and communications interoperability cannot be formally guaranteed at system level and functional level. Application modules on GPPs, DSPs and FPGAs could encode differently message parameters and could not understand each other. For CORBA-capable devices, this solution proposes a standard message passing API to access standard modem devices. However, *application* modules on non CORBA-capable DSPs and FPGAs are still encapsulated within black-box software proxies and are not treated as "first-class citizens" by software application objects. This approach is called "Component Level Adapter" approach in [PB07].

In order to support seamless communications between distributed CORBA applications and notably compliant with the SCA specification, two leading providers of software CORBA ORBs, PrismTech and OIS, propose to implement in hardware the main functionalities of CORBA ORBs that make adapters and HALs needless. A hardware ORB could be useful to satisfy hard real-time constraints as it is inherently deterministic in terms of latency and jitter compared to a software implementation especially with a TCP-IP transport layer. Communications performance could reach an improvement up to a hundred times over software ORBs on GPPs [Hum06], but as far we know there are no published details about benchmarks or descriptions about the experimental protocol (processor, memory and communication architecture, messages size, load conditions, ...).

5.6.4 PrismTechnologies, Ltd. Integrated Circuit ORB (ICO)

PrismTech, Ltd. [Pri08] is a company specialized in enterprise, real-time and embedded middlewares used in various application domains such as defense, aerospace, telecommunications and software defined radio.

The OpenFusion *Integrated Circuit ORB* (**ICO**) engine [HP06] [Hum06] [Hum07] [Fos07] is a hardware implementation of a CORBA ORB core depicted in figure 5.5. The OpenFusion ICO design environment consists of the ICO hardware communication engine, an IDL-to-VHDL compiler and optional Spectra MDE Tools for SCA compliant hardware implementations. ICO implements in VHDL the GIOP message protocol used by CORBA middlewares to provide communication interoperability among distributed software objects. ICO decodes GIOP request and reply message headers and converts the endianness of message data if required. Like the address decoding logic used in hardware bus interfaces,

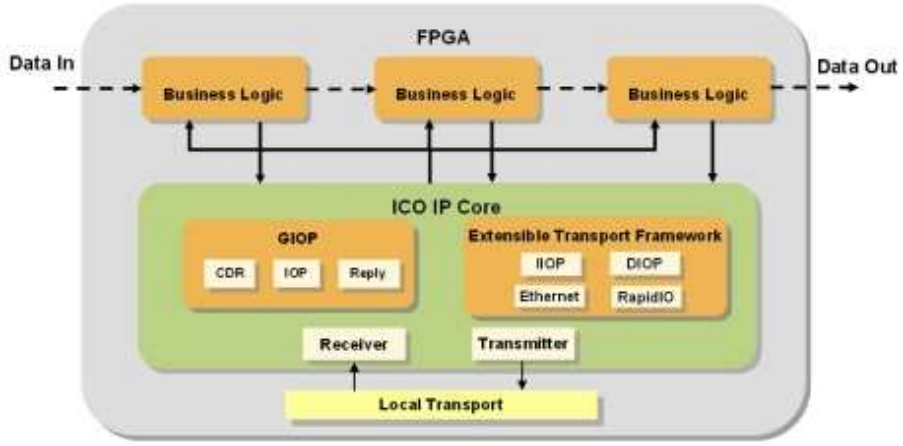


Figure 5.5: PrismTech Integrated Circuit ORB (ICO) [Fos07]

ICO routes GIOP requests to the appropriate hardware modules according to the object key and operation name contained in GIOP request messages. The direction of IDL operation parameters is interpreted as read/write primitives using read/write strobe signals on registers and memories within hardware components. User data in the GIOP message payload are decoded by removing padding and transferred to the desired hardware module. If a GIOP reply message is expected, the OpenFusion ICO engine reads the storage elements of the hardware modules to retrieve the response data if any. It then encodes the GIOP reply message header and the outgoing user-data into a GIOP reply payload. The reply message is finally written to the local transport layer via a typical FIFO interface. A bridge interface maps the ORB read/write strobe signals and data buses to the application modules interface. This bridge could be based on standard interfaces such as OCP [Fos07]. Hardware modules can also act as client. When a hardware module initiates a request, ICO reads the request data, encodes them into a GIOP request message and transfers this message to the required servant using the local transport layer. If a reply message is received by ICO, the response data are written to the client hardware module.

To allow hardware modules to communicate with the outside world through ICO, hardware designers have to describe each storage element of an application module i.e. register or memory by an operation name. The in, out and inout direction of an IDL operation parameter is respectively interpreted as writing, reading and writing/reading a storage element [Hum06]. IDL interfaces seem to be only described as low-level read/write primitives on register banks like hardware dependent software (HdS) drivers e.g. *void wrReg1 (in unsigned long p1)* as presented in [Fos07]. In order that IDL interfaces can be used at system level independently of their hardware or software implementations, we argue that they should be described at application level e.g. *void setMode(in unsigned char mode)*.

The IDL compiler maps IDL operation parameters to storage elements within an application module. It establishes a mapping between each IDL operation name and a single read or write strobe signal depending on operation parameters direction. The integration of hardware application modules with ICO is performed by connecting the read and write strobe signals and the data buses generated by the IDL compiler. The ICO IDL-to-VHDL compiler supports primitive IDL data types like char, octet, unsigned short, unsigned long and unsigned long long. It also supports constructed data types like simple strings, sequence and any of primitive types and structures of primitive types, strings and anys. Additionaly, ICO supports the SCA Resource and Device interfaces.

Based on the OMG Extensible Transport Framework (ETF) specification [OMG04], ICO proposes to support several transport layers via ETF interface modules. An ETF module may be introduce between

ICO and the local transport layer to encapsulate GIOP messages with transport headers.

In conclusion, PrismTech militates in favor of a "GIOP Everywhere" approach into which external as well as internal processors (GPPs/DSPs) communicate with hardware application modules in FPGA using the GIOP protocol. GIOP was not initially designed for embedded systems, even if it remains the default message protocol in CORBA for embedded a.k.a. minimum CORBA [OMG08a]. An obvious example of GIOP overheads is the encoding of operation names as sequences of characters in ORB messages instead of using numerical identifiers. These identifiers could be mapped into strings via lookup-tables in stubs and skeletons and accelerate operation demultiplexing in lieu of hashing as used in TAO. In real-time and resource-constrained environments, domain-specific message protocols such as MHAL for SDR could be integrated into CORBA as ESIOPs and provide similar functionalities, but may be more efficient in terms of hardware resources e.g. network bandwidth usage and encoding/decoding latency. Moreover, CORBA IDL could be used to generate stubs/skeletons for message-passing between different address spaces e.g. internal processor(s) within an FPGA, but also for memory-mapping within the same address space e.g. for internal processor(s) within FPGAs [MRC+00]. Such an approach could allow one to use the transfer model adapted to the distribution of objects, while preserving the same user-defined interface.

5.6.5 Objective Interface Systems, Inc. ORB*express* FPGA

Objective Interface Systems (OIS) [OIS08] is a firm which provides CORBA middlewares used in a wide range of application domains such as Telecom, Defence, SDR, Aerospace, process control, transportation and consumer electronics.

ORB*express* ®FPGA [Sys08] [Jac08] [JU08] is a hardware CORBA ORB written in VHDL. The ORB*express* ®IDL compiler could support all CORBA IDL data types. It generates hardware wrappers with "a simple, well-documented interface" to the FPGA designer, which support the dynamic and partial reconfiguration of Xilinx Virtex FPGAs. Hardware application modules could be disconnected from and reconnected to the hardware ORB without disturbing the processing of the ORB or other hardware modules. CORBA application functionalities could be transparently migrated between GPPs, DSPs and FPGAs. The latency to process CORBA messages would be inferior to a microsecond and the footprint would take a "small fraction" of FPGA LUT and gates [Jac08].

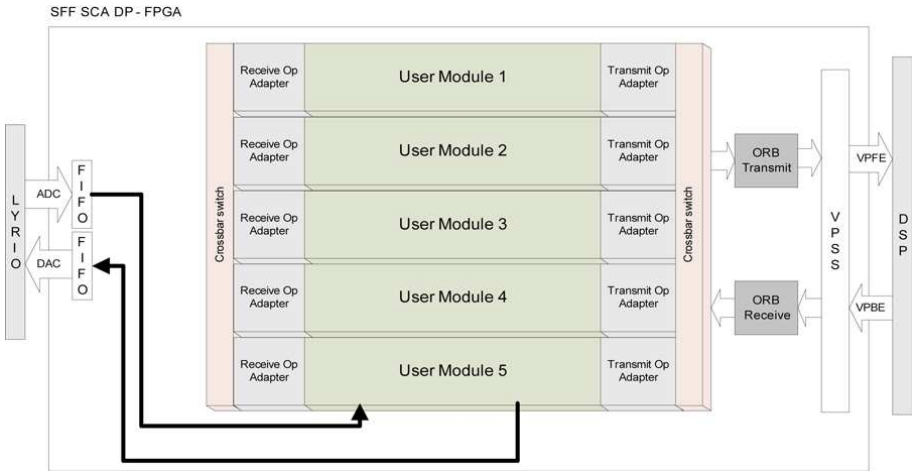


Figure 5.6: OIS ORB*express* FPGA [OIS08]

As a conclusion, OIS proposes a hardware implementation of a CORBA ORB like PrismTech. How-

ever, as far we know there are not many published details on its implementation and particularly on the hardware object interface, which are generated by their IDL-to-VHDL compiler.

5.6.6 OMG IDL-to-VHDL mapping

The two main providers of commercial real-time and embedded CORBA middlewares - PrismTech and OIS - have proposed a hardware implementation of a CORBA ORB core. The implementation of a common message protocol such as GIOP should provide communications interoperability. However, PrismTech and OIS have implemented their own IDL-to-VHDL mapping as no OMG standard exists at the moment. As a consequence, the portability of hardware application modules cannot be guaranteed for ORB users. The migration from one ORB vendor to another will require a manual integration between the application logic and the proprietary and incompatible stubs and skeletons. Hence, this is against one of the main motivations underlying object-oriented middlewares i.e. to automate the integration of application objects. To solve these issues, initial discussions about a standard IDL-to-VHDL language binding have started since December 2007 between four OMG members: on the one hand, the two ORB vendors that have implemented CORBA in hardware and on the other hand, two ORB users - Mercury and Thales - whose goal is to present industrial requirements.

In the scope of this work, we propose the mapping requirements for Thales based on the previous analysis of the state of the art about the application of the object concepts down to hardware [WSBG08]. These requirements will be presented later with our contribution.

Initially, PrismTech wanted to propose a Request for Comments (RFC) concerning the IDL-to-VHDL mapping [Bic07], but finally OIS and PrismTech were not focused on the interface mapping at this time, but more on using GIOP for heterogeneous communications with FPGAs [GP08].

5.7 Conclusion

A middleware aims at supporting transparent communications between heterogeneous systems regardless of their distribution, hardware computing architecture, programming language, operating system, transport layer and data representation. The main objectives of middlewares are to provide portability and interoperability to distributed applications. Since the CORBA object-oriented middleware is required by the SCA framework v.2.2.2 for SDR applications, we focused on the CORBA object model and presented its design flow, language mappings, message protocol and implementations. CORBA provides portability through the standard CORBA middleware API and middleware services, and standard language mappings from CORBA Interface Definition Language (IDL) to software programming languages. IDL describe user-defined messages as operation signatures. CORBA provides interoperability through a standard protocol called *General Inter-ORB Protocol* (**GIOP**), which is mapped to transport layers such as internet protocols or shared memory. CORBA also allows *Environment Specific Inter-ORB Protocols* (**ESIOPs**) to support communications in resource-constraint systems. We presented a state of the art on software implementations of real-time and embedded CORBA middlewares and on hardware implementations of object middlewares including CORBA. In embedded and real-time hardware/software platforms, middleware implementations have been lightened and customized to limit middleware functionalities to what is really needed by an application domain and to reduce middleware footprint. In the next chapter, we will present the component-oriented approach, which leverages the object and middleware approach.

Chapter 6

Component-Oriented Architecture

Contents

In software architecture, the component-oriented approach is a specialization of the object-oriented approach in which a component may be roughly viewed as a composition of objects, which are dedicated to a particular functionality or aspect such as computation, communication, (re-)configuration, life-cycle, etc.

The component-oriented approach has been successfully applied to embedded software. Our objective is to explore and propose a common object-oriented component-based approach for both hardware and software components, all along the design flow and at different abstraction levels.

The main goal of the component-oriented software architecture is the separation of concerns between what is the responsibility of the software designer (business code implementation) and all the other orthogonal preoccupations such as component management (instantiation, configuration, deployment...) and communication protocols.

In this chapter, we will introduce the main component models, which are used in the design of hardware/software embedded systems.

This chapter is organized as follows. Section 6.1 introduces the notion of component as used in hardware and software engineering. Section 6.2 presents classical definitions of software components and discusses their applicability to hardware components. Section 6.3 motivates the transition from an object-oriented approach to a component-oriented approach for both software and hardware entities. Section 6.4 describes the principles of the component approach such as architecture, integration and composition. Section 6.4 explains the technical concepts of the component approach such as component model, connection model and deployment model. Section 6.6 presents software component models for SDR such as SCA, general purpose component model such as CCM and software component models for real-time and embedded systems. Section 6.7 describes the hardware component model supported by hardware description languages and connection models Section 6.8 presents system component models such as the SystemC component model at different abstraction levels and actor-oriented components. Finally, section 6.9 concludes this chapter.

6.1 Introduction

Generally speaking, a component is a smaller, self-contained part of a larger entity [34]. Etymologically, *component* is the present participle of *compose* from Latin "componere" that means "to put together": com- ("together") and ponere ("to put"). Components are assembled to form larger components that compose a system. Hence, the term component is quite general and usually employed to designate any physical or logical piece of hardware and software in electronic devices.

Hardware engineering inherently follows a component-based approach at all design levels. From a macroscopic viewpoint, the motherboard of Personal Computer (PC) supports the assembly of various components such as microprocessor chips on CPU sockets, memory modules in memory slots, hard disk, dedicated boards such as graphics card, sound card, network card using internal buses like PCI and peripheral devices like mouse, keyboard on external buses such as USB and Firewire. Each of these components has well-defined hardware interfaces and implement standard protocols to provide portability from one motherboard to another and interoperability of communications between these components. Over the past 50 years, the continuous improvement of integrated circuits fabrication allows the miniaturization and integration of the main components of usual PCs onto a single chip to build System-on-Chip (SoC) on ASICs and FPGAs. From a microscopic viewpoint, hardware components in System-on-Chip (SoC) are basically called *Intellectual Properties* (**IP**) cores and include soft microprocessor cores (e.g. Xilinx MicroBlaze and Altera Nios II), bus controllers (e.g. UART[35], I2C[36] and RapidIO) memory controller (e.g. DDR2 SDRAM[37] and Flash memory), Ethernet MAC cores, Digital Signal Processing IPs such filters (e.g. FIR[38] and CIC[39] filters), encoder-decoder (e.g. Viterbi, Reed-Solomon, Turbo Code IPs) and much more. In ASICs, hardware components are custom IPs and standard cells, which implement basic functionalities such as NAND gates, FIFOs and RAMs.

Software engineering also follows a component-based approach through the use of various software libraries provided by languages e.g. C standard library, device drivers, OS, middlewares, and software platforms such as the Java Class Library (JCL) of the Java platform and the Base Class Library (BCL) of the .NET Framework.

Hardware components such as hardware devices and IP cores typically appear as software components through device drivers. A device driver is a software code that bridges the gap between high-level procedural or object-oriented application interfaces and low-level software interfaces based on read and write primitives to configure and control hardware devices.

A software library is merely a reusable set of classes and functions used by developers to obtain some functionality. In contrast, an object-oriented framework is a set of classes that provide a reusable software architecture dedicated to a given application domain [BHS07] [SB03]. Examples of frameworks include graphical user interface (GUI) frameworks such as Microsoft Foundation Classes (MFC), network frameworks such as ACE, middleware frameworks like TAO, application frameworks like the SCA [JTR06] for SDR applications.

The application of the concept of hardware component in software led to the emergence of "Component-Based Software Engineering (CBSE)". One of the first appearances of the concept of "software components" originates from McIlroy [McI68]. Instead of redeveloping from scratch, component-based hardware and software engineering consists of assembling new systems by reusing in-house business components or purchasing a license for prebuilt black-box components often called *Commercial Off-*

[34] http://en.wiktionary.org/wiki/component
[35] Universal Asynchronous Receiver Transmitter
[36] Inter Integrated Circuit
[37] Double Data Rate Synchronous Dynamic Random Access Memory
[38] Finite Impulse Response
[39] Cascaded Integrator-Comb

The-Shelf (**COTS**) components. Components can have a wide spectrum of granularity ranging from fine-grained components like device drivers and UART cores to large-grained components such as OS and soft CPU cores. Facing ever increasing design size and complexity, the reuse and assembly of previously validated components allows increased productivity and quality, while reducing development time and costs. Component-Based Design allows one to design or buy *once* a component and use it *multiple* times, amortizing its development or purchase costs. In classical software engineering, the traditional design flow consists of object-oriented analysis, design, programming, testing and integration.

In CBSE, designers activity is shifted from component specification, design, development and validation to component selection and integration. The selection of components is based on technical descriptions provided by datasheets. For hardware components, datasheets mainly describe hardware interface, timing behavior and footprint in terms of logical elements and/or gates. For software components, they typically contain memory footprint and supported instruction set architecture and compilers. The integration work depends on the quality of the chosen component, its documentation and optional testbenches and drivers. Due to proprietary standards, the use of COTS components results in a dependency on component suppliers or even vendor lock-in as users do not have any control on future component changes. For instance, the migration of a design from one FPGA vendor to another may require choosing functionally equivalent IP cores and lead to supplementary integration problems since each vendor uses its own bus interface and protocol. This clearly shows the limitations of the classical hardware component approach based on IP cores, on which we will focus in the following.

Due to the weak meaning of the term *component*, a "component-based system" does not have much more sense than a "part-based whole" [BBB+00]. Like the distinction made between object-based and object-oriented languages [Cap03], we make a distinction between component-based and component-oriented approaches. Indeed, Ada83 was defined as an object-based language because it provided the concept of object, but not the concept of inheritance which is characteristic of object-oriented languages [Cap03]. In the same way, the component-oriented approach goes further than the component-based approach as it provides in addition :

- standard **component model**s to define component structure and behavior like the interfaces component must implement,
- standard **connector model**s to specify component interactions, assemble components into larger components and into applications,
- standard **container model**s to mediate access to middleware services such as event, notification, transaction and persistence services,
- standard **packaging and deployment model**s to deploy distributed component-oriented applications.

Over the past 40 years, a cross-fertilization, i.e. a mutually productive and beneficial exchange, has occurred between component-based software engineering and component-based hardware engineering. Starting from the vision of McIlroy [McI68] in 1968, software components have been inspired by hardware components. Then, software components have independently evolved and led to a higher-level of abstraction and better separation of concerns that could improve hardware components. Our goal is to propose a common and unified component-oriented approach of design at system-level.

An important question is what are the differences between hardware and software components ? For Wang and Qian [WQ05], hardware components and software components are logically equivalent, but the software component approach needs a formal theory and associated tools to reach the maturity of hardware engineering. They propose an algebra for components in [WQ05] that is illustrated with EJB, CCM, OSGi and Web Services.

From a logical viewpoint, the same algorithm and model of computation can be clearly implemented in a hardware and/or software component. A system-level specification must be defined independently of hardware and software implementation languages. Examples of specifications languages are UML, SDL and Esterel. Various modeling frameworks such as Ptolemy [HLL+03], Metropolis [BWH+03] and BIP [Sif05] propose to model and simulate embedded systems through the hierarchical composition of computation models [GBA+07], which are implementation agnostic. However from an implementation viewpoint, hardware and software components have different execution model. Hardware components can be defined using logical equations according to the boolean logic and simulated as well-understood discrete events. Software components can be notably specified with the Hoare logic [Hoa69] using pre- and post-conditions, that have to be extended from procedural programming to component behavior and interactions [WQ05].

To support a common component-oriented approach at system-level which can be refined at implementation level as hardware/software components, system, hardware and software engineering must share the same abstract concepts and how these concepts can be implemented using system, software and hardware implementation languages. Such an approach will allow a better communication between system, hardware and software engineers.

6.2 Definitions

As in the case of objects, there is no real consensus on the definition of software components [Kra08]. The main reason is that each commercial, industrial and academic component technology applies the concept of component slightly differently for each execution environment (e.g. enterprise, embedded and/or real-time system) according to application domain requirements (e.g. financial, consumer electronics, automotive, avionics, software radio domain).

There is a lack of agreement about what software components are and how they are used to design and develop component-based applications [BBB+00]. Nevertheless, one of the most cited definition of software components is [SP96]:

"A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A component can be deployed independently and is subject to composition by third parties."

This definition underlines that the raison d'être of components is to be composed i.e. put together only through multiple well-defined interfaces and explicit dependencies forming a contract with their environment in which they are deployed. Note that this definition is sufficiently general to be applicable to both software components and hardware components including chips and IPs. Numerous discussions exist in the software literature about a definition of software components [WQ05], whereas there are no such debates in the hardware literature probably since hardware components are much better understood. Software component interfaces are typically operation-based software interfaces, while the component context in this definition is vague and may designate required interfaces or some software platform services such as periodic activation and communication services. Similarly, hardware component interfaces designate "wire"-based hardware interfaces, while the component context may designate required input/output wires or some hardware platform services such clock frequency, asynchronous reset and communication bus. Although this definition could be sufficient for hardware components, it needs to be extended with the closely related definitions of component model and software component architecture defined by Heineman and Councill in [HC01] and cited in [Kra08]:

"A **software component** is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

"A **component model** defines specific interaction and composition standards. A component model

implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model."

Bachmann et al [BBB+00] add that "the component model imposes design constraints on component developers, and the component framework enforces these constraints in addition to providing useful services."

These definitions stress the need for a component model, a composition standard and a supporting component infrastructure. The composition model includes a connector model and a deployment model. These definitions specify that components obey the interaction and composition rules of a component model, whose implementation allows one to enforce them. For hardware components, the interaction and composition standards may respectively designate standard communication protocols such as AMBA and hardware interconnection rules such as matching types and width of input/output ports for IPs or voltage for chips.

"A **software component infrastructure** is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications."

A component infrastructure guaranties that a component assembly will respect the performance constraints specified at a system level.

However, this formulation was more clearly refined by Krakowiak in [Kra08]:

"A *software component infrastructure* or *framework*, is a software system that provides the services needed to develop, deploy, manage and execute applications built using a specific component model implementation. It may itself be organized as a set of interacting software components."

A possible application of this definition to hardware components is the collection of EDA tools used by hardware designers to respectively write and simulate HDL code, synthesize, place, route and load bitstream down to FPGAs and/or produce ASICs.

A software component technology consists of a component model and component framework [BBB+00]. Each component technology has its own component model, component framework, languages and files to specify component interfaces, assembly, packaging, configuration and deployment.

This heterogeneity led the OMG to shift from Object Management Architecture [OMG95] centered around object request brokers to Model-Driven Architecture [OMG03] focused on models.

The integration of multiple heterogeneous component technologies requires component middlewares. Moreover, the concept of connectors has been proposed to provide interoperability between these technologies.

6.3 From Object-Oriented to Component-Oriented approach

The inherent limitation of the object-oriented approach has motivated the emergence of an evolutionary component-oriented approach. The main differences between the object orientation (OO) and component-orientation (CO) are listed below [WQ05] :

6.3.1 Object model vs. component model

The only constraint that is imposed on objects is to satisfy the principles of the object model such as encapsulation, inheritance, methods that can be invoked on an object interface i.e. messages that can receive an object. In contrast, a component model imposes much more constraints on components such as rules of naming, design, composition, configuration, packaging, deployment and interfaces to use and implement.

In the list below, we briefly review object concepts and contrast them with component concepts.

- **Identifiable Objects as Instances of Classes**: Like an object, a component is a self-contained and identifiable entity. While an object is said to be an instance of a class, there is not such a clear distinction between the static type of a component and its run-time existence. We will simply say that a "component instance" is an instance of a "component type". The term "component" generally refers to a component instance.

- **Method Invocation and Message Passing**: while an object is conceptually based on a single form of interaction i.e. method invocation, a component do not only communicate through message passing, but also via other communication models such as event and data stream.

- **Encapsulation**: Interface and Implementation(s): Like an object, a component encloses its own state and behavior that are only accessible through interface. Hence, a component also respects the encapsulation principle.

- **White-Box Reuse and Black Box Reuse**: Objects and components may support both white-box reuse through inheritance and black-box reuse through composition.

- **Polymorphism**: polymorphism can be extended to component interface type and component type.

- **Relationship**: As opposed to objects, components explicitly specify their dependencies. Components may support hierarchical containment to build composite components from primitive components. A component may be part of more than one assembly called configuration and shared between several components.

- **Concurrency, Synchronization and Coordination**: While interactions between objects are primarily based on method invocations, interactions between components may be explicitly mediated by special components called *connectors*.

- **Persistence**: As opposed to objects, components are hosted in containers that provide and mediate access to common services like persistence.

- **Distribution**: While an object is typically a fine-grained unit of distribution, a component may be from a fine to a large-grained unit of distribution thanks to hierarchical containment. Connectors can transparently deal with the non-functional aspect of distribution, while avoiding deployment anomalies [BÓ2].

Component concepts will be describe in more details in the following.

6.3.2 Separation of concerns

Objects directly use various technical services for naming, event, persistence, transaction, concurrency and security that are dependent on the underlying software platform and are interwoven with business logic. However, the same component could be reused in various applications and operating environments. The component-oriented approach introduces the concept of CONTAINER [BHS07] to mediate component access to platform services and automate their use instead of manual user interventions. These common services are provided by component frameworks that encapsulate middleware and OS services. Designers can thus focus on component functional logic rather than on the technical details of each platform on which components may be deployed.

6.3.3 Explicit dependency and multiple interfaces

An object only provides one interface that is the most-derived interface in the inheritance tree called principal interface in CORBA [OMG08b]. In contrast, a component provides and requires multiple interfaces that allow 1) an explicit specification of the external dependency of components, and 2) a separation of concerns between functional interfaces for business logic and non-functional interfaces for technical logic e.g. regarding configuration, deployment and activation. Component interfaces can be more easily added to extend component functionalities.

6.3.4 Interactions

OO basically supports a single type of interactions under the form of method invocations. CO goes further by taking into account multiple types of interactions such as invocations, events, and data streams. In CO, these interactions are reified i.e. materialized by the first-class concept of connectors. Connectors are used to mediate interactions between components and allow one to clearly separate component computation from component communication.

6.3.5 Composition rather than implementation inheritance

OO extensively relies on inheritance. While specification inheritance is beneficial to constrain implementations through abstract interfaces, implementation inheritance results in loose cohesion, tight coupling, may break the encapsulation principle and lead to inheritance anomaly [MS04]. Objects are loosely cohesive because attributes and methods implementation are implicitly scattered all over the inheritance hierarchy. Objects are tightly coupled since classes depend on each other implementation. Derived classes depend on the implementation of all classes on top of them in the inheritance hierarchy. Base classes may depend on the implementation of some functionalities that are left to derived classes or that derived classes should customize. Modifications in some classes may cause unexpected behavior and side effects in others classes. The encapsulation principle may be broken when derived classes directly access base classes attributes without using accessor methods as permitted by the public and protected qualifier of some OOLs such as Java and C++. Inheritance anomaly designates the abnormal modification of base classes that is required to guarantee the synchronization constraints of derived classes. In contrast, CO is based on the composition of components through their interfaces and results in tight cohesion, loose coupling and tight encapsulation.

6.3.6 Granularity of composition, distribution and reuse

In OO, units of composition, distribution and reuse are fine-grained objects, while CO allows a hierarchical composition of components from fine to large granularity. A component can be basically viewed as a composition of objects.

6.3.7 Language Independence

While OO mainly assumes object-oriented languages to notably provide inheritance, CO could be implemented in any language [WQ05]. One possible reason is that composition is basically supported by all languages as opposed to inheritance. Examples are software components in C such as in the Kaola, Fractal and PIN component models, in SmallTalk like in [FDH08], and hardware components in HDLs. Both C and HDLs are clearly not object-oriented languages, but they support component-based approaches. Nevertheless, non-OO languages must also preserve component concepts such as encapsulation.

6.3.8 Deployment and administration

OO is focused on implementation, while CO is centered on packaging, deployment (i.e. installation, connection), life cycle (i.e. initialization, configuration and activation), administration (e.g. monitoring and reconfiguration) in distributed systems [Kra08]. Indeed, objects and classes are not self-contained and self-deployable [WQ05].

6.3.9 Higher-level reuse

Due to their difference of granularity, OO is primarily based on low-level reuse of attributes, methods and classes through inheritance or composition, while CO originally supports higher-level reuse of component assemblies.

In summary, the software component-oriented approach is an evolution from the object-oriented approach. The object-oriented approach is focused on classes, objects and inheritance, whereas the component-oriented approach is focused on interfaces, interactions, dependencies and compositions.

6.4 Principles

Whatever the disagreements concerning their definitions, the principles underlying software components are basically the same in the literature and exposed in [WQ05]. Interestingly, these principles are also applicable to hardware components and are presented in an implementation-agnostic manner.

1. Components are based on decomposition and abstraction. Decomposition breaks a system into modules called components, which are self-described, self-contained and self-deployable. Abstraction determines the relevant level of details with which decomposition is performed.

2. During their design flow, components permits reuse at each level of their existence: component specification, component interface specification, component implementation in source code, components binaries registered in a repository and deployed component instances.

3. Component-based engineering increases quality and dependability since each component is supposed to have been tested and reused many times.

4. Component-based engineering could increase productivity since components are reused instead of being redeveloped from scratch at each time.

5. Component-based engineering promotes standardization to enable component "plug-and-play" and the emergence of component markets [Szy02]. Indeed, the need for portability and interoperability of hardware and software components has motivated the creation of international standards by various consortium such as IP-XACT [SPI08] specification for XML meta-data and tool interfaces from the SPIRIT (Structure for Packaging, Integrating and Re-using IP within Tool flows) consortium, interface and protocol standard such as AXI, VCI and OCP, the Corba Component Model (CCM) and Deployment and Configuration (D&C) [OMG06b] of Component-based Distributed Applications by the OMG.

6.4.1 From Hardware/Software Design to System Architecture

To tackle the complexity of modern embedded systems, it is increasingly accepted that hardware/software design needs to evolve towards system-level design [KNRSV00] [Dv04a] [SV07]. This evolution has motivated the emergence of system-level standards such as SystemC and SysML. In the same way that

hardware/software design evolves towards hardware/software architecture, system-level design needs to evolve to system-level architecture.

System-level architecture is the art and science of structuring a system, which may consist of heterogeneous parts such as a combination of hardware and software modules. A software architecture describes the subsystems and components of a software system, their relationships and their functional and non-functional properties [BHS07]. Software architecture separates multiple levels of concerns such as computation and communication thanks to two structural constructs: components and connectors. A software architecture can be dynamically changed through run-time allocation, instantiation and binding of components and connectors to provide a software system with adaptability and evolution. Similarly, the increasing industrial maturity of dynamic and partial reconfiguration of FPGAs enables to dynamically change a hardware architecture by loading partial bitstreams.

In [Per97], Perry justifies the required separation between (software) architecture and design. Architecture resides at a higher level of abstraction than design. It targets early engineering steps and provides a preliminary structure to a system. While design is focused on concrete implementation constructs such as algorithms, procedures, types and their interactions such as procedures calls and data access, architecture reasons about a system in terms of their components and their interactions. The constituents are represented by components, while their interactions are embodied by connectors. In our opinion, these principles are common to hardware and software architecture and could be applied at system-level.

An architectural style defines a set of rules that describe or constrain the structure of architectures and the way in which their components interact. Common architectural styles include pipes and filters, *Abstract Data Type* (**ADT**) like objects, events, messages and dataflow [GS94] [MMP00], layered style, shared memory with repositories (database and blackboard), distributed processes with client-server organization, FSM state transition systems. These common architectural styles have been formalized in architectural patterns such as LAYERS, PIPES AND FILTERS, BROKER, BLACKBOARD to build Pattern-Oriented Software Architecture (POSA) [BMR+96] [BHS07].

We believe that software architecture helps reason about hardware architecture through common, well-defined and well-understood architectural patterns and concepts such as connectors.

6.4.2 From Development to Integration

Component-based engineering accelerates development cycle and reduce costs. The assembly of components is based on their interfaces and semantics. The creation of a system from sub-system made of components is a natural work for numerous engineering domains such as aerospatial, avionic, automotive, mechanical, electrical and electronics systems.

6.4.3 Separation of Concerns

Separation of concerns allows the reduction of the conceptual size of each individual abstraction and makes it more cohesive [BME+07]. Components have multiple interfaces. Each component interface corresponds to an individual object interface and deals with a single aspect or facet. This decomposition favors separation of concerns and avoids monolithic or "god" objects. The multiple interfaces of components allow scalability and extensibility since a new interface can be added to a component to handle unforeseen aspects. *Aspect-Oriented Programming* (**AOP**) [KLM+97] provides another form of separation of concerns at source-code level rather than at interface level. In other words, an aspect is encapsulated within an interface in component-oriented programming rather than a piece of code as in aspect-oriented programming.

6.4.4 Reusability

Reuse allows one time implementation and multiple use, and increases productivity and quality [WQ05]. Reuse can be performed at different levels of granularity from low-level source code copy, inextensible function libraries, extensible class libraries that require a lot of understanding to high-level black-box reuse of components [WQ05].

6.4.5 Assembly and Composition

The key technical challenges of component-based design are formal approaches for *correct by construction* assembly of components [WQ05] and the prediction of the properties of a component-based system from the properties of the components themselves [BBB+00].

6.4.6 Component infrastructure or framework

A component infrastructure or framework consists of three models: a component model, a connection model, and a deployment model [WQ05]. The component model defines what a component is and how to create a new component under the component infrastructure. Component designers build components according to the component model. The connection model defines a collection of connectors and supporting facilities for component assembling. The deployment model describes how to install component-based applications into the execution environment. A component infrastructure or framework is also sometimes called "component technology" or "component architecture" in literature [WQ05].

Examples of component technologies include JavaBeans from Sun Microsystems, COM (Component Object Model) and DCOM (Distributed COM) from Microsoft and CORBA Component Model (CCM) from the OMG.

6.5 Technical Concepts

6.5.1 Component Infrastructure or Framework

A component infrastructure or framework consists of a component model, a connection model and component management. Component framework automates tedious tasks such as the configuration and deployment of component-based applications. These tasks are no longer required to be performed by the designers, who can now focus on business logic aspects. A component framework supports components with middleware services such as persistence, transaction, event service, security and quality of service such as availability and real-time performances. In many respects, component frameworks can be viewed as high-level and specialized operating systems [BBB+00].

Requirements

A component framework is required to manage a component-based application, notably its deployment into the execution environment. Application deployment consists in retrieving component artifacts (binaries, bitstreams) from a component repository, allocating the needed processing, memory and communication resources, installing component artifacts on the distributed execution nodes, then creating, initializing, connecting, configuring and activating component instances. The component framework makes component interfaces available to the end-users and monitors application activities to obtain its status, performances, load factors etc [Kra08]. When the end-users cease to use an application, the component framework can perform the reverse deployment actions: deactivation, disconnection, destruction of component instances, disinstallation and disallocation of component artifacts. Component framework

are required to provide common services to applications and guarantee their quality of services. Developers should have the capabilities to indicate the services their applications require, while component frameworks automatically inject the needed invocations in the application code and encapsulate components into containers to enforce separations of concerns [Kra08].

6.5.2 Component

A component provides its functionalities through required and provided interfaces. While most current component technologies only recognize software application components, we distinguish like the SCA two basic kinds of components: application components and platform components. A component can be viewed in multiple layers of a system. For instance, a software application component may be deployed on a software platform component called GPP logical device that represents the soft processor IP core (virtual hardware component) on which it is deployed. This IP core, its software and other hardware application components may be deployed on an FPGA chip and in an external flash memory chip (hardware platform components).

Application components implement business logic and require HW/SW platform services, while platform components implement technical logic and provides HW/SW platform services.

A component may have a functional or algorithmic implementation e.g. in executable UML, Matlab/Simulink, SystemC TLM; a software implementation e.g. in C, C++ or Java; or a hardware implementation e.g. in synthesizable SystemC RTL, VHDL or Verilog.

Logically, a software component has logical ports that provide and require properties, methods and events. Component properties also called attributes are either functional properties or non-functional properties also called extra-functional properties. Non-functional properties may be related to quality of service such as memory footprint and worst-case execution time (**WCET**) or to component metadata such as required processor, OS and software libraries or component name, version and source files. Events trigger the execution of an associated method called an event handler or callback. Methods may be synchronous or asynchronous i.e. blocking or non-blocking using oneway semantics, while events are typically asynchronous.

A hardware component has input, output, input/output hardware ports that provide and require one or several signal-based interfaces. Logically, it also provides and requires functional and non-functional properties and events. Functional properties or attributes are explicitly read and write through the component interface according to a custom or standard data transfer protocol and may be stored in registers or RAMs. Non-functional properties include footprint in terms of area, maximum execution frequency, latency, bandwidth regarding QoS or required FPGA, bus and hardware libraries, component name, version and source files. Writing on component ports triggers the activation of processes that implement component functionalities. Hardware components are typically synchronized by a clock port and initialized by an asynchronous reset port.

A component is governed by a component model, which may be based on a composition of models of computation, one for each of its different types of interfaces. For instance, control interfaces may rely on a FSM model of computation, while data stream interfaces may rely on a data flow model of computation.

6.5.3 Component Model

A component model is a set of standards, conventions and rules that components must respect. These rules assure that component developed by different designers and enterprises can be correctly installed in component repositories, deployed in an execution environment and interact at run-time with each other.

A component model defines the set of rules that describes the creation, configuration, composition and interaction of software components. A component model contains the interfaces that must implement a component.

Some works have been carried out to define formal component models using component algebra [WQ05] [BBS06] [dAH01] and models of computation [GBA+07] [RL02].

Requirements for component models

Requirements for a component model depend on the targeted application domains and application needs. General-purpose enterprise component technologies such as CCM and EJB provide full-featured component frameworks with advanced services such as persistence, transaction and security. In contrast, special-purpose embedded and real-time component technologies such as *PErvasive COmponent Systems* (**PECOS**) [GCW+02] require a small number of lightweight services to decrease memory footprint and may have stringent non-functional constraints regarding latency, bandwidth, deadlines, and priority.

We reformulate the requirements for a component model proposed by Bruneton et al. [BCS02] and Krakowiak [Kra08] at the hardware/software system level. In addition to requirements inherited from the object approach such as encapsulation and reusability, a component model should support:

- **hierarchical containment** of primitive components to build composite or compound components,
- **component sharing** to allow a component to be part of several component assemblies or configurations. Shared components may be used to abstract logical and hardware resources that are shared by multiple components such as processor, memory, communication bus,
- **explicit architectural description** of a component-based system to specify units of composition such as components and connectors using *Architecture Description Languages* (**ADL**s),
- **off-line or run-time reconfiguration** of component assemblies to add, remove, replace, modify component interfaces, components and connectors and allows system adaptation and evolution,
- **component control and management** during its life cycle to deploy components i.e. to install/uninstall, create/destroy, initialize, configure/query, connect/disconnect and start/stop components and during its execution for interception, monitoring and introspection.

Few component models such as Fractal support shared components. We add that a component model should enforce real-time constraints using scheduling and deal with concurrency within a component, while a connector model should cope with concurrency between components. Moreover, a component model should be able to describe software components as well as hardware components, but also application components as well as platform components. For instance, the SCA component model supports the description of hardware platform components as software components called logical devices.

6.5.4 Interfaces

Components specify the functionalities they provide and require through one or several interfaces.

Software component interfaces

Software interfaces typically define the set of operations they implement. In OO languages, a software component interface is generally a plain old object interface.

Two kinds of software component interfaces can be distinguished: user-defined application interfaces and standard interfaces from the component technology including the component model, the connection model, the deployment model and external middleware/OS services.

Component technologies may be based on implementation-independent interfaces such as CCM with CORBA IDL or implementation-specific interfaces such as EJB with Java and .NET with C# and VB.NET.

Hardware component interfaces

A hardware interface consists of the set of input, output and input-output ports offered by a hardware component. It is specified by an entity in VHDL or a module in SystemC and Verilog for virtual hardware components i.e. IP cores or a schematic in a datasheet for virtual and physical components.

Four dimensions for hardware interfaces can be distinguished: custom vs. standard, bus vs. core-centric.

Custom hardware interfaces are based on proprietary naming conventions and protocols that are particular to a designer, a group of designers or an enterprise. Hardware component ports are named according to traditional naming rules for clock (`clk`), active low asynchronous reset (`rst_n`), enable (`en`), strobe, data valid, chip select (`cs`), address (`addr`) and data etc. Interface protocols are graphically represented by timing diagrams in datasheets or in application specifications. Each usual and basic hardware components such as shifter, serial-parallel converter, FIFOs and RAMs has its own custom interfaces. As a result, designers need more time to understand, assemble and use hardware components.

In contrast, standard hardware interfaces provide common signal names and protocols among hardware designers. They favor component composition and avoid integration errors. Interface adapters are required to adapt custom interfaces to standard interfaces and may add performance overheads.

Contract

A contract determines the rights and duties of participants in a relationship. Beugnard et al. define four levels of contract [BJPW99] of increasing negotiability: basic or syntactical, behavioral, synchronization and quantitative 4.1.4.

A software component interface is often restricted to a syntactic contract [BJPW99] in standard component technologies, while academic component technologies may include syntactical, behavioral, synchronization and quantitative contracts. The definition of quality of service for components remains a major concern in component specification.

For Bachmann et al. [BBB+00], two kinds of contracts are required for CBSE: component contacts and interaction contracts. However beyond distinctions regarding these contracts, Bachmann et al. do not describe concretely what they should include.

A **component contract** specifies patterns of interaction that are specific to a component. Component contracts include the functional and non-functional requirements (rights) and provisions (duties) specific to a component as a whole and the interface contract of all component interfaces.

An **interaction contract** specifies patterns of interaction that are generic and independent from components. For naming consistency with the term "component contract", we consider that interaction contract can be more clearly renamed into **connector contract**. These patterns of interaction are represented by roles provided by connectors and used by components. Examples of interaction patterns include method invocation between client and server roles, event delivery between publisher and subscriber roles, data stream between source and sink roles, message passing between sender and receiver roles and shared memory between writer and reader roles.

Although the separation of responsibilities and contracts between components and connectors is not obvious, we consider that component contracts should include syntactical, behavioral and quantitative contracts through each of its interface contracts, while connector contracts should consist of synchronization and quantitative contracts. Indeed, a component provides typed service interfaces with well-defined usage rules and whose services may be required to execute in a given worst-case execution time. A connector may deal with communication, synchronization, arbitrating, scheduling between components with a given quality of service e.g. latency or fairness. It is independent from typed component interfaces.

Component-based system are assembled from the composition of components and connectors that respects each other's contract.

6.5.5 Behavior

A software component can be viewed as a composition of software objects. The same techniques used to specify object behavior can be extended to describe component behavior. Like an object, the behavior of a component can be divided into different views such as specified by UML diagrams. Temporal behavior can be specified by UML sequence diagrams for software components, and chronograms or timing diagrams for hardware components. Functional behavior can be specified by:

- state machines for hardware and software components e.g. using UML behavioral and protocol state machines [OMG07d], statecharts [Har87], interface automata [dAH01] or behavior protocols [PV02]. These state machines specify the order in which the methods provided and required by a component interface can be invoked. Based on these different formalisms, the compatibility and composition of components can be analyzed like in [dAH01] and [PV02].
- an implementation-dependent language such as C/C++/Java/SystemC for software and hardware components in case of High-Level Synthesis, and VHDL/Verilog for hardware components
- by an implementation-independent language such as an UML action language, the Matlab M language or through the assembly of pre-existing components like in Simulink,
- assertion-based constraints to specify pre-/post-conditions and invariants using e.g. OMG Object Constraint Language (OCL) [OMG06d] for software components or Hardware Verification Languages (HVL) such as *Property Specification Language* (PSL) [Acc04a] and Cadence Specman ***e*** for HW components.

6.5.6 Port

A component has ports that are service access points and the conceptual interface between the inner and outer view of a component. Component ports have a type such as an integer type e.g. for properties, a method-based interface type or an event type, a direction such as input, output or input-output or inout for short and interaction semantics such as synchronous or asynchronous method invocations in point-to-point mode, asynchronous event in publish-subscribe mode, or data stream in point-to-point mode.

6.5.7 Connection Model

In software architecture, a software connector is a component which binds the required and provided ports of two or more components. While components have ports, the interface access points of connectors are called roles [BÓ2]. In contrast to the object-oriented approach, the notion of connector reifies the relationships and interactions among components. Software connectors have been treated as first-class architectural entities in software architectures notably by Architecture Description Languages (ADLs).

Connectors separate computation from interactions. A connector may implement all functionalities except application functionalities. They can be used as a means of communication, mediation, coordination and control, and allows one to decouple and compose all these aspects [Per97]. While usual components implement application-specific functionalities, connectors implement generic and reusable communication and interaction patterns. Components and connectors can thus be reused independently.

Taxonomy of Software Connectors

In the taxonomy of connectors proposed in [MMP00], the three identified building blocks of software interactions are ducts, data transfer, and control transfer. A software connector mediates component interactions through one or more basic channels called **ducts** without associated behavior along which data and/or control are transferred among components. For instance, a pipe is a simple connector with one duct that only supports unidirectional transfer of raw data streams. Four main categories of connectors are identified according to the services they provide: communication for transfer of data, coordination for transfer of control, conversion regarding the type, number, frequency, or order of interactions, and facilitation for scheduling, arbitration, synchronization and concurrency control and load balancing. Both communication and coordination services may be provided by connectors such as function calls and method invocations. Like noted in [MMP00], transfer of control does not mandatorily imply loss of control from the caller to the callee. Moreover, eight types of connector are distinguished: procedure call for transfer of control and data; event for transfer of control and data; data access for transfer of data, conversion and persistence; linkage for facilitation; stream for data transfer; arbitrator for facilitation and coordination; adapter for conversion; and distributor for facilitation. Each connector type supports one or more interaction services. Connectors may thus adapt different interaction semantics and protocols.

Connectors as Middlewares

Connectors can provide components with distribution transparency. When components bound by a connector are deployed in the same address space on a single node, the connector may be implemented as simple pointers [Kra08]. When the same components are deployed on several nodes in different address spaces, the connector may use or implement the required middlewares services such as marshalling/unmarshalling, interface adaptation, interception, request dispatching etc. In this case, the connector is partitioned into two "fragments" that may represent a stub/skeleton pair or two middleware instances. For instance, software connectors have been employed in distributed embedded systems to drive the generation of dedicated middlewares thanks to their explicit expression of communication [SG07b].

Middlewares provide generic services to support various interactions among components, but it is often difficult to tailor a middleware to the specific needs of an application. Connectors typically provide services which are typically found in middlewares such as persistence, invocation, messaging and transactions. In the connector model proposed in [BÓ2], the three generic types of connectors that have been selected from [MMP00] are procedure calls, event delivery and data stream. Connector template architecture may include interceptors, stub/skeleton, dispatcher, arbiter and scheduler. Each connector template called frame is generic and parametrized by user-specific interfaces.

Composite connectors

To encapsulate various component interactions, connectors may implement from simple to complex functionalities. For instance, a connector may represent a POSIX mutex, a simple FIFO or use burst transfers and DMA. Composite connectors result from the composition of connectors and components [BÓ2].

An assembly of components and connectors is called a configuration. A configuration must comply with the composition rules of the connection model that describe how components and connectors

can be assembled. If the component model supports hierarchical containment, a configuration may be considered itself as a deployable component [Kra08].

Soundness of connectors

Some works such as [Kel07] [FDH08] have recognized that the distinction between components and connectors is not obvious or even artificial [Szy02]. We argue that the distinction between component and connector is relative to the considered level of abstraction. For instance, the Linux process scheduler can be viewed as a connector for application developers since it is not part of application functionalities [MMP00], but it can also be seen as a component for OS developers since it is part of OS business logic e.g. for innovative scheduling algorithms. In other words, components in the platform layer can be considered as connectors in the application layer and vice versa.

Related work on support of connectors in ADLs and component frameworks

The concept of connector has been used in Architecture Description Languages (ADLs) [MT00] and in some component frameworks such as SOFA [BÓ2] [BP04]. The use of explicit connectors has been proposed for distributed embedded systems in [SG07a] and in [RRS+05] with Lightweight CCM.

Connector model

A connector is governed by a connector model, which may be based on a composition of models of communication, one for each of its different types of roles. For instance, method-based interfaces rely on a message-passing model of communication, while event-based interfaces may rely on a publish-subscribe model of communication.

Although connectors have been studied for over a decade, they are not widely used in standard component technologies and have been proposed only recently by Thales and Mercury at the OMG for CCM and DDS.

6.5.8 Container

The CONTAINER design pattern [BHS07] allows the isolation of a component from the technical details of its possibly numerous operating environments. Components can access platform services from their container either through an explicit interface or via callbacks. Containers transparently integrate components into their execution environment. They use middleware services such as persistence, event notification, transactions, load-balancing and security as proxy of components. Containers also manage component life cycle such as creation/destruction using component factories, interconnection of component ports, configuration using configuration files typically in XML and activation e.g. using a timer service. A component typically provides a DECLARATIVE COMPONENT CONFIGURATION [BHS07] as in CCM, EJB and .NET that describes the platform services and resources it requires and how it wants to use them. The component framework or infrastructure automatically generates part of the container i.e. the glue code that is specific to the component requirements and the underlying platform.

6.5.9 Deployment model

The deployment model of a component infrastructure includes component instantiation, interconnection, configuration and activation. The deployment model is one of the fundamental differences among different component infrastructures [WQ05].

6.6 Software Component Models

In this section, we provide an overview of some component models.

6.6.1 Software Component Models for Software Defined Radio

JTRS JPEO Software Communications Architecture (SCA)

Introduction, Context and Objectives The Software Communications Architecture (SCA) [JTR01] [JTR06] [BJ07] is a software radio architecture published and maintained by the Joint Program Executive Office (JPEO) of the US Military Joint Tactical Radio System (JTRS) program. The SCA is considered as the de-facto standard for the SDR industry [OMG07e].

The SCA aims at enabling the portability, interoperability and reusability of software waveform applications that are deployed on different SCA-compliant radio platforms and reducing their development time and costs using software industry standards such as IEEE POSIX and its Application Environment Profile (AEP) for Realtime and Embedded Applications, OMG UML v.1.4.2, OMG Minimum CORBA v.1.0, W3C XML v.1.0 and OSF DCE v.1.1 UUID[40] .

UML is used to graphically represent SCA interfaces, operational scenarios, use cases, and collaboration diagrams using when possible the UML profile for CORBA v.1.0 [OMG02]. CORBA IDL is used to define the SCA interfaces. XML is used in the SCA Domain Profile, which identifies the capabilities, properties, dependencies, and location of software components and hardware devices of SCA-compliant radio systems.

The SCA has been primarily designed to satisfy the requirements of military radio applications, but it can also be used for commercial radio equipments. The SCA has received wide acceptance from military radio manufacturers, industry associations such as the SDRForum[41] and support from tools such as ZeligSoft Component Enabler and PrismTech Spectra tools. The ultimate goal would be that the SCA becomes a commercial standard maintained by an international organization such as the OMG. A concrete realization of this objective is the UML Profile for Software Radio a.k.a. PIM & PSM for Software Radio Components (SDRP) based on the SCA [OMG07e]. The SDRP specification leverages the SCA with the OMG Model-Driven Architecture (MDA) approach in which the Platform-Independent Model of SDR applications are decoupled from their realization in Platform-Specific Models for instance using a CORBA middleware. As opposed to the SCA, the SDRP specification is not yet widely used to design software radio applications and supported by commercial tools. The SCA follows a component-oriented software architecture approach through its own software component model and software architectural patterns (e.g. LAYERS and WHOLE-PART [BHS07]) and an object-oriented design and implementation approach using software design patterns (e.g. FACTORY METHOD, DISPOSAL METHOD, ADAPTER, PROXY, WRAPPER FACADE, OBJECT MANAGER, LIFECYCLE CALLBACK [BHS07]). The SCA masks the heterogeneity of hardware/software applications and platforms using a common object-oriented component approach. The SCA allows a functionality to be implemented either in hardware or software.

The SCA standardizes a software architecture for the deployment, management, interconnection, and intercommunication of software SDR application and platform components within radio communication systems.

The SCA specifies an implementation-independent set of interfaces, behaviors and rules to constrain the design and deployment of waveform applications and improve their portability. However, the SCA does not specify, like CORBA, how application components must be implemented notably in terms of

[40] Universally Unique IDentifier
[41] http://www.sdrforum.org

concurrency e.g. whether components must be passive or active, how they must be synchronized or whether their implementations must be thread-safe (reentrant), even if the SCA allows the use of POSIX.

The SCA software architecture defines an *Operating Environment* (**OE**), which consists of a *Core Framework* (**CF**), a minimum CORBA-compliant middleware and lightweight middleware services such as Naming service, Event service and Lightweight Log service, a POSIX-based Operating System (OS), a network stack and *Board Support Package*s (**BSP**).

The CF is a set of software interfaces defined in UML and CORBA IDL for the deployment, configuration and management of software application components on distributed real-time and embedded radio platforms.

In the following, we describe the last release of the SCA (version 2.2.2. [JTR06]) and its extensions at the time of this writing. Differences with SCA version 2.2 still in use are described when relevant. The SCA 2.2.2. extension addresses the deployment of non-SCA services other than Log, FileSystem, Event and Naming and a mechanism to manage and optimize application deployment.

Layered Software Architecture The SCA software architecture consists of several layers:

- Bus layer provided by Board Support Package (BSP) to use on-board transport buses such as VME, PCI and Ethernet.
- Network and Serial Interface Services to support multiple serial and network interfaces such as RS-232, RS485, Ethernet, 802.x and associated network protocol stacks such as an Internet Protocol (IP) stack from the OS layer.
- OS layer compatible with the SCA POSIX Application Environment Profile (AEP), which is based on the Real-Time Controller System Profile (PSE52) from POSIX 1003.13 [IEE04].
- Core Framework that is the set of application interfaces, framework interfaces and properties to abstract, manage and support hardware platform devices and software application components as a collection of distributed objects. All CF interfaces are defined in CORBA IDL.
- CORBA middleware is used by the CF to provide communication transparency among distributed computing objects and mask heterogeneity of languages and radio platforms.
- Application Layer in which SDR applications are an assembly of waveform components, which communicate through a CORBA ORB and perform functionalities organized as in the OSI model such as digital signal processing for modem components, data link for Link components, and routing processing for Network components, input/output access for I/O components and encryption/decryption for security components and radio services. Applications must implement some CF interfaces and use CF services and OS services restricted to the allowed SCA POSIX profile.

Core Framework The Core Framework v2.2.2. consists of four categories of software interfaces, which are graphically represented by UML class diagrams and textually defined in CORBA IDL.

- **Base Application Interfaces** contain the interfaces `LifeCycle`, `PropertySet`, `TestableObject`, `Port`, `PortSupplier`, `Resource` and `ResourceFactory`. The `Resource` interface inherits from the interfaces `LifeCycle`, `TestableObject`, `PropertySet` and `PortSupplier`. These interfaces are implemented and possibly extended by software application components called *Resources*. Application components inherit from the `Resource` base interface and are used by the SCA Framework Control interfaces for their management and control.

An SCA application is an assembly of `Resource` components. Examples of `Resources` include RouterResource, NetworkResource, LinkResource, BridgeResource, UtilityResource. Application components communicate either with each other or with the devices and services. The SCA does not specify how the functionalities of a Resource must be designed internal In the SCA, application components are only implemented in software, while in our approach we consider hardware application components i.e. signal processing IP cores as first-class components.

- **Base Device Interfaces** contain the interfaces `Device`, `LoadableDevice`, `ExecutableDevice`, and `AggregateDevice`. The `ExecutableDevice` interface inherits from `LoadableDevice` that inherits from `Device` that inherits from `Resource`. An `AggregateDevice` is an aggregation of device components. These interfaces are implemented by software platform components called *Devices*. The `Resource` components of an application uses `Device` components. Platform components inherit from the `Device` base interface and are used by the SCA Framework Control interfaces to manage and control hardware platform resources. `Device` components serve as software proxies [GHJV95] to represent real hardware platform devices such as ModemDevice, AudioIODevice and CryptoSecurityDevice. We consider that the `Device` interface is an instance of the WRAPPER FACADE pattern [BHS07] as this common high-level object interface encapsulated the low-level procedural or read/write interface of device drivers.

- **Framework Control Interfaces** contain the interfaces `Application`, `ApplicationFactory`, `Domain Manager`, and `DeviceManager` that are implemented by the SCA component framework to deploy and manage software application and platform components. These components are managed by an implementation of the Framework Control Interfaces.

- **Framework Services Interfaces** contain the interfaces `File`, `FileSystem`, and `FileManager` to provide distributed file systems [CDK01] using CORBA. Other software services not included in these interfaces include middleware services such as Log, Event and Naming services and radio services such as Time, Vocoder services.

Differences between the definition of the CoreFramework between SCA v2.2 and SCA v2.2.2 include the new "Base Device Interfaces" category that was initially included in the Framework Control Interfaces and the removal of the Domain Profile that do define software interfaces but deployment meta-data in XML.

As SCA devices and services called "System Components" are platform and domain specific, they are specified by standard APIs in dedicated documents separated from the SCA core specification. Where the SCA uses the term "system", we consider that the terms "platform" or "infrastructure" are more appropriate. For us, a system is a whole entity which results from the combination of both an application and a supporting platform.

Domain Profile The Domain Profile represented in figure 6.1 is a hierarchical set of XML files that describe the properties, interfaces, logical location, dependencies and assembly of `Resource` components with a Software Profile using a Software Profile, hardware `Device`s and services using a Device Profile.

The Domain Profile is a hierarchical set of XML files that describe the properties, interfaces, logical location, dependencies and assembly of Resource components using a Software Profile, hardware Devices and services using a Device Profile.

The Domain Profile is described by graphical UML class diagrams and textual DTD syntax. It has been adapted from the CORBA Components Specification (OMG version 3.0, formal/02-06-65: Chapter 6 - Packaging and Deployment). The Domain profile uses standard DCE UUID to uniquely identify XML

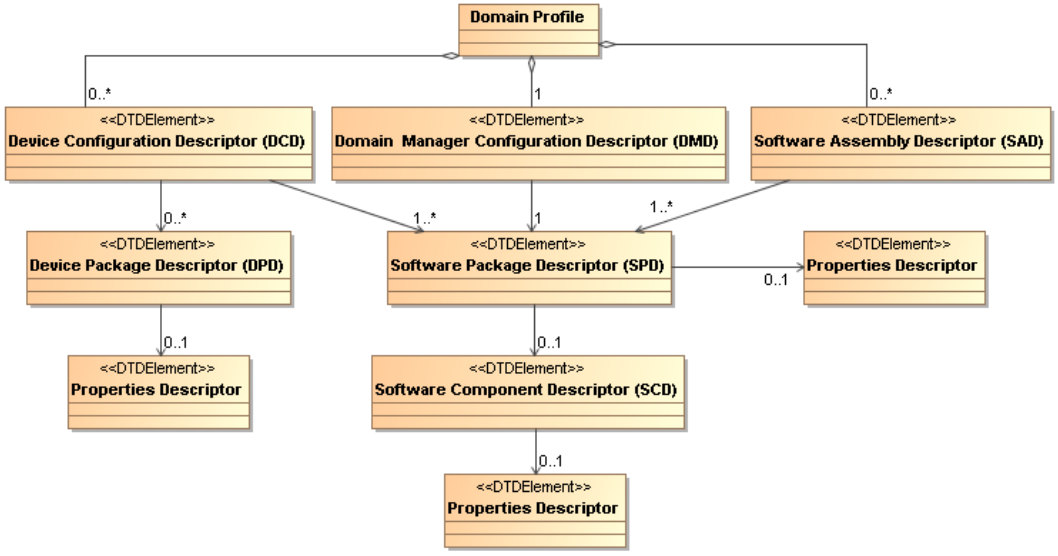


Figure 6.1: XML files from the SCA Domain Profile

elements and allows them to reference each other to express dependencies. A DCE UUID contains the characters "DCE:", the printable form of the UUID, a colon, and an optional decimal minor version number, for example: "DCE:700dc518-0110-11ce-ac8f-0800090b5d3e:1". A DCE UUID thus represents 42 characters or bytes. Besides, DCE UUID can be used for CORBA RepositoryIds [OMG08b].

Software components are described by a software profile which consists of one Software Assembly Descriptor (SAD) file that references (directly or indirectly) one or more Software Package Descriptor (SPD), Software Component Descriptor (SCD), and properties (PRF) files.

Software Package Descriptor (SPD) During application deployment, a SPD is used by the domain manager to load a component and its implementations. The SPD may describe several software implementations of a component for different types of processors, OS etc.

Software Component Descriptor (SCD) The SCD is based on the CORBA Component Descriptor specification. The SCD defines the component CORBA repository id (derived from CF `Resource`, `Device` or `ResourceFactory`), the component type (resource, device, resourcefactory, domainmanager, log, filesystem, filemanager, devicemanager, namingservice and eventservice), CORBA interfaces that the software component supports i.e. inherits (e.g. Resource or Device), provides and requires through component ports. Component ports have a name, a repid, a direction (provides or uses) and a type (data, control (default), responses or test).

Properties Descriptor The Properties Descriptor defines the value of configuration attributes for components and devices. Depending on the property kind (i.e. configure, test, allocation, execparam, factoryparam) and mode (i.e. readonly, readwrite, writeonly), `ApplicationFactory` and `DeviceManager` invokes the appropriate operations of the `Resource` (*configure* and/or *query*, *runTest*), `Device` (*de/allo- cateCapacity*, *execute*) and `ResourceFactory` (*createResource*) interfaces during deployment.

Device Package Descriptor (DPD) A device may have a Device Package Descriptor (DPD) file which provides a description of the hardware device along with its model, manufacturer, etc. The DPD belongs to the Device Profile and describes hardware device registration. Hardware components defined in DPDs can be from single hardware element to a whole hardware structure of a radio Hardware components should belong to the object-oriented hardware module classes defined by the SCA v2.2. section 4.2.2 e.g. Modem, RF, I/O, GPS, INFOSEC, Power Supply, and Processor. However, this part has been removed from SCA v2.2.2. and replaced up to some extend by the JTRS APIs (e.g. MHAL API, I/O device, GPS device) and used in [OMG07e].

Device Configuration Descriptor (DCD). A device manager is associated with a Device Configuration Descriptor (DCD) file. The DCD identifies all devices and services associated with a device manager by including references to the required SPDs. The DCD also defines properties of the specific device manager, enumerates the needed connections to services (e.g. file systems), and provides additional information on how to locate the domain manager.

Software Assembly Descriptor (SAD). The SAD is an XML-based ADL, which originates from the CORBA Component Assembly Descriptor (CAD). This descriptor specifies the interconnection and configuration of application components.

The SAD references all SPDs needed for an application, defines required connections between application components (connection of provides and uses ports / interfaces), defines needed connections to devices and services, provides additional information on how to locate the needed devices and services, defines any co-location (deployment) dependencies, and identifies a single component within the application as the assembly controller.

DomainManager Configuration Descriptor (DMD). The Domain Manager is described by the DomainManager Configuration Descriptor (DMD). It provides the location of the (SPD) file for the specific DomainManager implementation to be loaded. It also specifies the connections to other software components (services and devices) which are required by the domain manager.

Application Component Model The CF Base Application Interfaces depicted in figure 6.2 constitutes the SCA application component model. Application developers must implement these interfaces in order that application components are SCA compliant. The base interface for application components is the `Resource` interface that inherits from interfaces dedicated to a specific functionality.

Interfaces The `LifeCycle` interface allows one to `initialize` a component instance to a known initial state and to *release* a component instance from the CORBA environment. This interface is an application of the LIFECYCLE CALLBACK pattern [BHS07] The `TestableObject` interface allows one to *run* built-in tests (BIT) to check component implementation behavior as black-box. The `PropertySet` interface provides operations to `configure` a component and `query` its properties. The `configure` operation assigns a sequence of id/value pairs containing configuration `Properties` that may be defined in the Properties file of the Software Component Descriptor. The `query` operation returns a sequence of id/value pairs whose configuration Properties has been requested. The `Port` interface allows one to *connect* and *disconnect* component ports. Each requires Port called *uses* port has a type defined by an application-specific interface that must inherit from the Port interface. The `connectPort` operation of a requires Port associates the port name of a provides Port called `connectionId` to its object reference in order than the requires Port can invoke operations on the provides Port. A requires port may be connected to one or several provides ports using

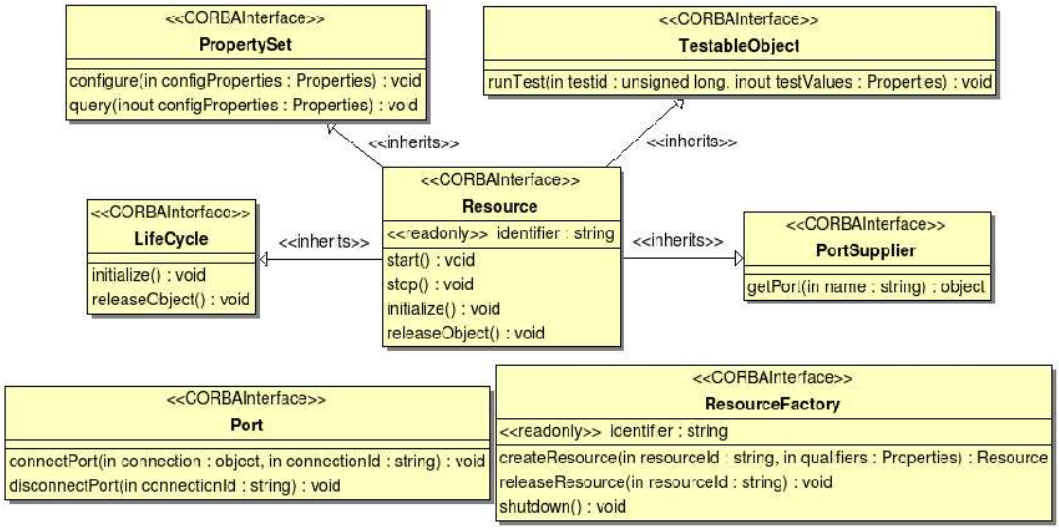


Figure 6.2: SCA Core Framework Base Application Interfaces

different `connectionId`. The `PortSupplier` interface is used to retrieve the object reference of provides and/or requires port by its name. The `Resource` interface inherits from the `LifeCycle`, `PropertySet`, `TestableObject`, and `PortSupplier` interfaces. This interface provides a common API to configure and control a component. It contains a read-only string identifier that uniquely identifies a `Resource` component. It also provides a `start` and `stop` operations to enable and disable its internal processing. The `ResourceFactory` interface allows the creation and destruction of a Resource component. A read-only string identifier is used to uniquely identify a `ResourceFactory` instance. The `createResource` operation allows the instantiation of a Resource component in the same address space as the resource factory or to retrieve the object reference of an already created resource. The `releaseResource` operation removes a given resource from the CORBA environment. The `shutdown` operation removes the resource factory from the CORBA environment. Each operation of Base Application Interfaces may raise associated exceptions that are not described due to space limitation.

Component Definition The SCA does not specify a standard way to define and implement a component. A component may be defined using UML and/or CORBA IDL 2.x. In UML, a SCA component may be defined as a component class that inherits from the Resource interface and may either inherit from the Port interface or contains Port objects. The SCA does not define a component concept using UML extensions such as a SCA UML profile and stereotypes. Such extensions are defined by the UML Profile for Software Radio [OMG07e]. The SCA uses CORBA IDL2.x, which only supports the concept of objects and not the concept of component, which is defined in CORBA IDL3.x. In IDL, a SCA component may be defined as a component interface that inherits from the Resource interface and may either inherit from the Port interface or separate port interfaces may be defined. In summary, a component is defined as an aggregation of objects instead of as a whole due to the lack of component constructs in the chosen versions of UML and CORBA IDL.

Hardware Platform Component Model As presented in figure 6.3, the CF Base Device interfaces constitute the SCA hardware platform component model. The SCA provides the concept of Logical Device to abstract physical hardware devices as software components. As the `Device` interface inherits

from the `Resource` interface, a logical device is also a resource and provides the same attribute and operations. Each device provides its own capacities (e.g. memory, bandwidth), which are allocated during deployment using the `allocateCapacity` and `deallocateCapacity` operations. A device references a Software Package Descriptor (SPD) which defines its connections to Resources and Services, and its configuration and capacity properties.

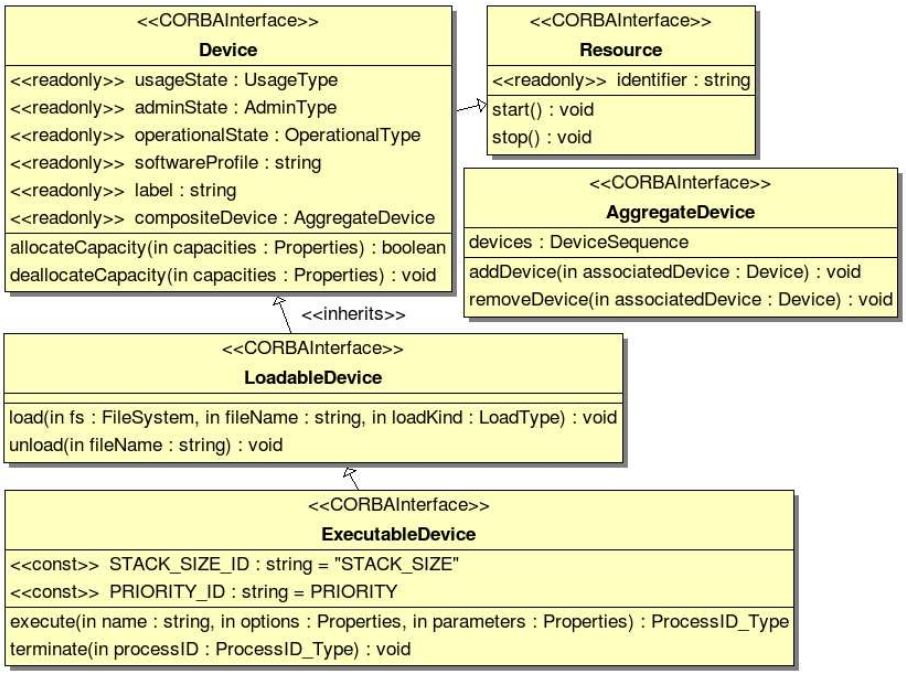


Figure 6.3: SCA Core Framework Base Device Interfaces

The Loadable Device extends the Device interface to provide the capability to `load` and `unload` to/from a device *software* binary artifacts such as kernel modules, drivers, shared libraries or executable files. Loadable Devices have been used to represent FPGAs [PH04] [SMC+07]. The Executable Device extends the Loadable Device interface to allow one to `execute` and `terminate` the execution of a *software* process on a device. An Executable Device may represent *Instruction Set Processors* (**ISP**) such as GPP, DSP, ASIP and soft processor cores.

The `Aggregate` interface allows child devices to register and unregister themselves from a parent device. For instance on FPGAs, an aggregate device may represent a modem device which contains sub-devices such as encoder/decoder and modulator/demodulator IP cores.

Hence, the SCA supports hierarchical composite platform components (`Devices`), but only flat primary application components (`Resources`).

Connection Model The SCA connection model is based on the `Port` and `PortSupplier` interfaces. Application components only interact through their ports whose interconnexions embodies the concept of connector. Ports provide channels through which data and/or control pass. The SCA specifies two kinds of point-to-point ports. A component port that requires services is a *uses port*, while a component port that provide services is a *provides port*. A uses port is defined as an user-defined UML class or IDL interface that inherits from the `Port` interface to be connectable to provides port(s). A provides port is defined as an user-defined UML class or IDL interface. A possible definition of two connected Resource components is presented in figure 6.4.

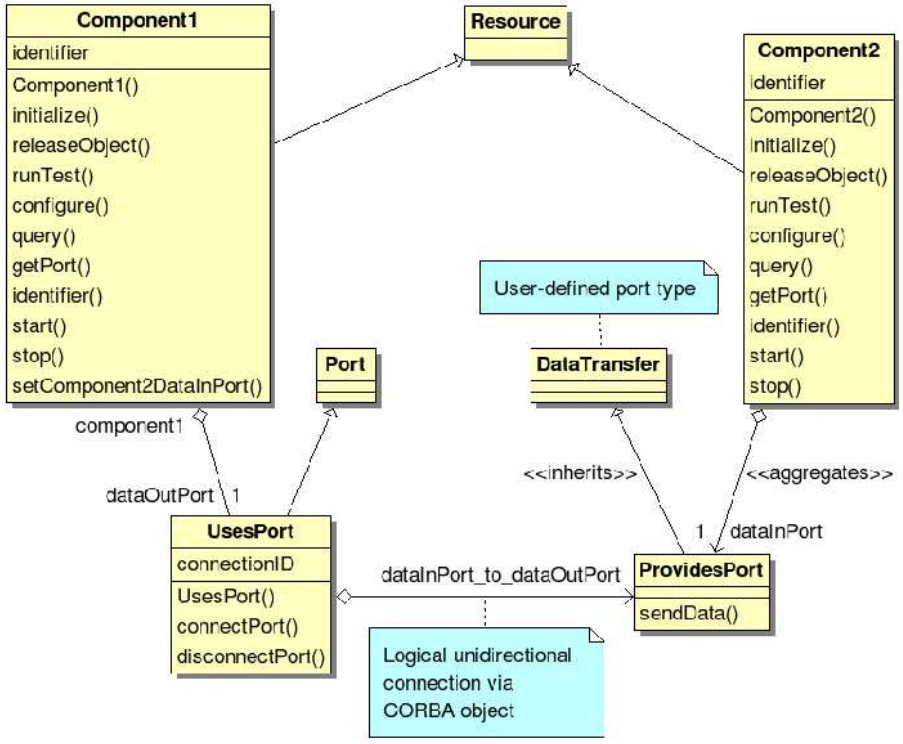


Figure 6.4: Class diagram of two connected Resource components

A connection from an uses port to a provides port is unidirectional from the invocation direction point of view. A uses port plays a client role, while the provides port acts as a server. A logical connection is established when the uses port obtains the object reference of the provides port in order to be able to invoke operations on it.

```
    // In Component1 class, implementation of PortSupplier interface
    CORBA::Object_ptr Component1::getPort(const char *name) {
        if(strcmp(name, "dataOutPort") == 0)
4           return CF::Port::_duplicate(dataOutPort.in()); // return
                dataOutPort reference
        else return CORBA::Object::_nil(); }
    // In UsesPort class, implementation of Port interface
    void UsesPort::connectPort(CORBA::Object_ptr connection, const char *
        connectionId) {
        DataTransfer_var port = DataTransfer::_narrow(connection); // cast
            to the port type
9       component1->setComponent2DataInPort(port.in()); } // whole-part
            relationship

    // In Component2 class, implementation of PortSupplier interface
    CORBA::Object_ptr Component2::getPort(const char *name) {
        if(strcmp(name, "dataInPort") == 0)
14          return DataTransfer::_duplicate(dataInPort.in());
        else return CORBA::Object::_nil(); }
    // In ProvidesPort class, implementation of user-defined DataTransfer
        interface
    void ProvidesPort::sendData(CORBA::UShort data) { //
```

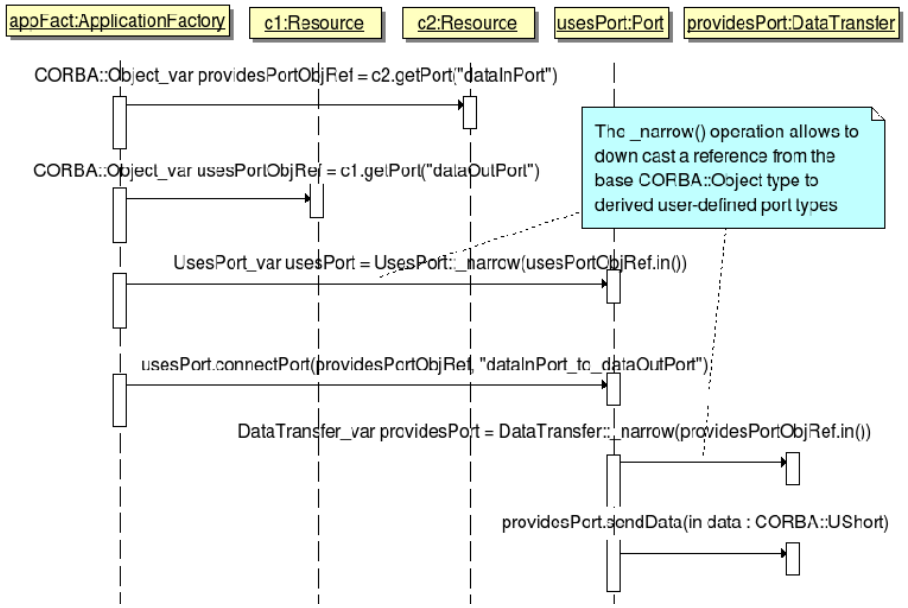


Figure 6.5: Sequence diagram for the connection between two Resource component ports

```
        cout << "component2.dataInPort received: " << dec << data << endl;
    }
```

Listing 6.1: Implementation example for Resource components and ports

The kind of each component port is defined in the Software Component Descriptor. Operations of the uses port interface, which inherits from the `Port` interface, are used to transfer user-defined control and/or data. The logical interconnection between requires and provides component ports is automatically performed by the component framework during application deployment using the Software Assembly Descriptor (SAD) and the Device Configuration Descriptor (DCD).

Deployment Model The Framework Control interfaces and the Domain profile represent the SCA Deployment model. The Framework Control interfaces use the meta-data from the XML files of the Domain Profile to deploy, manage and configure a software application on a radio platform. The Domain Profile embodies the SCA packaging model to package application components and platform devices as deployable software artifacts.

The `Application` interface allows one to control, configure and obtain the status of an deployed application in a SCA domain. It inherits from the `Resource` interface. An `Application` instance is returned by the `create` operation of an `ApplicationFactory` instance. The `Application` delegates invocations on the `runTest`, `start`, `stop`, `configure` and `query` operations to the *Assembly Controller* identified in the SAD file.

The `ApplicationFactory` interface is used to create an `Application` instance. The `create` operation takes configuration properties and a list of assignments of components to devices as parameters and returns an `Application` instance. An `ApplicationFactory` instance controls the deployment of application components on platform devices. When a user invokes the `create` operation on the application factory, it assigns application components on available devices and allocates the required device capabilities (e.g. memory, processor time). Then it loads and executes the application components on the devices using the software profile. As indicated in the SAD, the application factory retrieves the

object reference of a Resource either directly using the Naming Service or indirectly using a registered Resource Factory that will `create`s it. The application factory `initialize`s the obtained Resource, `get`s its Ports and `connect`s them. It `configure`s the assembly controller that can dispatch configuration commands to the assembled components. Finally, the `Application` instance is returned, a message may be writen to the Log service and a event may be pushed to the domain event service.

The `Device Manager` interface allows the management of a set of logical devices and services. A `Domain Manager` instance allows one to `register` and `unregister` devices and services with a device manager, `shutdown` a device manager and `get` the SPD implementation ID used by the Device Manager to create a component.

The `Domain Manager` interface serves for the control and configuration of a SCA domain. A `Domain Manager` instance allows one to `register` and `unregister` device managers with a domain manager, devices and services with a device manager, objects in an given event channel and `install` and `uninstall` an application.

Evolution of the integration of application components on non-CORBA capable processors

Adapters in SCA v2.2 in 2001 Adapters [GHJV95] are `Resources` or `Devices` that transparently convert CORBA invocations to `Resource` or `Device` operations into appropriate actions for application or platform components e.g. Modem and Security components that reside on non-CORBA capable processing units such as DSPs, FPGAs and ASICs. This adaptation allows the integration of business, legacy or COTS components which are implemented in hardware or software with CORBA-capable Resources in an uniform object-oriented approach. However, this integration is manually performed by application developers for each specific component. Resource adapters include dedicated code to control and communicate with non-CORBA capable application components and thus do not meet the portability and interoperability requirements of the SCA. Device adapters do not belong to predefined Device types such as AudioDevice and cannot implement their associated standard APIs. Moreover, these adapters only address the portability and interoperability of software application components, but not of non-CORBA capable application components on DSP, FPGA and ASIC.

Software integration approaches for software components on DSPs and hardware components on FPGAs has been successively proposed in [JTR04], [JTR05] [KB05].

Specialized Hardware Supplement (SHS) to the SCA specification v3.0 in 2004 The SCA specification version 3.0 was proposed to extend the version 2.2 and address the limitations of the SCA regarding the portability of application components on specialized processing units such as DSPs, FPGAs and ASICs. These dedicated processors suffer from a lack of standard operating environments to support portability such as standard OS on DSP and execution models on DSPs, FPGAs and ASICs. The SCA v.3.0 never became a standard release of the SCA and is not anymore available on the SCA JTRS web site [42]. The Specialized Hardware Supplement (SHS) [JTR04] to the SCA v3.0 specified four ingredients to improve the portability of components on non-CORBA capable processing units: a Hardware Abstraction Layer Connectivity (HAL-C), a reduced POSIX Application Environment Profile for DSPs, waveform functional blocks and an API for antenna subsystem.

The Hardware Abstraction Layer Connectivity (HAL-C) defines standard hardware and software communication interfaces between application components deployed on specialized hardware devices i.e. DSP and FPGA.

[42] `http://sca.jpeojtrs.mil`

The *HAL-C Infrastructure* refers to the software or hardware implementations of HAL-C interfaces that provides the operating environment for HCs and supports location transparent communication between HCs residing on the same and/or different PEs.

The HAL-C infrastructure may be considered as a message-passing middleware that supports portability through the HAL-C API like the socket API, but no built-in support for inter-HC communication interoperability using a standard message protocol like CORBA GIOP.

Extension for component portability for Specialized Hardware Processors (SHP) to the SCA v3.1x a.k.a. SCA Change Proposal (CP) 289 in 2005 A container-component approach is proposed in [KB05] for non-CORBA capable SCA components called *worker* on FPGAs and DSPs. We focus on the proposition for hardware components.

Workers have three kinds of ports: a control port, "business" ports associated to a subset of IDL and ports for local services provided by containers. An OCP profile is provided for each port category. Workers produce and consume messages through their ports. Mapping from IDL to OCP in this approach is not systematic since it depends on the meaning of the operations within an IDL interface. SCA control operations (i.e start/stop) are encoded on OCP address signals on thread 0, while invocations without parameter are interpreted as read. Configuration operations (i.e. configure/query) are not seen as invocations, but as regular read/write on the OCP MThreadID 1. Business operations are mapped on OCP Control or Status signals. For business interfaces, there is one port for requests and one port for responses. The logical container invokes and controls one or several components. It provides local services such as Reset, Clock and local memory services. This container only addresses waveform component portability. Interoperability is considered the integrator or platform provider responsibility. Willingly, no standard message format is proposed. A generic software proxy is proposed to adapt SCA Resource interface to a simplified control interface to manage hardware components.

We assume that the message header can be encoded by the container, while message body is encoded with CDR and has to be decoded a priori within the worker instead of within the container. Input and output ports only support read and write operations respectively. Provides component ports receive request messages on input ports and send response messages on output ports. Uses component ports send request messages on output ports and receive response messages on input ports. Port readiness can be indicated using the *SThreadBusy* signal on thread 0. On output ports, operation or exception are encoded on OCP Control signals, on input ports, operation or exception are encoded on OCP Status signals. Signals can be removed for one way operation (response port), single operation (*Control*/*Status*) or FIFO access (*MAddr*). Moreover, the granularity of the data unit transferred through component ports is a request or response message body rather than individual operation parameters.

Modem Hardware Abstraction Layer (MHAL) API in 2007 The SCA MHAL [JTR07] specification defines a standard MOM API and message header for GPP, DSP and FPGAs/ASICs. Hence, the MHAL can be considered as a hardware/software message broker for SDR platforms. The MHAL Logical Destination (LD) correspond to Message Endpoint [BHS07] and notably to HAL-C endpoints.

Development of a new SCA release in 2009 The JPEO JTRS annonced in August 2009 the development of a new SCA release [JTR09] in which the SCA specification will become technology independent and make CORBA optional. The change proposals for this new release notably concern the definition of SCA PIM and PSMs, SCA profiles with lightweight SCA components, operating environments other than for GPPs, real-time implementations.

Implementations

Software Implementations Open source implementations of the SCA Core Framework include SCARI[43] (Software Communications Architecture - Reference Implementation) from the Canadian CRC (Communications Research Centre) which is written in Java and was demanded by the SDRForum and OSSIE[44] (Open Source SCA Implementation::Embedded) from Virginia Tech which is written in C++. The OSSIE Core Framework has been ported on the TI OMAP SoC [Bal07]. Moreover, the SCA CF has been been used in mobile robots [HLEJ06].

Commercial implementations of the SCA Core Framework include Componennt Enabler from Zelig-Soft and Spectra SDR tools from PrismTech.

SCA Overhead Evaluation The overhead of implementing waveform with the SCA has been evaluated in [OS07] compared to a custom waveform implementation, in terms of latency [TBR07] and memory [BDR07].

Hardware Implementations Dynamic and partial hardware reconfiguration of FPGAs has been implemented under the control of the SCA CF [SMC+07]. Hardware implementations of SCA component have been proposed in [Hum06]. Standard interfaces for FPGA components using OCP are motivated in [NK07].

Extension for Quality of Services Some works such as [LPHH05] [LKPH06] have been carried out to extend the SCA CF to support the specification of Quality of Services.

OMG UML Profile for Software Radio

The adoption of the SCA by the SDRForum has contributed to the evolution of the SCA as a commercial standard within the OMG. Following the OMG Model-Driven Architecture (MDA) initiative, the UML Profile for Software-Defined Radio specifies a Platform-Independent Model (PIM) and a Platform-Specific Model (PSM) for CORBA of software radio infrastructures on top of which waveform applications may be developed. This profile is also known as PIM & PSM for Software Radio Components. The separation between the PIM and its implementation in PSM(s) facilitates the portability of waveform applications which may be deployed on various heterogeneous SDR platforms.

The UML Profile for Software Radio (SDRP) extends the UML modeling language with SDR domain specific concepts using stereotypes to model software radio components.

This specification defines a set of *facilities* to model the behavior of SDR systems and specify standardized APIs. A facility is defined as the realization of certain functionality through a set of well-defined interfaces. Facilities are independent from a particular middleware technology such as CORBA.

The SDRP specification is divided into 5 volumes: Communication Channel and Equipment, Component Document Type Definitions, Component Framework, Common and Data Link Layer Facilities, and POSIX Profiles.

The set of stereotypes of the UML profile defined in the Component Framework volume is organize in two main packages: the Applications and Devices package and the Infrastructure package.

The Applications and Devices package defines the concepts for the development of waveform applications and devices *Resource* and *Device* components. The Infrastructure package defines the concepts to deploy services and applications on a SDR platform.

The Application and Device Components package contains 5 packages:

- **Base Types** package defines types for applications, logical devices, and component definitions.

[43] `http://www.crc.ca/scari`
[44] `http://ossie.wireless.vt.edu`

- **Properties** package defines properties for configure, query, testable, service artifact and executable properties for components and executable code artifacts.
- **Interface & Port Stereotypes** package defines stereotypes for interfaces and components.
- **Resource Components** package defines stereotypes for the interfaces and components for the ResourceComponent, which is the basic component type for application and device components.
- **Device Components** package defines stereotypes for the logical device components on which components are deployed or which are provided by the radio platform.
- **Application Components** package defines stereotypes for the ResourceComponents within applications.

Within the UML profile for Software Radio, the UML Profile for Component Framework specification defines UML stereotypes for component property, interface and port. `Interface` stereotypes represent the type of interface provided or used by components through their ports. `Port` stereotypes represent the kind of ports a component may have. `Property` stereotypes represent the type of properties for interface and component attributes.

The `Interface` stereotype defines the following types of interfaces: IControl, IDataControl, IData, IStream and StreamControl. The `IControl` interface allows sending or receiving control. The `IData` interface allows sending data. The `IDataControl` interface allows sending data with control. The `IStream` interface allows managing streams. The `IStreamControl` interface allows controlling streams including user data and control information.

Based on these interfaces, the `Port` stereotype defines the following types of component ports: `ControlPort`, `DataControlPort`, `DataPort`, `ProvidesPort`, `RequiresPort`, `ServicePort`, `StreamControlPort` and `StreamPort`.

The `Property` stereotypes defines the following types of component properties: `Constant`, `ReadOnly` and `ReadWrite`.

The Communication Channel Facilities PIM is a non-normative specification of the physical layer and radio control facilities. It defines Physical Layer Facilities and Radio Control Facilities.

The Physical Layer Facilities specify a set of interfaces to convert the digitized signal into a propagating RF wave and vice versa, and to control and configure HW/SW physical layer components. The Physical Layer Facilities are divided into data transfer facilities, control facilities and I/O facilities. The data transfer facilities are defined in the Common and Data Link Layer Facilities::Common Layer Facilities. A physical layer component must realize i.e. implement data transfer interfaces to communicate with the upper OSI layers. The control facilities contain modem facilities and RF/IF facilities. The modem facilities include all digital signal processing elements required to convert bits into symbols and vice versa. They include baseband and passband digital modulation schemes such as QAM, PSK, Direct Sequence Spread Spectrum (DSSS), Orthogonal Frequency Division Multiplex (OFDM), and analog modulation schemes such as Amplitude Modulation (AM), Frequency Modulation (FM), and Phase Modulation (PM).

The modem facilities defines modem interfaces and components realizing these interfaces such as `PNSequenceGeneratorComponent`, `ChannelCoder`, `BlockInterleaver`, `Mapper`, `SourceCoderComponent` and `ChannelCoderComponent`.

The RF/IF facilities defines interfaces and components realizing these interfaces such as: `AmplifierDevice`, `SwitchDevice`, `HoppingFrequencyConverterDevice`, `FrequencyConverterDevice`, `AntennaDevice`, `DigitalConverterDevice` and `FilterDevice`.

The I/O facilities defines two types of I/O mechanisms: Serial IO and Audio IO. The Serial I/O package defines the `SerialIODeviceComponent` which provides and uses the `SerialIOSignals`,

`SerialIODevice`, `SerialIOControl` interfaces. The Audio I/O package defines an AudioIODeviceComponent that provides and uses the AudioIOControl and AudioIODevice interfaces.

The **ModemComponent** component is an abstract component from which all components in the modem facilities inherit. Modem components are stereotyped as «`resourcecomponent`» to indicate that they could either be implemented in software or in hardware via a «`devicecomponent`» component. ModemComponent provides one `ControlPort` and at least one `DataControlPort` or `DataPort`.

The Radio Control Facilities defines a set of interfaces to manage the radio domain and channels within the radio.

Finally, the Common and Data Link Layer Facilities defines the Quality of Service Facilities, Flow Control Facilities, Measurement Facilities, Error Control Facilities, Protocol Data Unit (PDU) Facilities and Stream Facilities.

GNU Radio

The GNU Radio[45] project, founded by Eric Blossom, provides an open-source software signal processing library to build software-defined radios primarily on desktop computers. A software radio application results from the interconnection of signal processing components. GNU Radio components are implemented in C++ and optimized using SIMD instructions, then they are wrapped in the Python object-oriented scripting language using SWIG[46](Simplified Wrapper and Interface Generator). GNU Radio targets the Universal Software Radio Peripheral (USRP) board which can be connected to a desktop computer via an USB cable (USRP1) or an Ethernet cable (USRP2).

6.6.2 General Purpose Component Models

Sun Enterprise Java Bean (EJB)

The Enterprise JavaBeans (EJB) is a commercial and general-purpose component model for desktop and server-side applications developed by Sun MicroSystems. EJB components are developped in Java. They extend the `EJBObject` base interface to add user-defined interfaces. Required interfaces are not explicitly specified. EJB does not provide the concept of port. EJB components are deployed and executed within EJB containers, which manage component life-cycle (creation/destruction, start/stop, activation/passivation) and mediate access to non-functional platform services such as the Java naming and directory service (JNDI), Java RMI, Java Message Service (JMS), transactions, persistence. Components are created using the *EJBHome* interface implemented by containers . The EJB component model does not support hierarchical composition or dynamic reconfiguration. The EJB deployment model is based on component packages in .jar files called *beans* and a deployment descriptor in XML. EJB does not exist in a lightweight version for embedded systems Moreover, EJB are based on the standard Java platform, which is not appropriate for resource-constrained embedded platforms.

OMG CORBA Component Model (CCM) and its Lightweight version (LwCCM)

The distributed object-oriented model of CORBA evolved to the distributed component-oriented model of the CORBA Component Model (CCM) [OMG06a]. New CORBA IDL keywords have been added to the object-oriented IDL 2.x to form the component-oriented IDL 3.x, which supports CCM. IDL 3.x keywords are intuitive and written in bold in the following.

[45] http://gnuradio.org
[46] http://www.swig.org

Component Model For synchronous or blocking interactions, a CCM component **provides** an interface through a *facet* port and **uses** an interface through a *receptacle* port. A receptacle can be either *simplex* or *multiplex*, if it is connected to a single facet or multiple facet(s). For asynchronous or non-blocking interactions, a CCM component **publishes** events (broadcast) or **emits** events (unicast) through an event source port and **consumes** events via an event sink port. Other OMG specifications propose additional functionalities such as the support for streams and quality-of-service.

The base interface implicitly inherited by all CCM components is the `CCMObject` interface. The `CCMObject` interface inherits from three others interfaces, which provides information on the component. The **Navigation** interface allows a client to retrieve references to the facet ports of a component. The **Receptacles** interface allows one to connect a receptacle to a facet, to retrieve receptacles of a component and existing connections. The **Events** interface allows a component to be connected to send and receive events, and to retrieve consumers, emitters and publishers. CCM components are created through dedicated interfaces called `CCMHome` and `KeylessCCMHome`.

Lightweight CORBA Component Model (LwCCM) The Lightweight CORBA Component Model (LwCCM) is a lightweight profile of CCM which targets distributed embedded systems. In particular, the following features are removed from the general purpose CCM model: Persistence, Introspection, Navigation, Transactions and Security.

Connection Model The CCM offers two kinds of messages: Synchronous messages provided by the communication between facet and receptacles, Asynchronous messages provided by event-based communication.

CCM D&C XML descriptors serve as an ADL and notably specify the configuration, assembly and allocation of CCM components onto execution nodes. CCM is based on a container, which is strongly inspired from the EJB. The CCM container is the execution environment, which transparently manage for component developers the life cycle of the hosted components, the management of events, the use of CORBA middlewares As the full CCM container is more dedicated to large distributed information systems, it may also manage advanced features such as persistence, security and transactions.

Deployment and Configuration Model A mapping from the abstract *Deployment and Configuration* (D&C) specification has been standardized to address the deployment of CCM based applications. The deployment process consist in:

- **Installation**: software packages are installed into the software repository.
- **Configuration**: this step consists in creating multiple assembly of the component-based application.
- **Planning**: according to application requirements, a component implementation is selected and associated to an execution node. The result of the planning is a "deployment plan".
- **Preparation**: this step consists in the installation of the binary files on the execution nodes.
- **Launch**: this step consists in instantiating components, interconnecting and configuring them, and starting application execution.

CCM is based on several XML descriptors:

- the **Component Interface Descriptor** (`.ccd`) contains data from the Component Interface Description. It is created by the *architect* and can be generated from the component IDL.

- The **Component Implementation Descriptor** (`.cid`) describes a single implementation and is created by the *developer* which creates a monolithic implementation, or by an *assembler* that creates a component assembly. It contains information about Component Implementation Description, Component Assembly Description, Monolithic Implementation Description.
- The **Implementation Artifact Descriptor** (`.iad`) contains information from Implementation Artifact Description and is created by the developer of the artifact.
- The **Component Package Descriptor** (`.cpd`) contains data from the Component PackageDescription. It is created by the *packager* that put together one or several implementations.

Implementations Open-source implementations of the CORBA component model include **OpenCCM**[47] written in Java and developed by LIFL, INRIA and Thales in various european research projects, and **MICO-CCM**[48] written in C++ and implemented by Frank Pilhofer. An open-source implementation of the Lightweight CORBA component model is **microCCM** developed by Thales and CEA in the COMPARE project. Commercial implementations of CCM have also been developed such as **MyCCM** from Thales and OpenFusion CCM from PrismTech.

In [JNH06], HW/SW components are described as CCM components to target a custom middleware for robotic. Indeed, the separation of concerns of the component-approach allows one to decouple components from their execution and communication environment e.g. CORBA itself.

Software CCM components have also been simulated in SystemC [KBG+08] [KBY+06]. Some tools have been created for the development, deployment and configuration of CCM components like Cadena[49] from the Kansas State University based on Eclipse and COSMIC[50] from the Vanderbilt University based on Model-Driven Engineering.

Microsoft Component Object Model (COM)

The Component Object Model (COM) is a commercial and general-purpose component model for desktop and server-side applications developed by Microsoft. The COM component model considers components as a set of binary object interface to support interoperability between components written in different language. The object interfaces of COM components are defined in a proprietary Interface Definition Language (IDL) derived from DCE IDL. The COM component model supports a limited form of introspection through interface navigation which allows one to retrieve a pointer to every interface supported by the object. COM technologies include COM, DCOM (Distributed Component Object Model) and COM+. DCOM (Distributed Component Object Model) extends COM to support distributed components. COM+ is an execution environment for COM/DCOM components provided with Windows. COM technologies are being replaced by the .NET framework. OpenCOM[51] is an open source implementation of COM developed by the University of Lancaster.

Fractal

Fractal [Fra08] [BCL+06] is a generic component model developed by France Telecom R&D and the french national institute for research in computer science and control called INRIA[52] . The Fractal component model provides advanced features such as hierarchical composition, introspection and sharing of

[47] http://openccm.objectweb.org
[48] http://www.fpx.de/MicoCCM
[49] cadena.projects.cis.ksu.edu
[50] http://www.dre.vanderbilt.edu/cosmic
[51] http://sourceforge.net/projects/opencom
[52] Institut National de Recherche en Informatique et Automatique

components to represent shared resources such as device drivers. A Fractal component has a content part containing the implementation of primitive components or sub-components for composite components and a control part implementing multiple control interfaces and other non-functional aspects. Fractal defines a component model template which may be customized to meet application needs. For instance, Fractal has been used to build a component-based operating system called Think [FSLM02] and a message-oriented middleware called DREAM [53]. The Fractal component model has been implemented in various programming languages such as Julia in Java, Cecilia and THINK in C, while implementations in .NET and SmallTalk are being developed.

Software Appliances (SOFA)

SOFA and its extension SOFA 2.0 [BHP06] is a component model developed at Charles University in Prague. SOFA supports the hierarchical composition of primitive components to build composite components. Primitive components are directly implemented in a programming language e.g. Java, while composite components are implemented by its sub-components. The SOFA component model defines a component by a *frame* and an *architecture*. The frame represents the black-box view of a component in which its provided and required interfaces are specified. The implementation of a SOFA component consists of a functional part implementing the *business* interfaces and a control part implementing the control interfaces required by the component model. These control interfaces provide non-functional services such as life-cycle, connection, reconfiguration and introspection which are implemented by controllers e.g. a lifecycle controller and binding controller. Controllers are used by the component framework (runtime environment, administration and deployment tools) and by the application business logic for some services e.g. to retrieve component properties. The control part is composed of *micro-components* which are objects implementing a control interface. The SOFA connection model reifies component binding by first-class *connectors* that support four communication styles: method invocation, message passing, streaming and distributed shared memory [BP04]. These communication styles are supported at both design and execution time. In SOFA 2, connectors allow provided-to-provided and required-to-required connections to integrate multiple communication styles. The assembly of components and connectors is described by an Architecture Description Language called *Component Description Language* (**CDL**) in SOFA and by a meta-model in SOFA 2. This meta-model allows the automatic generation of a repository, model editor, code skeletons for primitive components, XML descriptors (e.g. deployment plan) and automatic application deployment. The behavior of SOFA components can be formally specified by behavior protocols based on regular expressions [PV02]. Based on CDL description, the SOFA CDL compiler generates component templates to be filled by the programmer in a target programming language such as C++ or Java. The SOFA deployment model is based on a repository to store component description and implementation, a deployment plan to associate components to execution nodes and to allocate the required resources, and a distributed runtime environment called *SOFA node* composed of component containers called deployment docks. SOFA 2 supports dynamic reconfiguration of software components according to predefined reconfiguration patterns.

6.6.3 Software Component Models for Real-Time and Embedded Systems

Koala

The Koala [vOvdLKM00] component model is used at Philips for embedded software design in resource-constrained systems such as TV sets. Software component interfaces are described by a textual *Interface Definition Language* (IDL). Component configurations are specified by an *Architecture Description*

[53] http://dream.ow2.org

Language (ADL) having both a textual notation called *Component Description Language* (CDL) and a graphical notation. The component model supports *provides* and *requires* interfaces, and hierarchical containment of basic components to build compound components. *Diversity* interfaces are used to configure component properties like application modes. Component properties are required through such interfaces instead of being provided as in other component models such as CCM. Components access all non-functional properties such as memory management and device drivers via *requires* interfaces. The connection model supports both static binding at compile time and dynamic binding at run time. Static binding is implemented using naming conventions and renaming macros. Dynamic binding is implemented using *switches* which dynamically route function calls to the desired components according to its configuration. The deployment model is based on globally accessible interface and component repositories, and configuration management. For each component, the interface repository contains a formal IDL definition and an informal textual description for interface semantics. In addition, the component repository stores for each component a CDL definition, a textual description, and a collection of C files.

PErvasive COmponent Systems (PECOS)

In the PECOS IST project, a component model was proposed for embedded software on field devices [GCW+02]. A field device retrieves physical measures such as temperature or pressure from sensors to control actuators like valves or motors. Such devices have limited computing and memory resources and usually consist of a 16-bit microcontroller with 256KB of ROM and 40KB of RAM. The component model supports hierarchical composition of *leaf* components to build *composite* components. Components may have multiple *data-flow-oriented* ports which represent shared variables. Three types of components are distinguished: active components, passive components and event components. Active components have their own thread of control to schedule their activities and those of their passive subcomponents. Passive components have no thread of control and are executed synchronously for short periods. Event components are components whose behavior is triggered by events and represent hardware devices producing events such as timers, sensors and actuators. The PECOS component model has also a formal execution model based on Petri Nets to synchronize components with different threads of control and enforce their real-time constraints e.g. deadlines. The *CoCo* composition language is a textual Architecture Description Language (ADL) used to specify components, their assemblies and related families of component-based architectures. The PECOS composition tool can formally check component assemblies using semantics rules expressed as Horn clauses, and first order predicate logic. All component instances are statically created and scheduled at system start-up. Components and their instances may have functional and non-functional properties organized as *property bundles* such as initial values for ports, memory-consumption and scheduling information (worst-case execution time, execution period and priority). Component ports are interconnected through **connectors**. To specify component behavior, the PECOS component model defines three operations referred to as *hooks* which are automatically generated by a code generator and filled by component developpers in a target language (C++ or Java). The `initialize` operation is called at system start-up for component initialization e.g. to configure port initial values. The `execute` operation computes the output values from the internal component state and the input values. It is invoked either synchronously by a scheduler for passive components or continuously for active components. The `sync` operation is implemented by active and event components. It is synchronously called by a scheduler to synchronize the data exchange between asynchronous active and event components and the synchronous environment. Families of component-based applications may be specified in CoCo using *abstract* components which define a template architecture implemented by concrete components.

The authors have defined language mappings from CoCo to target languages such as C++ and Java. PECOS components are mapped to classes, while their instances are mapped to the member variables.

The PECOS execution model is mapped to the prioritized, pre-emptive and multithreaded PECOS Execution Environment that provides a common API to abstract the underlying RTOS from C++ and Java. The scheduler is automatically generated from the component timing properties. Active and event components are executed in their own thread, while passive components are executed in the thread of their parent component. Component ports are mapped to getter and setter methods according to their direction. Connectors are mapped to an automatically generated data structure called *Data Store*. The connectors between components in the same thread use the Data Store as a shared memory. The connectors between components in different threads copy the data from the Data Store in one thread to the Data Store in other thread using the `sync` operation for synchronization.

The PECOS approach shows that a component framework may be efficiently implemented on highly embedded and real-time systems using formal methods for component execution and assembly.

BIP (Behavior, Interaction and Priority)

The BIP framework and language address the modeling of heterogeneous real-time components [BBS06]. The BIP component model considers a component as the superposition of three orthogonal layers: **Behavior** described as a set of transitions, **Interactions** between transitions of the behavior using *connectors*, and **Priority** used to select an interaction among the feasible ones. BIP components are *atomic* components whose behavior is specified as a set of transitions with empty interaction and priority layers. Atomic components may have a set of ports used for synchronization, control states to indicate synchronization points, variable to store local data, transitions to represent atomic computation executed if a guard is true. The composition of two component is performed by composing each component layer of the same type two by two. A compound component results from the composition of its sub-components. The connection model supports connectors to specify interaction patterns between atomic component ports based on strong synchronization i.e. rendezvous and/or weak synchronization i.e. broadcast. The BIP execution platform manages the execution of BIP components written in C/C++. The execution may be multithreaded (one thread per atomic component) or single threaded (the thread of execution engine).

6.6.4 Architecture Description Language (ADL)

A component-based application is a component assembly, which is typically specified thanks to an Architecture Description Language (ADL) [MT00]. ADLs provide two main architectural abstractions *component* and *connector* to separate respectively computation from communication. Examples of ADLs include Acme, Darwin and Wright, UML2 and AADL. ADLs may have a textual syntax like XML-based ADLs, a graphical syntax like UML2 or both like AADL. They may model from static configurations to dynamic reconfigurations. Some ADLs such as Acme[GMW00], Darwin and Wright target the modeling and analysis of component-based architectures at design time, but do not provide a support at run-time. Other ADLs such as UML2 and AADL supports also code generation through model transformations.

6.6.5 Comparison of Software Component Models

An important gap exists between the features supported by the industrial component models such as EJB, COM, CCM, SCA, Koala and the academic component models such as SOFA, Fractal and BIP. Industrial component models aim at providing standard and stable component frameworks with less, but mature features. In contrast, academic component models aim at supporting advanced features such as hierarchical composition, behavior specification and verification, dynamic reconfiguration and several interaction semantics. However, even if these features are supported at design-time, there is no or little

support for them at run-time in the academic component frameworks [BHP06]. Recent comparison of software component models have been proposed in the literature such as [BS05] [CCSV07] [LW07].

6.7 Hardware Component Models for FPGAs and ASICs

In this section, we focus on the component models of hardware components on FPGAs and ASICs called *Intellectual Property* (**IP**) cores. We first present hardware components models through the modeling capabilities of hardware description languages such as VHDL and Verilog. Then, we describe the two main connection models of hardware components, which are either centered on bus interfaces or on core interfaces. Moreover, High-Level Synthesis (HLS) of hardware components from software programming languages is described to motivate the need of a highly configuration mapping from object-oriented software interfaces to signal-based hardware interfaces. The packaging model of hardware components is also presented based on an emerging standard called Structure for Packaging, Integrating and Re-using IP within Tool flows (SPIRIT). Finally, the deployment model of hardware components is described through the full static configuration and partial dynamic reconfiguration of FPGAs.

6.7.1 Component Models supported by Hardware Description Languages

VHDL

VHDL [IEE02] was developed in the 1980s under the aegis of the United States Department of Defense (DoD) in a program called Very-High-Speed Integrated Circuits (VHSIC) at the origin of its acronym VHSIC HDL (VHDL). It was first standardized by the IEEE in 1987. New versions of the language were released in 1993, 2000 and 2002. VHDL is commonly used in Europe, but also in the United States. VHDL is a concurrent language with strong and static typing whose syntax is inspired by the Ada programming language. Digital hardware is modeled in VHDL by `component`s communicating through wires represented by `signal`s. A VHDL `component` is a black box with a hardware interface declared in an `entity` which consists of input and output `ports`. A component `entity` is associated with one or several component implementations defined by an `architecture`. The association between an `entity` and an `architecture` to build a `component` is declared in a `configuration`. VHDL supports modularity through `package`s, `function`s and `procedure`s. A `package` contains globally defined data such as data types and subprograms Declarations of `entity`, `package` and `configuration` are put together into a `library`. A `library` represents a directory on the development workstation in which compiled hardware components are stored. [Smi98] A `package` may have a `package` body for the declaration of subprograms.

Listing 6.2 provides an overview of the VHDL syntax for a simple hardware counter and its testbench used to simulate and test its implementation.

```
library ieee;
use ieee.std_logic_1164.all; -- required packages
use ieee.numeric_std.all;
-- Interface declaration
entity counter is
    generic( COUNT_WIDTH: natural ); -- entity parameter
    port( reset           : in  std_logic;
          clock           : in  std_logic;
          clock_en        : in  std_logic;
          count           : out std_logic_vector(COUNT_WIDTH-1 downto 0));
end;
-- Interface implementation
```

```
architecture counter_behavior of counter is
    -- shared signal inferred as register
    signal value: unsigned(COUNT_WIDTH-1 downto 0) := ( others => '0' );
begin
  -- sequential execution statement
    cnt_proc: process(reset, clock, clock_en)
    begin
        if reset = '1' then
      value <= ( others => '0' );
  elsif rising_edge(clock) then
      if clock_en = '1' then
          value <= value + 1;
      end if;
  end if;
    end process;
    -- concurrent assignment statement
    count <= std_logic_vector(value);
end;

entity counter_testbench is
    generic(N: natural :=4);
end entity counter_testbench;

architecture behavior of counter_testbench is
    -- component declaration
    component component_counter is
    generic(COUNT_WIDTH: natural);
    port( reset           : in  std_logic;
          clock           : in  std_logic;
          clock_en        : in  std_logic;
          count           : out std_logic_vector(COUNT_WIDTH-1 downto 0));
    end component;
    -- component configuration
    for dut_inst: component_counter
        use entity work.counter(counter_behavior);
    -- component configuration
    for test_inst: component_test_counter
        use entity work.test_counter(test_behavior);
    -- . . .
    -- 'wire' declaration
    signal reset_s      : std_logic ;
    signal clock_s      : std_logic ;
    signal clock_en_s   : std_logic ;
    signal count_s      : std_logic_vector(N-1 downto 0);
begin
    -- component instantiation and interconnection
        counter_inst: component_counter
        generic map ( COUNT_WIDTH => N)
        port map (reset    => reset_s,
                  clock    => clock_s,
                  clock_en => clock_en_s,
                  count    => count_s);
    -- . . .
end architecture;
```

Listing 6.2: A counter in VHDL

Concurrent behavior is basically implemented either in sequential `process`es or in concurrent statements. Concurrent statements and processes are simulated concurrently, but run in parallel on FPGAs/ASICs. The simulation model of a process is an infinite loop, whose execution is conditioned by discrete events to which the process is sensitive and that are declared in a *sensibility list*. In addition to user-defined signals, `process`es are typically sensitive on a reset signal to initialize signal values and a clock signal to synchronize their execution. Intra-communications in component between processes and concurrent statements is performed through shared signals declared in the architecture.

The current revisions of VHDL called VHDL-200x are focused on verification [BMB+04]. They should include abstract data types like FIFOs, lists and queues to ease testbenchs writing, fixed and floating point packages [Lew07] and assertion-based verification based on the Property Specification Language (PSL), The VHDL Analog and Mixed-Signal (AMS) language extends VHDL to support the description and simulation of analog, digital, and mixed-signal heterogeneous circuits.

Verilog

Verilog was the first modern HDL created in 1985 by Gateway Design Automation. Its syntax is inspired by C. Verilog and VHDL share the same basic hardware constructs of modules, ports and wires. Verilog seems easier to learn than VHDL, because it is less verbose and typed, but it provides less constructs than VHDL for large scale development e.g. architectures and configurations. Listing 6.3 presents the same counter example in Verilog.

```
module counter (reset, clock, clock_en, count);
    parameter COUNT_WIDTH = 3;
  input clock;
  input reset;
  input ena;
  output [COUNT_WIDTH-1:0] count;
  reg [COUNT_WIDTH-1:0] value;
  wire count;
  // sequential execution statement
  always @(posedge reset or posedge clock)
  begin
    if (reset)
      value = 0;
    else if (clock_en)
      value = value + 1;
  end
    // concurrent assignment statement
  assign count = value;
endmodule
```

Listing 6.3: A counter in Verilog

SystemVerilog

Verilog has been enhanced with object-oriented extensions to build SystemVerilog 3.x [Acc04b] to facilitate the verification of abstract hardware models. As Verilog syntax is inspired by the C language,

SystemVerilog naturally provides OO extensions close to C++. A *class* construct was introduced to declare instance variables called *properties* and member functions named *methods*. SystemVerilog supports class inheritance, polymorphism and dynamic allocation of classes, but the synthetizable semantics of these advanced features are not defined.

6.7.2 Connection Model for hardware

Two main integration approaches are usually distinguished concerning the interfaces of IP cores: bus-centric and core-centric approaches [CLN+02].

Bus-Centric Interfaces

In the bus-centric approach, hardware component interfaces conform to a computer or SoC bus specification such as ARM AMBA, IBM CoreConnect or Altera Avalon for virtual IP core components, or PCI Express or USB for physical components. Dedicated wrappers must be developed or bought to reuse existing IPs on another bus. Such an approach is not satisfactory for a standard mapping for IDL to HDL.

Core-Centric or Socket-based Interfaces

In the core-centric or socket-based approach, hardware component interfaces conform to a hardware interface specification such as VCI [Gro01] or OCP [OI06]. Socket-based design allow designers to decouple IPs cores from each other and from the on-chip interconnect. Sockets favor reuse and portability of IP cores. Interface adapters are required to convert the common interface and protocol to proprietary interfaces and protocols. However, this adaptation is inferior or equal to the efforts required by the bus-centric approach, as the socket interface is designed to only capture the communication needs of each IP core [OIa] [OIb]. The core-centric approach is compatible with the portability requirement of hardware business components, that is why we propose in chapter 7 an IDL-to-HDL mapping inspired by the socket-based approach.

6.7.3 High-Level Synthesis (HLS)

With equivalent functionalities, writing HDL code is often much more verbose, error-prone and time consuming than coding in a high-level software programming languages. This productivity gap is similar to the use of assembly languages instead of high level languages like C. The main reason is the low level of abstraction offered by HDLs. Since the 2000s, numerous works have thus proposed to translate high-level languages into HDLs through *High-Level Synthesis*(**HLS**). Thanks to the maturity of HLS tools, HLS recently seems to have gained more acceptance in the electronics industry. Advances in HLS allow to obtain comparable or even better results in terms of speed and area than handwritten HDL code, since designers may focus on design space exploration instead of HDL coding.

Numerous commercial tools perform high-level synthesis mostly from C-like languages such as Cynthesizer [Sys] from Forte Design Systems, PICO [Syn] from Synfora Inc., CoDeveloper [Tec] from Impulse Accelerated Technologies Inc., DK Design Suite [Sol] from Agility Design Solutions Inc., C-To-Hardware [Alt] from Altera and Catapult C [Gra] from Mentor Graphics Corp, CyberWorkBench [NST] from NEC System Technologies Ltd., XtremDSP from Xilinx and Bluespec SystemVerilog (BSV) [Blu] from Bluespec Inc.

There are also various academic tools such as Stream-C [FGL01] from Los Alamos National Laboratories, ROCCC [GMN06] from the university of Riverside, SA-C [RCP+01] from the university of Colorado, SPARK [GDGN03] from the university of San Diego and GAUT [CCB+06] from Lester in France. Some of these commercial and academic tools are presented in [CM08].

HLS from sequential languages

G.A.U.T. GAUT[54] [CCB+06] is an academic HLS tool targeting digital signal processing applications. GAUT generates RTL VHDL and cycle-accurate SystemC for the SoCLib[55] virtual platform from a bit-accurate algorithmic specification in C/C++. After choosing a library of hardware operators characterized for specific FPGA/ASIC technology targets and compiling the C/C++ code, GAUT generates a control-data flow graph describing the dependencies between input/output data and hardware operators like multipliers and adders. Designers may specify hardware synthesis constraints such as clock period, cadence i.e. throughput, target technology, allocation strategy (e.g. distributed or global) and scheduling strategy. (e.g. As Soon As Possible (ASAP) or no pipeline). Designers may also specify I/O timing and memory mapping constraints. Memory constraints allow assigning variables and constants in registers or in memory banks at user-defined addresses and with initial values. After synthesis, a GANTT diagram presents the scheduling of accesses to operators, variables and registers. The architecture may be automatically pipelined. The generated accelerator may be simulated with ModelSim. The architecture of the produced accelerators contains of a functional *Processing Unit* (**PU**), a *Memory Unit* (**MU**), a *Communication and Multiplexing Unit* (**COMU**) and a GALS/LIS[56] interface. The interface of a PU has clock, reset, enable and internal input/output data signals. The PU consists of operators, registers and FSM with Datapath (FSMD). The MU has the same type of interface and contains RAM(s), address generators, mux, read and write registers and tri-states connected to the internal input/output data buses. The COMU has also the same type of interface and provides external signals (clock, reset and I/O ports). The generation of the memory unit is configured with the assignment of variables to memory banks, their address and initial values of constants. The generation of the configuration unit is configured by choosing the number of dedicated I/O ports, their latency and by selecting between different kinds and role of hardware interfaces (LIS master, LIS slave, FIFO and Sundance[57]-specific interfaces).

HLS from concurrent languages

Instead of using sequential C code, some HLS tools such as Cynthesizer and BlueSpec use natively concurrent languages resp SystemC and BlueSpec SystemVerilog (BSV).

Forte Cynthesizer Cynthesizer [Sys] [CM08] from Forte Design Systems is a HLS tool which takes as input SystemC code and generates SystemC RTL and Verilog RTL code. As opposed to ANSI C, SystemC allows designers to explicitly describe concurrency. Supported SystemC constructs include `SC_CTHREAD`, `SC_METHOD`, `MODULE` and `sc_in`/`sc_out` ports. Designers must implement the communication protocol in SystemC RTL using sc_in/sc_out ports. The member variables in MODULEs are inferred as storage elements. The supported data types include `sc_int`/`sc_bigint`, fixed-point and floating-point types. Examples of generated designs go from control-oriented applications such as USB/SATA controllers to signal-processing IPs. Cynthesizer also supports architecture exploration. To optimize the synthesis, each operator such as multipliers has been characterized in terms of timing in particular latency, area and power consumption.

BlueSpec BlueSpec HLS tool relies on a formal and parallel language called BlueSpec SystemVerilog (BSV). This language is inspired from works achieved in concurrent programming, notably the Haskell functional language, and Hoe and Arvind's works [HA04] at MIT on Term Rewriting Systems (TRS) for

[54] http://www-labsticc.univ-ubs.fr/www-gaut
[55] https://www.soclib.fr
[56] Latency-insensitive system
[57] http://www.sundancedsp.com

hardware design. BSV is based on the concept of parallel transactions. As in transaction middlewares, an atomic transaction is a collection of sequential actions which are either all executed (commit) or not at all (cancelled or rollback). As opposed to blocking synchronization based on locks using mutex and semaphores, transactions support non-blocking synchronization avoiding deadlocks. Atomicity of transactions allows safer composition and better scalability than the lock-based approach. The BSV language allows the definition of guarded atomic methods whose execution is conditioned by execution rules such as empty/full FIFO. BlueSpec automatically generates the control logic to shared resources. The hardware interface of method invocations contains an input `enable` signal, an output `ready` signal, input signals for method parameters and output signals for results. Implementations in BSV can be parametrized by generic data types similar to templates and module composition. Methods may be regrouped into interfaces provided and required by modules. IP cores in Verilog can be imported in the tool and converted to BSV description to ease IP reuse. The tool has been used to design control-oriented applications such as processor, DMA, peripheral devices and data-oriented applications such as H264, OFDM 802.11a (WiFi) and 802.16 (WiMax) [58].

6.7.4 Hardware Interface Synthesis

The application of the object model downto hardware presented in chapter 4.2.3 shows that a single kind of hardware interfaces, i.e. write-only interfaces, is proposed for hardware objects and such an interface represents one point in the mapping space. Works achieved in hardware interface synthesis [RBK00] motivate the need for more flexibility when mapping object interfaces to hardware interfaces. In [SKH98], the following classification of hardware interfaces is proposed **send and forget** interfaces for data buses, **strobe** interfaces to indicate when data are valid on a data bus, **handshake** interfaces and **FIFO** interfaces. In Mentor Catapult®tool [Gra07], designers explicitty associate to one or several parameters of a C function hardware interfaces similar to the ones proposed in [SKH98]. Input function parameters are read, while output parameters are written. In Altera C2H ®compiler [Alt07b], the generated hardware accelerators have read-only Avalon Memory-Mapped ports for input parameters and write-only ports for output parameters.

Position of this work

Classical works have been carried out in hardware interface synthesis [RBK00] to generate the hardware component interfaces such as network interfaces in MPSoCs [GRJ04] or the wrappers to adapt the interfaces of hardware components [PRSV98] [DRS04]. However, all these works are based on low-level interface specification typically at RTL level, while we target high-level interface synthesis and interface refinement.

An IDL-to-VHDL mapping should generate standard core interfaces to support reuse. In addition to the low-level contract of interconnection protocols, a higher level approach requires a functional contract, which may be specified in IDL and formally mapped onto a message whatever the abstraction levels. Our proposition consists in mixing the higher semantics level of hardware components and the flexibility of HLS.

There are still open questions regarding HLS tools such as the portability of C-like source codes from one tool to another due to the use of proprietary libraries, the lack of a standard mapping from C-like languages to HDLs, and the integration of the generated hardware modules within existing SoCs on FPGAs.

[58] http://csg.csail.mit.edu/oshd/index.html

6.7.5 Packaging Model

An emerging standard packaging model for hardware components at different abstraction level is the IP-XACT specification. IP-XACT is a set of specifications proposed by the SPIRIT (Structure for Packaging, Integrating and Re-using IP within Tool flows) [SPI08] consortium, which is composed of EDA vendors such as Cadence, Mentor Graphics and Synopsis, and end-users such as ARM, ST, NXP and TI. The objective of this consortium are to improve IP reuse and enables design automation. Another objective is that the same hardware component could be described at several abstraction levels, notably TLM and RTL, that offer different views according to the level of details required by the design phase (design, validation, integration) [Ghe06]. The IP-XACT specification defines XML files formats (XML Schema) to describe the characteristics of Intellectual Properties such as hardware component interfaces (signal direction and width, bus interfaces), registers (address, width and bitfields) and the interconnection between hardware components (business components and bus components).

An API called *Tight Generator Interface* (**TGI**) is also specified to make this information (meta-data) accessible to generation tools. Based on this API, a chain of generators can be developed to automate and customize each phase of enterprise design flow notably the documentation, configuration and integration of IP blocks. These generators may be scripts or software programs that instantiate hardware components in a top-level component and generate the required interface and protocol adapters and drivers. These specifications are freely available and are studied by the IEEE Working Group P1685 for standardization. We consider that the IP-XACT specification could correspond to a Platform Specific Model (PSM) for the Deployment and Configuration (OMG D&C) of hardware components. The IP-XACT API allows access to repositories of hardware components. Compared to software packaging models, SPIRIT does not define the location of hardware artifacts such as bitstream.

6.7.6 Deployment Model

Hardware IP core components can be either statically deployed on an FPGA at reset-time or dynamically deployed at run-time. Static FPGA reconfiguration means the complete configuration of the FPGA chip using a total bitstream. In dynamic partial reconfiguration, IP cores belonging to the so-called static part within the FPGA may be still running, while other IP cores are being physically configured by a partial bitstream.

FPGA devices are configured by loading a set of configuration bits referred to as a *bitstream* file into the SRAM-based configuration memory of the FPGA. As the SRAM[59] memory is volatile i.e. its content may be lost when the memory is not powered, FPGA devices need to be configured at each power-up.

Static Total Reconfiguration of FPGAs

Static total reconfiguration of FPGAs is typically triggered outside the FPGA by a third party e.g. the end users when the chip is reset either when powered-up (a.k.a. power-on reset) or while running. Another classical way to totally reconfigure an FPGA is via a JTAG [60] port but this is only used by the designers for debug purpose and not by the end users during system operation. Usually, the total bitstream is persistently stored in an on-board EEPROM (Electrically-Erasable Programmable Read-Only Memory) which may be programmed by FPGA vendor programmer tools such as Xilinx iMPACT or Altera Quartus Programmer.

As the reconfiguration time is proportional to the bitstream size, the width of the physical configuration bus and the bitstream transfer rate, a total reconfiguration of an FPGA is time consuming. Moreover,

[59] Static Random Access Memory
[60] Joint Test Action Group

a total reconfiguration of an FPGA does not satisfy the system availability requirement of some real-time and critical applications [SMC+07].

Dynamic Partial Reconfiguration (DPR) of FPGAs

Despite the intrinsic capability of FPGAs to be partially reconfigured at run-time, the dynamic partial reconfiguration is industrially supported only recently by the design tools of a single FPGA vendor, Xilinx, due to its growing need in professional applications for signal and image processing.

Dynamic partial reconfiguration may be triggered from outside the FPGA via an external bus transferring reconfiguration messages or from inside the FPGA by a software or hardware entity. The partial bitstream can be stored outside the FPGA in a local persistent storage like a hard disk drive or a memory card, or a remote persistent storage like a bitstream server [BDGC08]. The partial bitstream may then be transferred via a DMA controller from persistent storages in the main memory of an FPGA board to perform the partial reconfiguration.

The main benefits of the dynamic partial reconfiguration of FPGAs is the reduction of size, cost and power consumption of an FPGA chip [DPML07]. To some extend, partial reconfiguration offers the same benefits than software reprogrammation such as easier maintenance (e.g. bug fixing), on-demand loading of hardware components, extensibility and flexibility.

Dynamic partial reconfiguration of FPGAs provides the opportunity to support the dynamic deployment of both software and hardware waveform components in FPGAs according to the operational needs under the coherent control of a component framework such as the SCA Core Framework.

We proposed in [GMS+06] a system-level design flow taking into account the partial and dynamic reconfiguration of hardware modules. Other SystemC methodologies like [QS05], [SON06] and [BKH+07] have been proposed to model and simulate reconfigurable systems.

Dynamic Partial Reconfiguration Capability of Xilinx FPGAs

As an example, the Virtex family of FPGAs from Xilinx supports the dynamic partial reconfiguration of FPGAs via the ISE and Plan Ahead design tools. Virtex-II Pro, Virtex-IV and Virtex V FPGA devices contain an on-chip *Internal Configuration Access Port* (**ICAP**) [Xil07] which provides the user logic with a hardware configuration interface to the FPGA internal configuration logic. The ICAP interface allows the user logic to write a partial bitstream to the configuration memory of reconfigurable areas and thus to perform the partial reconfiguration. It also allows designers to read back the content of the configuration memory e.g. to save the configuration context. Figure 6.6 presents the interface of the ICAP_VIRTEX2 primitive which must be instantiated in the user design to be able to use the reconfiguration capability of Xilinx FPGAs.

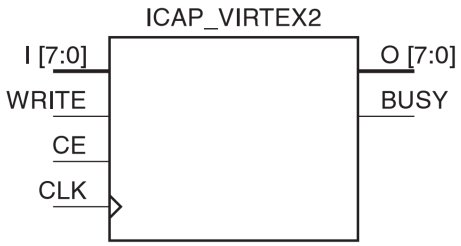


Figure 6.6: Internal Configuration Access Port (ICAP) [Xil07]

The ICAP interface consists of a clock (CLK) signal, a Clock Enable signal (CE), a read-write control signal (WRITE), a write data bus (I[7:0]), a read data bus (O[7:0]) and a status signal (BUSY) for read

commands. The data bus widths of the ICAP interface can be configured to 8 bits in Virtex II FPGAs, 8 or 32 bits in Virtex IV FPGAs and 8, 16 or 32 bits in Virtex V FPGAs. Hence, ICAP has a simple hardware interface which can be wrapped to any bus protocol and used by any hardware component (IP core) or any software component using software drivers. ICAP is a shared hardware resource which must be guarded by an HW/SW arbiter to serialize the reconfiguration requests. In summary, ICAP provides access to the dynamic reconfiguration service of Xilinx FPGAs. This service may be used during the deployment of hardware components under the control of component frameworks such as the SCA Core Framework.

In February 2006, Xilinx and ISR Technologies announced a SDR kit supporting a SCA-enabled SoC and the partial reconfiguration of Virtex IV [61]. The partial reconfiguration is controlled by a SCA CF running on an embedded PowerPC 405 processor. The PowerPC executes the SCA Operating Environment (OE) with the POSIX-compliant Integrity RTOS from Green Hills, the ORBexpress CORBA ORB from Objective Interface Systems (OIS) and the SCA Core Framework from the Canadian Communications Research Center (CRC). This kit includes two modem applications implementing a wideband waveform for video communications and a narrowband waveform for voice communications. A similar experiment was conducted in [SMC+07] [MMT+08] with a different OE. This experiment is described in the following.

Example of Dynamic Partial Reconfiguration of FPGAs under the control of the SCA Core Framework In [SMC+07], a partially reconfigurable hardware module was dynamically deployed on a Virtex IV FPGA device under the control of the SCA Core Framework.

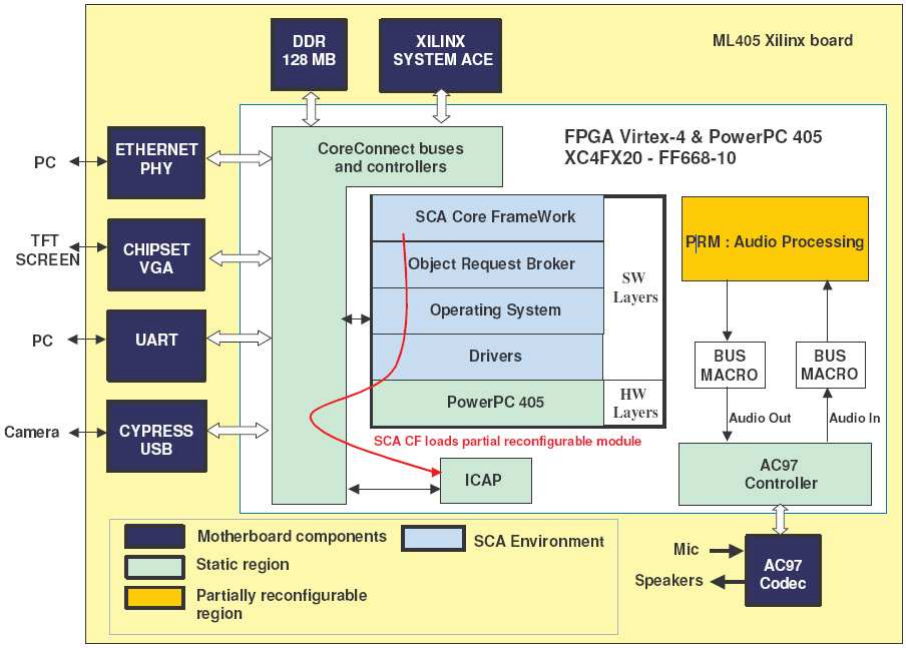


Figure 6.7: Dynamic partial reconfiguration by a SCA Core Framework [SMC+07]

Figure 6.7 illustrates the HW/SW SoC architecture of the demonstration platform. This platform is based upon a ML405 evaluation board from Xilinx. This board contains a Virtex-4 FX20 device including a PowerPC 405 hard processor core with 128MB DDR RAM, several I/O components for

[61] http://www.xilinx.com/prs_rls/dsp/0626_sdr.htm

communications such as USB chip and Ethernet PHY transceiver, and audio and video facilities via VGA[62] and AC[63]97 chips.

Hardware architecture The hardware architecture consists of a CoreConnect On-Chip Peripheral Bus (OPB) bus on which are connected an ICAP controller and various IP cores such as an AC97 controller to control external I/O chips.

The embedded PowerPC manages the dynamic partial reconfiguration through the ICAP controller driver, the continuous display of a video on a TFT screen connected to the VGA chip and the communication with a GUI residing on a laptop via an Ethernet cable. A microphone and a speaker are connected to the board.

An audio processing module is implemented as a dynamically and partially reconfigurable module. The user may choose via the GUI the audio effects to be applied on the voice which is captured by the microphone. This choice triggers the partial reconfiguration of a dynamic reconfigurable area with the partial bitstream of the audio processing module.

Software architecture The software architecture of the demonstrator is compliant with the SCA specification [JTR01] The SCA OE is based on the MontaVista[64] Linux Real-Time Operating System, the open-source MICO[65] CORBA ORB and Thales SCA-compliant Core Framework.

The static bitstream of the hardware platform components (i.e. the OPB bus and the I/O IP cores), the dynamic bitstreams of the hardware application components (i.e. the audio processing module), the sofware image of the OE and the software application components are stored on a Compact Flash card using System ACE[66]. The static bitstream and the sofware image of the OE are loaded on the FPGA when the board is powered up. During application deployment, the Core Framework dynamically loads the partial bitstream stored in the Compact Flash and writes its content to the ICAP controller using the ICAP driver appearing as a Linux kernel module.

The SCA Core Framework manages both the deployment of software application components on the PowerPC processor and the deployment of the hardware audio component in the reconfigurable area. The PowerPC processor is abstracted by a GPP Executable Device which provides an OS-independent API to load and execute the software application on the processor. The reconfigurable area is abstracted by an FPGA Loadable Device which provides an FPGA-independent API to load a partial bitstream in a reconfigurable region.

An open design decision about the software architecture concerns the granularity of an FPGA logical device i.e. the number of reconfigurable areas which must be assigned to an FPGA logical device. Indeed, a SCA logical device may be assigned to all reconfigurable areas or to each area. In this case, the number of logical device instances depends on the number of reconfigurable areas. In the next chapter, we will consider that an FPGA logical device for all reconfigurable areas is a better solution as 1) a single logical device coherently represents the configuration logic (e.g. the ICAP controller) as a shared resource which can only be used by a single software driver and can only satisfy a single reconfiguration request at a time, and 2) the bitstream is self-contained and **self-deployable** as it contains a header with addressing information indicating the area to be reconfigured. The main avantage is that there is always a single FPGA logical device instance regardless of the number of reconfigurable areas. Hence, the software architecture is independent from the hardware reconfiguration architecture and both

[62] Video Graphics Array
[63] Audio Codec
[64] http://www.mvista.com
[65] http://www.mico.org
[66] Advanced Configuration Environment

architectures may be designed in parallel and seamlessly integrated. An FPGA Device is an FPGA-specific platform component which may be developed once and reused multiple times with different waveform applications as it implements the standardized Loadable Device API.

6.8 System Component Models

In this section, we presents system component models such as the SystemC component model at different abstraction levels and actor-oriented components.

6.8.1 SystemC components

SystemC may be seen as a HW/SW component model in which SystemC modules must implement standard C++ interfaces. A SystemC component is an `sc_module` with required ports called `sc_in` and `sc_export` and provided ports called `sc_port`. Ports are bound to object-oriented interfaces called `sc_interface`. These interfaces may model from hardware interface at register transfer level to read from and write to signals to logicial interface at functional level to invoke services on another components. The counter example is presented in SystemC at Register Transfer Level in listing 6.4 and at Functional Level in listing 6.4.

```
#include <systemc.h>

const int COUNT_WIDTH=8;
SC_MODULE(counter)
{
  sc_in<bool> reset;
  sc_in<bool> clock;
  sc_in<bool> clock_en;
  sc_out<sc_uint<COUNT_WIDTH> > count;
  sc_uint<COUNT_WIDTH> value;

  void cnt_proc()
  {
    if(!reset) {
      value = 0;
    } else if(clock_en) {
      value++;
    }
    count = value;
  }

  SC_CTOR(counter)
  {
    SC_METHOD(cnt_proc);
    sensitive << reset.neg() << clock.pos() << clock_en.pos();
  }
};
```

Listing 6.4: Counter in SystemC at Register Transfer Level

```
  class Count : public virtual sc_interface {
    public:
    virtual void count(int value&)=0;
```

```
  };
  class Client : public sc_module {
  public :
    sc_port<Count> pC;
    int value;
    SC_HAS_PROCESS(Client);
    Client(const sc_module_name &n);
    void invoke_service() {
      pC->count(value);
      }
  };
  class Connector : Count, sc_module {
    sc_port<Count> pC;
    sc_export<Count> pS;
    OctetSequence s;
    SC_CTOR(Connector) : pS("pS") {
      pS( *this );     // bind sc_export->interface by name
    }
    void count(int value&) {
        pC->count(value);
    }
  };
  class Server : Count, sc_module {
    public:
    sc_export<Count> pS;
    SC_CTOR( Modem ): pS("pS") {
      pT( *this ); // bind sc_export->interface by name
    }
    void void count(int value&) {
      value++;
  };
};

SC_MODULE(Top)
{
  Client client;
  Connector connector;
  Server server;
  SC_CTOR(Top) :
      client("Client"),
    connector("Connector"),
    server("Server")
  {
    client.pC( connector.pS );
    connector.pC( server.pS );
  }
};
```

Listing 6.5: Counter in SystemC at Functional Level

A SystemC connector may be SystemC channels `sc_channel`, transactors, bus functional models and OSSS channels [GBG+06].

However, the SystemC component model does not support a packaging and deployment model, since it targets centralised system modelling and simulation.

6.8.2 Actor-Oriented Components

To bridge the conceptual gap between signal-based hardware design and method-based software design, Lee et al. propose the concept of *actors*. Actors are abstract system components which are concurrent and reconfigurable. They provide a data-flow oriented interface composed of ports and configurable parameters. A hierarchical composition of actors is called a *model*. An actor provide an executable behavioral specification based on operational semantics. The component model of actor-oriented components are model of computations. Each model of computations contains operational rules which govern the execution of a model in particular when components perform computation, update their state and perform communication. The Ptolemy II tool supports the hierarchical composition of heterogeneous models of computation.

The concept of Lee's actor is related to the actor concurrency model introduced by Hewittt and developed by Agha. The differences between Lee's and Agha's actors are that Agha's actors are active components with their own thread of control and communicate through asynchronous message passing, while Lee's actors are passive components without their own thread of control and communicate through asynchronous or synchronous message passing.

The differences between object-oriented components and actor-oriented components are that object interfaces do not describe explicitly time, concurrency, communication and usage patterns as opposed to actors. The differences between signal-based hardware components and actor-oriented components are that patterns of communication and interaction are difficult to specify in HDLs. The set of signals provided and required by hardware component interfaces describe a low-level transfer syntax but not the high-level semantics needed for component composition.

For Lee et al., software methods represent a *transfer of control* and sequential execution, while hardware signals represent a *transfer of data* and parallel execution. We consider this separation as too simplistic. Any application has both control-oriented and data-oriented processing at various degrees. We consider that software methods and hardware signals may represent a transfer of control, a transfer of data or a transfer of control and data. Transfer of control does not mandatorily imply loss of control from the caller/master to the callee/slave [MMP00]. Software methods may be blocking or non-blocking using *future* variables to perform computation and communication in parallel. Hardware interfaces typically contain control and/or data signals to implement synchronization and communication protocols.

However, the refinement and generation of hardware and software components from actor-oriented models are not described in details as far as we know. While actor-oriented components have data ports, object-oriented components have higher-level object-oriented interfaces that provide functional services. In contrast to the execution model of actor-oriented components based on formal models of computation, the object-oriented component approach relies on more models for component, connector, configuration, packaging and deployment. In this work, we focus on the interface refinement of abstract object-oriented components to synthetizable hardware object-oriented components.

6.9 Conclusion

In this chapter, we presented the main concepts underlying the component oriented approach for software and hardware engineering.

Hardware components provide and require low-level signal-based interfaces, which consist of input, output and input/output RTL ports. Software components provide and require high-level method-based interfaces, which consist of operations with input, output and input/output parameters.

The software component-oriented approach is an evolution from the object-oriented approach. Whereas the object-oriented approach is focused on classes, objects and inheritance, the component-oriented approach is focused on interactions, dependencies and composition.

In software architecture, the component-oriented approach relies on four main standard models: **component model**s defining component structure and behavior, **connector model**s to specify component interactions, **container model** to mediate access to middleware services, and **packaging and deployment model**s to deploy distributed component-based applications. The composition model includes the connector model and the deployment model. The component model is a set of standards, conventions and rules that components must respect. The connection model defines a collection of connectors and supporting facilities for component assembling. The deployment model describes how to install component-based applications into the execution environment.

A software component provides and requires services through multiple interfaces that allow an explicit specification of external dependencies and a separation of concerns between functional interfaces for business logic and non-functional interfaces for technical logic e.g. regarding configuration, deployment and activation, packaging, deployment (i.e. installation, connection), life cycle (i.e. initialization, configuration and activation), administration (e.g. monitoring and reconfiguration). A software component has logical ports that provide and require properties, methods and events.

A software connector is a component which binds the required and provided ports of two or more components. It mediates component interactions. Examples of interaction patterns include method invocation between client and server roles, event delivery between publisher and subscriber roles, data stream between source and sink roles, message passing between sender and receiver roles and shared memory between writer and reader roles. It may deal with non-functional services such as communication, coordination, conversion, synchronization, arbitrating, routing, and scheduling. Connectors separate computation from interactions. The notion of connector reifies the notions of relationships and interactions in the object model. When the same components are deployed on nodes in different address spaces, the connector may use or implement the required middlewares services such as marshalling/unmarshalling and interface adaptation. In this case, the connector is partitioned into two "fragments" that may represent a stub/skeleton pair or two middleware instances.

Connector template architecture may include interceptors, stub/skeleton, dispatcher, arbiter and scheduler.

A software container transparently integrates components into their operating environment, mediate access to middleware services and manage component life cycle.

Like the SCA, we distinguish two basic kinds of components: application components and platform components. Application components implement business logic and require HW/SW platform services, while platform components implement technical logic and provides HW/SW platform services.

A component may have a functional or algorithmic implementation e.g. in executable UML, Matlab/Simulink, SystemC TLM; a software implementation e.g. in C, C++ or Java; or a hardware implementation e.g. in synthesizable SystemC RTL, VHDL or Verilog.

For instance, the SCA component model defines an Application Component Model with the `LifeCycle` and `PropertySet` interface withih the `Resource` interface and a Hardware Platform Component Model with the `Device` interface. The SCA connection model is based on the `Port` and `PortSupplier` interfaces. The SCA deployment model is based on the Framework Control interfaces and the Domain profile defining XML descriptors.

In the hardware component models, a hardware component has input, output, input/output RTL ports that provide and require one or several signal-based interfaces. The connection model of hardware components is based on standard bus interfaces and communication protocols. An emerging packaging model for hardware components is the IP-XACT specification, which defines XML schema files to describe the characteristics of IP core components such as hardware interfaces, registers and interconnection. The deployment model of hardware IP core components can be either statically deployed on an FPGA at reset-time or dynamically deployed at run-time.

In the next chapter, we will expose how we propose to unify hardware, software and system compo-

nents by mapping high-level method-based software interfaces onto the signal-based hardware interfaces to reduce the abstraction and implementation gap.

Chapter 7

Unified Component and Middleware Approach for Hardware/Software Embedded Systems

Contents

In the previous chapters, we saw that different concepts, models and methodologies were proposed to unify the design of hardware/software embedded systems. The distributed object model was notably applied to hardware/software modules to provide common concepts between hardware and software designers. Language mappings were proposed to map method-based software interfaces to signal-based hardware interfaces, but each mapping proposition only represents one solution in the mapping exploration space. The component-oriented model was proposed to address the limitations of the object model notably to impose strict design constraints to designers in order to ease component reuse and integration. We showed that the component-based design is natural for both hardware and software engineers, but few language mappings were proposed to support the mapping of abstract component-oriented models to hardware, software and system component models. The purpose of this chapter is to define such language mappings, while exploring the mapping exploration space and providing transparent communications between the mapped components.

7.1 Introduction

The CORBA Interface Definition Language (IDL) 3.x allows designers to specify the software interfaces that a software component provides and requires through ports. Language mappings or bindings specify mapping rules to translate IDL3 interfaces into software interfaces in a target implementation language.

Our objective is to propose an IDL3-to-VHDL mapping and IDL3-to-SystemC mapping at different abstraction levels to apply the same middleware and component concepts down to hardware and provide a common design flow for hardware, software and system components. The goal is also to propose a systematic refinement of interfaces. Hence, IDL3 interfaces would be used to specify business component interfaces at system-level independently of their hardware or software implementation.

The specification from an operation-based abstract software interface to a signal-based hardware interface is not an easy task due to the important abstraction and conceptual gap.

One goal of the following chapter is to define the necessary requirements to enable the generation of hardware component interfaces from a HW/SW neutral IDL specification. These requirements are based on the analysis of the state of the art presented in the previous chapters. The key questions raised by such a high-level mapping are: What is a hardware component from a software component-oriented point of view ? How a software operation invocation is translated into VHDL code ? What is the VHDL entity interface of a hardware component, which should be generated from the IDL specification of this component ? How component attributes and operation parameters appear to hardware designers in his/her business code ? Our mapping proposition tries to answer these mapping requirements.

The second objective is to propose a middleware architecture framework to host the mapped hardware components and provide interoperability between hardware/software components.

This chapter is organized as follows. Section 7.2 presents the Component-Oriented Design Flow we propose. Section 7.3 describes the proposed mapping from object-oriented component-based specification to system components at transactional level and hardware components at Register Transfer Level. Section 7.4 proposes to extend the Software Communications Architecture for hardware IP core components in FPGAs. Section 7.5 presents a hardware middleware architecture framework to support transparent communication between hardware/software components Finally, section 7.7 concludes this chapter.

7.2 Component-Oriented Design Flow

In this section, we present the conceptual design methodology for SDR, which existed at Thales Communications in 2005. Then we propose to leverage this design flow with SystemC TLM to simulate the platform independent model of waveform applications, and the platform specific model of the waveform and the platform. Finally, we propose a common design flow for system, hardware and software components based on MDA, SystemC TLM, language mappings from logical interfaces to simulation and implementation languages.

7.2.1 Existing design flow and methodology for SDR

The conceptual design methodology is based on two concepts: Model Driven Architecture (MDA) **MDAGUIDE03** and the component/container model. As presented in section 2.8.1, MDA advocates the separation between system functionalities specification defined in a Platform Independent Model (PIM) from their implementation specification on a target platform captured by a Platform Specific Model (PSM). The PIM can be expressed in UML, while the associated PSMs are described in target programming languages such as C, C++ and VHDL. The MDA approach is useful in a SCA context because of the amalgam in the SCA specification between business choices (interfaces and behavior) and technical/implementation choices (type of middleware, POSIX profile). The component/container model enables a clear separation of concerns between behavioral and technical properties.

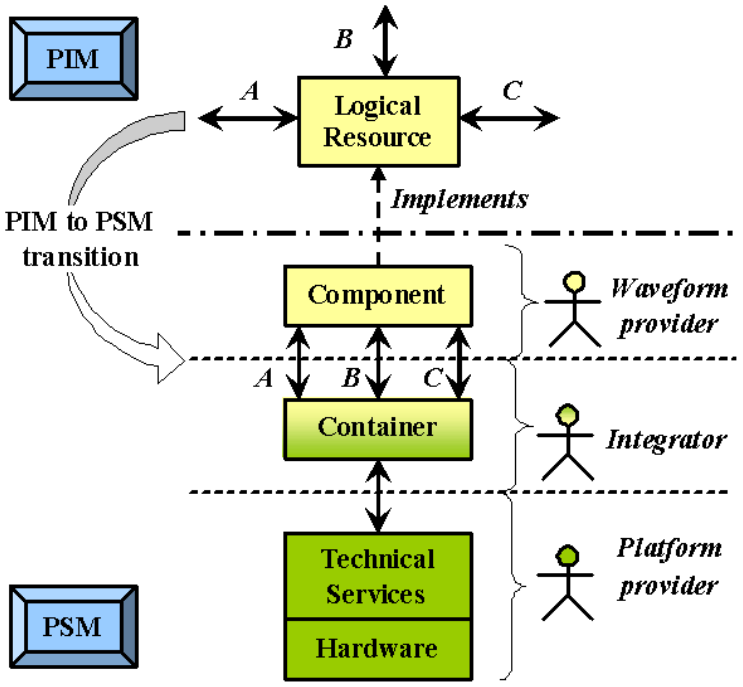


Figure 7.1: Existing methodology inspired from MDA and component/container paradigm.

The methodology illustrated in figure 7.1 consists in defining in the PIM the logical interfaces, e.g. A, B and C, between abstract processing resources (waveform management components, waveform resources, radio services and radio devices). Then the PIM-to-PSM transition is performed. A component implements the useful treatments specified at logical level and communicate with its environment thanks to a container through the mapped interfaces A, B and C. The container performs the adaptation between

logical interfaces and available technical services. These latter provide the software and hardware services used by the implementation of the logical semantics. This approach is compliant with SCA for GPPs and has been successfully applied on DSPs. It seems to be promising for FPGAs.

The design flow based on this methodology starts at PIM level with waveform specifications. The logical interfaces and real-time constraints between the logical waveform components themselves and with the radio sub-system are modeled in UML thanks to use case, class, statechart and scenario graphical diagrams. The breakdown into waveform logical components are performed according to system requirements (characteristics of usable GPPs, DSPs, FPGAs) and business experience. Then subsequent hardware and software design flow can begin. The waveform application can be lately validated on an SCA compliant workstation. However, the final validation can only be done once the hardware platform is available. Thanks to SystemC TLM, we propose a PIM/PSM design flow to design and validate earlier waveform applications on virtual SDR platforms.

7.2.2 Proposed simulation methodology

MDA and component/container paradigm approach can be simulated thanks to SystemC TLM as illustrated in figure 7.2. The business services of logical components are implemented in a C/C++ model. This independent and reusable model is encapsulated in a SystemC TLM container, which abstracts it from its virtual execution environment. For a software component, the container could use an API, while for a hardware component, it could contain a register bank (see our experiments in chapter 8). Developers can validate that the SystemC TLM virtual module provides the required behavior and perform a partitioning based on performances estimation. The model can be either refined until RTL for a hardware component or extracted to be optimized and cross-compiled for a software component. If the partitioning was already performed, the C/C++ model can be used without a SystemC TLM encapsulation directly on an Instruction Set Simulator (ISS) if available, otherwise the SystemC kernel could be used. This SystemC TLM based methodology allows to simulate the heterogeneous architecture of SDR platforms.

Thus, we propose SystemC TLM for the simulation and validation of PIM and PSM. In the SDR context, an important need is to be able to simulate the real-time constraints of a radio system, from its specifications to its final validation. SystemC appears, as far as we know, as the only language addressing this kind of needs.

At PIM side, SystemC at functional level can be used to model the logical interfaces and simulate the real-time constraints between the waveform logical components themselves and with the radio subsystem. The resulting SystemC TLM modules could include a behavioral model or acts as dummy black box. This model can be used as a useful and undeniable executable specification between hardware and software teams that anticipates the PSM and accelerates the development cycle. The concepts of required and provided interfaces specified in the PIM can be natively found in SystemC 2.1 and used to simulate a PIM.

Event and time driven systems can be modeled with `sc_event` and `wait` statements, while UML synchronous and asynchronous calls can be respectively modeled with SystemC TLM blocking and non-blocking calls.

At PSM side, SystemC at cycle level enables performance estimation (profiling) of waveform software components executed on a virtual radio platform composed of ISS for GPP and DSP and *Bus Functional Models* **(BFM)** if available. Thus, real-time constraints simulated in the PIM model can be reused to validate the PSM model. SystemC TLM allows one to simulate and to validate only critical parts instead of the entire radio system model. A concrete example will be seen in our experiments to illustrate the modeling of PIM and PSM in SystemC TLM (see chapter 8).

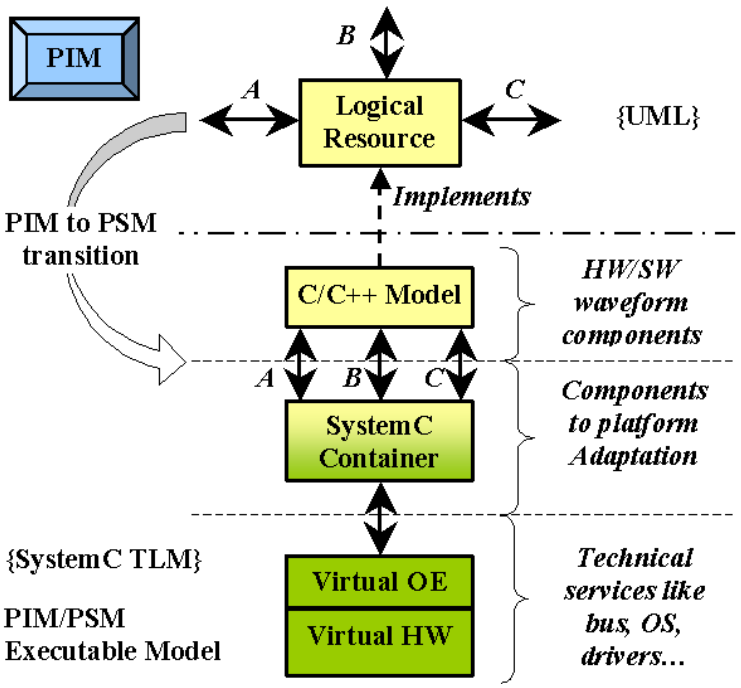


Figure 7.2: SystemC TLM model of PIMs/PSMs.

7.2.3 Proposed Design Flow

As the SCA specifications [JTR01] [JTR06] and the deprecated SCA developer's guide [JTR02] rely on the UML standard for the design of waveform applications and as Thales is an active OMG member which has participated in the UML profile for software radio [OMG07e] based on the SCA, the UML profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [OMG08h] and the CORBA Component Model, we propose a coherent design flow with state-of-the-art and standard technologies notably following an OMG MDA-based design flow. Indeed, our proposition must integrate well into and extends the existing SCA design flow. The proposed design flow is depicted in figure 7.3

Other UML profiles may be used for an MDA design flow for embedded systems such as the UML Profile for CORBA and CORBA Components Specification [OMG08g], the PIM and PSM for Smart Antenna [OS09] from OMG and SDRF, and the UML Profile for System on a Chip (SoC) [OMG06e] inspired from SystemC, and Systems Modeling Language (SysML) [OMG08f].

Face to the number and complexity of UML profiles, the relative maturity of tools, the required training of software, hardware and system engineers and the associated costs, a coherent HW/SW co-design flow is not easy to define and to apply within an enterprise or a set of specialized enterprise departments.

The proposed design flow applies the MDA-based Y design flow proposed by the LIFL [DMM+05] to the SCA-based SDR domain.

We make a clear separation of concerns between the component-oriented application (e.g. with SCA and/or CCM personalities), the software platform (e.g. CORBA) and the hardware platform (e.g. with GPP(s), DSP(s) and FPGA(s)). Indeed, an application may be defined independently from the underlying software platform, and the software platform may be defined independently from the underlying hardware platform. Clearly, an application is specified as an assembly of hardware/software components as in the SCA and CCM.

The proposed design flow starts with the capture of waveform applications and hardware platform

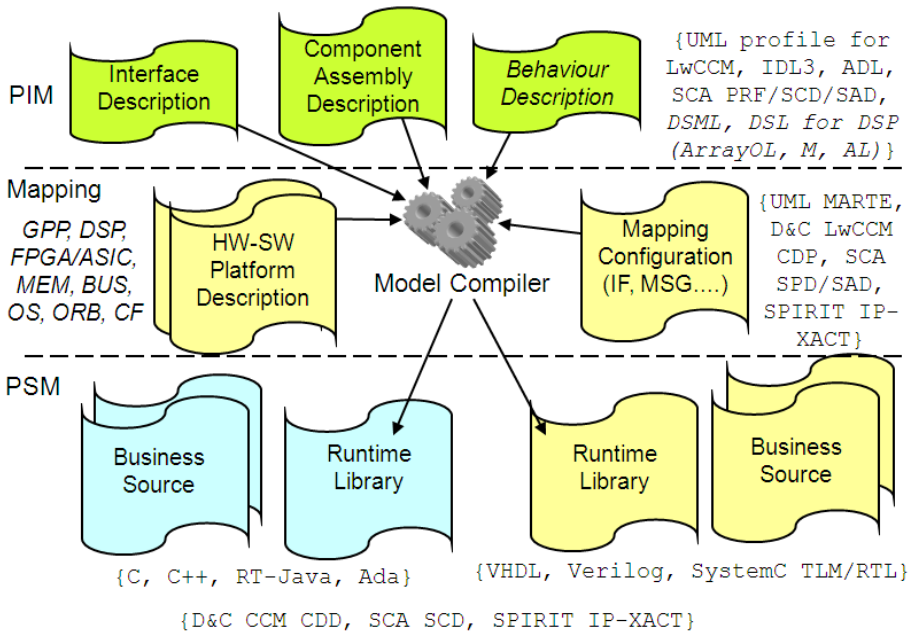


Figure 7.3: Proposed Design Flow

requirements in UML PIMs.

A component-oriented application may be specified by the waveform application component designers in a HW/SW Platform-Independent Model (PIM) using standard UML. Then this PIM may be annotated or marked [MSUW04] using the UML interfaces standardized in the PIM of the UML profile for Software Radio and SysML to model non-functional system requirements.

In parallel, the component-oriented hardware platform may be specified by the platform component i.e. device designers with the PIM of the UML profile for Software Radio, MARTE for the Non-Functional Properties (NFP) and hardware architecture, UML4SoC for virtual hardware platform in SystemC and the emerging UML profile for SPIRIT to describe hardware signal-based interfaces and the memory mapped registers.

The PIMs may then be translated in SystemC TLM containers, where a functional model is encapsulated to validate algorithms choice, by simulating the behavior of the system and measuring its performances (e.g. in terms of the *Bit Error Rate* (**BER**) for a modem application). This executable specification can be reused for example to validate waveform compliance to specifications, in particular the real-time constraints.

During hardware/software partitioning, several architectures and partitioning choices can be evaluated for the system regarding for example the interconnection network and the memory dimensioning. Associated tools perform for instance profiling or bottlenecks check. This partitioning enables to ease the definition of waveform breakdown/granularity, and thus reusable/reconfigurable components.

During architecture exploration, a feasibility study may be conducted. If the requirements cannot be met, a feedback is given to the system engineers in order to review their partitioning or eventually their algorithms. After several iterations, this step results in a system architecture definition, which provides a breakdown of the application with associated hardware targets and the interfaces of each component.

Then hardware modeling and software developments can start in parallel. The system can be refined progressively to platform specific models in SystemC TLM, for both hardware and software modules, keeping the same testbench to functionally verify the successive translations. This refinement can be re-

alized through timing annotations or use of another bus/topology and results in a better design. Moreover, these models can be reused earlier in future developments. In software, application code is developed, eventually for managing the partial reconfigurability (bitstream download, application deployment, configuration...).

Co-simulations can be performed to check the behavior of the application in a specific operating mode. Then the hardware workflow can start (synthesis, place and route) and finally the platform integration on the real target is performed. A validation shall then be led to check that all system requirements are met.

In the next section, we will present how the mapping of object-oriented component-based specifications to software, system and hardware components can be achieved in the proposed design flow.

7.3 Mapping object-oriented component-based specification to system, software and hardware components

As the mapping of object-oriented component-based specifications to software components already exists using standard software language mappings e.g. CORBA IDL3 to C++, we propose a unified approach to also map such specifications to system components at transactional level and hardware components at register transfer level. We propose the use of hierarchical component ports to ease interface refinement and better manage the difference of abstraction levels between software components at logical level in UML, SCA and CCM; system components at transaction level in SystemC and hardware components at RTL level in HDLs like VHDL and SystemC RTL. Hence, a hierarchical port provides different views of the same high-level functional component port. Table 7.1 presents the mapping between software, system and hardware component ports at different abstraction levels for a given RTL mapping configuration.

7.3.1 IDL3 to SystemC Language Mapping

IDL3 to SystemC at Functional Level

The proposed mapping from IDL3 to functional SystemC is relatively straightforward. It extends the IDL-to-C++ mapping with additional mapping rules summarized in table 7.2 to fit specific SystemC language constructs. A SystemC component is a `sc_module` component with required `sc_export` ports and provided `sc_port` ports associated to `sc_interface` interfaces.

An alternative mapping for SystemC consists in using the IDL-to-C++ mapping and inheriting from the SystemC interface. However, we prefer to consider SystemC not as a C++ library, but as a language and use its syntax and semantics.

The same functional interface in IDL may be used at each abstraction level without interface redefinition.

In the SPICES project, the mapping of CCM components is based on multiple inheritance. The mapped CCM component inherits from the mapped IDL interfaces to C++, the `CCMObject` interface, the `sc_module` interface, and the `sc_interface`s for the connectors. Facet and receptacles are respectively mapped to `sc_export<the_provided_interface>` and `sc_port<the_used_interface>` types.

IDL-to-SystemC Mapping Illustration at Functional Level

The Platform Independent Model (PIM) of a partial waveform application is represented in UML in Figure 7.4a and Figure 7.4b. In the class diagram of this waveform depicted Figure 7.4a, a circle represents an abstract interface, which is required (use dashed arrow) by a component and provided (realize

Port	**Logical Port**	**Message Port**	**Signal Port**	**Signal Port Description**
	in provides	in request	in push	receive request from caller
			in addr	identifier of the callee operation
			in data	operation in/**in**out parameters
			out ack	request accepted by callee
		out reply	out push	send reply to caller
			out data	return value and out/in**out** parameters
			in ack	reply accepted by caller
			out exc	exception triggered by callee
	out requires	out request	out push	send request to callee
			out addr	identifier of the callee operation
			out data	operation in/**in**out parameters
			in ack	request accepted by callee
		in reply	in push	receive reply from callee
			in data	operation out/in**out** parameters
			out ack	reply accepted by caller
			in exc	exception triggered by callee
Examples				
	SCA, CCM, SystemC (FV, PV, OSCI TLM 1.0 transport)	SystemC TLM (OCP, OSCI TLM 1.0 request-reply, TLM 2.0)	OCP RTL	

Table 7.1: Mapping of component ports at different abstraction levels

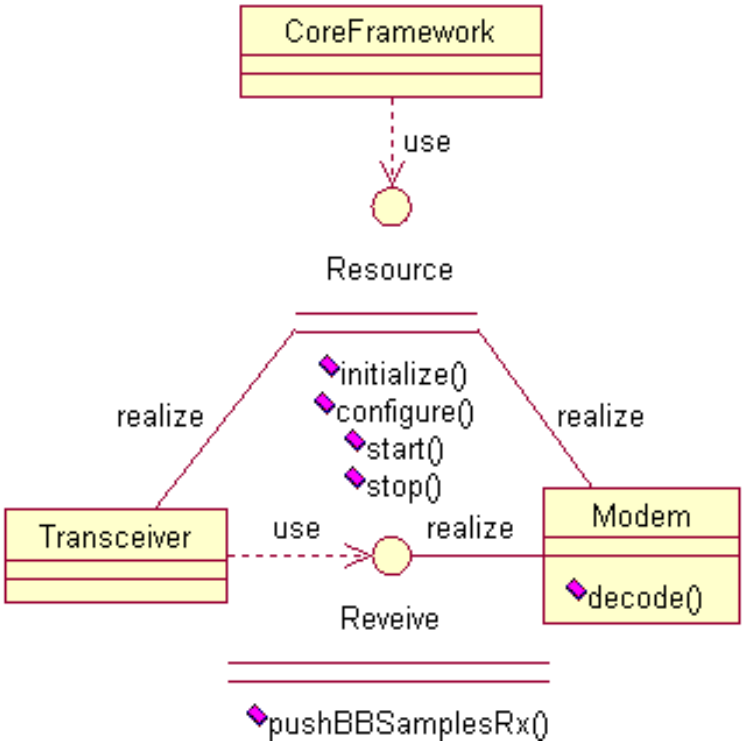
IDL3 constructs	**SystemC constructs**
`module M { ... };`	`namespace M { ... };`
`interface I { ... };`	`class I : virtual sc_interface { ... };`

`component C {`
`provides I1 P1;`
`uses I2 P2;`
`};`
	`class C: sc_module {`
	`sc_export<I1> P1;`
	`sc_port<I2> P2;`
	`};`

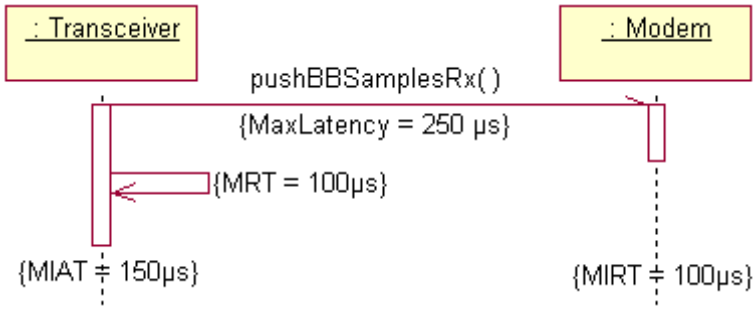
`connector C { ... };`	`class X: sc_channel { ... };`

Table 7.2: Summary of the proposed IDL3-to-SystemC mapping at functional level

association) by another one. Some real-time constraints are also captured by a UML sequence diagram given in Figure 7.4b. According to this PIM, a Transceiver Resource shall asynchronously call (truncated arrow) the Modem Resource method with a maximum latency of 250μs to push the received baseband samples. This call shall return with a *Maximum Return Time* (**MRT**) of 100μs. Moreover, the *Minimum Inter-Arrival Time* (**MIAT**) is 150μs between two Transceiver calls, and the *Minimum Inter-Return Time* (**MIRT**) between two Modem returns is 100μs.



(a) Class diagram



(b) Sequence diagram

Figure 7.4: Platform Independent Model (PIM) of a partial waveform application

Based on UML specifications, interface definition in CORBA IDL3 can be generated as shown in listing 7.1. An octet sequence is transferred from the `Transceiver` to the `Modem` component by calling the oneway `pushBBSamplesRx` operation. The CORBA sequence Abstract Data Type (ADT) is implemented in C++ and the same API is used to use CORBA sequences in SystemC. Two connec-

tors are defined to respectively support blocking and non-blocking (oneway) invocations. We use the `component` IDL3 keyword to define connectors for backward compatibility, but a new `connector` keyword could be introduced in CORBA IDL.

```
module Example {
    typedef sequence<Octet> OctetSequence;
    interface Receive {
          oneway void pushBBSamplesRx(in OctetSequence samples);
    };
    component Transceiver {
        uses Receive pR;
    };
    component SynchronousConnector {
        provides Receive pR;
        use          Receive pT;
    };
    component AsynchronousConnector {
        provides Receive pR;
        use           Receive pT;
    };
    component Modem {
        provides Receive pT;
    };
```

Listing 7.1: Transceiver Modem Example in IDL3

In the PIM, the `Transceiver` component requires the `Receive` interface implemented by the `Modem` component. In the SystemC functional model of the PIM depicted in Figure 7.5, the `Receive` interface (line 4 in listing 7.2) is required by the `Transceiver` module port (l. 10) and is implemented inside the `Modem` module (l. 67), which is called by the `Transceiver` port through a `SynchronousConnector` (l. 25) or `AsynchronousConnector` (l. 38) channel. These channels model the communication real-time constraints e.g. throughput and latency (l. 33) using `wait` statement. In fact, real-time constraints can be encapsulated in the SystemC TLM container itself (Explicit Timing Annotation) or in a transactor between the module and the channel, which can be better to separate timing from behavior (Implicit Timing Annotation) [THA05]. A transactor is a hierarchical connector channel, which acts as a TLM protocol and interface converter. Traffic generators can be used to transfer samples according to various probability laws to observe the behavior of the communication chain following operational conditions (link overload, Signal-to-Noise Ratio decrease...). After the validation of real-time constraints, the SystemC TLM models of the containers and components in the PIM can be refined to their PSM models.

```
1 namespace Example {
  using namespace CORBA;
  typedef UnboundedSequence<Octet> OctetSequence;
  class Receive : public virtual sc_interface {
    public:
6   virtual void pushBBSamplesRx(const OctetSequence& samples)=0;
  };
  class Transceiver : public sc_module {
  public :
    sc_port<Receive> pR;
11  SC_HAS_PROCESS(Transceiver);
    Transceiver(const sc_module_name  &n);
    void transmit() {
```

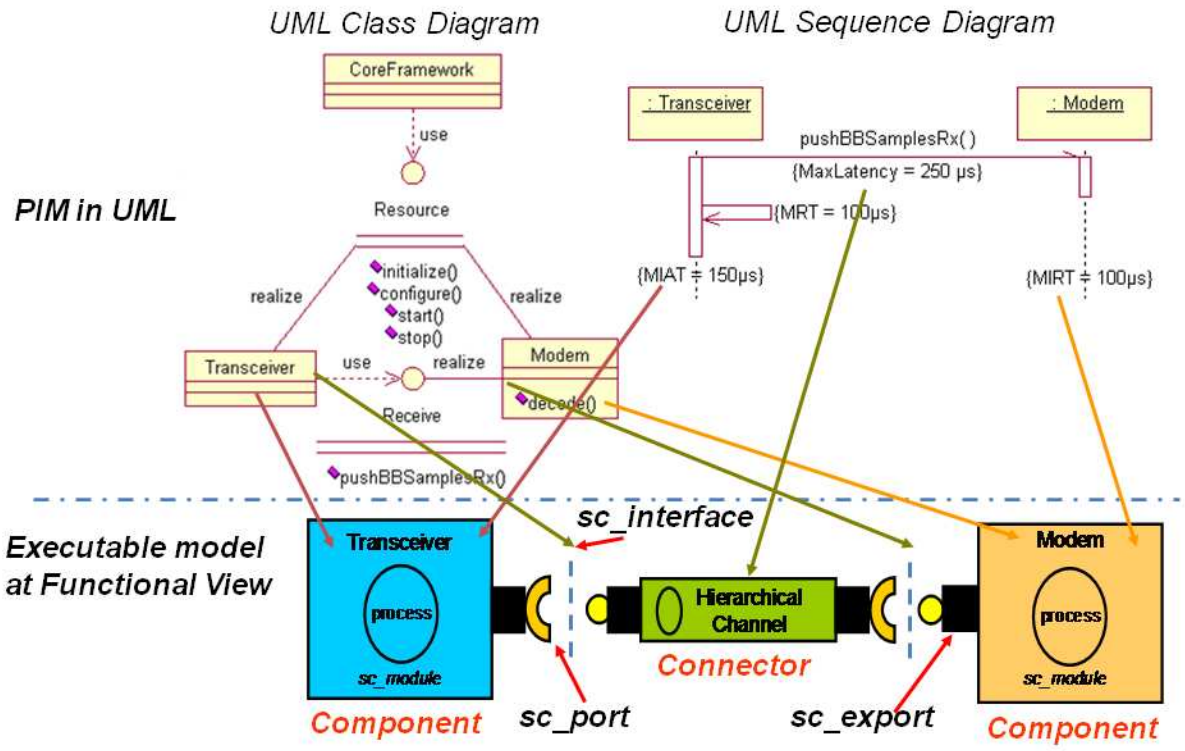


Figure 7.5: Executable specification in SystemC at Functional View

```
        wait(TRANSCEIVER_LATENCY, SC_NS);
        for(UShort j = 0; j< 2; j++) {
16      OctetSequence samples;
        samples.length(3);
        for(UShort i=0; i<samples.length(); i++) {
          samples[i] = i+j*3;
        }
21    pR->pushBBSamplesRx(samples);
      wait(100, SC_NS);
      }
};
class SynchronousConnector : Receive, sc_module {
26  sc_port<Receive> pR;
  sc_export<Receive> pT;
  OctetSequence s;
  SC_CTOR(SynchronousConnector) : pT("pT") {
    pT( *this );      // bind sc_export->interface by name
31  }
  void pushBBSamplesRx(const OctetSequence& samples) {
    assert(SYNC_CHANNEL_LATENCY<SYNC_CHANNEL_MAX_LATENCY);
      wait(SYNC_CHANNEL_LATENCY, SC_NS);
      pR->pushBBSamplesRx(samples);
36  }
};
class AsynchronousConnector : public Receive, sc_channel {
    public:
```

```
    sc_port<Receive> pR;
41  sc_export<Receive> pT;
    SC_HAS_PROCESS(AsynchronousConnector);
    AsynchronousConnector( const sc_module_name & n ) :
      sc_module(n),   pR("pR"), pT("pT") {
      pT( *this );      // bind sc_export->interface by name
46    SC_THREAD(transfert);
      sensitive << s; dont_initialize();
    void pushBBSamplesRx(const OctetSequence& samples) {
      tmp.length(samples.length());
      for (CORBA::ULong i=0; i<samples.length(); i++) {
51      tmp[i] = samples[i];
      }
      s.notify();
    }
      private:
56      sc_event s;
        OctetSequence tmp;
    void transfert() {
        while(true) {
        assert(ASYNC_CHANNEL_LATENCY<ASYNC_CHANNEL_MAX_LATENCY);
61      wait(ASYNC_CHANNEL_LATENCY, SC_NS);
        pR->pushBBSamplesRx(tmp);
        wait();
        }
    }
66  };
  class Modem : Receive, sc_module {
    public:
    sc_export<Receive> pT;
    OctetSequence s;
71  SC_CTOR( Modem ): pT("pT") {
      pT( *this ); // bind sc_export->interface by name
    }
    void pushBBSamplesRx(const OctetSequence& samples) {
      wait(MODEM_LATENCY, SC_NS);
76    // C/C++ model functions calls
      cout << " Samples received :" << endl;
      for (CORBA::ULong i=0; i<samples.length(); i++)
        cout << " samples["  <<  i << "] = " << (unsigned short)samples[i]
           << endl;
  };
81 };

SC_MODULE(Top)
{
  Transceiver transceiver;
86 #ifdef ASYNC
  AsynchronousConnector connector;
#else
  SynchronousConnector connector;
#endif
91  Modem modem;
  SC_CTOR(Top) :
```

```
      transceiver("Transceiver"),
    connector("Connector"),
    modem("Modem")
96  {
    transceiver.pR( connector.pT );
    connector.pR( modem.pT );
  }
};
```

Listing 7.2: Transceiver Modem Example in SystemC at Functional View

IDL3 to SystemC at Transactional Level

The IDL-to-HDL mapping can be seen as a refinement problem in SystemC. User-defined interfaces in IDL may be mapped to SystemC at Functional Level. Transactional connectors called transactors may adapt these interfaces to standard TLM interfaces. The key point is that the invocation messages derived from IDL interfaces are identical and carry the same semantics at all abstraction levels. For instance in TLM 2.0, invocation messages may be sent in the generic payload of TLM transactions. At all abstraction levels, the syntax and semantics of the business interfaces in UML or IDL and the associated messages will the same, only the manner to transfer the messages will change according to the abstraction level. The equivalence between operation invocation and message transmission provides a systematic approach for the refinement from one abstraction level to another. This object-oriented approach to refinement contrast with the classical ad-hoc refinement in SystemC where the syntax and semantics of business interfaces are lost during refinement.

This mapping proposition will be illustrated in chapter 8 with the refinement of a business component from a functional model in C to a SystemC component at Programmer View (PV) level (see §8.2.2), the refinement of a transactional connector from an OCP TL2 to an AHB bus functional model (see §8.2.2), and the refinement at different abstraction levels from SystemC FIFO to OCP TLM and OCP RTL (see §8.3.2).

The mapping from IDL3 to SystemC RTL is presented in the following subsection with the IDL3 to HDL Mapping and detailed in chapter

7.3.2 IDL3 to HDL Mapping

In this section, we present a language mapping from component specification in CORBA IDLs to component specification in HDLs in particular VHDL.

Differences between a SW-to-SW interfaces mapping and a SW-to-HW interfaces mapping

Basically, software IDLs like CORBA IDL have been designed for software programming languages. There is only a systematic and one-to-one mapping between an IDL interface and a software interface in the target implementation language, which reside at close abstraction levels. Defining a standard mapping for HDLs such as VHDL is much more difficult.

In software programming languages, a procedure call convention [PH07] specifies the execution model of an operation call for compilers and transparently hides this model from the application developer's point of view. For instance in MIPS processors [PH07], the first four procedure arguments are passed in registers, while the remaining arguments are pushed on the stack. The return address of the caller is saved in a register and the program counter jumps to the callee procedure. The callee performs the computations and place the results in dedicated registers. The program counter finally jumps to the

saved return address to return the control to the caller. Compared to a hardware designer, a software designer has little room in its design to optimize the execution time or memory consumption of an operation call. Indeed, this optimization is performed by the compiler according to the procedure call convention and directives given by the user. In hardware description languages, an inherent design trade-off exists between area, performance (e.g. bandwidth, latency) and power consumption for hardware modules. Hardware designers require a complete control over this trade-off. An example is the numerous knobs offered by usual synthesis tools and/or high-level synthesis tools to explore and optimize the design space. Note that we focus on the mapping space for hardware component interfaces and not on the design space related to the implementation of component business functionalities. An IDL-to-HDL mapping requires to select and configure non-functional properties such as hardware interfaces, timing and protocols. As opposed to software mapping, the final hardware interface may depend on the set of methods declared in the IDL interface for instance to configure a address bus used for method invocations. Because the CORBA IDL3 only specifies functional properties i.e. behavior, a standardized mapping configuration file should explicitly define these non-functional properties. This file could be generated manually, via a graphical tool or inferred by an IDL compiler front-end.

Mapping Requirements

In the following, we present some mapping requirements based on the compilation of the state of the art on hardware and software implementations of objects, middlewares and components. They are notably focused on concurrency, message and transport.

Component-oriented hardware architecture A component interface is represented by a VHDL `entity`. A component implementation is represented by a VHDL `architecture`. A component implementation is selected by a VHDL `configuration`.

Encapsulation The value of component attributes shall be retrieved and modified only through dedicated accessor and mutator operations called "getter" and "setter". No direct access to internal component attributes should be allowed from component outside. A `writeonly` keyword is required for IDL3 attributes and would implicitly infer a setter method. For instance, a hardware register may be write-only for software and read-only for hardware. In this case, the external bus interface may be a write-only hardware port, while the internal component interface may be a read-only hardware port.

Concurrent implementation of a method There should be no assumption on component hardware implementation, sequential processes and concurrent statements may be used. Sequential implementations only based on FSMs such as in [Rad00] are not required. Implementation may be hand-coded or generated by some HLS tools. The only constraint is to respect the interface contract both at system-level in IDL and at RTL-level with the mapped hardware interface.

Concurrent method invocations and executions within a component Multiple clients may invoke simultaneously the same method or distinct methods that may be executed in both cases sequentially or in parallel. Multiple methods may be invoked and executed independently to enable pipelining of configuration and computation and exploit hardware parallelism.

Figure 7.6 illustrates that parallel method invocations may increase performance by pipeling operation executions. In case A), sequential method invocations with no pipeline require 2 cycles per iteration. In case B), parallel method invocations allows one to achieve 1 cycle per iteration using a simple pipeline with 2 stages that is a gain of 1 cycle compared to case A). Such a requirement militates in favor of a

component approach instead of an object approach as each component aspect e.g. configuration and computation may have a dedicated interface with different qualities of service e.g. timeliness and may require a distinct hardware interface to allow parallel method invocations.

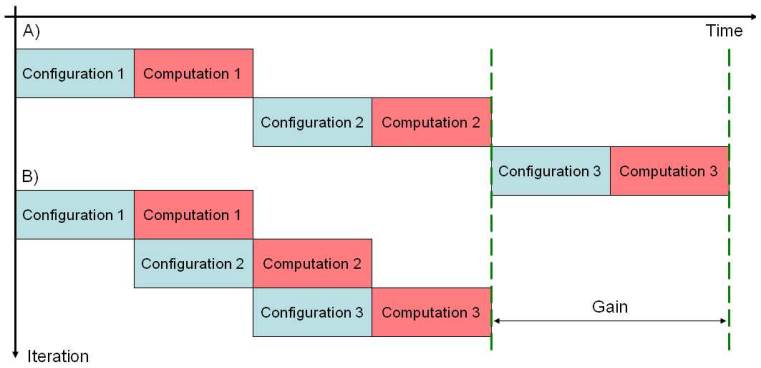


Figure 7.6: Concurrent invocations of methods to enable pipeline of configuration and computation

Concurrent accesses to internal attributes and method parameters Component attributes may be accessed concurrently from the outside by clients via method invocations or from the inside by method implementations themselves. Moreover, a method implementation may require parallel accesses to method parameters. The mapping of IDL abstract data types to storage elements e.g. register, internal/external RAM and FIFOs should be flexible enough to use dedicated or shared resources for multiple attributes and parameters depending on parallel access constraints, data dependencies, area and performance.

Concurrent invocations of methods from multiple clients A centralized or distributed scheduler may be required to grant access to one client invocation at a time on one component port as illustrated in figure 7.7. For instance, schedulers are used in [Rad00] and [ICO08] to arbitrate concurrent access to hardware components from respectively only hardware or hardware/software clients. Priorities may be associated to client requests. Like in Real-Time CORBA, scheduling policies could be standardized. The scheduling policy should be explicitly chosen and configured in an hardware ORB configuration file or in D&C XML files. Such schedulers may be generic and reused in several applications. These dedicated components correspond to hardware connectors.

Concurrent requests and replies of method invocations To simplify hardware component implementation, responses to method invocations are assumed to be strictly ordered inside a component port.

Interaction Semantics As opposed to the literature on hardware objects that only support a single hardware interface and protocol without flow control, component interfaces may require to support various interaction semantics. For instance, a data producer may require a write only output port with *push* semantics in which the sender is the initiator, while a data consumer may require a read only input port with *pull* semantics in which the receiver is the initiator.

Component implementations should be reusable even if the execution environment changes e.g. in terms of clock domains, data production and consumption rates, etc. Connectors may be used and reused to adapt different interaction semantics and mask such execution environment changes.

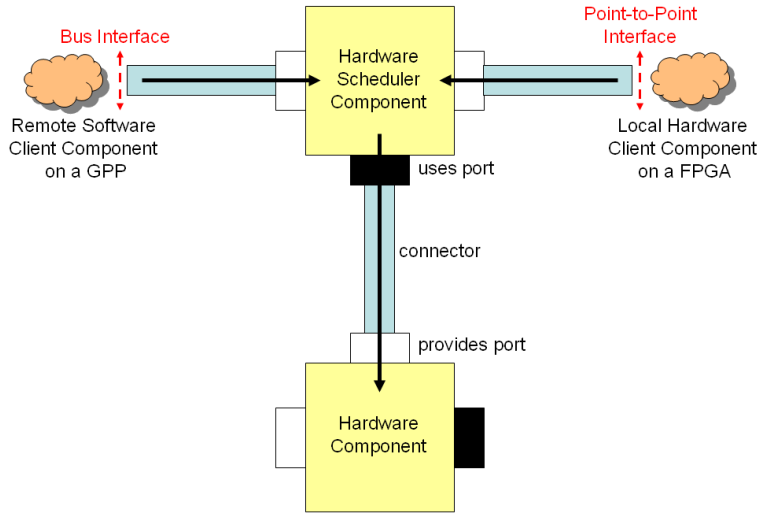


Figure 7.7: Example of concurrent invocations from hardware and software clients

Hardware interface and protocol framework To improve the portability and interoperability of components, a standardized family of interfaces and protocols is required. A component may need to manage its input/output control and data flow if this management is part of the application logic. An example is a SCA Resource component that use the `pushPacket` API. Otherwise, connectors may transparently performed flow control e.g. by buffering data and may deal with other non-functional aspects.

Synchronization

Blocking invocations By default, method invocations should be blocking i.e. the caller thread of control is blocked until the invocation results are returned. This behavior is coherent with the usual software programming model and simplify development and validation at the expense of performance.

Non-blocking invocations without reply (oneway) Oneway operations only have input parameters and a void return type. As opposed to unreliable oneway operations, reliable oneway operations should use a hardware handshake on their invocation interface to acknowledge the acceptation of input parameters. Like CORBA Synchronization Scope (SynchScope) policies, the synchronization point in the communication path where an acknowledge is returned may be defined at stub, ORB, transport, skeleton or server level. This synchronization policy could be configured in the mapping configuration file or an ORB configuration file.

Non-blocking invocations with reply: Asynchronous Method Invocation (AMI) An input handshake should be used to acknowledge the acceptation of input parameters. Conversely, an output hardware handshake may indicate the availability or validity of output parameters. Connectors may be used to adapt different synchronization schemes, for instance, between blocking software and non-blocking hardware.

Message passing

Data types encoding rules Data types encoding rules are required to be with or without padding to enable a trade-off between encoding/decoding complexity and efficient use of bandwidth. CORBA CDR (Common Data Representation) align all IDL data types on their natural boundaries. The resulting padding may not be required for embedded computing. Examples of other encoding rules without padding are: CCDR (Compact CDR) [KJH+00a], ASN.1 Packed Encoding Rules (PER) [IT02] and ZeroC Internet Communication Engine (ICE) Protocol [HM07]. Designers should be able to configure the mapping or ORB to specify what encoding rules they want their messages to use.

Extensible Message Framework CORBA GIOP (General Inter-ORB Protocol) messages are too expensive for embedded computing. Special purpose message format with only the necessary functionality such as the ESIOPs (Environment Specific Inter-ORB Protocols) are required. A family of ESIOPs could be defined using IDL structures to support messages interoperability. Examples of domain-specific message formats include the SCA MHAL messages [JTR07] for Software-Defined Radios, and EIOP [KJH+00a] and CANIOP [LJB05] for the CAN bus based automotive systems. A comparison between the hardware implementations of CORBA GIOP and SCA MHAL will be presented in chapter 8.

Streaming Inter-ORB Protocol Data stream oriented applications may require that messages can be treated on the fly at wire speed. The beginning of a reply can be received as soon as the request transfer is started [MVV+07]. In other words, requests and replies may be pipelined. Dedicated communication channels may be allocated to request/reply messages with guaranteed services in terms of throughput, latency and jitter [Bje05]. Two solutions can be envisaged. *Time-Division Multiplexing* (**TDM**) allocates time slots to interleave requests and replies. *Spatial-Division Multiplexing* (*SDM*) uses dedicated physical bus for requests and replies. The required message format should be selected from different standard message formats in the mapping or ORB configuration file.

Extensible Transport Framework In addition to the family of application-level hardware interfaces, a generic bus interface for messages transfer should be defined to keep the same connector interface regardless of the underlying bus protocol e.g. AMBA and CoreConnect. This bus interface could enable the establishment of point-to-point connections through virtual channels like in OCP.

Data types mapping Only the needed IDL data types must be efficiently mapped to VHDL data types. General-purpose IDLs need extensions to be "hardware-aware" and to notably support bit-accurate data types like in SystemC.

Reference A client component port needs to identify the servant component port with which it wants to communicate like object identifiers in [RMB+05]. Component instances and their ports may be identified by a system unique identifier. This identifier may be encoded in an enumerated type in the same order as it appears in the D&C component deployment plan. During method invocations, servant component identifiers may be transferred on an output signal called *servantId* . The width of this signal could correspond to the number of bits required to binary encode the maximum number of components allowed in a system. This number may be known from the component assembly description.

Implementation As shown in figure 7.8, several servant component instances may be implemented in the same hardware module to save area [RMB+05].

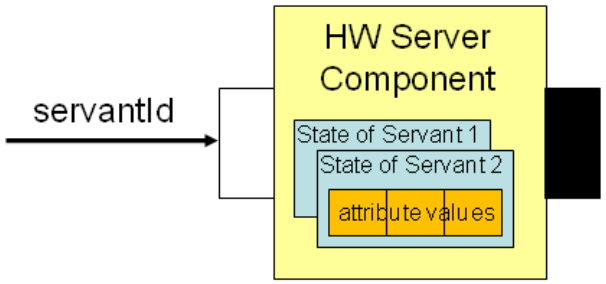


Figure 7.8: Multiple component instances in the same hardware module

Persistence The component state may be automatically stored and restored by the execution framework by calling getter and setter methods according to the servantId [RMB+05]. However, only the public component state can be accessed by these methods inferred from the public attributes specified in IDL interfaces. A dedicated persistence API may be implemented by component developers to store and restore both public and private component properties using some standard serialization format. Serialization could be based on the same data encoding rules than the ones used for message marshalling/unmarshalling.

Mapping simplification The IDL-to-HDL mapping may be simplified. For instance, servantId signals can be removed in point-to-point connections between two components. OperationId signals can be removed if an IDL interface only supports one operation [JTR05].

Inheritance In IDL3, the inheritance consists in reusing the definition of a parent component interface to define a derived component interface. During implementation, the child component can reuse the parent implementation or re-implement another behavior using the same interface definition as illustrated in figure 7.9. Inheritance could be implemented in hardware by extension of the parent component hardware interface and implementation [Rad00]. The child component has additional attributes and methods implementations. The inherited attributes and methods can share resources with the new ones.

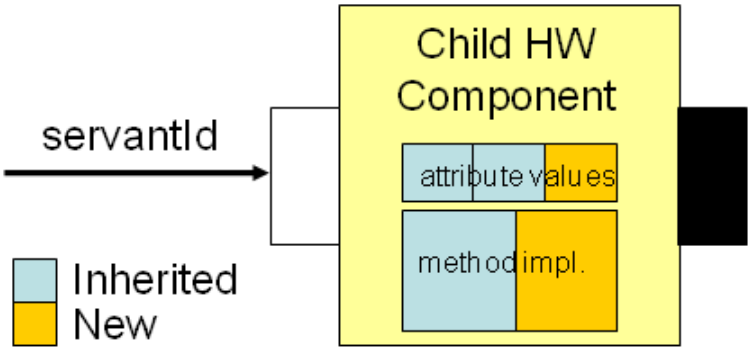


Figure 7.9: Example of inheritance implementation

Polymorphism Polymorphism of a component reference enables to call a parent or child method implementation without distinction. The implementation of the method is selected at run time (late binding

or dynamic binding) depending on the type of the referenced component. Hardware implementations of polymorphism have been presented in [Rad00], [GHM03] and [Frö06].

Systematic mapping The transformation rules to map software interfaces to hardware interface should be (semi-)formal to be automated and not on a case by case basis as in [JTR05]. A systematic mapping allows one to map every IDL interfaces from application objects and components to lightweight ORB services such as Name service, Event service and Extensible Transport Framework (ETF).

Mapping Proposition

We propose to map abstract object-oriented components in a PIM to object-oriented hardware components in PSMs. The mapping of concepts from software to hardware components is general enough to address software components such as SCA, CCM and UML components and hardware description languages such as VHDL, Verilog and SystemC RTL. In this chapter, we focus on language mappings from CORBA IDL3 [OMG08b] to VHDL and SystemC RTL. CORBA IDL3 notably includes the component concepts of the CORBA Component Model. We give an overview of the mapping proposition in the following. The language mappings are described in details in chapter A. The IDL-to-HDL mapping focuses on the interface between business components and the connector fragments a.k.a. stubs/skeletons without making any assumption on the middleware implementation. The underlying hardware middleware architecture framework will be presented in section 7.5.

Our mapping proposition is based on a structural interpretation of abstract object-oriented components. A hardware CCM component is represented by a VHDL `entity` and its implementation corresponds to a VHDL `architecture`, which is selected by a VHDL `configuration`. An abstract CCM component port is mapped to a logical hardware component port, which is a set of VHDL input and output ports. Each hardware CCM component port is based on one or more named hardware interfaces for the invocation request (incl. operation identifier and input parameters) and reply (incl. output parameters, return value and optional exceptions).

Hardware CCM component port allows the separation of concerns between computation in the component and non-functional properties like synchronization and storage. To separate concerns and improve its reuse, a hardware component port may correspond to a VHDL `entity` with its `architecture`. The component core `entity` and its logical port `entities` may be put together in a global component entity. In this case, a logical component port is a subset of the global entity ports.

A hardware connector may represent non-functional properties such as synchronization and communication: flow control, clock domain crossing, interaction semantics, protocols of point-to-point buses, shared buses or NoCs. A connector may also handle interruption and DMA services to transfer messages between two connector fragments. The implementation of the connector fragments may be software-to-hardware or hardware-to-hardware.

A hardware container provide all remaining non-functional services such as persistence and lifecycle. The lifecycle service may manage component creation/destruction e.g. with dynamic partial reconfiguration, component initialization e.g. with a asynchronous or synchronous reset, parameters configuration with register banks, and activation/deactivation e.g. with a clock or enable signal.

Besides, we consider that IP-XACT is to hardware components what D&C is to software components. OMG D&C and SPIRIT IP-XACT XML files describe the implementation, configuration and interconnection of respectively hardware and software components. The IP-XACT specification could correspond to a D&C PSM for the deployment and configuration of hardware components. D&C mainly specifies application software components, while IP-XACT describes application and platform IP core components. We argue that D&C concepts should be extended with SCA and IP-XACT concepts to

address the description, configuration and deployment of hardware/software application components on distributed hardware/software platforms.

In the following, we present the mapping from abstract component concepts such as *component*, *port*, *interface*, `attributes`, *operation parameters*, and `connectors` to concrete component concepts in HDLs.

Component Implementation An abstract object-oriented component at system-level may have a functional implementation e.g. in executable UML, Matlab/Simulink, SystemC TLM; a software implementation e.g. in C, C++ or Java; or a hardware implementation e.g. in synthesizable SystemC RTL, VHDL or Verilog.

Component Port Owing to the abstraction gap between hardware and software constructs, an abstract component port corresponds to several HDL I/O ports. The HDL block construct such as VHDL `entity` or Verilog `module` permits a logical grouping of related signals to form a hardware interface. A HW component port may be either a logical set of I/O ports in a hardware module e.g. VHDL in/out `ports` in an `entity`, or a hardware module e.g. a VHDL `entity`. In the first approach, all software interfaces of an abstract component are mapped to a single hardware interface. Due to the lake of name scoping in HDLs, signal names must be artificially prefixed by the port name of the abstract component to ensure their uniqueness. The second approach allows the reuse of component ports as hardware modules which implement reusable hardware interfaces and protocols. This is coherent with object-oriented software components, in which each port is represented by a class in software and therefore a module in hardware. This ensures the separation of concerns between different component ports. From a synthesis viewpoint, the hierarchy of blocks in the second approach is broken by the synthesis tool and becomes thus similar to the first mapping approach. A hardware mapping should be configurable to permit both alternatives depending on the user preferences. Regardless of the mapping choices, both approaches allow a transparent interconnection of hardware modules as they finally provide and require the same signals.

Moreover, input and output hardware ports may be passive or active. An input component port may be active by pulling/reading data like the consumer in the producer/consumer model e.g. SystemC input port. An input port may also be passive, when the outside pushes/writes data on it. Conversely, an output port may be active by pushing/writing data like the producer in the producer/consumer model e.g. SystemC output port. An output port may also be passive, when the outside pulls/reads data on it. Connectors may adapt and decouple data production and consumption rates.

Component Interfaces A component interface is represented by a VHDL `entity`, a Verilog or SystemC `module`. We distinguish three kinds of hardware interfaces: the business component core interface, the invocation interface of component ports and the connector interface. These interfaces may be configured by VHDL generics, Verilog parameter and SystemC static class variables or `#define`.

Internal component interface to access operation parameters The component interface is the internal interface directly visible by the hardware designer. This interface is explicitly chosen and configured from a family of standardized hardware interfaces. These choices are described in a standardized mapping configuration file used by IDL3-to-VHDL compilers to generate hardware component. Such a standard configuration file can be inspired by High-Level Synthesis tools and could also be used in these tools instead of custom configuration file, scripts or GUI. For instance, a read-write component attribute or property may be associated to a register, RAM or FIFO interface.

External component interface to send/receive invocations The invocation interface is the external interface of component ports that is visible from the outside world i.e. the operating environment, component users, bus etc. An operation call is invoked on this interface by a client through a connector. The Operation Identifier (OpID) enables to select the method to execute as in numerous works such as [KR98], [Rad00] and [BGG+07].

The distinction between internal component object interfaces and external component port object invocation interfaces can be useful for RAMs to explicitly express an external dependency thanks to a generic RAM interface and to enforce reusable components independent of proprietary RAMs inside a component.

Concurrent method invocations By default, operations defined in the same object interface are mapped to the same hardware logical port. To simplify the mapping, method invocations on the same hardware logical port are sequential, while method invocations on different hardware logical ports are parallel. To enforce these mapping constraints, operation identifiers are assigned on a single shared address bus e.g. "operationID" like in [Rad00]. Protocol state machines may specify the order in which methods are allowed to be called. We also consider to extend OMG IDL or UML to explicitly specify the concurrency constraints among operations like in [GMTB04] and [JS07] and to map accordingly sequential invocation to the same hardware logical port. The objective is to allow parallel execution of non-concurrent methods and sequential execution of concurrent methods.

Interface Family We propose a family of named hardware interfaces with well-defined semantics to enable a flexible, extensible and efficient IDL3-to-VHDL mapping. Indeed, such a mapping must be highly flexible to satisfy the various performance-cost trade-offs inherent to the hardware domain. In our proposition, the same IDL interface could be mapped to several hardware interfaces, because the same functionality may require different hardware invocation protocols depending on the performance trade-off.

Selection of the abstract interfaces or operations to be mapped A hardware mapping should allow the selection of the abstract interfaces or operations, which are meaningful in hardware. Even if theoretically any abstract interface could be mapped in hardware, the hardware cost would be too important in practice.

Attributes and operation parameters In the same way that CORBA provides transparent access to attributes and operation parameters, a hardware mapping should mask the access to and storage of component properties and method parameters. To allow concurrent access to operation parameters from method implementation, each parameter must have unshared data signals.

The width of the data bus used to transfer parameters may be explicitly chosen during IDL compilation to save area. Operation parameters, whose size are larger that the data bus width, must be serialized. Component interfaces should have unshared data buses.

Attributes and method parameters stored within hardware component ports The location of the memory to store attributes and method parameters may be within the component, in the component port or in an external on-chip or off-chip memory.

Attributes and methods parameters may be stored in registers or RAM/ROM inside a component port to separate functional aspects such as computation from non-functional aspects such as memory storage and concurrent access control.

The nature of the memory (register, internal or external RAM/ROM) used to store attributes and parameters value may be inferred by an IDL compiler from their types (scalar like integer or constructed like array and struct). However, the resulting hardware component interface shall be explicitly chosen in a mapping configuration file used by IDL3 compilers. A family of named hardware interfaces shall be standardized and parameterized e.g. by VHDL generics. The internal component interface for attributes stored in registers may be composed of only a data bus since no control signals may be required. The component interface for attributes stored in RAM require a generic RAM interface to be independent from proprietary memory interfaces on FPGA/ASIC platforms.

The memory address mapping and data layout of parameters should be explicitly defined in the mapping configuration file. The method invocation interface requires control and data buses to invoke the hardware method implementation. The control bus contains an virtual address bus to select the method to execute as in [Rad00].

Implementation A component implementation is represented by a VHDL `architecture`.

Configuration A component implementation is selected by a VHDL `configuration`.

Connectors Software connectors are typically represented in programming languages with procedure calls and shared data, and usually take the form of libraries and frameworks. In hardware description languages, we can notice that interactions among hardware modules are specified using VHDL signals, Verilog wires and SystemC channels, memory modules (registers, FIFO, RAM) and more rarely procedure calls.

Taxonomy of hardware connectors We consider all these hardware languages constructs as hardware connectors that are used to bind hardware components. Like abstract components, an abstract connector at system-level may have a functional implementation e.g. in executable UML, Matlab/Simulink, SystemC TLM; a software implementation e.g. in C, C++ or Java; or a hardware implementation e.g. in synthesizable SystemC RTL, VHDL or Verilog.

Moreover, we argue that the taxonomy of software connectors proposed in [MMP00] can be easily applied to hardware to provide a unified concept of connector at system-level. Indeed, a hardware connector consists of one or more ducts i.e. wires or signals used for control transfer e.g. for `clock`, `reset`, `data_valid`, `enable`, control bus signals, and data transfer e.g. data bus signals for packets serialized on a parametrized data width. The separation between control and data in software connectors is a common practice in hardware design with the control path e.g. implemented using conditional statements and FSM, etc., and the data path e.g. implemented using data flow operators such as adders, shifter, etc.

We propose some examples of hardware connectors for each connector type identified in [MMP00].

Procedure call connectors have been proposed in the application of object concepts down to hardware [Dv04a]. They were studied in the section 4.2. Examples of such connectors include [KRK99], [Rad00] and [BRM+06].

Event connectors include the heterogeneous composite connector for interruptions, which consists of Interrupt Service Routines (ISRs), Programmable Interrupt Controllers (PICs) and interruption lines. Transfer of control is supported by IRQ signals that make the processor Program Counter (PC) jump to the ISR, while transfer of data are supported by usual bus transfers and DMA controllers.

Data access connectors are used to read/write data from transient memories such as registers, FIFO, internal and external RAM or persistent memories like flash memories, Direct Memory Access (DMA) controllers, parallel-to-serial data converters using shift registers, rate converter using FIFOs, protocol

converters such as bus bridges. A *data access connector* may be bound to a standard bus interface to configure hardware components.

Linkage connectors include wires connecting IP cores to on-chip/off-chip buses and NoC links.

Stream connectors include hardware pipes i.e. the point-to-point wires used in custom data bus to bind data flow-oriented IP cores within the same FPGA chip that act as hardware filters in the PIPES AND FILTERS architectural pattern [BMR+96], serial bus controllers such as RapidIO and Ethernet controllers, and burst based buses. A stream connector that communicates behind the FPGA chip boundaries may be directly bound to the physical interface i.e. pins. of FPGA chips.

Arbitrator connectors include hardware multiplexers (mux), arbiters, schedulers and address decoders that are typically found in on-chip, off-chip bused such as AHB and NoC routers, or in OSSS guards [BGG+07] and channels [GBG+06].

Adaptor connectors include hardware interface and protocol adapters in bridges and NoC Network Adaptor. Many works in the literature deal with the synthesis of hardware adapters such as [MS98].

Distributor connectors are represented by the broadcast functionality of NoCs and off-chip bus like the CAN bus and socket interface like OCP.

Connector Duct Family We keep the same connector model than [MMP00]. A hardware connector has one or more ducts used to transfer control and data. However, the taxonomy in [MMP00] does describe ducts and the direction of data and control flow. Based on our previous family of interfaces, we refine our taxonomy of hardware connectors by identifying atomic ducts whose roles may have different interaction semantics as shown in figure 7.10. A connector results from the composition of hardware interfaces to represent different interaction semantics. Connector ports or roles implement the chosen hardware interfaces.

In this section, we explore the mapping space for procedure call connectors, which are required by the client-server object model of CORBA. As a consequence, interactions are mainly based on point-to-point and one-to-one communications.

The transfer of control is either blocking for synchronous invocations or not blocking for asynchronous invocations. A procedure call connector is composed of two ducts: one duct for invocation requests and one duct for invocation replies. These ducts may support buffering (e.g. register, FIFO, LIFO or local/system RAM), flow control, reliability, and ordering of transfers. A duct may be a simple set of point-to-point wires, an on-chip bus such as AHB or on-board bus like TI DSP EMIF (External Memory Interface), HPI (Host Port Interface) and Expansion (XBUS) buses. The roles of duct are a composition of two atomic push or pull interactions. The direction of the transfer of control goes from the client to the server and is indicated by the direction of the push and pull arrows in figure 7.10. The direction of the transfer of data goes from a component output port to a component input port. In push interactions, the direction of the transfers of control and data is the same from the client to the server. In pull interactions, transfer of control goes from the client to the server, while transfer of data goes from the server to the client.

The **push connector duct** is only based on push interfaces. Two push connector ducts are used to transfer requests and replies in hardware objects [Frö06] [Rad00] [KRK99] [BRM+06]. The "wire" push connector contains a duct without memory, while the "delay" push connector has a duct with register(s) to delay the arrival of data. This type of connector is also used in software to send messages in the message passing model.

The **pull connector duct** is only based on pull interfaces. We distinguish the *proactive* pull connector in which the master input port initiates the pull interaction on its own and the *reactive* pull connector in which an event triggers the pull interaction. For instance, the *proactive* pull connector is used for polling the value of a register i.e. to perform blocking read. The *reactive* pull connector is used to trigger a read

after an event e.g. an interrupt. In this case, the client is master on the bus, whereas the server is slave on the bus. The server initiates the interaction instead of the client: there is an *inversion of control* [Kra08].

The **push-pull connector duct** uses both push and pull interfaces. It may provide memory and flow control. This connector is used in the producer/consumer communication model like in Unix pipes and sockets, in SystemC channels, OSSS channels [GBG+06] and HLS tools such as [Gra07] [Alt07b]. Producers and consumers act as clients, while the duct acts as a server, which receives push requests from the producer and pull requests from the consumer.

The **pull-push connector duct** represents an active connector duct in which the duct acts as a client, and input and output component ports acts as servers. The duct reads data from an output port and writes data to an input port. For instance, this connector is used by the SCA Core Framework to `get` a provides port object reference and `connect` it with a uses port.

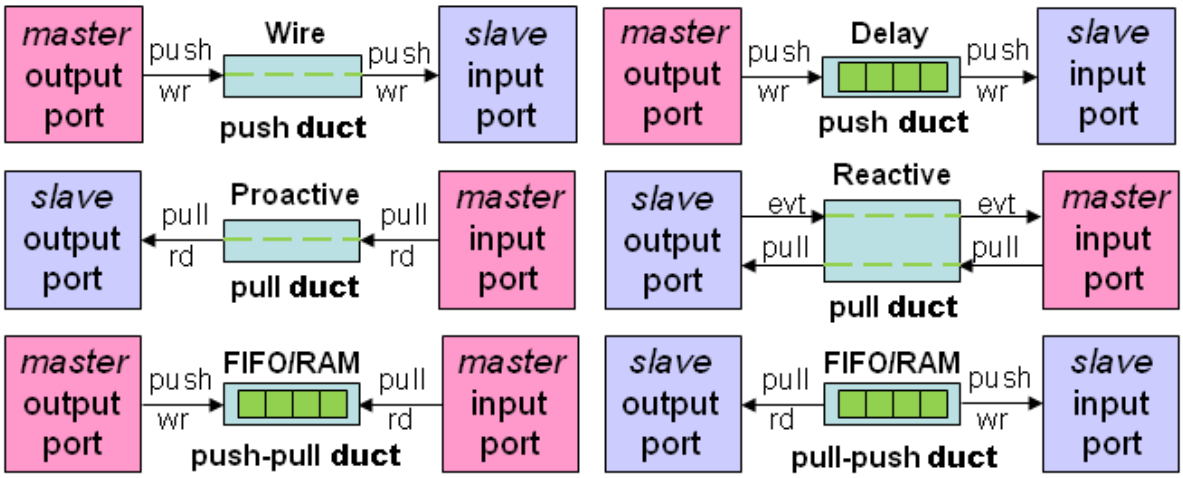


Figure 7.10: Collection of connector ducts supporting different interaction semantics

Mapping configurations Based on our previous family of hardware connectors, we define a set of non-exhaustive mapping configurations to take into account different combinations: hardware components request and reply ports may be master or slave, perform read or writes for requests and responses, and share interface signals. A mapping configuration results from the composition of hardware/software components and connectors. The request and reply messages of an operation call may be transferred in one or two connectors. We identify four mapping configurations for procedure call connectors that are represented in figure in 7.11: 1) two push connector ducts for the request and the reply (peer-to-peer), 2) a push connector duct for the request and a pull connector duct for the reply (client-server or master-slave), 3) one connector duct for request and reply; the hardware interface is thus shared, but requests and replies are not pipelined, 4) a non-blocking connector duct for non-reliable oneway operations, for which no reply is expected, and a blocking connector duct for reliable oneway as in Real-Time CORBA.

Figure 7.12 presents two possible mapping configurations for two component ports. In figure 7.12a, two wire push connector ducts are used for requests and replies, while two FIFO push-pull connector ducts are used in figure 7.12b.

More generally, we argue that the mapping exploration space for connectors include the solutions proposed in [LvdADtH05]. Lachlan et al. identify 16 types of interactions for one way message-passing communication according to time, space and synchronization decoupling, and 16*16 = 256 possibilities for two way interactions. We consider that these interactions should be represented by and "capitalized"

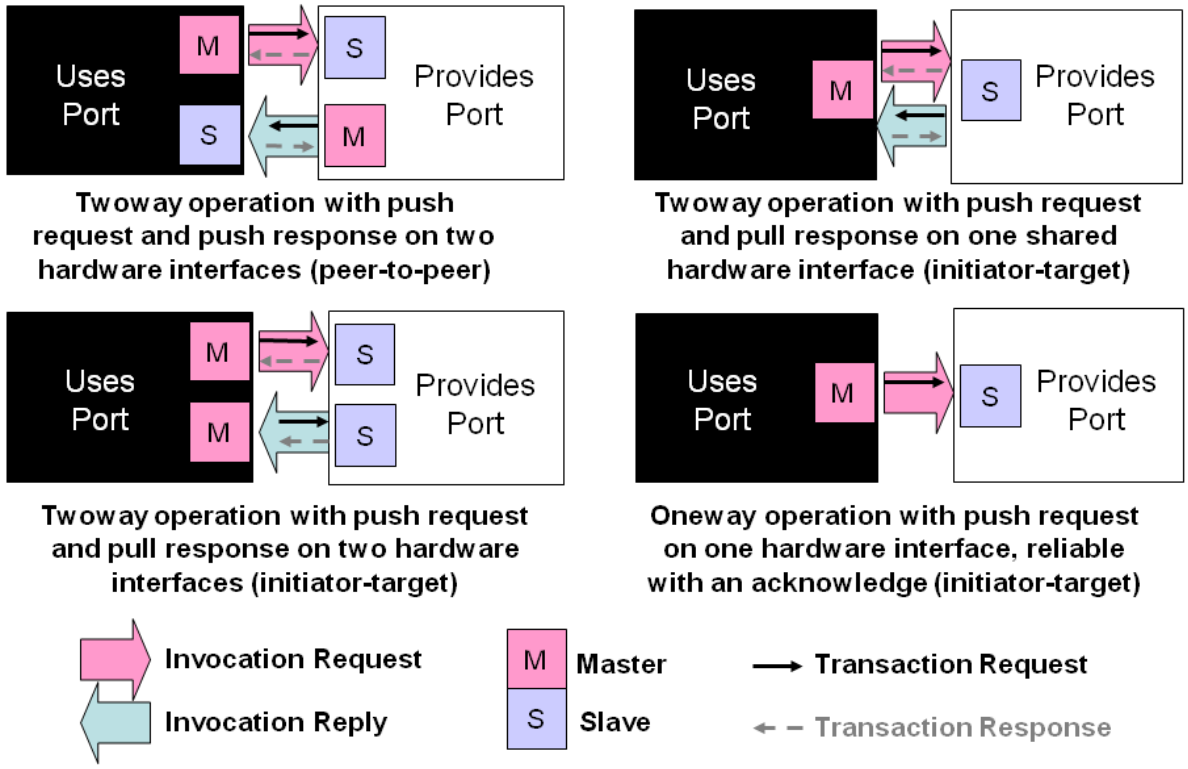
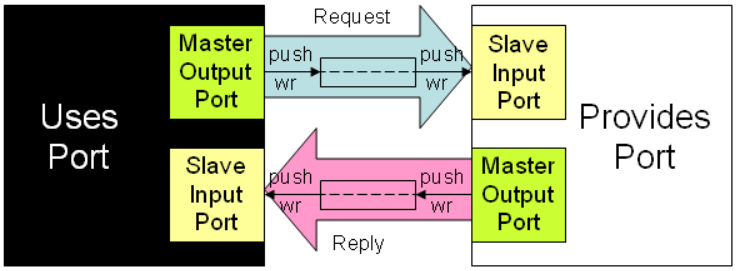
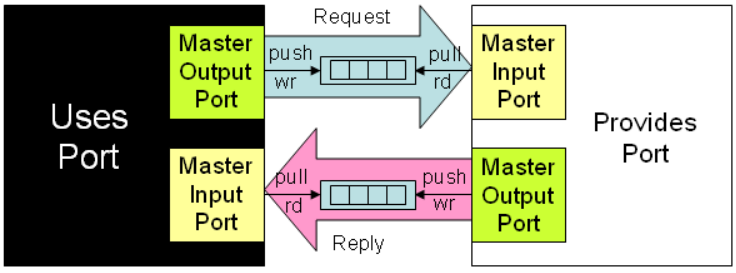


Figure 7.11: Different component interface configurations



(a) With two wire push connector ducts



(b) With two FIFO push-pull connector ducts

Figure 7.12: Two possible mapping configuration for two component ports

in connectors.

One of our objectives is to show that the mapping exploration space is much more vast and the hardware/software interactions much more rich than the design points represented by the works from the hardware object literature .

Hardware Invocation Protocol To provide a coherent programming model for HW and SW CCM components, method invocations on the same CCM port i.e. defined within the same IDL interface are sequential. The hardware invocation interface is thus shared between all methods as in [Rad00]. However, the method invocations on different CCM port may be concurrent as in [Frö06] [BRM+06], consequently they are mapped onto different hardware CCM component ports. These mapping assumptions are coherent with the software interpretation of CCM port as a communication interface. Thanks to this interpretation, the invocation concurrency is explicitly expressed in IDL3 and the state coherency modified by getters and setters is guaranteed thanks to sequential invocations.

Chu presents in [Chu06] handshaking protocols and data transfer protocols. Even if these protocols are presented in the scope of data transfers between hardware components driven by different clock frequencies, they are also applicable to hardware components in the same clock domain. The two main handshaking protocols are based on four or two protocol phases.

The advantage of the four-phase handshaking protocol are that both the master and the slave do not need to know the data transfer rate of each other and/or the clock frequency at which they operate.

The two-phase and four-phase handshaking protocol can be used for a two-way hardware operation call without parameters and a void return value e.g. `void start()` and a reliable oneway operation e.g. `oneway void start()`. Some possible mapping configurations include one pair of `req`/`ack` signals forming one interface for the request-reply, two interfaces with the same direction (push-push) or the opposite direction (push-pull) for the request and the response phase in parallel, one interface for the request then the response phase in serial. These mapping configurations correspond to different trade-off in terms of number of wires, handshaking overhead and reliability.

The two-phase and four-phase handshaking protocol allows the transfer of control like an enable pulse between a master and a slave. The addition of a data signal can be introduced to transfer not only control, but also data. Based on the previous handshaking protocols, three data transfer protocols can be identified with decreasing reliability and overhead: four-phase handshaking transfer, two-phase handshaking transfer and one-phase transfer. The four-phase handshaking transfer has a high latency and blocks the master longer than the other protocols, but both participants make no timing assumption on each other. For synchronous masters and slaves, a data transfer takes 5 cycles as master and slave FSMs have three and two states respectively.

The one-phase transfer allows the maximum throughput of one transfer per clock cycle, but cannot receive back-pressure feedback from the slave to control the data flow. The master and the slave need a timing agreement to guarantee that no data is lost or duplicated during the transfer. The two-phase handshaking transfer is a trade-off between both previous solutions. It can divide the latency of the four-phase handshaking transfer by 2 as a single handshaking phase occurs instead of two.

Interestingly, writing data from one asynchronous sub-system to another is called a *push* operation, while reading data is called a *pull* operation [Chu06]. We have used the same terminology in our mapping proposition.

Unidirectional data transfers can be implemented by a *push* operation and a *pull* operation depending on the data signal direction. Bidirectional data transfers can be implemented by a push-pull operation using two unidirectional data signals, a command signal to indicate the type of operation (push or pull) and an optional address signal to designate a memory location. Since the push and pull operations are mutually exclusive, a single command signal can be shared. Any number and kind of signals can

be further added to the basic handshaking protocols to exchange additional control, address and status information and build more complex protocols like bus protocols.

The two-phase and four-phase handshaking protocols requires synchronization during each transfer and result in a high latency. To transfer a huge amount of data, the synchronization overhead can be reduce using data transfer protocols based on a memory buffer such as a FIFO or a shared memory (RAM). A FIFO only requires synchronization and blocks when a write transfer is initiated when the FIFO is full or when a read transfer is initiated when the FIFO is empty. Otherwise, a data transfer can be performed at each clock cycle. The shared memory buffer may be based on a single port RAM or a dual port RAM to support concurrent read/write at different memory locations. Synchronization only occurs when simultaneous access to the same address is requested.

We presented the basic communication building blocks used to transfer control and data between two hardware modules: four phase, two phase, one phase, FIFO-based and shared memory-based protocols. In conclusion, the choice of a hardware interface and protocol depends on the amount of data to transfers and a trade-off between latency and area consumption as presented in [SKH98]. Our mapping proposition is based on the semi-automatic choice and composition of these building blocks to map CORBA IDL and UML interfaces to hardware interfaces.

Local transfer of operation parameters on the invocation interface As several operation parameters may be mapped onto the same hardware component port, the way parameters are transferred on hardware interface signals should be explicit. The mapping configuration file should specify informations like: the data bus width allocated to each parameter, the endianness to determine if transfers are LSB or MSB first and the data layout of parameters transfer. Indeed, different transfer modes may be distinguished: Parallel or Spatial-Division Multiplexing (SDM) like in [Rad00] and [BRM+06], and Serial or Time-Division Multiplexing (TDM) like in [BRM+06]. Different data type encoding and alignment rules may also be distinguished e.g. unaligned without padding or aligned on data types size with padding as in [Frö06] and CORBA CDR (Common Data Representation). The mapping configuration depends on several trade-offs between timing/area, en/decoding complexity and bandwidth efficiency.

Transfer models to exchange messages on the communication interface Defining a family of hardware interfaces is not sufficient to guarantee communication interoperability. Indeed, messages derived from the IDL may be transparently transferred according to two models: **Address Mapping** (AM) or **Message Passing** (MP).

In the Address Mapping model used in [Frö06], a message containing all packed parameters is serialized according to the bus data width and each message chunk is sent at multiple network addresses. There is at most one address per message chunk. The address decoder inside the skeleton splits message chunks into parameters depending on their data type size. These parameters may be written into registers, FIFOs, RAMs or wires. A last write is required to signal the last message chunk and locally invoke the operation. An IDL specification could be used to generate HW/SW interfaces like [MRC+00].

In the Message Passing model used in [BRM+06], an operationId is mapped to a single address, on which each message chunk is sent. This model only requires $\lceil log_2(\#operations) \rceil$ address signals per IDL interface. The address decoder triggers a dedicated FSM for each invocation message, which splits message chunk into parameters and may write them into storage elements. There is one FSM state per message chunk. This FSM statically knows the last message chunk and no additional write is required to invoke the method.

Common Interfaces and Protocols

Defining a mapping from IDLs to HDLs is a high-level interface synthesis problem. This mapping requires to be flexible enough in reason of the inherent hardware trade-off between timing, area and power consumption.

To support portability and interoperability of hardware components and propose a flexible and extensible mapping, we propose common hardware interfaces and protocols to provide the building blocks of a mapping framework. Such a framework will allow to customize a mapping to fit application requirements.

The purpose of this mapping framework is to explicitly associate with one or more operation parameters the required hardware interface and protocol that best correspond to the implicit communication model of the parameter (control or data, continuous or buffered stream, with or without control flow, memory-mapped access and so on). These interfaces present the mapping framework approach, but can not be considered as exhaustive. New interfaces could be added to address unforeseen needs. The names of the presented signals may be changed by the user to correspond to specific coding rules. User-defined names are alias for the real name of the standard interfaces signal. The only important point is to respect the standard interface protocol.

Figure 7.13 depicts this non-exhaustive and generic set of interfaces: from simple data interface for registered attributes to addressable blocking interface for sequential synchronous invocations. Each successive interface adds semantics: blocking or non-blocking, push semantics for write or pull for read, with or without flow control, addressable or not. These interfaces can be easily mapped onto socket protocols like OCP. Instead of the raw OCP interface, its $\tilde{1}00$ parameters, its bridging profiles and some OCP subsets, these named interfaces explicitly specify the invocation semantics and ease mapping configurability.

These interfaces are presented in more details in annexe A which describes the CORBA IDL3 to VHDL and SystemC Language Mappings.

Mapping of Data Structure

As depicted in figure 7.14, the data address offset, data layout and endianness must be explicitly defined in the mapping configuration file.

Transfer of operation parameters

A logical hardware input port is a a set of VHDL entity I/O ports that is associated with one or several input parameters It implements a standard hardware interface (syntax) and protocol (semantics) e.g. push or pull, blocking or non-blocking, with/without data or flow control and addressable interface. This mapping should be flexible and extensible, because the nature of operation parameters is totally implicit in IDL: control flow, data flow or a mix, continuous or discontinuous, with or without flow control etc. The association between parameters and input ports is explicitly specified in a mapping configuration file. The component-oriented nature of CORBA can be used to define dedicated IDL interfaces for control, configuration or data-oriented aspects. The nature of communication could be expressed by new IDL keywords to better characterize CORBA component ports such as `buffered`, `addressable`, `stream` and infer hardware interfaces based on register, fifo, and RAM. When several input parameters are associated to an input port, the way input parameters are transferred on hardware interface signals should be explicit. The data bus width is specified in the mapping configuration file. It is not required to be a power of two. The endianness is configured to determine if transfers are LSB or MSB first. The data layout of parameters transfer shall also be configured.

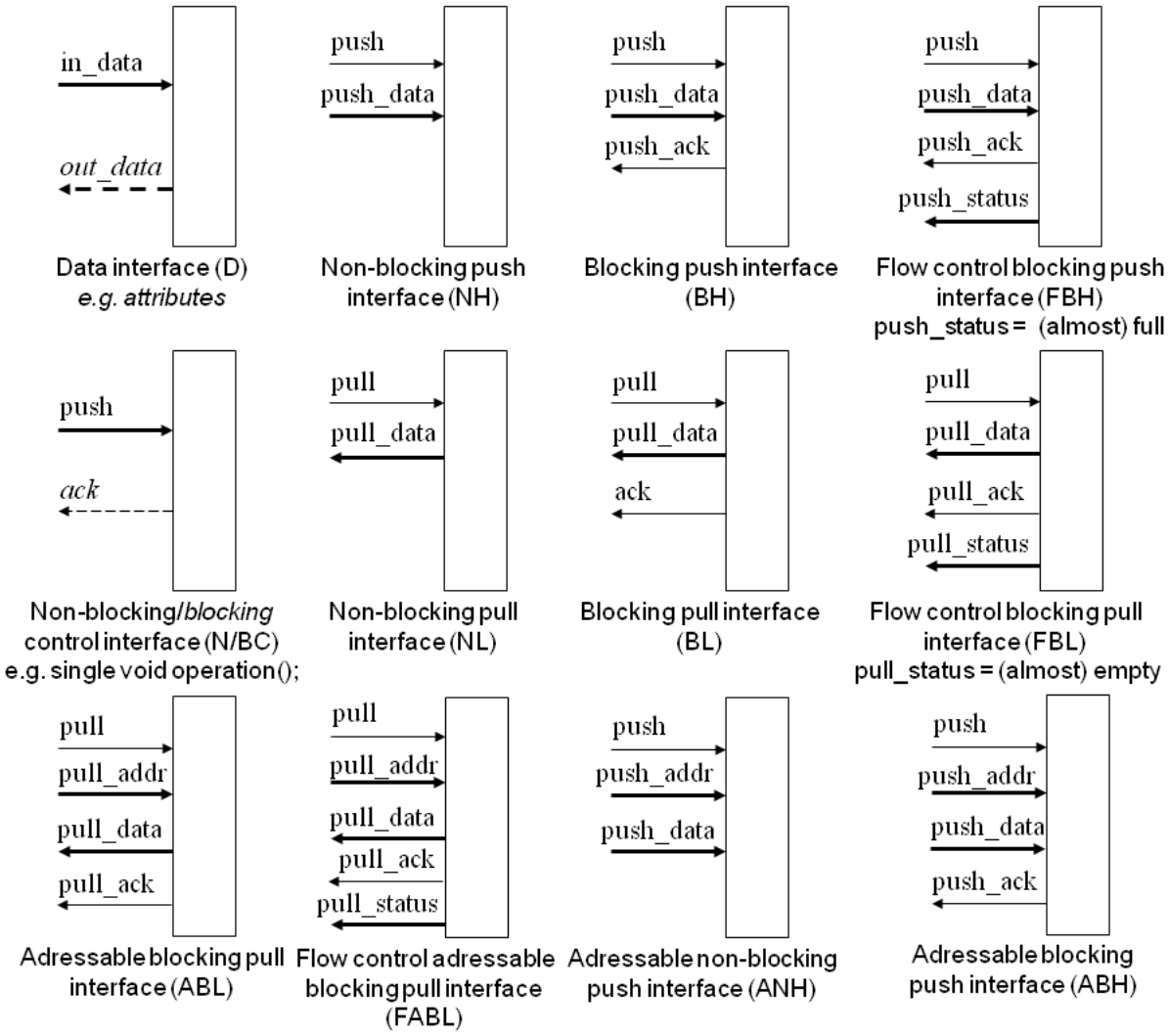


Figure 7.13: Non exhaustive standard interfaces required for invocation and parameters transfer

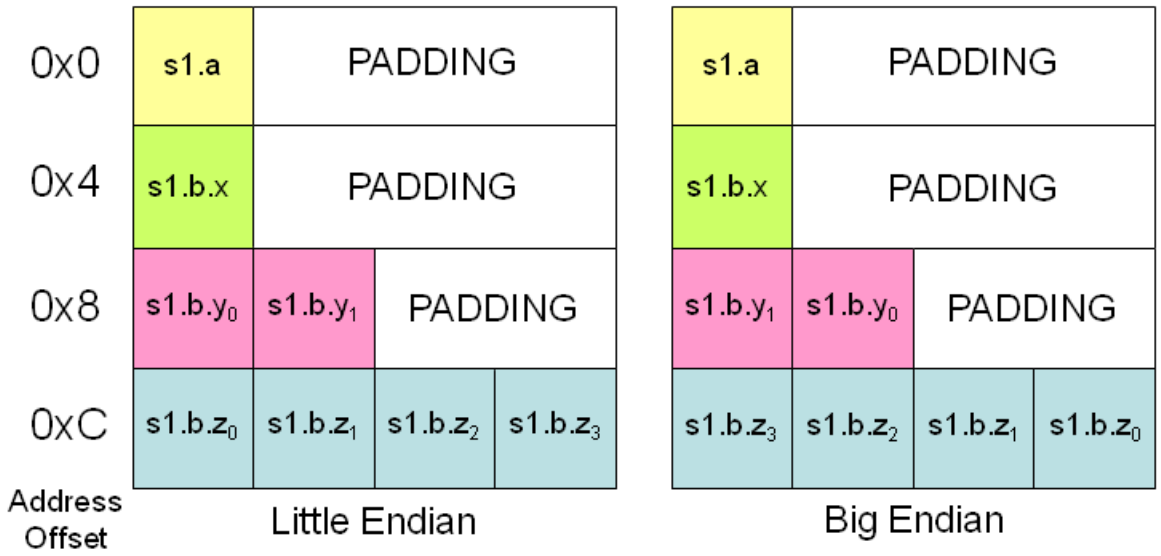


Figure 7.14: Data layouts for data structures aligned on 32bits

Different transfer modes can be distinguished in particular: *Serial* or *Time-Division Multiplexing* (**TDM**), where one parameter (chunk) is transferred at a time, and *Parallel* or *Spatial-Division Multiplexing* (**SDM**) where (one chunk of) each parameter is transferred at a time [Bje05]. Serial transfers require less hardware wires therefore less silicium area and power consumption, but are slower. Parallel transfers require more hardware wires therefore more silicium area and power consumption, but are faster.

Different encoding rules can be distinguished in particular: *with padding* where non-significant data are inserted to align parameters on address boundaries as in CORBA CDR [OMG08b], and *without padding* where no data are inserted to align parameters as in ICE [HS08] [Hen04]. As in ASN.1 Packed Encoding Rules (PER) [IT02] support both alternatives. Unaligned data are more compact but are slower for middlewares to encode and decode. Aligned data are easier to encode and decode but use more bandwidth.

We propose to support all these solutions to best fit each particular application requirements in DRE systems. Designers should indicate to IDL-to-HDL compilers which one they want to use during mapping configuration. For both transfer modes and encoding rules, reusable connectors can be envisaged to convert parameter data from one configuration to another.

Figure 7.15 illustrates the different data layouts to transfer input parameters and the associated tradeoffs in terms of performance (timing, area and power consumption).

Hence, we identify two main solutions to transfer parameters: 1) a dedicated signal for each parameter whose width correspond to the size of the parameter data type. An appropriate naming convention is required to avoid a name clash. This solution is user-friendly and performant, but requires a lot of area.

2) shared wires for all parameters going in the same direction whose width is fixed by a constant (e.g. VHDL generic). This may generalized to all parameters of all operations of an interface. This approach is the most generic, area-efficient, but less user-friendly as parameters are serialized.

The data width used to transfer parameters in parallel or serial may be chosen at instantiation time. The endianness and alignment rules - with or without padding - should be defined. The VHDL data type e.g. record inferred from UML/IDL definitions could not be used in entity/component interface, but within VHDL implementation (architecture) where these data types should be serialized in a generic

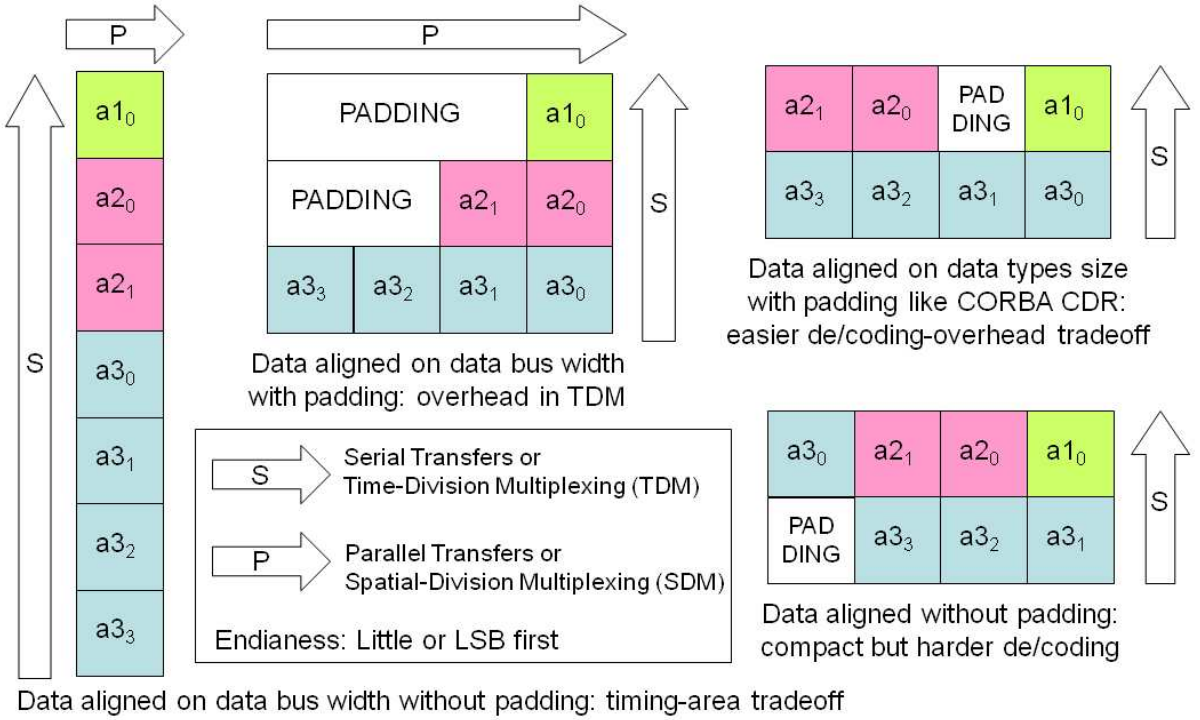


Figure 7.15: Transfer of parameters

way on the shared data bus (solution 2). Each approach has its strengths and weaknesses. It is a trade-off between flexibility, complexity to guaranty portability/interoperability and performance (area, timing, power consumption).

By default, the transfer of parameters should be *Globally Serial Locally Parallel* (**GSLP**). Indeed, request messages are typically globally serialized on buses, while operation parameters may be locally transferred in parallel to maximize performance and avoid a bottleneck regardless of the use of serial or parallel buses. Adapter connectors would be in charge of serialization/deserialization between the bus interface and the hardware invocation interface.

Mapping Illustration and Hardware Component Architecture

We consider a filter component that provides one configuration interface `Config` to configure the mode `m`, the precision `p`, the length `l` and the coefficients `cf` and to `initialize`, `start` and `stop` its processing. This filter also provides and requires a data transfer interface called `DataStream` to receive and send an octet sequence.

```
    interface Config {
      attribute int3 m; attribute octet p;
      attribute short l;
      void initialise(); void start(); void stop();
5     void setCoeff(in sequence<octet, 128> cf);
      //alternatives definitions:
      //void configure(in int3 m, in octet p, in short l);
      //void setMode(in int3 m);void getMode(out int3 m);
    };
10  interface DataStream {
```

```
    void pushOctet(in sequence<octet> strm);
};
component Filter {
  provides Config            configPort;
  provides DataStream  inputPort;
  uses       DataStream  outputPort;
};
```

Listing 7.3: Interface and component definitions for the FIR filter example

The hardware filter component is depicted in Figure 7.16. This figure illustrate the hardware component architecture we propose and a possible architecture template to be filled by an IDL-to-VHDL compiler. As all configuration attributes and methods are defined in the same IDL interface, they share the same *Addressable Blocking Push* (**ABH**) interface for invocation request and Blocking Push (BH) interface for reply. After configuration, these parameters may be accessed concurrently by the IP core as required.

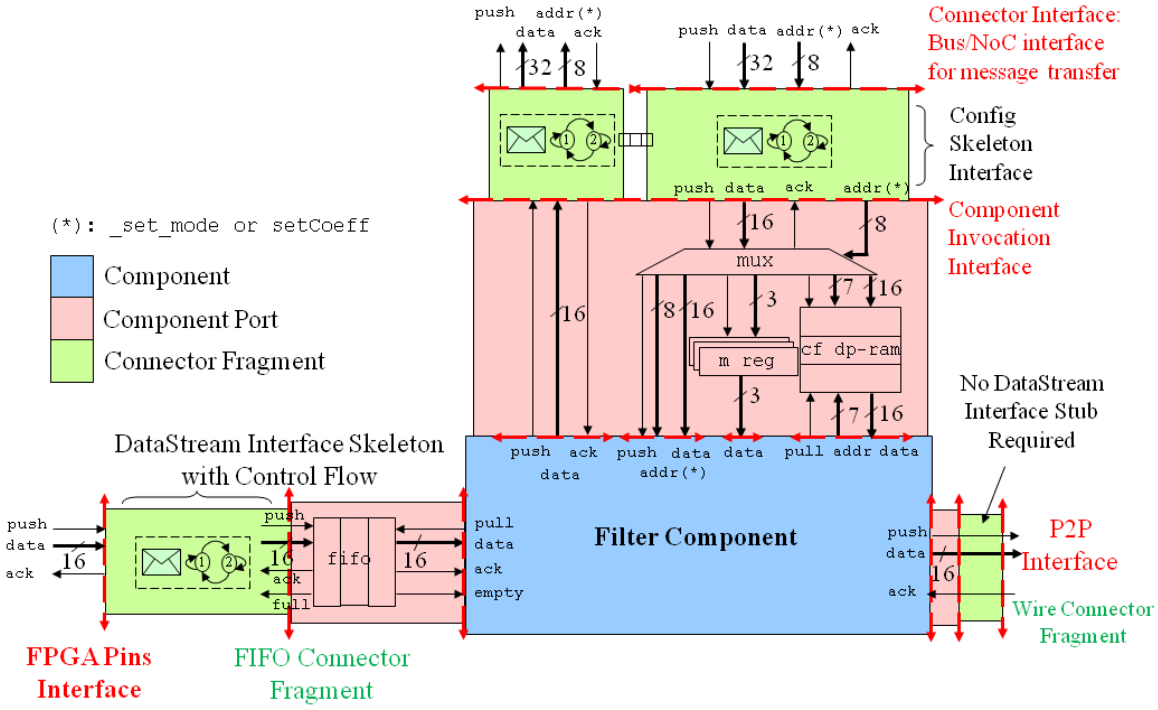


Figure 7.16: Hardware object-oriented component architecture of the FIR filter

Blocking push interfaces are also selected for `DataStream` ports. Connectors are fragmented into so-called connector fragments representing notably stubs and skeletons. The `Config` connector fragment includes a skeleton, which translates invocation messages received from a bus into invocations. The input `DataStream` connector fragment includes a skeleton and a FIFO to buffer data. Due to the point-to-point connection with another component, no stub is required for the output `DataStream` connector fragment. Note that the same sequence data type may be either mapped into RAM(s) for `cf` or a FIFO for `strm` and thus requires different hardware interfaces.

7.3.3 Hardware Mapping Applications

In the following, we illustrate the application of the IDL-to-VHDL mapping to a Transceiver and to hardware CORBA components with the Dining Philosophers problem.

Transceiver

To improve the portability of waveform applications, the SDRForum investigates the standardization of APIs to manage the *Transceiver* of radio platforms.

For instance, the Universal Software Radio Peripheral (USRP) used in the GNU radio project is a hardware radio platform including an FPGA and DAC/ADC on which different transceiver boards may be plugged in to transmit and receive radio signals in predefined frequency bands such as AM/FM radio, cell-phones (GSM), WIFI, GPS or HDTV bands. The USRP boards may be connected through an USB port in the first version or a Gigabit-Ethernet port in the second version to any desktop computer, which control this board using the GNU radio software library. Basically, the GNU radio software applications send and receive IQ samples to and from the USRP board. Interestingly, the GNU radio APIs remains the same regardless of the transceiver board.

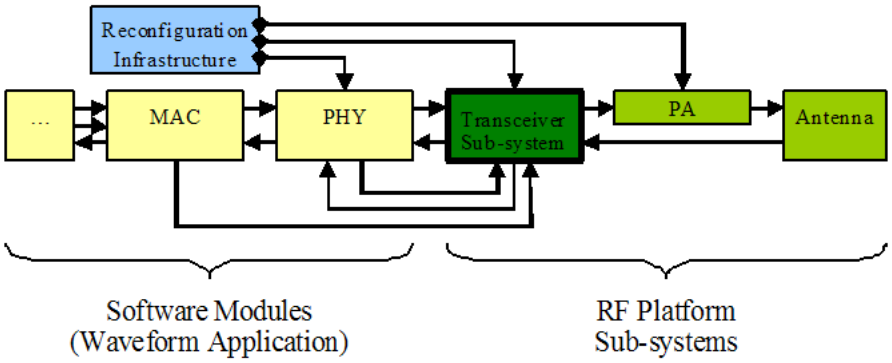


Figure 7.17: Position of the transceiver sub-system in a radio chain

As illustrated in figure 7.17, a Transceiver transforms complex baseband samples from the physical layer (PHY) into a low-power RF analogue signal to the Power Amplifier (PA) in the transmit channel and conversely in the receive channel [NP08]. In addition, the Transceiver perimeter tends to be extended to the PA and the Antenna. The Transceiver may be configured by the Physical layer, the MAC layer or a reconfiguration controller e.g for cross-layer optimization in cognitive radio. Typically, a RF front-end contains one or several sets of transceiver, Power Amplifiers (PA) and antennas for instance in Multiple Input Multiple Output (MIMO) radio systems using multiple antennas. For instance, in the open source hardware design written in Verilog of the USRP v1, the FPGA contains a pair of DUC/DDC including hardware filters and a CORDIC-based NCO.

The *Transceiver Facility* specification proposed by Thales [NP09] defines a set of radio-related *Common Concepts* and *Features*, which are organized as a toolbox allowing the control and configuration of a Transceiver. The Common Concepts define radio-specific notions such as `Base-band Samples`, `Channelization`, `Dwell Structure`, `Frequency Set`, `RF Level` and `Gain`.

Each Feature represents a capability within the Facility which may be selected to satisfy waveform application needs and implemented by the Transceiver. In particular, the Transceiver Facility defines nine features: `Transmit`, `Receive`, `HalfDuplexControl`, `FrequencyHoppingControl`, `RxAdaptiveGainControl`, `TxAdaptiveLevelControl`, `AsynchronousTxFlowControl`,

`DeviceManagement` and `ErrorsManagement`. For more information, the reader may refer to [NP08]. Each Feature specifies a set of APIs and non-functional QoS properties called "performance attributes" to allow application or platform developers to manage the transceiver capabilities. A Transceiver may be viewed as an abstract component having a set of optional ports, which are bound to a chosen Feature interface. Following a MDA/MDE approach, the objective of this specification is to define abstract interfaces in a Platform-Independent Model (PIM) and map them to software and hardware interfaces in Platform-Specific Models (PSM). These abstract interfaces are primarily defined in UML class diagrams, while definitions in CORBA IDL were also requested by SDRForum partners to implement them in a SCA CoreFramework.

In this work, we participated with an hardware engineer from Thales and Indra to the definition of the hardware interfaces for the *Transmit* and *Receive* Features. This experiment allows me to apply my mapping proposition and evaluate it on a concrete example from the beginning of our interface and component-centric specification and design flow. Figure 7.18 provides an overview of the Transmit Channel feature. A `Transmit Channel` of a Transceiver subsystem performs the up-conversion of an input Baseband Signal burst into an RF Signal burst. It basically consists of a FIFO to store Baseband samples and an up-conversion chain which performs the upsampling and filtering of baseband samples to generate the RF signal.

The Transmit Channel is managed by two APIs: `TransmitDataPush` and `TransmitControl`. The `TransmitDataPush` interface contains an abstract operation called `pushBBSamplesTx` to send packets of base-band samples in the Transceiver FIFO. Its performance attributes include the expected sample rate and the size of the base-band samples packet. The `TransmitControl` interface is used by a Waveform Application to determine when and how the RF signal is transmitted during a *Transmit Cycle*.

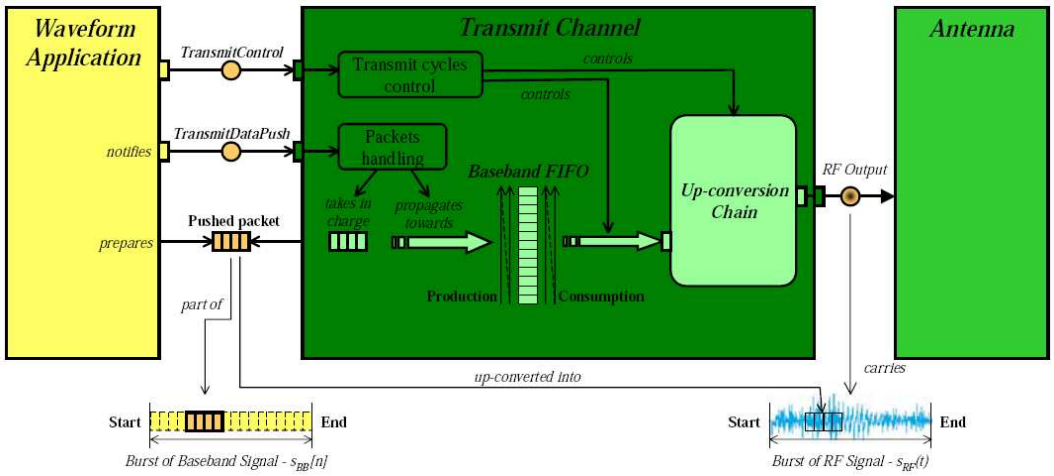


Figure 7.18: Overview of the Transmit Channel Feature [NP09]

The specification has deliberately been written to be simple as possible to ease its software and hardware implementation. Basically, *bidirectional* IDL operations with `in`, `out` and `inout` parameters and exceptions may be translated into *unidirectional* operations with only `in` parameters to simplify synchronization problems. As no IDL interface was defined in the first version of the specification [NP09], listing 7.4 presents a possible interface.

```
module TransceiverFacility {
  // BBSample and BBPacket Concepts
3 typedef struct BBSampleStruct {
```

```
    short valueI;
    short valueQ;
  } BBSample;
  typedef sequence<BBSample> BBPacket;
  // Transmit Feature
  module Transmit {
    interface TransmitDataPush {
      void pushBBSamplesTx(in BBPacket thePushedPacket, in boolean
          endOfBurst);
    };
  };
};
```

Listing 7.4: `TransmitDataPush` interface in CORBA IDL

Listing 7.5 presents the proposed hardware `TransmitDataPush` interface described in VHDL.

```
library IEEE;
use IEEE.std_logic_1164.ALL;
entity TransmitDataPush is
generic(
  G_CODING_BITS : natural := 16 -- for instance
);
port (
  -- Common Signals
  clk : in std_logic; -- Clock signal
  rst_n : in std_logic; -- Reset signal
  -- pushBBSamplesTx
  BBSample_I : in std_logic_vector(G_CODING_BITS-1 downto 0);
  BBSample_Q : in std_logic_vector(G_CODING_BITS-1 downto 0);
  BBSample_write : in std_logic;
  BBPacket_start : in std_logic;
  BBPacket_end : in std_logic;
  BBSample_ready : out std_logic
  -- Other signals
);
end entity TransmitDataPush;
```

Listing 7.5: Hardware `TransmitDataPush` interface in VHDL

A blocking push data interface based on a two-phase handshake was chosen to allow the communication between a client and a server with different data production and consumption rates. A `BBSample` is transmitted only if both `BBSample_write` and `BBSample_ready` signals are asserted. The `BBSamplePacket` concept is described in VHDL using additional signals to indicate the start and the end of a sample packet. Every feature using the `BBSamplePacket` concept has to implement the following protocol: the `BBSamplesPacket_start` signal is asserted with the first sample of a packet, the `BBSamplesPacket_end` signal is asserted with the last sample of a package. In any other case, `BBSamplesPacket_start` and `BBSamplesPacket_end` signals are deasserted.

During the elaboration of this mapping with hardware engineers, I observed that VHDL records are not flexible enough as they cannot be parametrized by VHDL `generic`s. Obviously, a "software-oriented" naming convention like `pushBBSamplesTx_push` instead of `BBSample_write` is not natural for hardware engineers, that is what an IDL-to-VHDL mapping should allow the renaming of signals like in OCP configuration file [OI06], while respecting a standardized interface protocol like the blocking push data interface in this case. It is also more natural to define a VHDL `entity` rather than

a VHDL `component` in a VHDL `package` even if both basically define the same hardware interface. Hardware engineers do not understand the necessity of defining a hardware interface in HDL to support the portability of RTL code at source-level like for software APIs, even if the SCA MHAL [JTR07] already proposes such interfaces for FPGA/ASIC.

As shown in figure 7.19, a simple table describing the direction, size and meaning of signals are sufficient for hardware engineers.

Signal Name	Direction	Signal Format	Signal Units	Signal Min Value	Signal Max Value	Notes
Clk	in	1-bit Discrete	Clock	0->1 = Active Edge	1->0 = Passive Edge	XX MHz Transmit Clock
rst_n	in	1-bit Discrete	Level	0	1	Active-low, asynchronous reset.
BBSample_I	In	CodingBits-1 bit Signed	N/A	-32,768	32,767	Imaginary Sample value
BBSample_Q	in	CodingBits-1 bit Signed	N/A	-32,768	32,767	Quadrature Sample value
BBSample_write	in	1-bit Discrete	Level	0	1	Data valid flag to request sample transmission
BBSamplesPacket_start	in	1-bit Discrete	Level	0	1	Packet start flag
BBSamplesPacket_end	in	1-bit Discrete	Level	0	1	Packet end flag
BBSample_ready	out	1-bit Discrete	Level	0	1	Accept/ready flag to activate sample transmission

Figure 7.19: TransmitDataPush VHDL entity signals

Defining hardware interface signals is not sufficient to use them, that is why their timing behavior has also been specified by a timing diagram presented in figure 7.20.

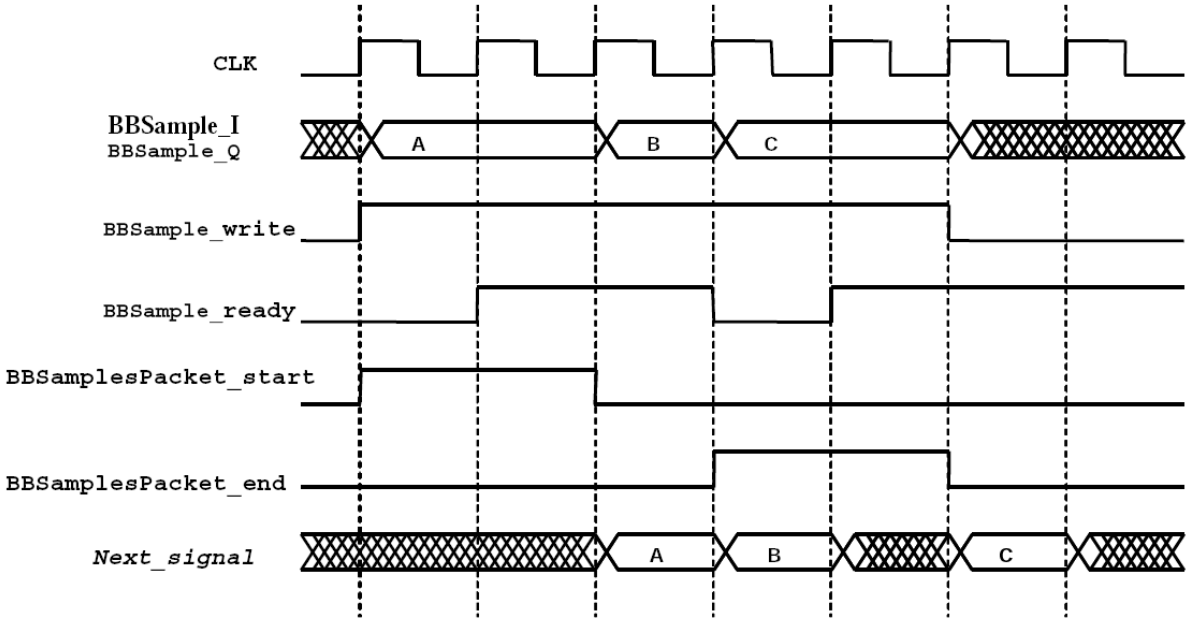


Figure 7.20: BBSamplePacket TransmitDataPush Timing diagram

In a similar way, the mapping of the `Receive` interface has been proposed. The `pushBBSamplesTx` interface signals have only the opposite direction.

Even if we also proposed internally IDL interfaces for the configuration services, they did not have

been incorporated in the first release of the specification. Nevertheless, we present our proposition for the `TransmitControl` IDL interface in listing 7.6. In this interface, the `createTransmitCycleProfile` operation [NP09] is not represented as this kind of "factory" method is useful for a software implementation, but not for a hardware implementation. Indeed, such an operation is used to create a software data structure, which is already available in the hardware implementation e.g. under the form of registers. Interestingly, this shows that a IDL-to-RTL mapping should allow to choose which operations are required to be implemented in hardware instead of blindly generating RTL code for all IDL operations.

```
module TransceiverFacility {
  typedef Time          ULong; // for instance
  typedef Frequency     ULong; // for instance
  typedef AnaloguePower ULong; // for instance
  // Transmit Feature
  module Transmit {
    interface TransmitControl {
      void configureTransmitCycle (
        ULong                 targetCycleId,
        Time                    requestedTransmitStartTime,
        Time                    requestedTransmitStopTime,
        Frequency           requestedCarrierFrequency,
        AnaloguePower requestedNominalRFPower );
      void setTransmitStopTime (
        ULong                 targetCycleId,
        Time                    requestedTransmitStopTime );
    };
  };
};
```

Listing 7.6: `TransmitControl` interface in CORBA IDL

Listing 7.7 presents the proposed hardware `TransmitControl` interface described in VHDL. We choose a non-blocking push interface to locally transfer in parallel all operation parameters. According to our mapping proposition, the configuration registers are located in the `TransmitControl` skeleton providing storage and synchronization services. The `_write` signals are used to forward the invocation to the application component and inform it that configuration registers may have been modified. The configuration values may be written to the skeleton either as a message containing all operation parameters in a FIFO or as memory-mapped accesses for each parameter. Regardless of the message-passing or memory-mapping interface chosen to transfer parameters, the hardware application component may continuously read register values via the `TransmitControl` interface signals (*data interface*) and keep inform for their modifications. To not restrict the performance of hardware application components regardless of the platform communication service, we propose the concept of *Globally Parallel Locally Serial* (**GPLS**) transfer. Globally at the system level, application data are serialized on platform communication buses e.g. on 16 or 32 bits. Locally at the IP core level, a serial data transfer to and from an IP represents an inherent communication bottleneck that could not efficiently use a parallel bus. Conversely, a local parallel transfer provide the maximum performance regardless of the use of a serial or parallel bus. We conclude that the transfer of operation parameters between an application component and a platform adapter (stub/skeleton or container) should be by default parallel, while transfers between this platform adapter and another platform adapter via an hardware ORB and the available transport layers (on-chip/off-chip bus/NoC) are generally serialized. In this way, the routing constraints are only local between a component and its adapter, which may adapt the data widths of the application invocation interface and the platform connectivity interface. In any case, the final IDL-to-RTL mapping should be configured according to the specific QoS needs of the application and the capabilities of the platform.

```
library IEEE;
use IEEE.std_logic_1164.ALL;
entity TransmitControl is
generic(
  G_TIME                    : natural := 32 -- for instance
  G_FREQUENCY               : natural := 32 -- for instance
  G_ANALOGUE_POWER : natural := 32 -- for instance
);
port (
  -- Common Signals
  clk : in std_logic; -- Clock signal
  rst_n : in std_logic; -- Reset signal
  -- configureTransmitCycle
  configureTransmitCycle_write : in std_logic; -- push interface
  configureTransmitCycle_targetCycleId : in std_logic_vector(31 downto 0);
  configureTransmitCycle_requestedTransmitStartTime : in std_logic_vector(
     G_TIME-1 downto 0);
  configureTransmitCycle_requestedTransmitStopTime : in std_logic_vector(
     G_TIME-1 downto 0);
  configureTransmitCycle_requestedCarrierFrequency : in std_logic_vector(
     G_FREQUENCY-1 downto 0);
  configureTransmitCycle_requestedNominalRFPower : in std_logic_vector(
     G_ANALOGUE_POWER-1 downto 0);
  -- setTransmitStopTime
  setTransmitStopTime_write : in std_logic; -- push interface
  setTransmitStopTime_targetCycleId : in std_logic_vector(31 downto 0);
  setTransmitStopTime_requestedTransmitStopTime : in std_logic_vector(
     G_TIME-1 downto 0);
  -- Other signals
);
end entity TransmitControl;
```

Listing 7.7: Hardware `TransmitControl` interface in VHDL

Hardware CORBA Components with the Dining Philosophers problem

To illustrate the IDL3-to-VHDL mapping, we reuse the same example which has been implemented in CCM in OpenCCM[67] and MicoCCM[68]: the Dining Philosophers.

Dining Philosophers The Dining Philosophers example is a classical concurrency problem. Philosophers sit around a round table. There is a fork between each Philosopher. Philosophers think until they become hungry, then they try to take two forks in their left and right side in order to eat. Obviously, there are not enough forks for all philosophers to eat at the same time. The objective is that philosophers do not die from starvation. In the following section, we present the CORBA IDL3 modeling of this problem and a possible mapping in VHDL.

IDL3

[67] http://openccm.ow2.org

[68] http://www.mico.org, http://www.fpx.de/MicoCCM

```
--philo.idl3
import Components;
module DiningPhilosophers {
    exception InUse {};
    interface Fork {
      void get () raises (InUse);
      void release ();
    };
    component ForkManager {
      provides Fork the_fork;
    };
    home ForkHome manages ForkManager {};
    enum PhilosopherState {
      EATING, THINKING, HUNGRY,
      STARVING, DEAD
    };
    eventtype StatusInfo {
      public string name;
      public PhilosopherState _state;
      public unsigned long ticks_since_last_meal;
      public boolean has_left_fork;
      public boolean has_right_fork;
    };
    component Observer {
      consumes StatusInfo info;
    };
    home ObserverHome manages Observer {};
    component Philosopher {
      attribute string name;
      uses Fork left;
      uses Fork right;
      publishes StatusInfo info;
    };
    home PhilosopherHome manages Philosopher {
      factory new (in string name);
    };
};
```

RTL mapping in VHDL We apply our mapping proposition to the previous IDL3 definitions in *philo.idl3* file according to the mapping rules defined in the annexe. The following code has been manually generated in the *philo.vhd* file. Each VHDL component is associated to a dedicated VHDL file which contains VHDL entities: *ForkManager.vhd*, *Observer.vhd* and *Philosopher.vhd*. Obviously, these files are not represented for space reasons. The *Corba.vhd* contains the definitions of CORBA types.

```
--philo.vhd
library IEEE;
use IEEE.std_logic_1164.all;
library CORBA;
```

```
use CORBA.CORBA_types.all;
library philo;

package DiningPhilosophers is
  constant getId : std_logic_vector(1 downto 0) := "01";
  constant releaseId : std_logic_vector(1 downto 0) := "10";
  constant DiningPhilosophersExc : natural := 1;
  constant InUse : std_logic_vector(DiningPhilosophersExc-1 downto 0) := "1";
  component ForkManager port (
    clk : in std_logic;
    rst_n : in std_logic;
    the_fork_req : in std_logic;
    the_fork_opID : in std_logic_vector(2 downto 0);
    the_fork_ack : out std_logic;
    the_fork_exc : out CORBA_exception;
    the_fork_exc_id : out  std_logic_vector(DiningPhilosophersExc-1 downto 0)
  );
  end component ForkManager;
  type PhilosopherState is (EATING, THINKING, HUNGRY, STARVING, DEAD);
  type StatusInfo is record
    name : CORBA_short;
    state : PhilosopherState;
    ticks_since_last_meal : CORBA_unsigned_long;
    has_left_fork : CORBA_boolean;
    has_right_fork : CORBA_boolean;
  end record StatusInfo;
  component Observer port (
    clk : in std_logic;
    rst_n : in std_logic;
    info_req : in std_logic;
    info : in StatusInfo
  );
  end component Observer;
  component Philosopher port (
    clk : in std_logic;
    rst_n : in std_logic;
    get_name : in std_logic;
    set_name : in std_logic;
    left_req : out std_logic;
    left_opID : out std_logic_vector(2 downto 0);
    left_ack : in std_logic;
    left_exc : out  CORBA_exception;
    left_exc_id : out  std_logic_vector(DiningPhilosophersExc-1 downto 0);
    right_req : out std_logic;
    right_opID : out std_logic_vector(2 downto 0);
    right_ack : in std_logic;
    right_exc : out  CORBA_exception;
    right_exc_id: out std_logic_vector(DiningPhilosophersExc-1 downto 0);
```

```
    info_req : out  std_logic;
    info : out  StatusInfo
  );
  end component Philosopher;
end package DiningPhilosophers;
```

7.3.4 Mapping Implementation and Code Generation

Although the OMG promotes an MDA approach based on standards such as MOF, UML, XMI, and transformation languages like QVT (Query/View/Transformation) and MOF-to-Text (M2T), no standard meta-model is standardized for standard programming languages such as C, C++ and Ada. As far we know, only a Java meta-model has been standardized by the OMG in the scope of the UML Profile for Enterprise Distributed Object Computing (EDOC). As a result, standard CORBA IDL language mappings are not standardized using a MDA approach with standard model-to-model transformations, but using language-to-language mappings with non-executable textual specifications and without any reference implementation [OMG08c] [OMG01] [OMG99] [OMG08d].

As illustrated in [JTR05] with OCP and by high-level synthesis tools from C-like languages, we argue that the mapping between software interfaces in UML and CORBA IDL to hardware interfaces in HDLs needs to be much more flexible and configurable than classical one-to-one mappings between software programming languages.

Besides, some works have proposed in the literature meta-models and UML profiles for VHDL [Dv04a] [Frö06] [Beu07] [WAU+08] and SystemC [RSRB05] [WZZ+06]. Such works may be a starting point for standard OMG specifications.

To specify standard language mappings from textual or graphical specification languages such as UML, CORBA IDL, AADL, Domain Specific Languages (DSLs) to software programming languages such as C, C++, Ada, and Hardware Description Languages (HDLs) such as VHDL, Verilog, SystemC and SystemVerilog which are compliant with a MDA approach:

1. a meta-model must be specified using OMG Meta-Object Facilities (MOF) for each of these input and output languages to efficiently exploit their concepts and semantics in model transformations

2. model-to-model transformations must be defined from input meta-models to output meta-models using MOF QVT transformation language,

3. model-to-text transformations must be defined using the MOF Model to Text Transformation Language (MOFM2T) to generate source code from models according to language syntax.

In this work, we focus on mapping object-oriented software components defined in UML and/or CORBA IDL3 to hardware components at register-transfer level (RTL) in VHDL and transactional level in SystemC.

Interestingly, the UML profile for CORBA and CCM [OMG08g] already specifies a standard meta-model and UML stereotypes for CORBA objects and components. Parts of the CORBA/CCM meta-model are depicted in figure 7.21. To apply an OMG MDA approach for software to hardware language mappings, a meta-model and a UML profile for VHDL, Verilog and SystemC RTL/TLM should also be standardized. Such meta-models define the structured data model of the language and language keywords are data types (classes) of the meta-model.

The CORBA and CCM meta-model contains four packages:

- the **BaseIDL** package is a meta-model of the CORBA Interface Repository (IR) [OMG08b] which was initially defined in IDL for CORBA objects. It is depicted in figure 7.21.

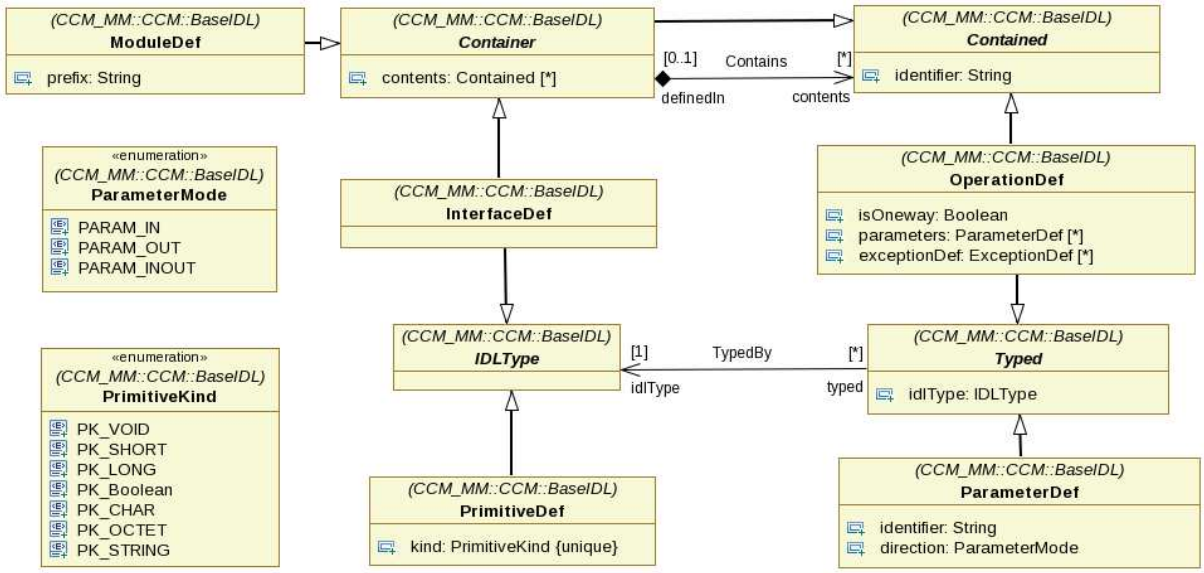


Figure 7.21: Simplified UML class diagram of the BaseIDL package of the CCM meta-model

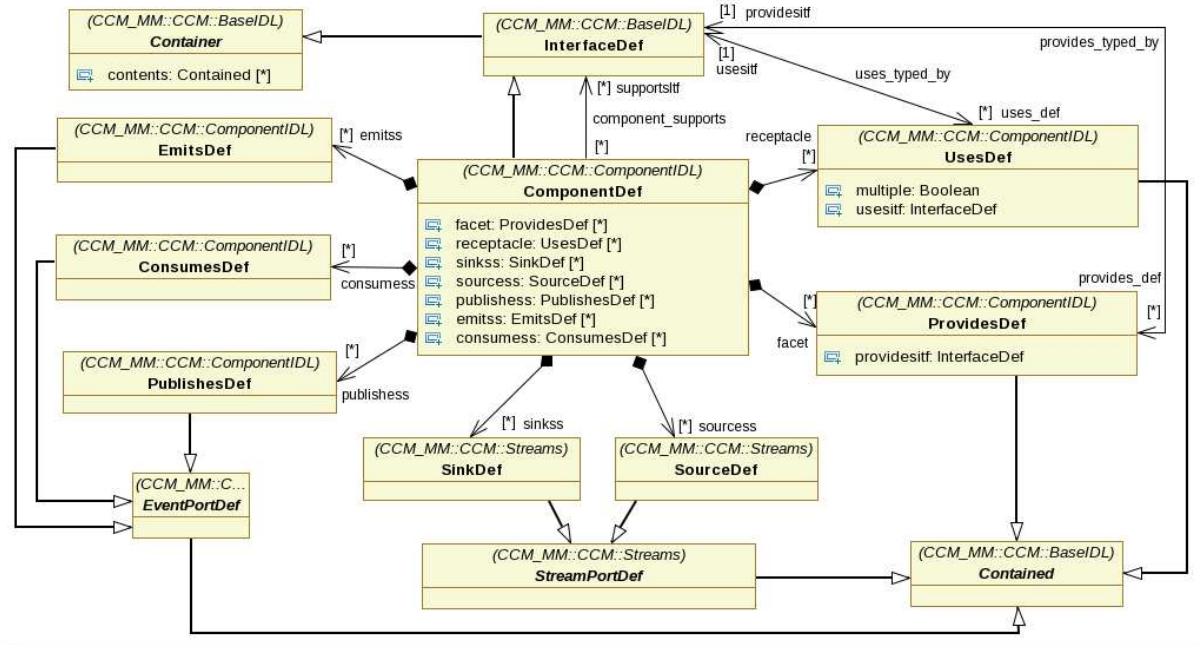


Figure 7.22: Simplified class diagram of the ComponentIDL package of the CCM meta-model

- the **ComponentIDL** package extends the BaseIDL package with the CORBA Component Model (CCM) concepts.
- the **CIF** package is a meta-model of the language used to describe component implementations.
- the **Deployment** package is a meta-model of the Deployment and Configuration (D&C) concepts for CCM such as the assembly of CCM components. It can be used to generate CCM XML descriptors.
- the **Streams** package extends the CCM meta-model to describe the exchange of continuous data streams between CORBA components. These streams seem to initially target multimedia streams for music and video contents over computer networks, but could be relevant for data streaming in Software Defined Radio applications.
- the **CCMQoS** package extends the CCM meta-model with the definition of QoS properties for CORBA components.

Note that in the Lightweight CORBA Component Model (LwCCM) meta-model defined in [OMG08g], the event ports and provides/uses ports are included, but not the stream ports.

In a similar way, to apply an MDA approach to the SCA, a UML profile for SCA concepts and a meta-model for the SCA component model need to be defined. The UML profile for SCA is already standardized within the UML profile for Software Radio [OMG07e]. Moreover, since SCA uses the object-oriented CORBA IDL 2.x to define SCA component interfaces, the meta-model for these interfaces is already standardized in the BaseIDL package of the UML profile for CORBA and CCM. Finally, since the SCA component model evolved from an early version of the CORBA Component Model (CCM), the meta-model for the SCA component model can be considered as a subset of the CORBA Component Model which is standardized in the ComponentIDL package of the UML profile for CORBA and CCM. For instance, the uses and provides ports of SCA components that are described in SCA Software Component Descriptors (SCD) can be considered as the facets and receptacles of CCM components. Furthermore the SCDs specify some types of uses and provides components ports such as control, data, responses, and test ports. These types of ports have been refined and included in the UML profile for Software Radio as stereotypes for component ports such as `ControlPort`, `DataControlPort`, `DataPort`, `StreamControlPort` and `StreamPort`. Besides, the SCA and CCM component models rely on different base interfaces, respectively `Resource` and `CCMObject`, but these interfaces are internal to the component model and are not for instance part of the CCM meta-model. Hence, the meta-model for SCA and Software Radio components may be defined as the CCM meta-model extended to `Control` ports, `DataControl` ports, `Data` ports and `StreamControl` ports.

The necessity for interoperability between multiple input and output standards requires a flexible IDL compiler architecture, which can certainly benefit from an MDA approach. Each input and output formalism needs respectively a dedicated front-end and backend centered around central models. Input and output formalism are described by a meta-model and IDL compilation can be revisited as a transformation of models between between input models to output models.

Architecture of the IDL3-to-VHDL compiler prototype

From a general point of view, we propose the following architecture for an IDL3-to-HDL compiler:

- multiple input standards such as domain-specific UML profiles like UML for SDR and MARTE, CORBA IDL, Domain-Specific Languages (DSLs) and Domain-Specific Modeling Languages (DSMLs).

- object-oriented component meta-models e.g. for SCA, CCM, but also VHDL, Verilog and SystemC.
- multiple output standards for VHDL, Verilog, SystemC, IPXACT SPIRIT.

Obviously, already standardized language mappings to software languages can be included notably to target SDR embedded platforms using C, C++ and Ada.

Model transformations can be used from input formalisms to software component models such as SCA and CCM and from software component models to hardware component models for VHDL, Verilog and SystemC.

We have built a prototype for an IDL3-to-VHDL compiler using the MDA principles. The implementation of our prototype uses the front-end of an open-source project developed at Thales called MyCCM High Integrity (HI) [69] available on source forge web site. MyCCM-HI is a C component framework for critical distributed real time embedded software. MyCCM-HI contains a meta-model of the Lightweight CORBA Component Model described using the meta-language of the Eclipse project called Ecore.

Eclipse is an open-source community which develops a Java-based software development platform. The Eclipse Project was originally initiated by IBM in 2001. Eclipse provides an Integrated Development Environment (IDE) which can be extended by means of user-developed or free/commercial third-party components called plug-ins or bundles. These components are besides developed against a service-oriented component framework called OSGi whose implementation is called Equinox. The Eclipse Foundation hosts a number of Eclipse projects notably the Eclipse Modeling Project. The Eclipse Modeling Project contains subprojects such as the Graphical Editing Framework (GEF) to build domain-specific editors, Graphical Modeling Framework (GMF) project to build Domain-Specific Modeling Languages (DSML), the Eclipse Modeling Framework (EMF) project to build Domain-Specific Meta-Model, the Model-to-Model Transformation (M2M) and the Model To Text (M2T) projects, Textual Modeling Framework (XText) project to build Domain-Specific Languages (DSLs). The EMF project allows developers to describe a structured data meta-model in UML, in XML as XML Schema Definition (XSD) or Java and to generate the associate Ecore meta-model. Based on this meta-model, Eclipse can generate Java code to manipulate and serialize a compliant model in XML and to create and edit models with an automatically generated simple editor.

Extension of the CCM meta-model for RTL mapping configuration

We have extended the Ecore meta-model of LwCCM from MyCCM-HI to allow the flexible configuration of the IDL3-to-HDL mapping. In addition to the existing `baseidl`, `cif` and componentidl packages in the root `ccm` package, we introduce a `RtlMappingConfiguration` package which is depicted in figure 7.23.

This package contains a data model with the following mapping information:

- the `FlowControlKind` enumeration indicates how a hardware component interface handles flow control. For instance, flow control may be *Proactive* e.g. using a accept signal or *Reactive* e.g. using a busy signal.
- the `ProtocolKind` enumeration indicates a signal-based interface protocol such as *TwoPhaseHandshake* or *FourPhaseHandshake*.
- the `AlignmentKind` enumeration indicates how the alignment of data in a message and on the wire. The alignment may be *OnZeroBytes*, *OnTwoBytes*, *OnFourBytes* or *OnDataTypeSize* like in CORBA GIOP.

[69] http://myccm-hi.sf.net

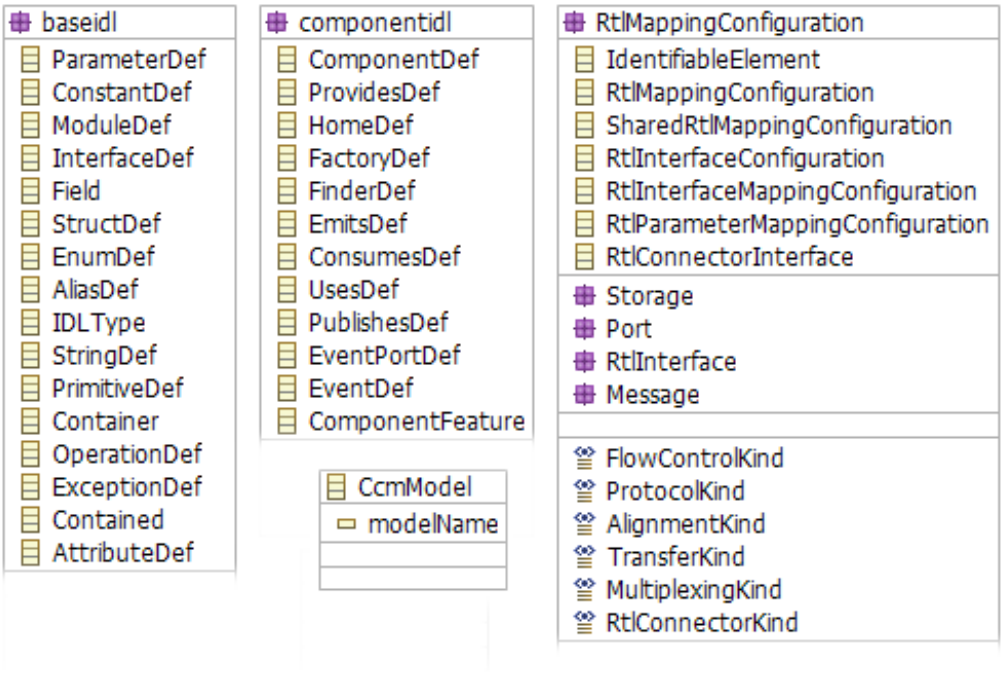


Figure 7.23: CCM meta-model with RtlMappingConfiguration class

- the `MultiplexingKind` enumeration indicates how message data are multiplexed in time and space on the wire for instance using *TimeDivisionMultiplexing* or *SpatialDivisionMultiplexing*
- the `MultiplexingKind` enumeration indicates how message data are multiplexed in time and space on the wire for instance using *TimeDivisionMultiplexing* or *SpatialDivisionMultiplexing*
- the `RtlConnectorKind` enumeration indicates the kind of hardware connectors used to interconnect two or more hardware or software components. For instance, a connector may *Point2Point*, *AMBA_AHB*, *AMBA_APB*, *CoreConnect_OPB*, *CoreConnect_PLB* or *OCP*.
- the `RtlConnectorKind` enumeration indicates the kind of hardware connectors used to interconnect two or more hardware or software components. For instance, a connector may *Point2Point*, *AMBA_AHB*, *AMBA_APB*, *CoreConnect_OPB*, *CoreConnect_PLB* or *OCP*.

As depicted in figure 7.24, the `Storage` subpackage defines all concepts related of the storage of operation parameters in memory elements like registers within hardware component ports and contains the following information:

- the abstract `Memory` class represents a memory element like a register or FIFO. It is the base class of all memory element classes. It contains an integer attribute called *addr* representing the relative address of the memory element on a bus e.g. 0x4 and an integer attribute called `width` representing the data width of the memory element e.g. 32 bits.
- the `Register` class represents a register and inherits from the `Memory` class.
- the `FIFO` class represents a First-In First Out (FIFO) memory element and inherits from the `Memory` class. Its attributes include the *depth* of the FIFO and various boolean flags such as

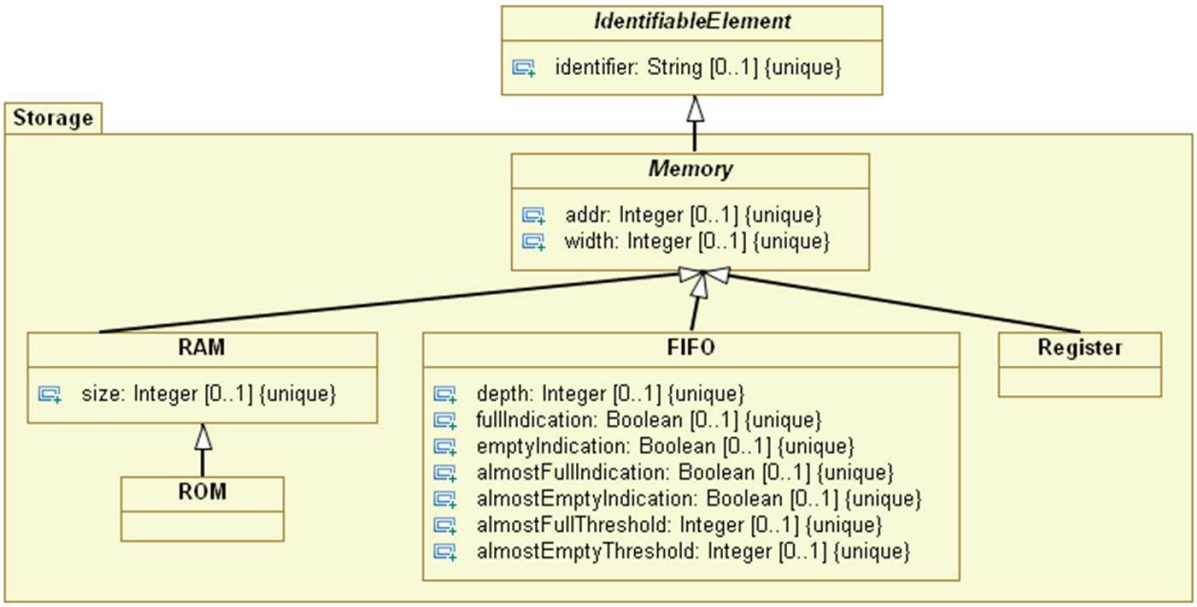


Figure 7.24: Storage UML package

fullIndication, *emptyIndication*, *almostFullIndication*, *almostEmptyIndication* and the corresponding integer threshold called *almostFullThreshold* and *almostEmptyThreshold*.

- the `RAM` class represents a Random Access Memory (RAM) and inherits from the `Memory` class. It contains an integer attribute called *size* representing the size of the RAM in octets.
- the `ROM` class represents a Read-Only Memory (ROM) and inherits from the `RAM` class.

The `Port` subpackage defines the concepts related to the hierarchical ports of software, system and hardware components and contains the following information:

- the `DirectionKind` enumeration indicates the direction of a component port such as `in`, `out` or `inout`.
- the abstract `Port` class represents a component port. It is the base class of all component port classes. The component port classes are defined in dedicated packages and represent component ports at different abstraction levels. The `Port` class contains a *direction* attribute of `DirectionKind` type to indicate the direction of the component port.
- the `LogicalPort` subpackage defines the kind of logical ports that a software, hardware or system component may have at logical or functional level.
- the `MessagePort` subpackage defines the kind of message ports that a software, hardware or system component may have at message or transactional level.
- the `RtlPort` subpackage defines the kind of RTL ports that a hardware or system component may have at RTL or signal level.

As depicted in figure 7.25, the `MessagePort` subsubpackage defines the following kind of message component ports:

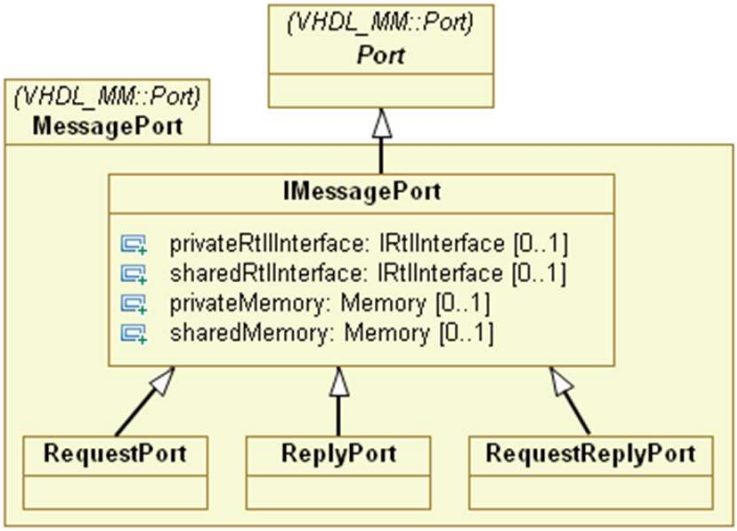


Figure 7.25: MessagePort UML package

- the abstract `MessagePort` class represents a message component port and is the base class of following message component port classes. The messages may be middleware messages such as GIOP or MHAL messages, transactional messages such as OSCI TLM, OCP TLM, PV/AV messages, or on-the-wire messages with memory-mapped read/write messages Its *rtlInterface* attribute of type `Interface::RtlInterface` may contain a list of RTL ports representing a signal-based hardware interface. The abstract `Interface::RtlInterface` class belongs to the `Interface` package and is described in the following.

- the `RequestPort` class represents a message component port through with request messages may be sent or received. Its inherited *rtlInterface* may be a push-only or pull-only RTL interface i.e. write-only or read-only RTL interface.

- the `ReplyPort` class represents a message component port through with reply messages may be sent or received. The inherited *rtlInterface* has the same characteristics than the RequestPort attribute.

- the `RequestReplyPort` class represents a message component port through with request and reply messages may be sent or received. Its inherited *rtlInterface* is a push/pull RTL interface i.e. write/read RTL interface.

As depicted in figure 7.26, the `LogicalPort` subsubpackage defines the following kind of logical component ports:

- the abstract `LogicalPort` class represents a logical component port and is the base class of all logical component port classes. This class contains three logical port attributes which may have an instance each i.e. with [0..1] multiplicity. The *requestPort* attribute of type `MessagePort ::RequestPort` allows a logical port to send or receive request messages. The *replyPort* attribute of type `MessagePort ::ReplyPort` allows a logical port to receive or send reply messages. The *inoutRequestReplyPort* attribute of type `MessagePort::RequestReplyPort` allows a logical port to transport both request and reply messages. A logical port may have either a request port and a reply port, or an inout request-reply port. Such constraint may be expressed

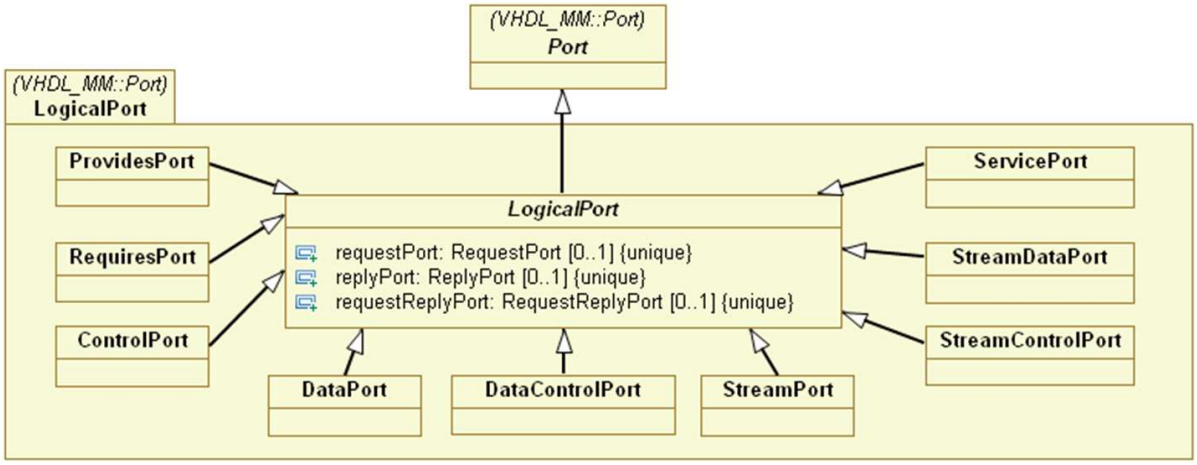


Figure 7.26: LogicalPort UML package

using the OMG Object Constraint Language (OCL) [OMG06d]. The request and reply ports may be mapped to two dedicated set of RTL ports, while the request-reply port may be mapped to a single shared set of RTL ports.

- the `ProvidesPort` class represents a logical component port providing an operation-based software interface. This kind of ports may represent the provides port of SCA components, the facet port of CCM components, the TLM port of SystemC components at transaction level, the `sc_export` port of SystemC components and/or the logical port of object-oriented hardware components. The inherited input *requestPort* allows a provides port to receive request messages. The inherited output *replyPort* allows a provides port to send reply messages. The inherited *inoutRequestReplyPort* allows a provides port to receive request messages and send reply messages.

- the `RequiresPort` class represents a logical component port requiring an operation-based software interface. This kind of port may represent the uses port of SCA components and the receptacle port of CCM components, the TLM port of SystemC components at transaction level, the `sc_port` port of SystemC components and/or the logical port of object-oriented hardware components. The output *requestPort* allows a requires port to send request messages. The input *replyPort* allows a requires port to receive reply messages. The *inoutRequestReplyPort* allows a requires port to send request messages and to receive reply messages.

- the other logical port classes include the kind of ports defined in the UML profile for Software Radio [OMG07e] such `ControlPort`, `DataControlPort`, `DataPort`, `ServicePort`, `StreamControlPort`, `StreamPort` and `StreamDataPort`, and in the UML profile UML profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [OMG08h] such as `MessagePort` and `FlowPort`. The configuration of the RTL mapping for these logical ports is the same as the `ProvidesPort` and `RequiresPort` thanks to the inheritance of the generic attributes of the `LogicalPort` class. Other kinds of logical port classes may be defined by extending the `LogicalPort` class.

As depicted in figure 7.27, the `RtlPort` subsubpackage defines the following kind of RTL component ports:

- the `RtlPort` class represents a component port at Register Transfer Level (RTL) and is the base class of all RTL component port classes. It contains an integer attribute called *width* which indi-

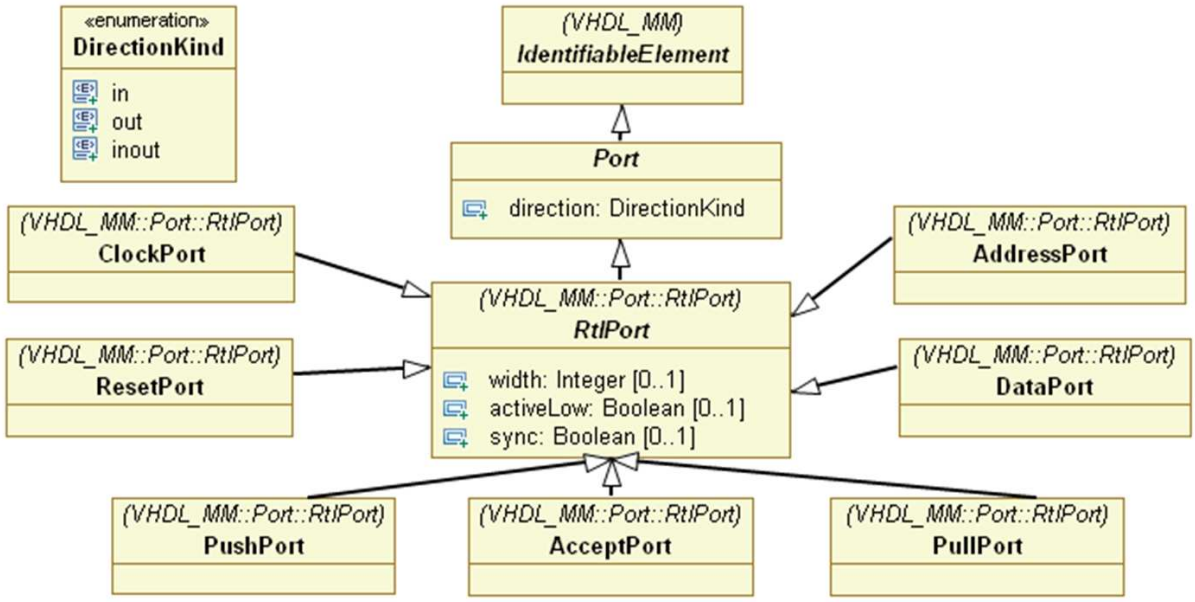


Figure 7.27: RtlPort UML package

cates the width of the RTL port whose value is -1 for a wire e.g. for VHDL `std_logic` and the data width for a signal bus e.g. for VHDL `std_logic_vector`, a boolean attribute called *activeLow* that indicates at which electrical level (0 or 1) the signal is considered as active with false as default value, and a boolean attribute called *sync* which indicates whether the signal is synchronous with a `ClkPort` with true as default value.

- the others RTL component port classes explicitly define a generic and extensible family of RTL ports which are typically used by hardware designers. These port classes include `ClockPort`, `EnablePort`, `ResetPort`, `PushPort`, `PullPort`, `AddrPort`, `DataPort`, `AccPort`, `ReqPort`, `AckPort`, `StatusPort`, `BusyPort`, `ClearPort`, `LoadPort`, `FullPort`, `EmptyPort`, `AlmostFullPort` and `AlmostEmptyPort`. The `AlmostFullPort` and `AlmostEmptyPort` have an integer `threshold` which indicates from which number of tokens a full or empty flow control signal is activated. Other kinds of logical port classes may be defined by extending the `RtlPort` class.

As depicted in figure 7.28, the `RtlInterface` subpackage defines a family of named and typed RTL interfaces:

- the abstract `RtlInterface` class represents a signal-based hardware interface at Register Transfer Level (RTL). It inherits from the `IdentifiableElement` class and the `RtlMappingConfiguration` class and is the base class of all RTL interface classes. We propose two means to configure the mapping of a logical component port to the RTL interface(s) of message component port(s). The configuration of the RTL interface mapping may be either defined explicitly by adding RTL port instances in the `IHwInterface` class or by choosing a predefined interface from the family of interfaces.

- the `IHwInterface` class may contain an instance of each kind of ports defined in the `RTLPort` package [0..1] cardinality and an optional generic list of RTL ports to add RTL ports not defined in the `RTLPort` package. This inbterface is depicted in figure 7.29.

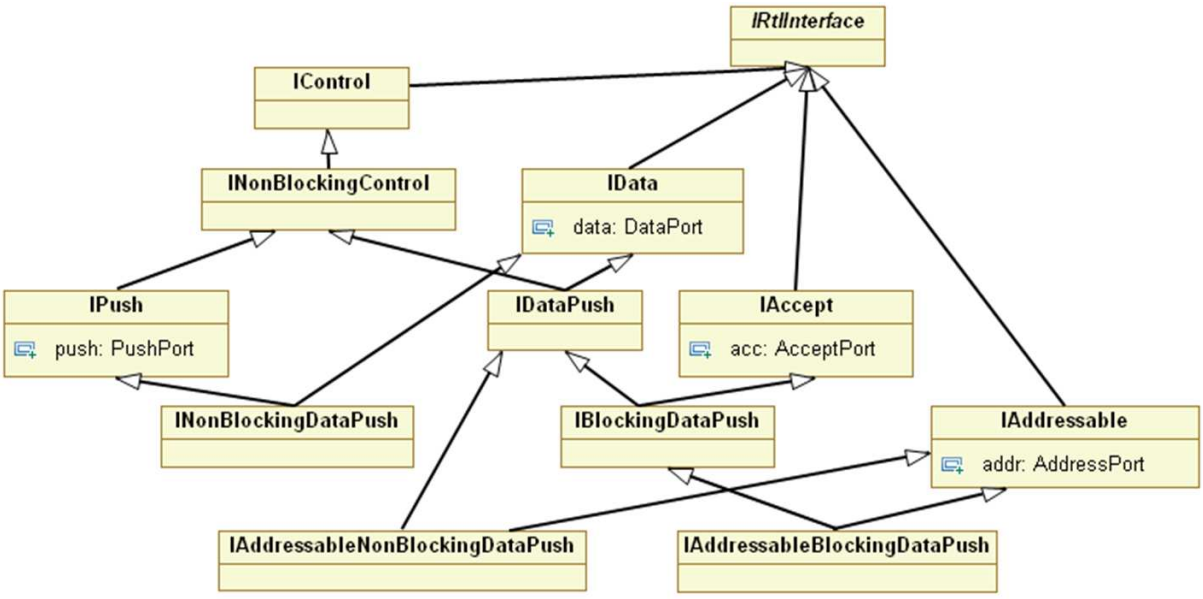


Figure 7.28: InterfaceFamily UML package

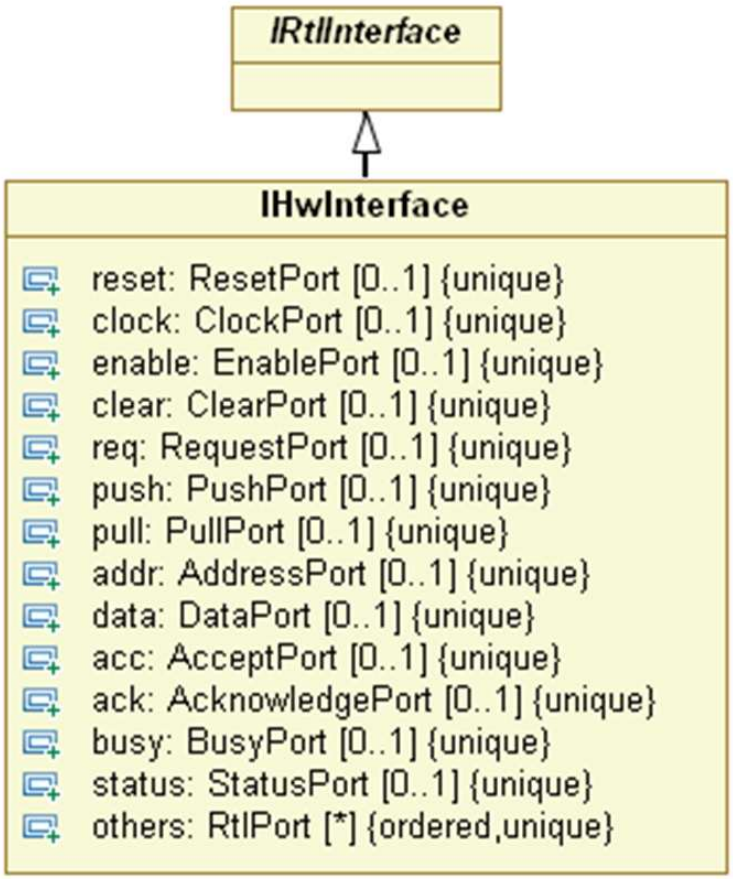


Figure 7.29: HwInterface UML class

- the other classes are the object-oriented modeling of the named hardware interfaces previously proposed in [GBJV08]. Inheritance is used to factor out RTL ports from RTL interfaces. Each hardware interface adds a RTL port and its associated semantics. For instance, the `IData` class contains a *data* attribute of type `DataPort`. Then, the `IPush` class inherits from the abstract `INonBlockingControl` interface and contains the *push* attribute of type `PushPort`. Finally, the `IPushData` class inherits from the `IPush` and `IData` interfaces.

The `Message` subpackage defines all concepts related to messages.

- the `MessageKind` enumeration defines the kind of messages which can be transported through message ports. It includes messages such as *CORBA_GIOP*, *SCA_MHAL*, `ReadMessage` and `WriteMessage` for TLM and RTL messages, and `DataMessage` for raw data message.

- the `Message` class contains an integer `version` attribute to indicate the version of the message protocol and a `message` attribute of type `MessageKind` to indicate the message protocol.

The `RtlMappingConfiguration` class is the base class of RTL mapping configuration classes called `RtlInterfaceMappingConfiguration`, `RtlParameterMappingConfiguration` and `RtlConnector-MappingConfiguration`. The `RtlMappingConfiguration` class contains an optional instance of the `RTLInterface` class called *privateInterface* which is contained by value and another optional instance called *sharedInterface* which is contained by reference. This distinction allows designers to define either a dedicated RTL interface defined at local scope or a shared RTL interface defined at global scope.

The `RtlInterfaceMappingConfiguration` class allows designers to configure the mapping of logical ports to message ports containing RTL ports. It contains the following attributes. The *requestPort* attribute of type `RequestPort` may contain a RTL interface through with requests messages are sent or received. The *replyPort* attribute of type `ReplyPort` may contain a RTL interface through with reply messages are sent or received. The *sharedConnectorInterface* attributes of type `RtlConnectorInterface` may contain a hardware connector which is shared by several hardware component ports like an on-chip or off-chip bus. The *privateConnectorInterface* attributes of type `RtlConnectorInterface` may contain a hardware connector which is dedicated to a single hardware component port like a point-to-point connection. The *message* attribute of type `MessagePort` may define the kind of messages transferred through the request and reply ports.

The `RtlParameterMappingConfiguration` class allows designers to configure the mapping of operation parameters to shared or dedicated memory element(s) such as register(s), FIFO(s) or RAM(s). The *sharedMemory* attribute of type `Memory` may indicate that a memory element is shared by one or several parameters in the same operation or in different operations. These operations may be defined in the same software interface or in different software interfaces. In other words, the designers can allocate a memory element at the parameter scope, operation scope, interface scope and component scope. The *privateMemory* attribute of type `Memory` may indicate that a memory element is dedicated to a single operation parameter.

The `RtlConnectorInterface` class describes the RTL interface of hardware connectors such as on-chip and off-chip bus and point-to-point connection. The *connectorKind* attribute of type `RtlConnectorKind` may indicate the kind of the connector such as AMBA AHB, CoreConnect OPB or an OCP interface. The optional *IPXACT_SPIRIT_filename* attribute of type string may indicate the absolute path to the IPXACT_SPIRIT description of the connector interface.

Finally, the abstract `IdentifiableElement` class contains a string identifier which is used to identify instances of numerous classes such as RTL mapping configuration classes, `Memory` classes, `Port` classes, `RtlInterface` classes and `Message` classes.

Our proposition can be viewed as an extension and generalization of the OCP-based hardware SCA component model proposed in [JTR05] to generic software, hardware and system components using the model-based approach defined by the OMG called MDA.

A UML profile for RTL mapping configuration may be easily derived from this meta-model by extending the UML profile for CORBA and CCM [OMG08g]. The basic principle to build a UML profile from a domain-specific meta-model consists in creating a UML stereotype for each concept of the meta-model like shown in [OMG08g]. Moreover, each UML stereotype must extend the correspond meta-class in the UML meta-model or in a UML profile. For instance, the Port class from our meta-model may be defined as a UML stereotype in a UML profile for RTL mapping configuration that may extends the Port meta-class of the UML meta-model, the Port class from the UML profile for CORBA and CCM or from the UML profile for Software Radio [OMG07e]. In this way, a software component defined in UML may have LogicalPorts, MessagePorts and RTLPorts and designer may configure the mapping of these ports using their attributes which appear as UML stereotype tags.

RTL Mapping Configuration Editor

We used Eclipse to automatically generate an editor compliant with the extended CCM meta-model. As depicted in figure 7.30, we configured the RTL mapping to correspond the configuration presented for the Filter example 7.16. The XMI file generated by the editor is similar to the XML file proposed in [GBJV08] as shown in listing 7.8.

```
<?xml version="1.0" encoding="ASCII"?>
<ccm:CcmModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:
    xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ccm="http://www.
    thalesgroup.com/ccm/1.0" ... identifier="Filter" absoluteName="Filter"
    modelName="Filter">
  <contents xsi:type="ccm.baseidl:ModuleDef" identifier="Example"
      absoluteName="::Example">
4   ...
      <contents xsi:type="ccm.baseidl:OperationDef" identifier="setCoeff"
          absoluteName="::Example::Config::setCoeff" fileName="Filter.idl3"
          lineNumber="11" idlType="//@types.2">
        <parameters idlType="//@contents.0/@contents.0/@contents.4"
            identifier="cf">
          <rtlMappingConfiguration>
            <rtlInterface xsi:type="ccm.RtlMappingConfiguration.Interface:
                IAddressableNonBlockingPull">
9             <addr identifier="cf_addr_o" direction="out" width="7"/>
              <data identifier="cf_data_i" width="16"/>
              <pull identifier="cf_pull_o" direction="out" width="-1"/>
            </rtlInterface>
            <memory xsi:type="ccm.HwMappingConfiguration.Storage:RAM"
                identifier="cf_dpram" width="16"/>
14        </rtlMappingConfiguration>
        </parameters>
...
```

Listing 7.8: RTL Mapping Configuration XML file

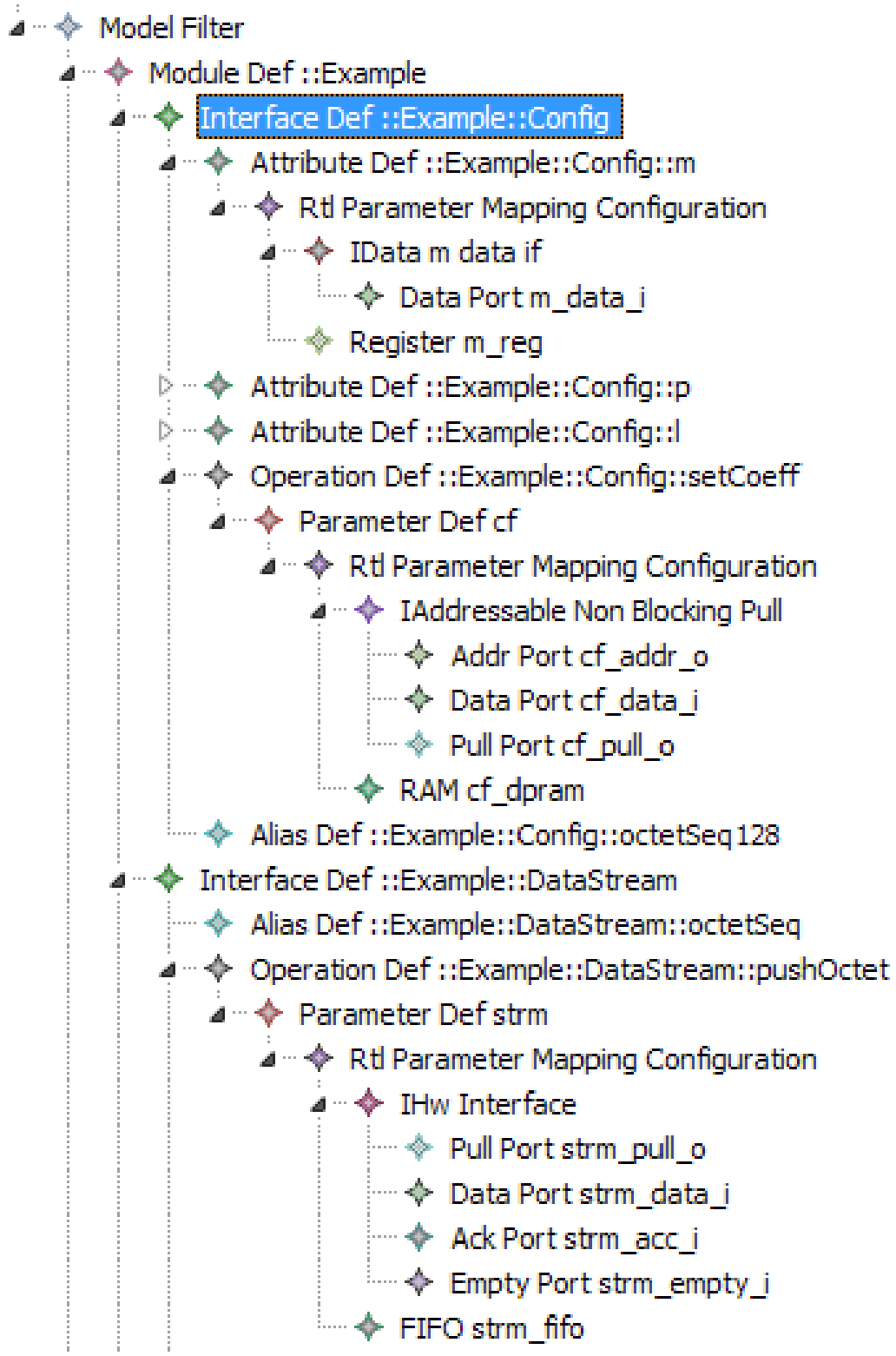


Figure 7.30: RTL Mapping Configuration Editor

CCM meta-classes	VHDL meta-classes
`ModuleDef`	`PackageDef`
`InterfaceDef`	`EntityDef`
`ParameterDef`	`PortDef`
`PrimitiveDef`	`Std_logic, Std_logic_vector`
`ConstantDef`	`ConstantDef`
`StructDef`	`RecordDef`
`EnumDef`	`EnumDef`
`FieldDef`	`FieldDef`
`IDLType`	`VHDLType`
`ParameterMode`	`PortMode`

Table 7.3: Mapping between CCM meta-classes and VHDL meta-classes

VHDL meta-model

Instead of defining a VHDL meta-model from scratch as in the literature, we decided to leverage the experience used to build the CCM meta-model and to keep a similar architecture and naming conventions to ease the definition of model transformations between the extended CCM meta-model and the VHDL meta-model. To specify the VHDL meta-model, we started from the standard UML model of the CCM meta-model which is available as an XMI file on the OMG website. After some minor modifications, the UML model has been modified in the free and open source Papyrus UML modeler, which is available as an Eclipse plugin, to build the VHDL meta-model in UML. Table 7.3 illustrates the mapping between CCM meta-classes and VHDL meta-classes. From the UML model of the VHDL data model depicted in figure 7.31, we generated the corresponding Ecore meta-model to automatically produce the java code of the VHDL meta-model. This code is further used by the model-to-model and model-to-text transformation tools described below.

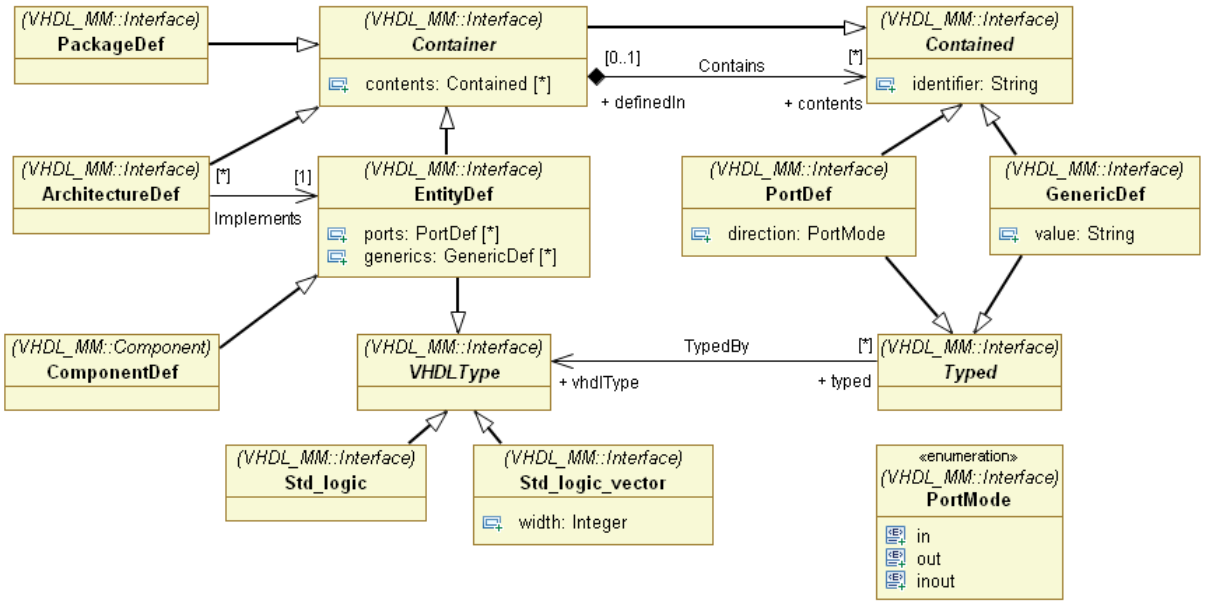


Figure 7.31: VHDL meta-model in UML

Model-to-Model Transformation

We have written Model-to-Model Transformations from the extended CCM meta-model to the VHDL meta-model using a free and open-source implementation of the OMG MOF Query/View/Transform (QVT) model-to-model transformation language [OMG08e] called SmartQVT[70]. SmartQVT is developed by France Telecom and is available as an Eclipse plugin. Listing 7.9 illustrate the mapping between a CCM component and a VHDL entity in QVT.

```
mapping CCM_MM::ComponentIdl::ComponentDef::component2entity() : VHDL_MM::
    Interface::EntityDef {
  identifier := self.identifier;
  absoluteName := self.absoluteName.substringAfter("::").replace("::","_");
  fileName := self.fileName.replace(".idl3",".vhd");
  var clkPort := object VHDL_MM::Interface::PortDef {
  identifier := "clk";
    direction := VHDL_MM::Interface::PortMode::in_;
  vhdlType := vhdlModel.objects()[VHDL_MM::Interface::Std_logic]->
     asOrderedSet()->first();
  comment := "Synchronous clock";
  };
  contents := clkPort;
  var rstPort := object VHDL_MM::Interface::PortDef {
  identifier := "rst_n";
    direction := VHDL_MM::Interface::PortMode::in_;
  vhdlType := vhdlModel.objects()[VHDL_MM::Interface::Std_logic]->
     asOrderedSet()->first();
  comment := "Active low asynchronous reset";
  };
  contents += rstPort;
  contents += self.facet->map facet2ports();
  contents += self.receptacle->map receptacle2ports();
  ...
}
```

Listing 7.9: Mapping between CCM component and VHDL entity in QVT

Model-to-Text Transformation

We have written Model-to-Text Transformations from the VHDL meta-model to VHDL code source using a free and open-source implementation of the OMG Model-to-Text transformation language (MTL) called Acceleo [71]. Acceleo is developed by Obeo and is available as an Eclipse plugin within the Eclipse M2T project. Listing 7.10 illustrates the generation of VHDL code from a VHDL model according to a template written in MTL. Listing 7.11 presents a excerpt of the VHDL code generated by the model-to-text transformation.

```
[template public importLibrary()]
library IEEE;
  use IEEE.std_logic_1164.all;
[/template]

[template public generateEntityDef(entityDef: EntityDef)]
```

[70] http://smartqvt.elibel.tm.fr

[71] http://www.eclipse.org/modeling/m2t/?project=acceleo

```
entity [entityDef.identifier/] is
  port ([for (portDef : PortDef | entityDef.contents) separator(';')]

    [if (not portDef.comment.oclIsUndefined())]
  [portDef.generateComment()/]
    [/if]
    [portDef.generatePortDef()/][/for]
 );
end [entityDef.identifier/];
[/template]

[template public generatePortDef(portDef: PortDef)]
[portDef.identifier/]: [portDef.direction/][if portDef.vhdlType.oclIsTypeOf
   (Std_logic)] std_logic[else] std_logic_vector([if portDef.vhdlType.
   oclAsType(Std_logic_vector).width=0][portDef.vhdlType.oclAsType(
   Std_logic_vector).width/][else][portDef.vhdlType.oclAsType(
   Std_logic_vector).width-1/][/if] downto 0)[/if]
[/template]

[template public generateComment(portDef: PortDef)]
-- [portDef.comment/]
[/template]

[template public generateEntity(vhdlModel: VhdlModel)]
[comment @main /]
[for (packageDef : PackageDef | vhdlModel.contents)]
  [for (entityDef : EntityDef | packageDef.contents)]
  [file (entityDef.identifier.concat('.vhd'), false)]
[fileHeader()/]

[importLibrary()/]

[entityDef.generateEntityDef()/]

  [/file]
  [/for]
[/for]
[/template]
```

Listing 7.10: Generation of VHDL code from a VHDL model in MTL

```
library IEEE;
  use IEEE.std_logic_1164.all;

entity Filter is
  port (
    -- Synchronous clock
    clk: in std_logic;
    -- Active low asynchronous reset
    rst_n: in std_logic;
    configPort_push_i: in std_logic;
    configPort_data_i: in std_logic_vector(7 downto 0);
    configPort_addr_i: in std_logic_vector(2 downto 0);
    configPort_acc_o: out std_logic;
```

```
    configPort_push_o: out std_logic;
    configPort_data_o: out std_logic_vector(7 downto 0);
    configPort_acc_i: in std_logic;
    inputPort_push_i: in std_logic;
    inputPort_data_i: in std_logic_vector(7 downto 0);
    inputPort_addr_i: in std_logic_vector(0 downto 0);
    inputPort_acc_o: out std_logic;
    inputPort_push_o: out std_logic;
    inputPort_data_o: out std_logic_vector(7 downto 0);
    inputPort_acc_i: in std_logic;
    outputPort_push_o: out std_logic;
    outputPort_data_o: out std_logic_vector(7 downto 0);
    outputPort_addr_o: out std_logic_vector(0 downto 0);
    outputPort_acc_i: in std_logic;
    outputPort_push_i: in std_logic;
    outputPort_data_i: in std_logic_vector(7 downto 0);
    outputPort_acc_o: out std_logic);
end Filter;
```

Listing 7.11: Example of generated code for the Filter

7.4 Hardware Application of the Software Communications Architecture

As far we know, the only hardware interface that has been proposed for the JTRS SCA `Resource` interface has been presented in [JTR05] based on OCP. This mapping proposition uses the specific capabilities of the OCP model such as Threads. It assumes that control and configuration operations can be invokes concurrently. Moreover, the resulting mapping is totally customized to the semantics of each `Resource` component. To guarantee a consistent behavior and simplify the hardware mapping, we consider that the invocations on the `Resource` interface are serialized. Indeed, the Core Framework should sequentially `initialize`, `configure`, `start`, `stop`, `configure`... components. The behavior of this interface protocol could be described as an UML protocol state machine, which only permits a predefined sequence of invocations to proceed.

7.4.1 Hardware SCA Resources, Device and Services

The main abstractions of the SCA can be divided into application abstraction with the Resources and platform abstractions with the Devices and Services including the radio devices and radio services.

The hardware implementation of a SCA `Resource` component represents a hardware application component. Examples of hardware application components include DSP IP cores such as convolutional en/decoders and Viterbi en/decoders. A hardware `Resource` component may either contain a new user-defined IP core developed from scratch, a reused user-defined IP core or a third-party COTS IP core. The hardware interfaces of the new IP cores directly result from the mapping of software UML or IDL interfaces (e.g. the SCA Resource interface and the user-defined interfaces) to HDLs, while the reused and COTS IP cores are encapsulated within a wrapper providing the resulting hardware interfaces. This top-down design flow is similar to the design of software objects and component with CORBA in which the software developers performs its implementation against the mapped software interfaces.

The hardware implementation of a SCA Device component represents a hardware device component. Examples of hardware device components include I/O IP cores such as Ethernet MAC, AC97 chipset,

RapidIO, I2C. In the same way as software application components access hardware platform components via logical devices with standard interfaces to increase application portability, hardware application components should access these hardware platform components via standard hardware interfaces. For instance, in the same way as logical Ethernet Devices have a standard software interfaces, all Ethernet MAC IP core should have the same hardware interfaces. However, these hardware interfaces must have stronger semantics than the existing hardware interfaces which are proprietary bus protocols such as CoreConnect for Xilinx IP cores and Avalon for Altera IP cores.

The interfaces of a SCA `Resource` component may be implemented in software, in hardware or in both. Beyond the custom proxies described in [JTR05], the standardization of a configurable mapping between software and hardware interfaces for component portability and of lightweight ESIOP(s) for component interoperability should shift the generation of proxies from end-users to middlewares.

7.4.2 SCA Operating Environment for FPGA

The Software Communications Architecture defines an Operating Environment (OE) which consists of a Core Framework (CF), a CORBA middleware, Name and Event middlewares services, a POSIX-based Operating System (OS), its network stack and Board Support Packages (BSP).

An OE needs to be standardized for non-CORBA capable processing elements such as DSP, FPGAs and ASICs. Extensions of the SCA have successively proposed with the Specialized Hardware Supplement (SHS) [JTR04] in 2004, the extension for component portability for Specialized Hardware Processors (SHP) [JTR05] in 2005 and the Modem Hardware Abstraction Layer (MHAL) [JTR07] API in 2007. Furthermore, an OE has been proposed for DSP in the scope of the IST End-to-End Reconfigurability (E2R) [72] european project for the OMG SWRadio specification which is inspired from the SCA [LN07]. OEs for GPP, DSP and FPGAs are also investigated in the European Defense Agency (EDA) European Secure Software Radio (ESSOR) project [73].

In this section, we propose an Operating Environment for the SCA in FPGA.

The functionalities required for an OE on FPGA are the following:

1. To guaranty the portability of hardware SCA components from one FPGA platform to another provided by different providers. This portability requires standard hardware interfaces on which hardware components may depend without any other dependencies to the FPGA platform. This functionality is provided by the Application Programming Interfaces of POSIX, CORBA and the CF.

2. To guaranty the interoperability of hardware SCA components with software SCA components distributed across the SDR platform. This functionality is provided by CORBA with message protocols.

3. To support the deployment of hardware SCA components in FPGA. This deployment correspond to the loading and unloading of total or partial bitstream in the FPGA. The partial and dynamic reconfiguration of FPGAs enables the application of the dynamic deployment approach of the SCA downto FPGA. This functionality is provided by the interfaces CF

A naive application of the SCA OE in hardware would to propose an substitute for POSIX, CORBA and CF. An Operating Environment for FPGA must only implement a subset of the required functionalities of the SCA OE which is specified for General Purpose Processor. Proposing an OE for FPGA

[72] http://www.ist-world.org
[73] http://www.eda.europa.eu

basically means proposing a hardware signal-based interface and implementation for the services provided by POSIX, CORBA and CF. However, hardware resources in FPGA are too expensive and scarce to allow and want the implementation of all these services in hardware. Only a hardware interface and implementation for the required services is needed and must be proposed. We assume that if a mapping between a software operation-based interface and a hardware signal-based interface is possible, then the mapping of POSIX, CORBA and CF interfaces is a particular case. We focus on the selection of the services which are meaningful and useful in hardware.

POSIX-like Services

In the following, we present the main services standardized by the POSIX specification and discuss their utility for hardware components.

1. Process and Thread management for the creation, control, termination, scheduling of processes and threads. Dynamic and partial reconfiguration of FPGA allows the dynamic configuration of FPGA logic blocks and routing resources with a partial bitstream. This configuration simulates the creation and termination of hardware process with a hardware module (VHDL entity). These processes communicate through signals acting as shared variables. For Xilinx FPGAs, the hardware interface for the creation and termination of hardware threads corresponds to the interface of the Internal Configuration Access Port (ICAP). A vendor-independent standard bitstream interface must be proposed and the proprietary interface like ICAP would be wrapped by this standard interface.

2. Inter-Process Communication (IPC): signal, pipes, message passing, shared memory. VHDL processes and modules primarily communicate via signals representing shared variables. Based on this signal-based interactions, message passing communications may rely on FIFOs and shared memory communications may rely on local or system RAM.

3. Time services: clocks and timers. In synchronous digital design, hardware modules are synchronized by a common clock signal within a synchronous domain. Using the clock signal, time delays may be obtained with counters or timers.

4. File management. Due to scarce hardware resources, a file management service is typically not implemented in hardware, but in software on a GPP dedicated to control, management and deployment tasks. File management is required to deploy an application using its deployment plan and associated (XML) descriptors. In hardware, it is needed to load a total or partial bitstream down to an FPGA.

5. C library services. The C library provides services such as timers, I/O, memory allocation. The VHDL library provides basic services such as data types and their conversion. The VHDL library do not offer services such as storage services with standard FIFOs, RAM and registers. Users must define their own storage elements based on the available hardware resources.

6. Synchronous and asynchronous I/O management. Hardware modules have typically synchronous I/O signals. In Globally Asynchronous Locally Synchronous (GALS) system, asynchronous communications take place between synchronous domain. From a hardware interface protocol viewpoint, the synchronous i.e. blocking communications are obtained with two-phase and four-phase handshaking protocol, while asynchronous communications are obtained with enable-based protocol.

7. Coordination/synchronization services: semaphores, condition variables, barriers. Intra-module and inter-module coordination is performed using explicit multiplexer according to the one-writer / multiple reader protocol.

Obviously due to the different nature of the hardware and software execution, all these services are not required in FPGA. No such standardized services exist for hardware implementations. However, hardware implementation of POSIX-like OS services have been proposed in the literature with include Hybrid Thread (HThread) [APS+07] for reconfigurable computing from Kansas University and MultiFlex [PPL+06b] for networking and multimedia from ST.

CORBA-like services

Basically, the SCA relies on the CORBA middleware to provide location transparency via opaque reference, a standard middleware API, a remote method invocation (RMI) service, and lightweight Naming, Event and Log Services. Location transparency may be provided to hardware components by applying the Proxy design pattern [GHJV95] in hardware. The hardware proxies provide the same signal-based interface as the business component for which they act as proxies. The client proxy stores the physical address of the server component port e.g. in a register which may be modified if the server component is migrated to another location. These proxies are responsible for the packing and unpacking of data transferred on the signal based interface to and from request and reply messages to implement a RMI protocol. If a systematic UML/IDL-to-HDL/SystemC mapping is defined, any software interface from CORBA services may be refined downto hardware if meaningful. In particular, this is true for the lightweight Naming, Event and Log Services. The application of CORBA middleware services downto hardware will be described in the next section.

Application of the Domain Profile

Due to the overhead of XML parsing, this functionality of the SCA CF deployment service will certainly remain in software on a control general-purpose processor. Nevertheless, the XML descriptors of the domain profile needs to be adapted to the description of hardware SCA components.

Services provided by the OE for FPGA

The services of an OE for FPGA are listed in the following. These services are conceptually provided by a hardware container to the hardware business components i.e. IP-cores. The portability of hardware modules relies on the standardized hardware interfaces of these services. An abstract interface in CORBA IDL may be mapped to each of these services.

- the **functional configuration interface** allows the deployment framework to configure the parameters of business components. The abstract interface in UML/CORBA IDL may contain a read-only, write-only or read/write attributes, explicit getter/setter methods such as `setParam()`, or a generic `configure` method like in the SCA . The hardware interface for a write-only attribute or setter method typically consists of an input control signal like `enable`, `write` or `push`, an optional address signal when the number and size of parameters is relatively big, and an input data signal per parameter or for all parameters respectively for a small or big number of parameters. An acknowledge signal is generally not required. Conversely, the hardware interface for a read-only attribute or getter method typically consists of an input control signal like `read` or `pull`, an optional address signal when the number and size of parameters is relatively big, and a output data signal per parameter or for all parameters respectively for a small or big number of parameters.

- the **hardware reconfiguration interface** allows the loading of one or multiple hardware waveform components using a dynamic and partial bitstream referenced by the SCA domain profile and the unloading of hardware components by loading blank bitstream. This interface may be implemented by a software FPGA Device using software drivers such as the one of the ICAP controller or a hardware FPGA Device wrapping the FPGA-specific reconfiguration interface such as the ICAP write-only interface. A single FPGA Device instance per FPGA chip may be required for all reconfigurable areas as the dynamic partial bitstream from Xilinx transparently includes the addressing information which indicate where the bitstream must be loaded in the FPGA and which reconfiguration area must be used. Hence, an FPGA Device per reconfigurable area may not be needed. A software or hardware `ResourceFactory` may also use the physical reconfiguration service interface to create and release a hardware waveform component. The hardware reconfiguration service interface should be independent from any FPGA vendor even if currently only Xilinx industrially supports dynamic partial reconfiguration of FPGAs in its PlanAhead placement tool although hardware reconfiguration is an inherent feature of FPGAs.

- the **power management interface** may rely on low-power techniques to reduce the power consumption of hardware modules. The power management interface may be based on a power control interface and protocol like in AXI [ARM04] and in OCP3.0, and may use clock gating for FPGAs to reduce dynamic power consumption [ZRM06]. The objective is to reduce the frequency or switch off the clock signal of idle hardware modules.

- The **data stream interface** is used to transfer a stream of data between hardware components such as signal processing blocks. This interface may support burst transfers like OCP [OI06], packet-oriented transfers and allow data flow control like the Avalon Streaming interface [Alt07a]. Such hardware interface corresponds the `pushPacket` operation of SCA building blocks or the `pushBBSamples` of the Transceiver Facility [NP08].

- The **control interface** allows the start or stop the processing of a hardware components. This interface usually consists of an `enable` signal. Such an interface correspond to the `start` and `stop` operation of the SCA Resource interface.

- the **activation service** or clock service provides a periodic clock used to synchronize a set of hardware modules. It may be provided by a DCM (Device Clock Module) on Xilinx FPGAs. Advances implementation may support dynamic frequency scaling to reduce the power consumption of hardware components. This service is typically incarnated by a clock signal in the interface of hardware modules.

- the **initialization service** or reset service allows the initialization of hardware modules. This service is typically incarnated by a reset signal in the interface of hardware modules.

- the **storage service** is used by the hardware module to read and write user-data to standardized storage elements such as single-port and dual-port RAMs, FIFOs with flow control and registers. The storage elements store the input and output data of the business components such as RMI messages and raw data.

7.4.3 Hardware SCA Resource Interface

We propose the following hardware signal-based interface for the SCA `Resource` interface. We choose an addressable blocking data push interface for requests and a non-blocking data push interface for replies.

31 ... 26	27 ... 4	5 ... 0
OP_ID	PARAM_ID	EXC_ID
PARAM_VALUE		

Table 7.4: Memory-Mapped Hardware/Software SCA `Resource` Interface

```
library ieee;
use ieee.std_logic_1164.all;
library CORBA;
use CORBA.CORBA_types.all;

entity Resource is
generic (
EX_DW : natural := 6;
OP_DW : natural := 4;
DATA_DW     : natural := 8;
);
port (
    -- common signals
clk : in  std_logic;
    rst_n : in  std_logic;
    --  shared Resource interface
    -- request
    push_i : in  std_logic;
    data_i : in  std_logic_vector(DATA_DW-1 downto 0); -- Operation Id
    addr_i : in  std_logic_vector(OP_DW-1 downto 0); -- Operation Id
    ack_o : out    std_logic;
    -- reply
    push_o     : out  std_logic;
    data_o     : out  std_logic_vector(DATA_DW-1 downto 0);
    exc : out  CORBA_exception;
    exc_id : out  std_logic_vector(EX_DW-1 downto 0)
);
end Resource;
```

7.4.4 Memory-Mapped Hardware/Software Resource Interface

Depending on the chosen trade-off between performance and area, several alternatives exist to define a memory-mapped Hardware `Resource` Interface. For instance, one register may be allocated per operation parameter or one register bank per operation [Frö06]. However, we chose to constrain the hardware/software interface to only permit one invocation at a time and guarantee the sequentiality of operation calls to avoid race conditions. We first analyzed the requirements of the `Resource` Interface. The number of operations in the `Resource` interface is 10 therefore 4 bits are basically required to indicate the operation to be invoked. In the same way, the number of possible exceptions that may be raised by the `Resource` operations is 56 hence 6 bits are basically required to indicate an error code. If we consider a typical data bus width of 32 bits, 21 bits are available to indicate an operation parameter.

Table 7.4 presents the proposed memory-mapped register bank to allow the software `Resource`

stubs to invoke the Resource operations of hardware Resource or Device components. The OP_ID field is a binary encoded numerical identifier that indicates the hardware operation to be invoked. The PARAM_ID field is a binary encoded numerical identifier that indicates an operation parameter. The PARAM_VALUE field contains the value of an operation parameter. This hardware/software interface is generic and requires few hardware resources. Only two writes and reads are required to invoke an operation without parameter such as initialize, start and stop.

For instance for the initialize operation, an invocation request from a software component to a hardware component may consist for the software stub in writing "1" in the OP_ID register to indicate the initialize operation and in clearing the EXC_ID register with "1" i.e. by writing "0x3F". For the invocation reply, the software stub may either be proactive by polling the value of the EXC_ID register to know whether the initialization has failed (EXC_ID$\neq$0), or it may be reactive and wait for an interrupt. The same mechanism may be applied to all SCA operations with no parameter i.e. initialize, start and stop. The most difficult Resource operations to map are the configure and query operations, because they use as parameter a variable-length sequence of id-value structures containing a sequence of bytes (string) and a polymorphic data type (any). We propose that the string identifier is transparently replaced in software stubs by a numerical identifier written in the PARAM_ID field. The data width of this fields allows a comfortable number of parameters (2^{21} i.e. 4+ million). If a parameter does not fit in the PARAM_VALUE register e.g. a long long integer or a sequence of bytes, several writes are required with the same values for the OP_ID and PARAM_ID fields. In this case, the hardware skeleton would have a demarshalling FSM to place the content of the PARAM_VALUE register in its internal registers, which could then be read by the hardware application component after all parameters were received. I notably discussed this interface with Bruno Counil, Rémy Chau and Benjamin Turmel.

7.4.5 Extension of the SCA Domain Profile for hardware components

SCA XML descriptors can be extended to take into account the non-functional properties of hardware SCA components such as memory map like in IPXACT, simulation and synthesis tools used, maximum frequency, area in gates or logic elements, reference model in C, transaction-level model in SystemC, etc.

Software Package Descriptor (SPD)

During application deployment, a SPD is used by the domain manager to load a component and its implementations. The SPD could contain several software and hardware implementations of a component for respectively different types of processors and OS, and different FPGAs. Like IP-XACT, the SPD could describe different views of the same component at different abstraction levels for instance functional, transactional, software and hardware component implementations. The SCA domain profile must be extended to address hardware SCA components on FPGA and ASIC. As opposed to [JTR05], we propose the following extensions to the SPD file descriptor:

- **view**: functional, TLM, software and hardware implementations,
- **target**: FPGA or ASIC,
- **family**: Xilinx VirtexV, Altera Cyclone II,
- **tool**: ModelSim, ISE/Quartus, Precision, Catapult and their version,
- **language**: SystemC, VHDL, Verilog, SystemVerilog, Catapult C,

- **performance**: frequency, size,
- **hardware/software interface**: memory map, device drivers,
- **hardware interface** e.g. name, direction and size of VHDL entity ports,
- **bitstream**: total or partial bitstream for reconfiguration

Software Component Descriptor (SCD)

The SCD describes the IDL interfaces supported, provided and required by a software SCA component. We propose a Hardware Component Descriptor (HCD), which describes the signal-based interfaces supported, provided and required by hardware SCA component. The signal-based interfaces result from the mapping of the operation-based interfaces identified by CORBA repository ids of the corresponding software SCA component. This mapping include the memory mapping of the operation-based interfaces according to a given alignment constraints. The chosen alignment rule must be the same for the alignment of fields in message payload and the alignment in memory. As illustrated in [Frö06], operation parameters may be aligned on 0, 2 or 4-bytes that describes the relative physical addresses where operation parameters may be read and written according to their direction (in, out, inout parameter) and access rights (read-only, write-only, read/write attribute). The absolute address of operation parameters is only known by the proxies like in [Frö06], but these proxies remain as transparent as usual CORBA proxies unlike [Frö06]. If the implementation of an abstract SCA component is changed from a software to a hardware implementation and vice versa or if its location is changed, only the reference in the proxy encapsulated the physical location needs to be updated. Client components will continue to access server components using the same interface to ensure location and implementation transparency. The SCD and HCD describe the interfaces of concrete components for the SCA Platform-Specific Model (PSM) that are derived from the common abstract component in the Platform Independent Model (PIM).

The type of hardware SCA components may be resource, device, resourcefactory, log, namingservice and eventservice. A hardware resource (resp. device) component implements the `Resource` (resp. `Device`) interface. A hardware resourcefactory component may be a hardware reconfiguration controller loading partial bitstreams from a memory to a reconfigurable area. A hardware log component may be a hardware controller writing to a persistent memory like memory card, flash or disk. Hardware implementations of naming service and event service will be presented in the following. Like the SCD, the HCD describes the component ports via their name, repository id, direction (provides or uses) and type (data, control (default), responses or test). The SCD and HCD may be extended with other component type such as event and debug.

Properties Descriptor

The Properties Descriptor defines the value of configuration attributes for components and devices. Basically, the same Properties Descriptor should be used for the SCD and HCD, but the it may include additional information for the HCD such as clock frequency.

It contains five types of XML elements: *simple* property (id, value, data type and min-max range, read/write mode, units, kind), *simplesequence*, *test*, *struct* or *structsequence* elements. These elements may have id, name, value(s), data type and min-max range, read/write mode, units, kind etc. The *simple* element is used to define id-value pairs. It correspond to CORBA basic data type such as short. The *Simplesequence* element is a list of *simple* elements with different values of the same type. It correspond to CORBA sequence data type. The *test* element is a list of input and result values to use with the *runTest* operation. The *struct* element is used to aggregate simple element of different types. It correspond to

CORBA struct data type. The *structsequence* element is a list of *struct* elements with different values. It correspond to CORBA sequence of structs.

Depending on the property kind (i.e. configure, test, allocation, execparam, factoryparam) and mode (i.e. readonly, readwrite, writeonly), `ApplicationFactory` and `DeviceManager` invokes the appropriate operations of the `Resource` (*configure* and/or *query*, *runTest*), `Device` (*de*/*allocateCapacity*, *execute*) and `ResourceFactory` (*createResource*) interfaces using these elements as parameters at application creation during deployment.

Device Package Descriptor (DPD)

A DPD may describe a board on which a waveform application is deployed. A DPD contains an id, name, version and group of title, author, description and hardware device registration. A hardware device registration element may describe an FPGA chip such as the manufacturer (e.g. Xilinx), the model number (e.g. XC4VFX20-FF672), the device class (e.g. Modem) and the description (e.g. Virtex IV FX20). The FPGA device registration element may have child hardware devices which describe the hardware devices contained in the FPGA such as processor devices (e.g. PowerPC or MicroBlaze), memory devices (e.g. Block RAM), bus devices (e.g. CoreConnect PLB and OPB), I/O devices (e.g. RocketIO), etc. Hence, the content of an FPGA device registration element may be similar to the description of the SoC architecture in design tools such as Xilinx EDK and Altera SoPC Builder.

Device Configuration Descriptor (DCD)

A device manager is associated with a Device Configuration Descriptor (DCD) file. The DCD identifies all devices and services associated with a device manager by referencing the associated SPDs. The DCD also defines properties of a device manager, enumerates the needed connections to services (e.g. file systems), and provides additional information on how to locate the domain manager.

The DCD may describe the deployment of hardware waveform components to FPGA devices using the component placement elements, the instantiation of hardware waveform components, the connections of hardware waveform components to hardware services such as naming, event and log service to retrieve their object reference i.e. logical address. The device manager may be implemented as a software object or a hardware object. However, such a CF management entity is likely to be implemented in software to reduce the hardware overhead.

Software Assembly Descriptor (SAD)

The SAD references all SPDs needed for an application, defines required connections between application components (connection of provides and uses ports / interfaces), defines needed connections to devices and services, provides additional information on how to locate the needed devices and services, defines any co-location (deployment) dependencies, and identifies a single component within the application as the assembly controller.

The SAD indicates the collocation of application components on the same device. If hardware application components are colocalized on the same FPGA device, their ports may be interconnected in point-to-point and communicate with each other via hardware function calls instead of messages. If hardware and software application components are colocalized on the same FPGA device, they may communicate through the available on-chip bus using address mapping or message passing.

This descriptor specifies the interconnection and configuration of application components. The SAD describes an id, name, component SPD files, partitioning information to indicate component placement (e.g. components collocation on the same device) and component instances (id, name, component properties, details on how object references are obtained - by a resource factory or by the Name Service),

an assembly controller that dispatches invocations from the user via a GUI to Resource operations delegated by the `Application`, connections between component provides and uses ports to exchange `Port` object reference, and externally visible ports of the application.

7.4.6 Deployment of Hardware SCA Resource Components

Some works have studied the management of the dynamic and reconfiguration of IP cores i.e their deployment such as run-time 2D placement. The deployment process of the SCA may be built on top of these works. The deployment and undeployment process of the SCA CF must be adapted to support the deployment and undeployment of hardware SCA Resource components. The heart of the deployment process is implemented in the `ApplicationFactory` interface. Concerning the deployment of a hardware SCA component whose implementation is packaged as a partially and dynamically reconfigurable bitstream, the CF must check that the size of the bitstream and the number of required hardware resources in terms of FPGA Configurable Logic Blocks (CLB) and RAM blocks, DSP blocks etc. are inferior or equal to the resources available in the reconfiguration areas and their size to authorize the hardware component instantiation.

We update the deployment process described in [JTR06] and presented in Figure 7.32.

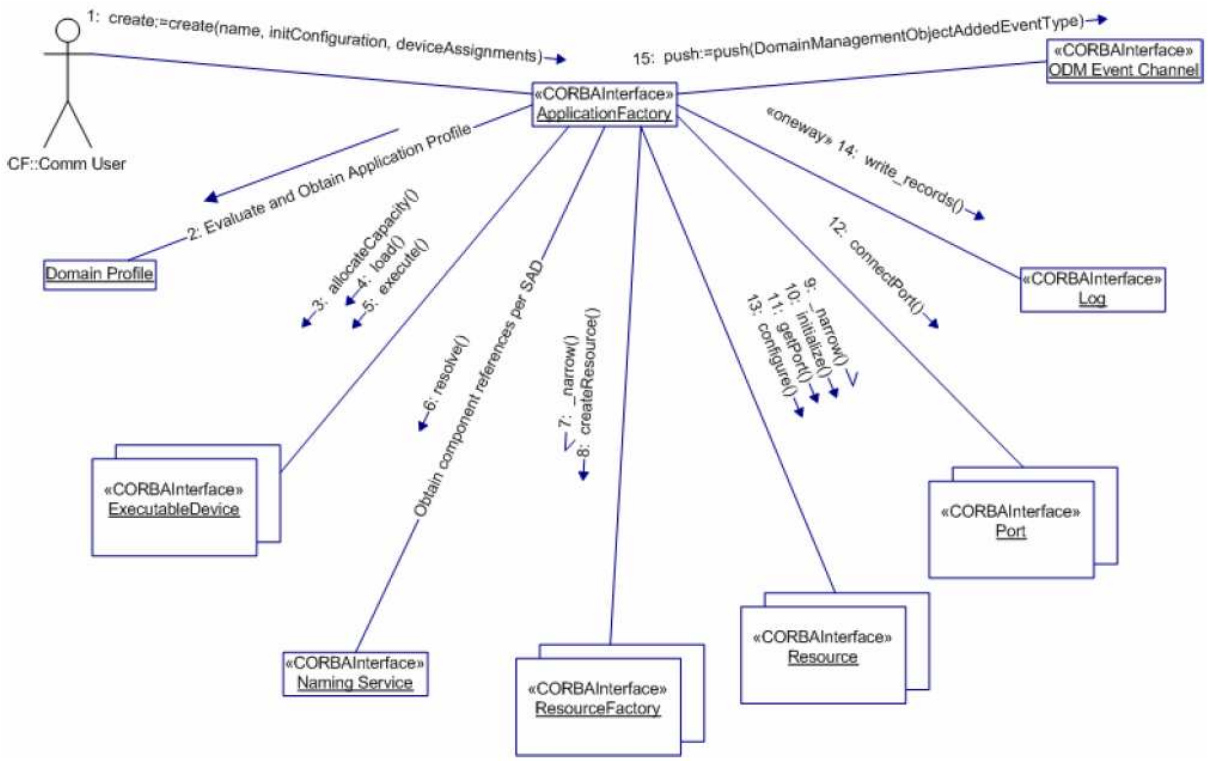


Figure 7.32: SCA deployment process implemented by an `ApplicationFactory`

The deployment process not only needs the filename of the bitstream but also metadata i.e. the synthesis results after Place and Route on different target FPGAs to exactly know the required hardware footprint i.e. the hardware resources needed for the configuration of the chosen FPGA reconfigurable area with the partial bitstream.

7.5 Hardware Middleware Architecture Framework

We tried to apply the same design patterns than those typically used to design software middleware implementations and to identify the same architectural elements. As depicted in figure 7.33, we propose a hardware middleware architecture framework based on five layers: the application layer, the presentation layer, the dispatching layer, the messaging layer and the transport layer.

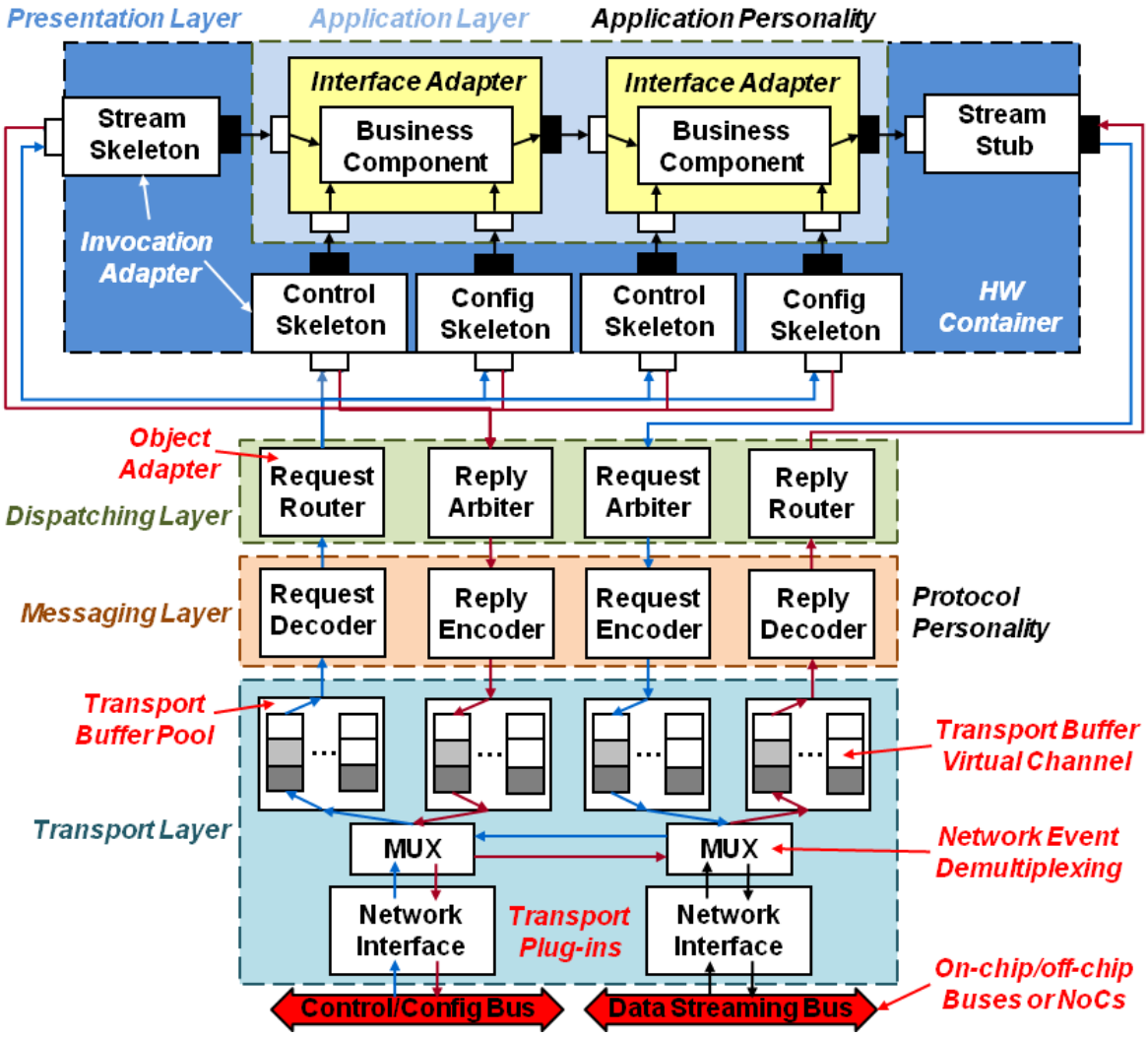


Figure 7.33: Middleware Architecture Framework

The objectives of such a layered architecture is to separate the functionalities of a hardware communication middleware into replaceable components. In the chapter 8 presenting our practical experiments, we will describe three configurations of this middleware architecture framework based on the classical memory mapping and message passing using SCA MHAL and CORBA GIOP 1.0. The proposed middleware architecture framework is a generalization of these middleware configurations. This architecture framework must be seen as a generic configuration or assembly of middleware platform components.

7.5.1 Application and Protocol personalities

We reuse the notion of application and protocol personalities proposed for PolyORB [VHPK04]. We consider the hardware SCA components as part of a SCA application personality. Another application personality could be based on hardware CCM components or other component models. Two protocol personalities have been implemented using SCA MHAL and CORBA GIOP 1.0. The request and reply paths may be merged if the message protocol does not perform no distinction between requests and replies like MHAL. The hardware middleware architecture presented in figure 7.33 is generic enough to include the studied middleware configurations. However, application and protocol requirements may require exploration of the middleware architecture to satisfy given QoS constraints. For instance, the number of buffers and network interfaces and the allocation of physical addresses i.e. *network endpoint*s to component ports may change from one pair of application and platform to another one. Indeed, the final middleware configuration depends on the mapping of connectors, which bind application component ports, to the platform communication buses. Various application to platform mapping may be possible according to the communication constraints. For instance, the application information flows like control, configuration and data stream may be mapped onto shared or dedicated buses for each or all flows. Moreover, the platform typically provide proprietary buses for DSP-FPGA, FPGA or GPP-FPGA communications No encapsulation into messages of data streams may be required if the real-time constraints are too stringent to accept the overhead in latency needed for message decoding and encoding. The overhead of message encapsulations depends on the ratio between the message format and raw user-data.

7.5.2 Application layer

The *application layer* contains the hardware application components. These components may be for instance hardware SCA `Resource`s or CCM components. They implement in hardware the most relevant operations required by the software component model interfaces. The other operations can be transparently implemented in software stubs to save hardware resources. The components in the application layer can be reused independently of the other middleware layers.

7.5.3 Presentation layer

The *presentation layer* performs the encoding/decoding of message payloads. It consists of hardware stubs and skeletons which accomplish the hardware marshalling/demarshalling of message byte streams. They act as invocation adapters [PRP05] which adapt the generic interface of the middleware and the specific application components. IDL compilers may automatically produce these stubs/skeletons by generating dedicated FSMs. Based on the operation signatures defined in CORBA IDL, encoding/decoding FSMs may extract a known set of bits from the message byte stream in one or several clock cycles depending on the mapping configuration (e.g. the data bus width). Like proposed by the Packed Encoding Rules (PER), message payloads may be encoded using data alignment like GIOP CDR or not like SCA MHAL and ZeroC ICE. Stubs decode message payloads into operation parameters which are stored in memories (registers, FIFOs or RAMs). Stubs may forward the invocation to the application components via a `push` interface, or the components can read the parameters values via a `data` interface or a `pull` interface. The collection of stubs/skeletons that isolate the application components from their operating environment can be viewed as a hardware *container* which provides activation (via writes for hardware invocations), synchronization (e.g. clock), storage (e.g. registers) services.

We investigate the implementation of the `Config` port in VHDL based on both MM and MP transfer modes. We present some code excerpt in listing 7.12.

```
-- Package resulting from the mapping configuration
```

```
package Filter_configPort_cfg is
constant MW: natural := 3; -- m data type size
constant D: natural := 5; -- paDding for p
constant PW: natural := 8; -- p data type size
constant HAW: natural := 8; -- PUSH_ADDR_WIDTH
constant HDW: natural := 16; -- PUSH_DATA_WIDTH
constant LAW: natural := 7; -- PULL_ADDR_WIDTH
constant LDW: natural := 16; -- PULL_DATA_WIDTH
constant M_P_L_ID: std_logic_vector(HAW-1 downto 0) := x"04";
constant CONFIGURE_ID: std_logic_vector(HAW-1 downto 0) := x"05"; -- ...
   end;
-- Hardware Component Port storing method parameters
use work.Filter_configPort_cfg.all;
entity Filter_configPort is
generic(HAW,HDW,LAW,LDW: natural);
port (
-- Component Port Invocation Interface
push_i     : in  std_logic;
push_addr_i: in  std_logic_vector(HAW-1 downto 0);
push_data_i: in  std_logic_vector(HDW-1 downto 0);
ack        : out std_logic;
-- Component Port Internal Interface
push_o     : out std_logic;
push_addr_o: out std_logic_vector(HAW-1 downto 0);
push_data_o: out std_logic_vector(HDW-1 downto 0);
m_data_o: out std_logic_vector(MW-1 downto 0);
cf_pull_i     : in std_logic;
cf_pull_addr_i: in  std_logic_vector(LAW-1 downto 0);
cf_pull_data_i: out std_logic_vector(LDW-1 downto 0);
-- ... ); end;
-- Memory Map (MM) 1
elsif push_addr = M_P_L_ID then
 m_data_o       <= push_data(MW-1 downto 0);
 p_data_o       <= push_data(PW+MW+D-1 downto MW+D);
-- Message Passing (MP) 1
elsif push_addr_in = CONFIGURE_ID then
 next_state     <= STATE_CONFIGURE_CF;
 m_data_o       <= push_data(MW-1 downto 0);
 p_data_o       <= push_data(PW+MW+D-1 downto MW+D);
-- Memory Map (MM) 2
elsif push_addr = SET_M_ID then
 m_data_o       <=push_data(MW-1 downto 0);
-- Message Passing (MP) 2 on top of Memory Map (MM) 2
elsif push_addr_in = CONFIGURE_M_P_L_ID then
 next_state     <= STATE_SET_P;
 ad_push_o      <= '1'; -- address decoder
 ad_push_addr_o <= M_ID;
 ad_push_data_o <= push_data(MW-1 downto 0);
-- ...
```

Listing 7.12: Mapping Illustration in VHDL for the FIR filter

Based on the mapping configuration, the **Filter_configPort_cfg** package parameterizes the **Filter_configPort** module with the bit width of the hardware interface signals, the IDL attributes and

parameters and operation identifiers. In the MM1 and MP1 mode, the skeleton decodes the same message containing m,p,l and ten coefficients. MP1 requires more hardware resources notably a DSP48 to increment the address of coefficients in RAM.

Thanks to logic optimization, the support of both schemes requires fewer resources than the sum of both. In the second experiment, we develop a generic address decoder to allow the separate configuration of each attribute and parameter on top of which a MP to MM adapter may be implemented. No RAM is thus required in this adapter. This approach provides more flexibility at the expense of more consumed resources. Based on these prototypes, more exhaustive investigations are necessary to observe how these schemes behave in time and space when scaling up. MM or MP or a combination of both could be chosen transparently and explicitly.

7.5.4 Dispatching layer

The *dispatching layer* performs the routing of incoming messages and the arbitration of outcoming messages. When the component port provides services and acts as a server port, the incoming messages are request messages, while the outgoing messages are reply messages. Conversely, when the component port uses services and acts as a client port, the incoming messages are reply messages, while the outgoing messages are request messages.

When ingoing messages are received, the input router dispatches incoming invocations to the appropriate skeleton according to the logical address in the request or reply message header e.g. an operation name or request identifier for GIOP or a Command Symbol for MHAL. For hardware object-oriented middlewares, the router acts as an *object adapter*. In memory mapping, the router is implemented as a classical address decoder that selects a skeleton based on the input physical address. This router shares the same design trade-offs as its counterpart in buses and NoCs. It may be implemented as a simple 1-to-N multiplexer where N is the number of skeletons, but this is not scalable when the number of component ports increases. To increase its scalability, the router can have a distributed implementation for instance as a tree of 1-to-2 multiplexer that makes a better use of FPGA routing resources. The router implementation may generic and configured at compile time depending on the number of component ports.

When outgoing messages are ready to be sent, the output arbiter selects a skeleton for replies or stub for requests according to some classical arbitration policies e.g. round-robin, priority-based or a combination of both.

7.5.5 Messaging layer

The *messaging layer* performs the encoding and decoding of message headers, encapsulates message payloads with message headers, and implements the protocol state machine with FSMs. The more the message format contains information and the messaging protocol is complex, the more there are states and the more hardware resources are consumed to implement it. For instance, the implementation of GIOP 1.0 request-reply messages and MHAL messages showed that GIOP requires 52% of total LUTs in addition. In memory mapping, this layer is empty since no message processing is necessary. The message payloads are directly transferred from the transport layer to the dispatching layer. The messaging layer provides an *Extensible Messaging Framework* (**EMF**) in which message plug-ins may be introduced to support various message protocols. Dynamic and partial hardware reconfiguration of FPGAs [SMC+07] may be used to dynamically loadable hardware messaging modules as they were dynamically linkable software libraries.

7.5.6 Transport layer

The *transport layer* provides a generic hardware interface to the underlying transport mechanisms e.g. on-chip or off-chip buses, Network Interfaces in NoCs, high-speed serial links, etc. It includes *transport buffer*s to store the incoming and outcoming messages to be sent and received from the network. In our experiments, the transport buffers have been implemented as generic hardware FIFOs which have been mapped to the FIFOs available from the target FPGA library. Transport buffers may be organized as a bounded pool of buffers with a configurable depth and width defined at compile time. Each transport buffer represents a *virtual channel* and may be associated with a predefined priority to avoid *priority inversion* like in Real-Time CORBA ORBs and NoCs. For instance, a low-priority control/configuration message should not be blocked by the processing of a high priority data stream message. These virtual channels allow to provide differentiated services with best-effort or guaranteed quality of services (QoS) regarding throughput, latency and jitter like in NoCs [RDP+05] [Bje05]. Typically, the transfer of video, voice and data has different qualities of services. This layer contains *transport plug-in*s like bus bridges to adapt the generic interface of the transport buffers and the specific interface of the standard bus. Each transport plug-in constitutes a *network interface*. Such transport plug-ins require stable hardware interfaces which can be derived from CORBA IDL interfaces according to our mapping proposition. In our experiments, we evaluate OCP to provide a standard interconnection socket. For instance, an Xilinx OPB-to-OCP bridge have been implemented and used in our middleware prototypes. The transport layer provides an *Extensible Transport Framework* (**ETF**) like in Real-Time CORBA in which transport plug-ins may be introduced to support various transport protocols. Dynamic and partial hardware reconfiguration of FPGAs [SMC+07] may be used to dynamically loadable hardware transport modules as they were dynamically linkable software libraries.

7.5.7 Performance Overhead

We consider that the services provided by each of these layers are more or less already implemented by classical communication buses. Indeed, communication buses need to transfer messages from a communicating entity to another, route messages to the appropriate entity, which must marshal and unmarshal commands and parameters to and from messages. These services are simply organized as a set of well-defined layers. This organization does not necessarily imply an important performance overhead in itself. The potential performance overhead will primarily depend on the implementation of each layer. The proposed middleware framework is generic and must be tailored to the specific communication requirements of an application. For instance, a designer may choose memory mapping instead of message passing or prefer a lightweight message format to reduce communication latency. The increasing overhead of three configurations of this middleware architecture framework will be illustrated in chapter 8.

7.6 Limitations

As already said previously, the CORBA Interface Definition Language suffers from inherent limitations to address hardware design. The standard CORBA IDL is a declarative language, which does not address behavioral specification. As a number of language mappings to software implementation languages are already specified, a natural approach was the specification of a IDL-to-HDL and IDL-SystemC TLM mappings. The other standard use by the SCA which could be used is UML, but no standard mapping exist between UML and software programming languages. Since a standard mapping exists between UML and CORBA IDL using the UML for profile CORBA and CCM, definition a mapping from IDL-to-HDL is similar to defining a mapping from UML-to-HDL. CORBA IDL can be considered as a concrete textual syntax to the UML graphical language. Nevertheless, UML provides a standard mean to specify

behavior of abstract components independently of their implementations. It notably provides Action Language, Sequence Diagram and State Diagram. Another approach would be to include language construct proposed in the academic literature, but these extensions would be not standard. These extensions include for instance behavioral protocol [PV02]. Moreover, the use of the MDA approach is conceptual rather than a strict implementation of the OMG approach e.g. using UML.

7.7 Conclusion

In this chapter, we presented a common component approach for system, hardware and software components based on the language mappings of a high-level interface specification in CORBA IDL or UML and a common middleware approach through a hardware middleware architecture framework.

We propose to leverage the conceptual SDR design methodology based on MDA and the component/container approach, which existed at Thales Communications in 2005 [GNSV07] , with SystemC Transaction Level Modeling (TLM) to simulate the platform independent model of waveform applications (see §7.3.1.0), and the platform specific models of waveform applications and SDR platforms (see §8.2.2). The component/container approach allows one to transparently integrate functional models in C as SystemC components in virtual platforms. SystemC TLM allows the simulation and validation of PIMs and PSMs at different abstraction levels. We propose a common design flow for system, hardware and software components based on MDA, SystemC TLM, and language mappings from business interfaces to simulation and implementation languages at different abstraction levels. The MDA approach is coherent with the development of the new technology-independent SCA release

To support the systematic refinement of object-oriented component interfaces in CORBA IDL or UML to system, software, and hardware component interfaces in C/C++, SystemC and VHDL, we propose to define new language mappings from CORBA IDL to SystemC TLM/RTL and VHDL.

The mapping from IDL3 to functional SystemC extends the IDL-to-C++ mapping with additional mapping rules to fit specific SystemC language constructs. A SystemC component is a `sc_module` component with required `sc_export` ports and provided `sc_port` ports which are associated to `sc_interface` interfaces. We illustrate the UML/IDL-to-SystemC mapping at functional level through the transition from a Platform Independent Model (PIM) in UML of two waveform components to a Platform Specific Model (PSM) in SystemC (see §7.3.1.0).

The mapping from IDL3 to SystemC at Transactional Level is based on transactional connectors called transactors, which adapt the mapping of user-defined object-oriented interfaces to standard TLM interfaces. The cornerstone of this mapping relies on the equivalence between operation signatures and message definition in the object-oriented approach. Regardless of the abstraction level, the syntax and semantics of the business interfaces in UML or IDL and the associated transactions are the same, only the manner of transferring messages changes with the abstraction level. This object-oriented approach to refinement contrast with the classical ad-hoc refinement in SystemC, where the syntax and semantics of business interfaces are lost during refinement.

For the mapping from CORBA IDL to Hardware Description Languages (HDLs), we emphasize the differences between interface mapping between software languages, and interface mapping between hardware and software languages. Existing CORBA IDL language mappings are generally systematic and one-to-one mapping, which share a common procedure call abstraction. Software-to-hardware mapping require more flexibility due to the inherent hardware trade-off between area, performance and power consumption. An IDL-to-HDL mapping requires to select and configure non-functional properties such as hardware interfaces, timing and protocols, and explore the mapping space. We propose the use of a standardized mapping configuration file to explicitly and finely configure the mapping.

We formulate user and implementation requirements for a standard IDL3-to-HDL mapping. The

most important requirements concern concurrency (e.g. concurrent method invocations and executions within a component, concurrent accesses to internal attributes and method parameters, concurrent invocations of methods from multiple clients); connectors to adapt different interaction semantics or synchronization schemes; hardware interface and protocol framework to improve the portability and interoperability of HW components; blocking and non-blocking synchronization; message passing with aligned/unaligned data types encoding rules; stream-oriented Inter-ORB Protocol to pipeline requests and replies with Time or Space Division Multiplexing (TDM/SDM); Extensible Transport Framework (ETF) with a generic bus interface like OCP; Extensible Message Framework (EMF) to support domain-specific message protocols like SCA MHAL, and a subset of IDL data types with bit-accurate types.

In our mapping proposition, a hardware object-oriented component (e.g. SCA, UML, CCM) is represented by a hardware module, whose implementation (e.g. VHDL `architecture` may be selected by a hardware configuration e.g. VHDL `configuration`. An abstract component port is mapped to a logical hardware component port, which is a set of RTL input and output ports.

We distinguish three kinds of hardware component interfaces: the business component core interface or internal component interface to access operation parameters, the invocation interface of component ports or external component interface to send/receive invocations, and the connector interface to send/receive messages. By default, methods in the same object interface share the same hardware port and thus method invocations of the same interface are sequential.

A hardware connector may represent non-functional properties such as synchronization and communication such as flow control, clock domain crossing, interaction semantics, and protocols of point-to-point buses, shared buses or NoCs.

A hardware container provide all remaining non-functional services such as lifecycle and persistence to store component state. For instance, the lifecycle service may manage component creation/destruction e.g. with dynamic partial reconfiguration, component initialization e.g. with a asynchronous or synchronous reset, parameters configuration with register banks, and activation/deactivation e.g. with a clock or enable signal.

The mapping is based on a common and independent family of hardware interfaces and connectors. We apply our mapping proposition to the interfaces of a FIR filter, a Transceiver standardized at the SDRForum, a convcoder and bitinterleaver SCA components and the dining philosopher problem in CCM and a modulator in [GBSV08].

An IDL3-to-VHDL compiler prototype has been implemented according to a MDA approach. Based on the LwCCM meta-model, we define a VHDL meta-model. We extended the standard CCM meta-model and proposed a VHDL meta-model based on the architecture of the first one. The RTL mapping is finely configured by an editor generated from the CCM meta-model. Model-to-model transformations rules were defined in the OMG QVT language. Model-to-text transformations were defined by code templates using OMG MOF2Text language. Our hardware middleware architecture framework allows hardware components to transparently communicate with software and/or hardware components using either classical memory mapping or message passing with CORBA GIOP or SCA MHAL. The proposed middleware architecture applies the software middleware design patterns down to hardware. The hardware middleware architecture framework contains five layers: the application layer consists of the hardware application components, the presentation layer performs the encoding/decoding of message payloads, the dispatching layer deals with the routing of incoming messages and the arbitration of outcoming messages, the messaging layer takes care of the encoding and decoding of message headers, encapsulation of message payloads and the protocol state machine, and the transport layer provides a generic hardware communication interface and transport buffers. The hardware middleware architecture framework is based on the concepts of application and protocol personalities. An application personality corresponds to a component model personality e.g. SCA or CCM . A protocol personality corresponds to a message protocol such as SCA MHAL and CORBA GIOP.

Our mapping proposition was first dedicated to CORBA IDL, but can be easily extended to any object-oriented interface description notably in UML.

Chapter 8

Experiments

Contents

8.1 Introduction

This chapter presents the experiments on which the propositions of the previous chapter are based. These experiments are part of our contributions. The two studied applications are signal-processing applications in wireless digital communication systems. These systems are typical products developed by Thales Communications, where most of our thesis took place. In particular, these experiments focus on the physical layer of radio modems, which are implemented on FPGA.

This chapter is organized as follows. First, section 8.2 presents the modeling of an existing RTL IP core component as a reusable SystemC component at Programmer View (PV) level, its integration and software validation in a virtual platform of a subset of an MPSoC and the refinement of this virtual platform to be closer to the real platform. Then, section 8.3 describes the modeling at different abstraction levels of a subset of a high-data rate OFDM modem using SystemC FIFO channels, OCP TLM and OCP RTL, and the implementation of the proposed hardware middleware framework using memory-mapping and message-passing with CORBA GIOP and SCA MHAL. Finally, section 8.4 concludes this chapter.

8.2 DiMITRI MPSoC for Digital Radio Mondiale (DRM)

In this section, we present the modeling of a Viterbi decoder in SystemC TLM and the modeling of part of the DiMITRI MPSoC for Digital Radio Mondiale (DRM) as a virtual platform. This virtual platform is then refined by replacing the processor model and the Bus Functional Model (BFM), and by using adapters called transactors.

8.2.1 Application and Platform presentation

The DiMITRI MPSoC [QSC04] implements a digital radio receiver which supports two digital radio front-ends for *Amplitude Modulation* (**AM**) and *Frequency Modulation* (**FM**) analog demodulation, *Orthogonal Frequency-Division Multiplexing* (**OFDM**) digital demodulation and multimedia codecs. DiMITRI consists of two ARM processors, one dedicated to the physical layer and the other to the upper layers, and hardware accelerators notably a Viterbi decoder and two *Digital Down Converter*s (**DDC**) as depicted in Figure 8.1. Available materials for this application were software code including drivers, VHDL code and related specifications.

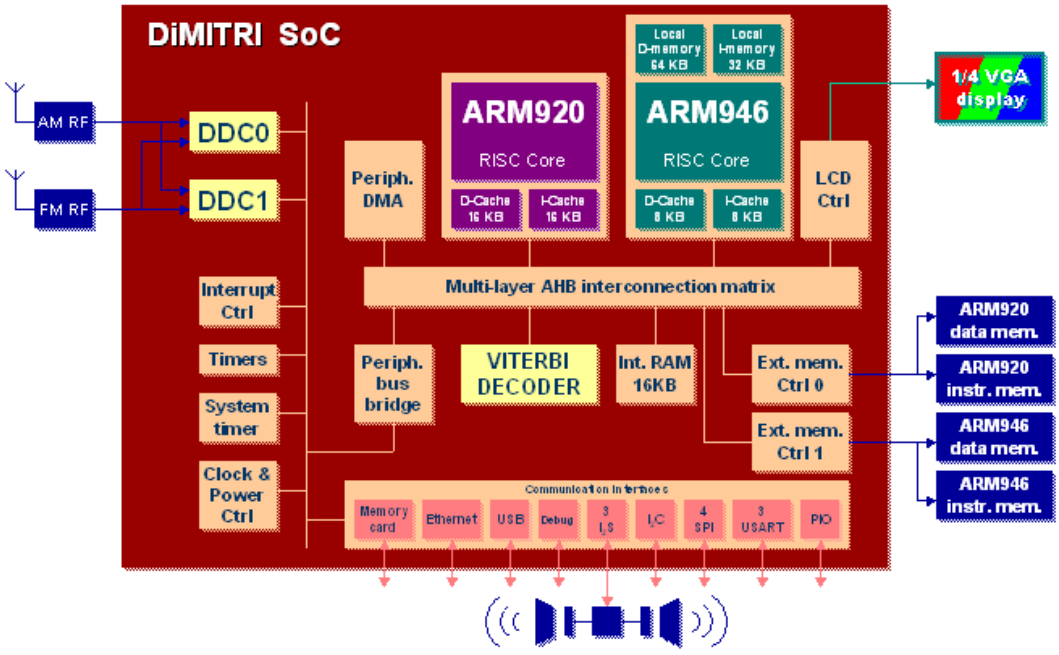


Figure 8.1: DiMITRI MPSoC architecture overview

8.2.2 Executable PSM in SystemC TLM

This sub-section presents the modeling of a business component, a Viterbi decoder, using the component/container approach and the refinement of a connector, a Bus Functional Model, in SystemC TLM. We built a virtual platform of a subset of the DiMITRI MPSoC. We consider this virtual platform constitutes an executable Platform Specific Modem (PSM) of the DiMITRI MPSoC. Note that we use the term "PSM" according to its literal meaning, but not in a strict OMG MDA context. We modeled the previously validated RTL model of the Viterbi decoder as a SystemC Component and integrated it in the virtual platform. The business code in C of the Viterbi decoder has been encapsulated in a SystemC TLM

container with a register/bit accurate hardware interface at Programmer View (PV) level. The resulting SystemC TLM IP has been reused without any modification during the refinement and exploration of the virtual platform depicted in Figure 8.2. The first virtual platform is composed of an ARM968-ES Instruction Set Simulator (ISS), an OCP TL2 channel [OI07], two memories (ROM and RAM) with an OCP TL2 interface and an OCP TL2-to-PV transactor connected to the Viterbi model. According to the taxonomy of connectors proposed in [MMP00] (see 6.5.7), we consider that the OCP TL2 channel represents an arbitrator/distributor connector, while the OCP TL2-to-PV transactor is a conversion connector.

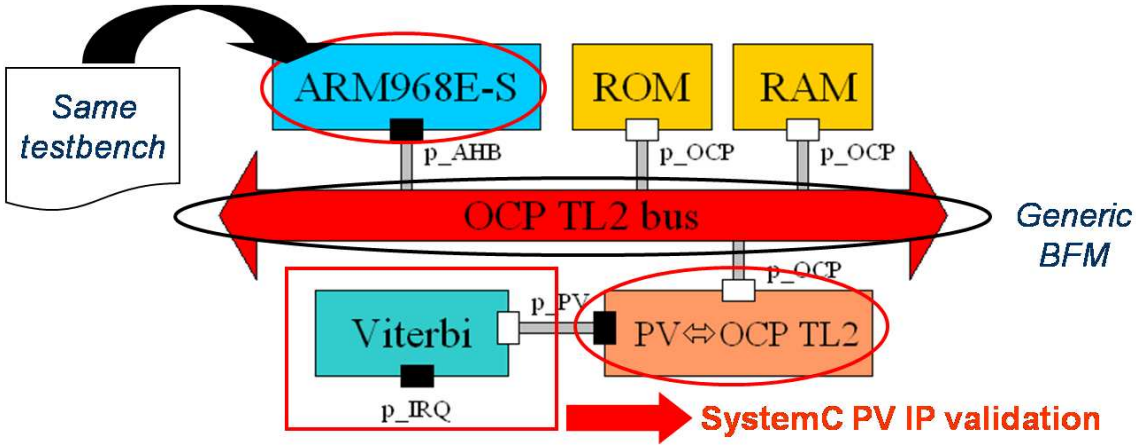


Figure 8.2: Generic Virtual Platform at Programmer View (PV) level

Business Component Refinement: From C functional model to SystemC component at PV level

The hardware engineer at Thales, Bertrand Mercier, who developed the Viterbi decoder IP core extracted from the reference simulation in C a functional model in C of the Viterbi IP. We encapsulated this C model in a SystemC TLM IP at Programmer View (PV) level. At this abstraction level, SystemC models are untimed, memory map and register accurate, and can support interrupt handling. We model the Viterbi registers with the same bit accuracy as in the Viterbi IP core specifications using the SystemC Modeling Library (SCML) [SCM08]. A register bank and its registers are declared and bound in the Viterbi module constructor. Bitfields allow one to select a precise set of bits inside a register. A read or write callback function is associated to each register or bitfield. A read or write access to the register or bitfield triggers the callback. The register bank is bound to a PV target port, which uses the OSCI TLM 1.0 API [RSPF05].

As shown in Figure 8.3, the C model implements the business logic of the Viterbi decoder and its services are called by the container through register and bitfield callback functions. These callbacks functions implement the technical logic such as marshalling/unmarshalling, storage and synchronization. Indeed, the Viterbi component decodes sequences of octets, which are sent in burst four octets at a time on the 32bit AHB bus. Hence, read/write callbacks act as marshalling/unmarshalling procedures, which unmarshal words as bytes to be decoded and marshal decoded bytes as words. The C code file is compiled with SystemC files keeping the model intact and constitutes a reusable golden source. The SystemC container encapsulates the C model. There is a clean separation of concerns between business and technical logic. The resulting SystemC TLM model is untimed and event-driven. The Viterbi model was validated with the testbench already used to verify the hardware IP core. The ISS executes the testbench code and accesses to the Viterbi registers thanks to the original driver in C. The modeling

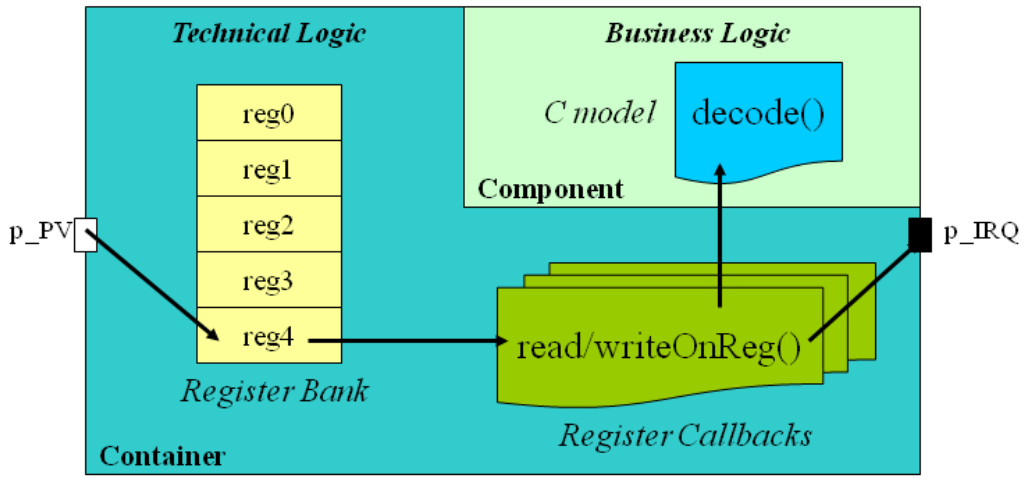


Figure 8.3: Encapsulation of a C behavioral model within a SystemC PV container

effort to follow this component/container paradigm from a C model is not really important, but it enables portability and reuse of business logic.

Transactional Connector Refinement: From OCP TL2 to AHB bus model

The refinement of the first virtual platform was performed, while keeping the same SystemC TLM component as depicted in Figure 8.4.

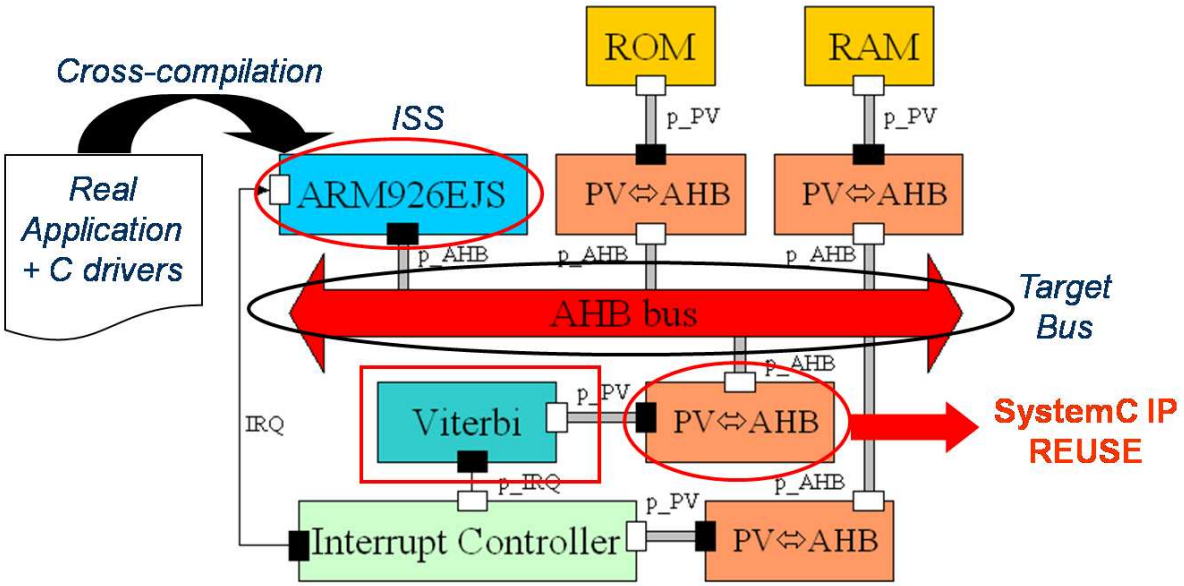


Figure 8.4: Transparent refinement using transactors

The transactional connector refinement consisted in replacing the OCP TL2 bus model by an AHB bus model, and the virtual hardware component refinement in replacing the ARM968-ES core by an ARM926-EJS core. Both steps enable us to be closer to the real platform. Thanks to the use of an AHB-to-PV transactor and cross-compilation, a true reuse of the Viterbi SystemC TLM component was

performed since no modification was necessary. The refinement from one platform to another is thus highly simplified.

Moreover, a real software application has been executed on this second virtual platform. We successfully validated an iterative decoding application previously executed on the real MPSoC, which uses the Viterbi decoder many consecutive times.

Unlike the OCP-based platform which used polling, the second platform contains an interrupt controller. The Viterbi decoder uses it to inform the ARM processor core that decoded samples are available and can be read from registers. This represents another step in the accuracy of the virtual platform.

Thus the contract at PV level - into which waveform software component developers could enter - has been fulfilled: software validation, interrupt handling and modeling accuracy have been achieved. This SystemC TLM IP component can be reused to model more quickly and easily other Viterbi-based virtual SDR platforms.

This test case was not sufficiently complex to perform architecture exploration or obtain significant profiling results. To be able to go further with exploration and profiling, another hardware engineer at Thales, Bertrand Caron, continued the modeling of the DiMITRI MPSoC. Using the same modeling approach, he modeled the DDC and DMA IP cores of the real SoC and integrated more software application code onto the ARM core. His results showed that the Instruction Accurate ISSs we used in the presented virtual platforms significantly increases the simulation speed of an order of magnitude (x10) in comparison to an Cycle Accurate ISS.

8.2.3 Results Analysis

Compared to the classical approach in which a C functional model is developed when manually translated into VHDL to build an IP core, a SystemC container allows the reuse of the C functional model to build a SystemC component, which can be integrated within a virtual platform. A SystemC component at Programmer View level allows an earlier validation of the hardware/software interface and software drivers, which are error-prone code to develop. Modeling hardware accelerators in SystemC at lower abstraction level such as cycle accurate level may be not required if the main purpose of the simulation is software validation, and since several models of the same component at different abstraction levels are difficult to maintain and keep synchronized in terms of functionalities. Moreover, the C functional model may be translated by HLS tools into synthesizable HDL code with some coding rules or after some code rewriting e.g. using bit-accurate data types. Even if HLS tools offer a direct refinement from C to HDL, SystemC components may still be required since they provide higher simulation speed than RTL components. Finally, the generation of a SystemC container, which maps the software interface of the C functional model to the register bank of the SystemC IP at PV level may be automated using a memory-mapped IDL-to-SystemC mapping. For instance, each operation parameter may be explicitly mapped to a memory-mapped register.

In parallel to the modeling of hardware accelerators as SystemC components at PV level, the communication architecture of the virtual platform may be explored by analyzing bus contentions and estimating power consumption using different high-level but accurate bus functional models. This exploration is at Architect View (AV) level. SystemC transactors are adaptor connectors which adapt two software interfaces at different abstraction levels e.g. a hardware accelerator at PV level and a BFM at AV level. These transactional connectors supports SystemC component reuse and are themselves reusable independently of the business components themselves. They allows a clean separation between computation in hardware accelerator models and communication in BFMs.

8.3 High-Data Rate OFDM Modem

In this section, we describe the modeling at different abstraction levels of a subset of a high-data rate OFDM modem using SystemC FIFO channels, OCP TLM and OCP RTL, and the implementation of the proposed hardware middleware framework using memory-mapping and message-passing with CORBA GIOP and SCA MHAL.

8.3.1 Application and Platform presentation

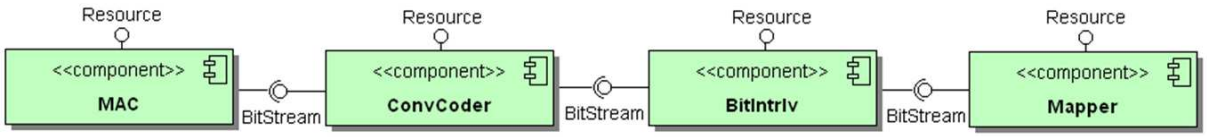


Figure 8.5: Example of component-based applications

As depicted in figure 8.5, I selected a subset of a complete High-Data Rate OFDM Modem and focused on the coder part of the OFDM transmitter. This modem was designed in 2003 to favor a one-to-one mapping between a waveform functional component and its software and hardware implementation, and reuse. The application is a custom OFDM waveform, which was developed according to a traditional design flow without a software or hardware component-oriented approach in the sense of this work e.g. without SCA, CCM or SystemC TLM. The sample waveform is composed of four functional components: a *Media Access Control* (**MAC**) component, a Convolutional Coder (**CC** or ConvCoder), a Bit Interleaver (**BI** or BitIntrlv) and a bit-to-symbol Mapper (**MAP**). The first component is part of the MAC layer, while the three last blocks are part of the modem in the physical layer. The modem has two main operational modes. In the mode 1, the MAC layer is hosted on a GPP, the ConvCoder and the BitIntrlv on an FPGA and the Mapper on a DSP. In the mode 2, the BitIntrlv is on DSP, while the others functional components remain on the same computation unit than in the mode 1. Within the mode 1, there are four sub-modes corresponding to different modulation schemes and thus data throughput.

In our experiments, the MAC component is a straightforward bit generator that produces data from a file according to the mode. The Convolutional Coder introduces redundancy to ease error correction. For each input bit, it produces two output bits. It is implemented as a shift register in which some intermediate bits are XORed (eXclusive ORed) with the output of the shift register. The Bit Interleaver shuffles bits to avoid that errors affect continuous bit of information and spread these errors on several unrelated bits. The bit-to-symbol mapper establishes a mapping of a number of bits according to the mode onto a symbol of information to reduce the amount of information to be transmitted. In this test case, it is a simple observer that stores the data produced by the bit interleaver and compares them with the results from the reference simulation.

This application is based on *Time-Division Multiplexing Access* (**TDMA**) Radio communications are divided into time *slot*s which are temporal windows during which a transmitter can transfer user application information (voice, video, data, etc.) to a receiver in one direction (half-duplex) or in both directions (full-duplex). As depicted in figure 8.6, each slot is itself divided into eight sub-slots called *burst*s in which data are transmitted at different frequencies. This technique is called *Frequency-Hopping Spread Spectrum* (**FHSS**). As its name suggests, the total transmission spectrum is spread across multiple wideband channels having its own carrier frequency. The transmission frequency *hops* from one value to another one according to some predefined schemes. This mechanism allows one to protect communications from interference e.g. due to multiple communication paths, signal fading due to frequency-

selective channels, jamming and eavesdropping. In figure 8.6, the dwell period is the frequency hopping period plus the burst period. The slot period includes the propagation period to avoid interferences.

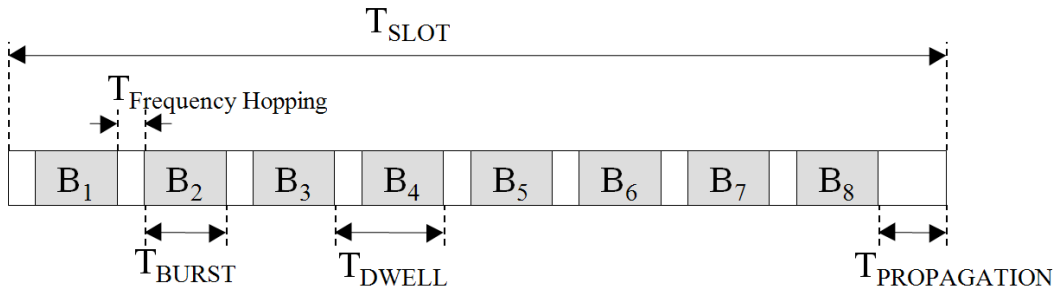


Figure 8.6: Slot Format of the High-Data Rate OFDM Modem

The available materials for this waveform application were the specifications, the reference simulation in C fixed point for the whole communication chain, the software implementation in C for the functional blocks hosted on DSP and the hardware implementation in VHDL for the functional blocks hosted on FPGA.

The initial objectives of this example was to find a concrete and representative example of a waveform application to study:

- the transition from a traditional design to a SCA-compliant approach;
- the hardware transposition of the SCA Resource interface for FPGA;
- an extremely fine granularity of hardware SCA Resources i.e. an IP component;
- the hardware transposition of component, container, connector approach for FPGA;
- the transparent migration of a waveform functional component, here the BitInterleaver, from DSP to FPGA and vice versa to represent two mappings of the same waveform.
- distribution transparency for components on GPP, DSP and FPGA: access transparency, communication transparency and location transparency.

During this work, we implemented these waveform building blocks as SystemC components, CORBA objects, software SCA Resource components in C++ with the SCA Core Framework from Thales [SMC+07] and OSSIE[74], and hardware SCA Resource components in VHDL.

8.3.2 Modeling at Different Abstraction Levels

Kahn Process Network with SystemC FIFO Channels

In the scope of the ANDRES project [AND08], a hardware engineer at Thales, Frédéric Colon, encapsulated the reference model in C of the High-Data Rate OFDM Modem in SystemC at Functional View (FV). Each functional block of the waveform application was translated into a SystemC module and each function call between functional blocks was replaced by a SystemC FIFO channel to transfer data between the consecutive blocks. This model represents a Kahn Process Network (KPN) in which sequential SystemC processes communicate through asynchronous message passing using unbounded SystemC FIFO channels.

[74] http://ossie.wireless.vt.edu

As depicted in figure 8.7, I selected a subset from the complete waveform model for which we had the RTL implementation of the functional components mapped to FPGA. The sample waveform is composed of a controller, which broadcasts configuration invocations on the previously mentioned components. The controller acts as the SCA Assembly Controller [JTR06]. Application components are represented by SystemC `sc_module`, while connectors correspond to `sc_fifo` channels. The components follow a producer/consumer model or Kahn process network in which data source components write data on an `sc_fifo`, while data sink components reads data from the `sc_fifo`. The FIFO connectors implement the `read`/`write` operations of the FIFO `sc_interface` bound to component ports.

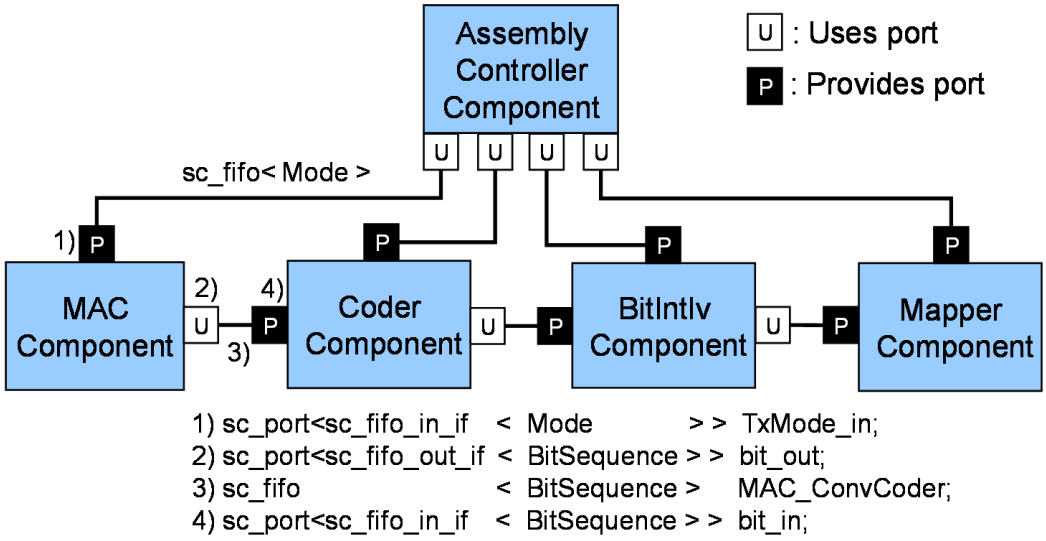


Figure 8.7: Functional modeling with Kahn Process Network (KPN)

SystemC Stub/Skeleton and OCP TLM connectors

First, the controller configures the transmission mode of each business component by invoking the `setTxMode` operation. Then, a bit stream is alternatively sent and received to model the half-duplex transmission from the MAC component up to the Mapper component by calling the `pushBit` operation on the component ports. The `setTxMode` operation is a twoway synchronous operation, since the component must be configured before starting its processing, while the `pushBit` operation is a oneway asynchronous operation which "sends and forgets" data. The bit stream is modelled as a `BitSequence` and sent as one burst in parameter of the `pushBit` operation. The `BitSequence` data type is an implementation of the OMG IDL sequence type for bits. Synchronous and asynchronous method invocations are modeled in SystemC by dedicated connectors as illustrated in listing 7.1. For local invocations between software components deployed on GPP and DSP, functional invocations are based on *Interface Method Call*s (**IMCs**) [GLMS02] on SystemC channels. For local invocations between hardware components deployed on FPGA, synchronous and asynchronous invocations are respectively modeled as non-posted and posted writes in OCP TL2 [OI07] [GBSV08]. Remote invocations between hardware and software business components are transparently mediated by connectors fragments i.e stub/skeleton. The `BitSequence` parameter is marshalled and unmarshalled by the `PushBit` connector fragment as a sequence of words for transmission on 32bit OCP TL3 channels. The `Mode` parameter is marshalled and unmarshalled by the `Configure` connector fragment in a single word. The *push* and *pull* models were also implemented. In the push model, the client connector fragment acting as stub for the MAC

component writes each word of the marshalled bit sequence on the server connector fragment acting as skeleton for the Coder component. In the pull model, the BitInterleaver stub generates an interruption via the `IRQ` signal depicted in figure 8.8, then the Mapper skeleton acknowledges the interruption via the `ACK` signal, reads the length of the word sequence and finally each word of the sequence. The pull model models the slave role of the FPGA on the data bus and the master role of the GPP. The invocation messages containing the marshalled operation parameters are encapsulated in the `MData` field of OCP TLM messages, while the operations are identified in the `MAddr` field.

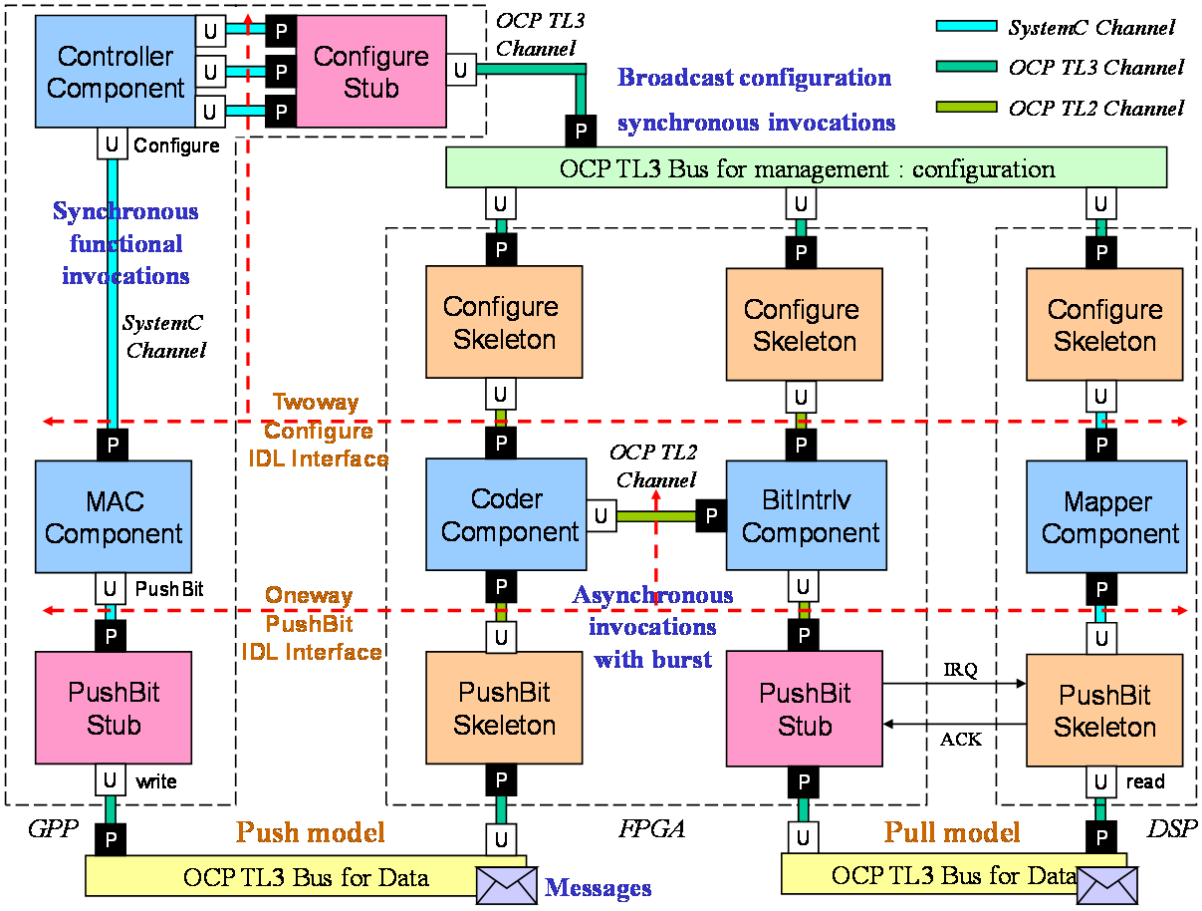


Figure 8.8: Multi-Level modeling using communication channels at different abstraction levels

Hardware SCA components and OCP at Register Transfer Level (RTL)

As depicted in figure 8.9, the Coder and Interleaver SystemC components were refined as hardware SCA components in VHDL using OCP RTL as a concrete mapping of our interface family. The Coder and Interleaver IP cores were reused with their custom interfaces from the existing OFDM modem. They were encapsulated by hardware stubs/skeletons resulting from the application of the IDL-to-VHDL mapping on OCP interfaces [GBSV08]. These stubs/skeletons as connector fragments. There is a hardware skeleton for each provided IDL interface and a hardware stub for each required IDL interface. The *equivalent* IDL interfaces implemented by the hardware SCA components are the following:

```
interface Control { // from SCA Resource interface
```

```
    void initialize();
    void start();
    void stop();
    Object getPort(in string portname);
};
interface CC_Config {
 writeonly attribute short redundancy;
};
interface BI_Config {
 void configure(in short mode, in short nbOfLines, in short nbOfColumns,
 in short TotalNbOfBits);
};
interface Stream {
    typedef sequence<bit> BitStream;
 oneway void pushSlot(in BitStream strm);
};
interface ConvCoder: Control, CC_Config, Stream {};
interface BitInterlv: Control, BI_Config, Stream {};
```

The `Control` interface is a subset of the `Resource` interface and is given for illustrative purposes. In reality, SCA components should provide the `Resource` interface and implement it. Default implementation of `Resource` interface operations could be automatically introduced in skeletons even with a empty implementation for instance for the `initialize`, `start`, `stop`, `runTest` operation. The `initialize` operation is interpreted as a software reset e.g. a synchronous reset for HW components. The `start`/`stop` operations may be forwarded to hardware components or may be interpreted as a clock enable.

As opposed to [JTR05] and in accordance with [HP06], we give designers the choice to implement or not the `getPort` operation to retrieve the physical or logical address of a SCA component. The `portname` string parameter may be transparently mapped to a numeric identifier using a look-up table in the generated stubs. The same information encoded differently may thus be transferred, but more efficiently to save communication bandwidth and latency. The SCA `getPort` operation and its CCM counterpart `getFacet` are normally required by the deployment framework to establish interconnections between component ports *at runtime*. For static hardware deployments, such operations are not necessary as the physical address of hardware components are predefined in the system memory map and the interconnections between them and the on-chip bus are fixed. For instance, in the AMBA bus implementation provided with the open source Leon[75] soft-processor-based SoC, the physical address of each peripheral devices is stored in a ROM used by the AMBA bus to perform address decoding.

However if we take into account hardware dynamic partial reconfiguration, the address of the reconfiguration zone or slot into which a hardware component is instantiated is not a priori known in advance as it depends on the number and location of the previously loaded components. The *getPort* operation may thus retrieve the hardware component port identifier and pass these information to all hardware and/or software components that need to communicate with this component. The port identifier represents a virtual address which can be mapped to a physical address. During deployment, the component framework could use a hardware reconfiguration service as proposed in [DRB+07] to determine an available reconfiguration zone within a reconfiguration slot pool, similar to the RT-CORBA thread pool, and load the partial bitstream of the hardware component in this slot. The address of the control port required by

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

the component model e.g. the SCA `Resource` or `CCMObject` interface port would be registered into a centralized or distributed Naming Service. This hardware name service can be implemented with few hardware resources as a *Content-Addressable Memory* (**CAM**) also called associative memory [Chu06]. While a RAM or a register file generates a data from an input address, a CAM generates an address, index or key from the input data. The generated address may be then used to access the desired data from a RAM or a register file. As opposed to the sequential search in a RAM, a CAM performs parallel searches across all addresses. CAM are used for high-speed searches such as network routing, cache memory management, and data compression. A CAM can be efficiently implemented in hardware using FPGA/ASIC libraries e.g. the Altera *altcam* megafunction or the Xilinx CAM LogiCORE using Core Generator. Using CAMs, a hardware/software component framework could obtain the address of each component port to invoke their provided services e.g. configuration, data stream, etc.

The `CC_Config` interface provides a *write-only* attribute called `redundancy` that determines the encoding rate. The new *write-only* keyword we proposed allows the inference of a setter operation without a *public* getter method. Method implementations can still access the component property using a *private* getter method.

Instead of using the generic `configure` operation of the `Resource` interface, we define a business configuration operation called `configure` in the `BI_Config` interface. This operation takes as input parameters the application mode (mode 1 or 2), the number of lines and columns for the interleavement and the total number of bits to transfer to save the hardware multiplication of both previous numbers.

The `Stream` interface defines a new type called `BitStream` representing a sequence of bits and a *oneway* `pushSlot` operation to transfer the data of a transmission slot. The oneway property indicates than no reply is required. Alternatively, the declaration of the `Stream` interface could be replaced by a `Stream eventtype` declaration if one-to-many communications are required in addition to asynchronous semantics.

```
eventtype Stream {
    public BitStream slot;
};
```

The `ConvCoder` and `BitIntrlv` interfaces specify the encoder and interleaver component objects. These interfaces are given for illustrative purposes, because they are not required to describe a SCA component. Indeed, the SCA does not rely on the `component` construct of CORBA IDL 3, but on its own XML-based ADL that describes the provided and used IDL interfaces using XML markups.

The equivalent IDL3 declaration for CCM using the SCA Resource interface would be:

```
component ConvCoder {
    provides     Control   theControlPort;
    provides     CC_Config theConfigPort;
    provides     Stream    inputStreamPort;
    uses         Stream    outputStreamPort;
    // alternatively with events:
    // consumes  Stream    inputStreamPort;
    // publishes Stream    outputStreamPort;
};
component BitIntrlv {
    provides     Control   theControlPort;
    provides     BI_Config theConfigPort;
    provides     Stream    theStreamPort;
```

```
        // alternatively with events:
        // consumes  Stream    inputStreamPort;
        // publishes Stream    outputStreamPort;
    };
```

Using IDL3, component dependencies are clearly expressed for *human* designers, whereas XML is more a machine-language than a human-readable language. However, both SCA and CCM use XML-based ADLs to describe component interconnections. A textual domain specific language would be more user-friendly as proposed in the ADL literature [MT00]. Moreover well-defined interfaces can be reused across component declarations as the `Config` and `Stream` interfaces in the `ConvCoder` and `BitIntrlv` components.

As illustrated on the top and bottom of figure 8.9, the `Control` and `Config` ports implement two write-only OCP subsets for the request and reply messages. The control and configuration message payloads are transferred through these OCP ports. On the left and right of the figure 8.9, two input and output stream data flow ports implement the FIFO write-only OCP subset without the `SCmdAccept` signal since no flow control is required.

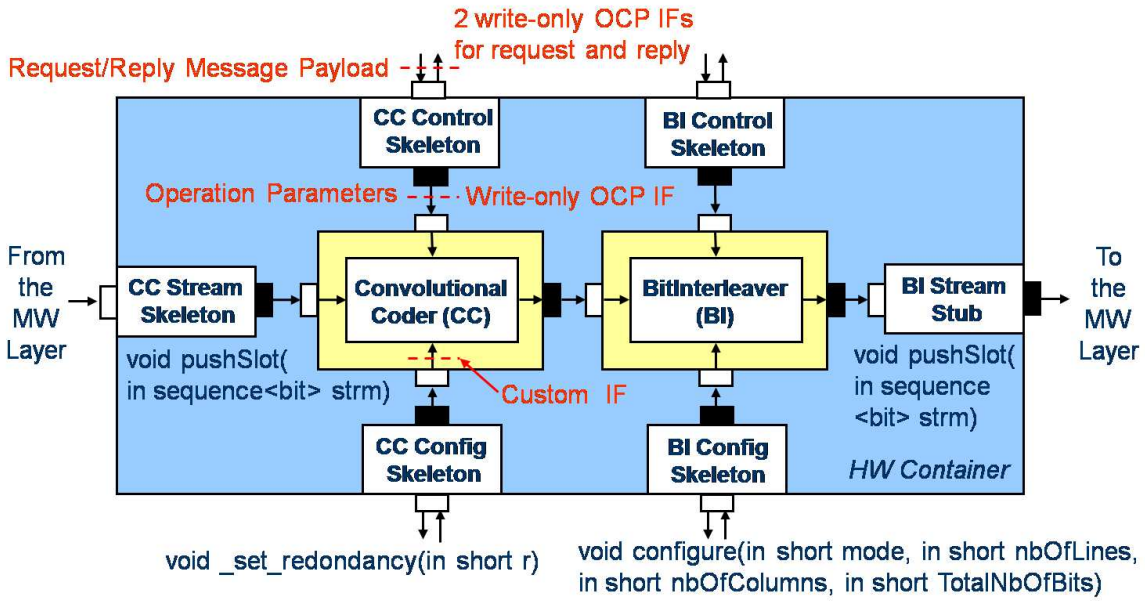


Figure 8.9: Hardware SCA Encoder and Interleaver components in Application Mode 1

Each component port interface is configured with generics which are defined in a VHDL package that contains all OCP profile parameters.

The component assembly depicted in figure 8.9 represents the configuration in the mode 1 in which the encoder and interleaver are deployed in FPGA, while the figure 8.10 corresponds to the configuration of the mode 2 where only the encoder is mapped in FPGA and the interleaver in software.

The introduction of hardware stubs/skeletons acting as proxies at distribution boundaries is transparent for hardware components. Indeed the hardware interfaces of the Encoder component are not modified between the mode 1 and 2 depicted in figures 8.9 and 8.10.

The hardware stubs/skeletons make the adaptation between the RTL interfaces from the IDL interfaces and the custom RTL interfaces of the existing IPs. This hardware reuse approach is strictly similar to the software CORBA wrappers developed to reuse legacy code with automatically generated stubs/skeletons.

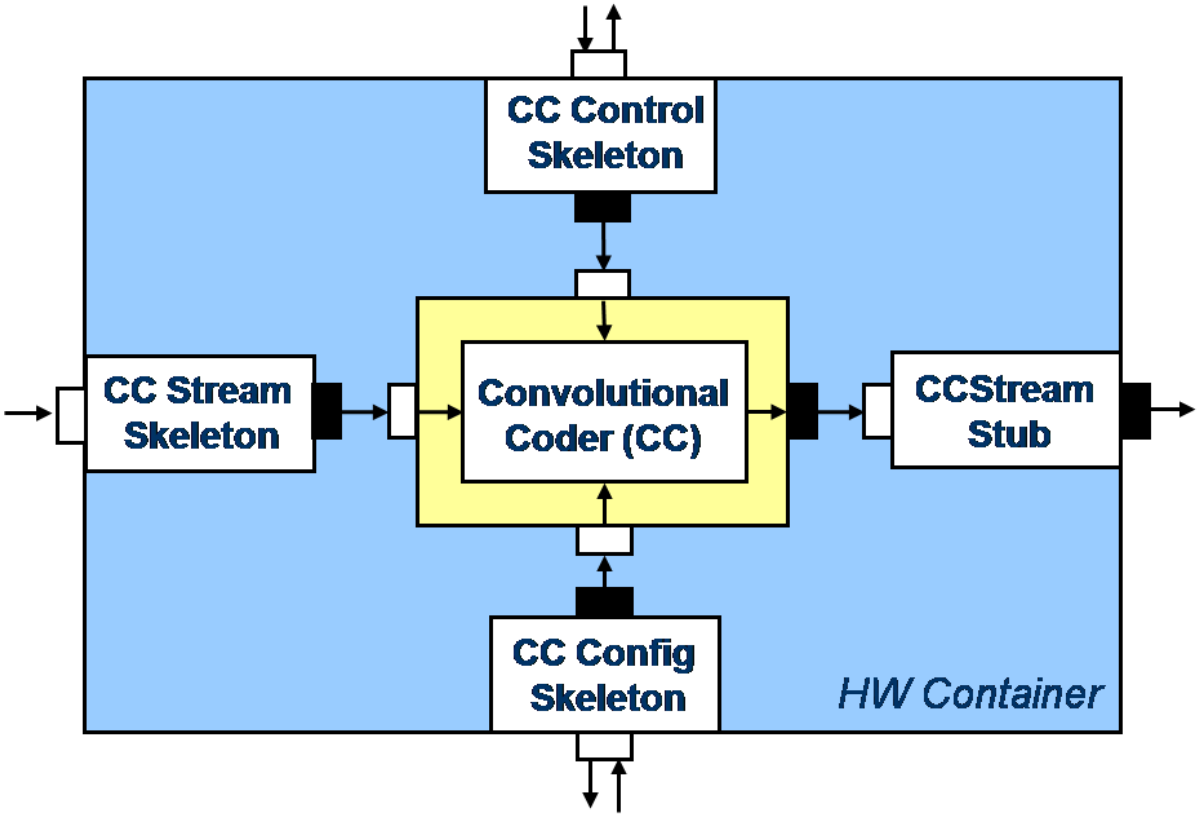


Figure 8.10: Hardware SCA Encoder component in Application Mode 2

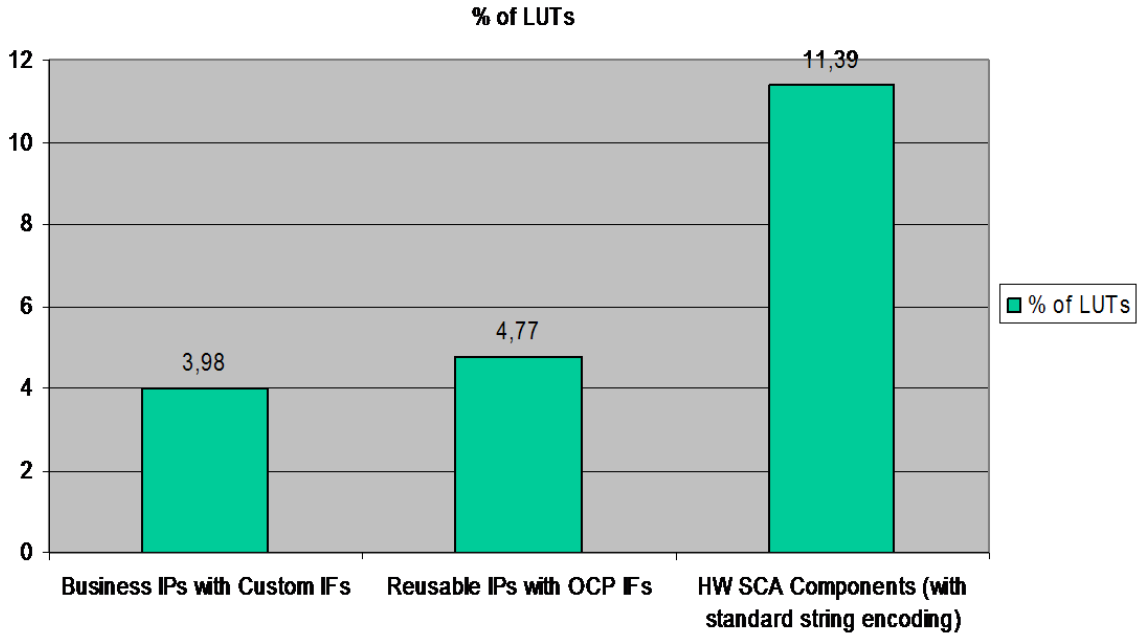


Figure 8.11: Synthesis results for Xilinx VirtexIV FPGAs

Figure 8.11 represents the synthesis results in number of Look-Up Tables (LUTs) for the Xilinx VirtexIV FPGAs. The initial business IP cores with their custom interfaces use 3.98% of the total number of LUTs available in the chosen Virtex IV. The encapsulation of these IP cores with OCP interfaces increases the LUT occupation by 17%.

These results show that the wrapping required by the socket-based approach is quite negligible as a socket interface such as OCP allows capturing only the communication needs of each IP. Even if the IP cores used for this example are relatively simple, the wrapping consists in a direct mapping of signal names between the custom and OCP interfaces. In comparison, the encapsulation of the OCP components as hardware SCA Resource components shows an increase of 58% without any optimization. This overhead is due to the standard encoding of IDL string e.g. in the `portName` parameter if the `getPort` operation as a sequence of ASCII characters. We deliberately implemented this encoding to quantitatively evaluate the overhead of commercial hardware CORBA middlewares. This encoding is obviously not network and hardware efficient. It is simply not appropriate for embedded systems. .

In addition, we successfully validated the SystemC/VHDL co-simulation of a transactional connector which converts the software invocation of the `pushBit` operation from a functional Stream SystemC interface to the hardware invocation of this operation on an OCP RTL component port using the ModelSim RTL simulator. ModelSim generates code to perform this co-simulation.

8.3.3 Hardware Middleware Implementations

In this section, we present three kinds of hardware middlewares. The first middleware is a hardware object-oriented middleware with a CORBA GIOP 1.0 protocol personality. It implements in hardware the encoding and decoding of the `Request` and `Reply` messages of CORBA GIOP 1.0. The second middleware is a hardware message-oriented middleware with a SCA MHAL protocol personality. It

supports the hardware encoding and decoding of SCA MHAL header messages and dispatches the same message payloads than the GIOP messages to provide transparent communications to application components. While the two first middlewares are based on message passing, the last hardware middleware implementation is based on memory-mapping and allows one to evaluate the overhead of the two previous approaches. We consider these middleware implementations as three configurations of the same middleware architecture framework which revisits bus and NoC concepts for hardware middlewares.

All hardware middleware implementations have been deployed on the Xilinx ML405 board. The hardcore PowerPC embedded processor within the VirtexIV FPGA executes a software testbench, which writes request messages in the middleware FIFOs across the Xilinx OPB bus and read reply messages to compare them with reference results. Note that the on-chip processor is only used to debug and validate the middleware prototypes. In a real implementation, instead of sending both message headers and payloads through the on-chip bus, only message payloads would be sent to hardware IP components to reduce bandwidth usage. In this case, addressing information in message headers would be mapped to bus addresses. In fact, this experiment simulates the reception of GIOP messages from outside the FPGA. The final combination of the hardware/software application and platform components is defined by the system engineers according to performance requirements and design trade-offs. For instance, memory mapping could be chosen for local communications within an FPGA, while message passing using GIOP or MHAL could be chosen for external communications outside the FPGA. GIOP or MHAL messages could be received from external high-speed communication links e.g. from the buffers of an Ethernet MAC IP core or a RapidIO serial link without the need of a embedded processor.

In all these implementations, OCP interfaces have been used in each middleware layer to evaluate the hardware socket-based approach which is close to the interface-centric and component-oriented middleware approach. We think that OCP provides a generic communication model and a common interface language for hardware designers, who only need to know one interface protocol. However, we also remark in our experiments that OCP and even its profiles are too large in scope and that our middleware implementation only requires OCP subsets such as write-only FIFO interfaces. We consider that standardized OCP profiles may address middleware interfaces at transport level for on-chip and off-chip buses, while the OCP subsets cited as examples in the OCP specification [OI06] may address application interfaces for business IP core components.

Moreover, OCP has some technical limitations for a standard mapping since it remains a bus-oriented interface protocol. Indeed, an OCP address is a byte address aligned on an OCP word whose size refers to the width of the OCP data bus [OI06]. For instance, if the data bus width is 32 bits, the address must be incremented by 4 since there are 4 byte addresses for each byte of a 32 bit word. Even if this correlation between the address and data buses is justified for physical addresses for bus transfers, this is a drawback when the address bus is used to carry a logical address representing an operation identifier. This logical address should be incremented by one to save area and remove useless address signals. Of course, other OCP signals could be used like `MReqInfo`, but this is counterintuitive for hardware designers, whereas the address signal provides a better mapping of semantics between RMI and OCP.

Like the address signal, the burst length signal designates a number of words. For instance, if we would like to transfer an IDL array of 16 `octet`s as a burst on a 32bit data bus, the burst length would be 4 words. If we would like to transfer an IDL array of 16 `octet`s as a burst on a 8bit data bus, the burst length would be 16 words. For instance, if we would like to transfer an IDL sequence of 4 `long`s as a burst, the burst length would be 4 words on a 32bit data bus and 8 words on a 16bit data bus. Hence, the address and burst length are not related to the payload but to the data bus width. This is required for bus transfers at the transport level, but useless for method invocations at application level.

The OCP interfaces we use help us to define our family of interfaces, which must be independent from any existing *proprietary* interface protocol to be used in a standard IDL-to-RTL mapping for the OMG.

Moreover, defining a standard OMG mapping based on OCP requires that commercial and open-source ORB implementers buy membership to OCP-IP to validate their implementation of the OCP protocol. In any case, these considerations are out-of-the scope of this thesis. In fine, these OCP interfaces can be considered as the mapping of our independent interfaces on the particular OCP protocol. Other hardware interface mappings could be defined for the deprecated VCI protocol or other interface protocols in the future.

GIOP RMI-oriented protocol personality

With the help of a trainee I supervised, we prototyped a GIOP message decoder/encoder to evaluate the overhead of an hardware implementation of the GIOP protocol as proposed by the two leading embedded ORB vendors PrismTech [HP06] and OIS [OIS08].

Thanks to the open-source network analyzer Wireshark[76] (formerly named Ethereal), I recorded the GIOP messages exchanged between the software SCA components. Then these messages have been sent to the hardware SCA components by a software testbench executed on the PowerPC embedded processor.

Figure 8.12 depicts the hardware implementation of an Object Request Broker (ORB) with a CORBA GIOP personality. We tried to apply the same design patterns and to identify the same architectural elements than those typically found in software middleware implementations. The architecture of this hardware ORB can be divided into five layers: the application layer, the presentation layer, the dispatching layer, the messaging layer and the transport layer.

The **application layer** contains the hardware application components i.e. the Convolutional Encoder and the BitInterleaver in our experiments. These components are SCA Resources. They implement the most relevant operations required by the `Resource` interface in hardware. The other `Resource` operations can be transparently implemented in software stubs.

The **presentation layer** consists of the manually generated hardware stubs and skeletons. They act as invocation adapters [PRP05] which perform the decoding and encoding of GIOP message payload. Stubs decode message payloads encoded with the Common Data Representation (CDR) encoding rules into operation parameters which are stored in registers. They can either forward the invocation to the application components via a write-only interface (`push` interface) to inform them that they can read parameter values, or the components can continuously read their values via a signal (`data` interface). The collection of stubs/skeletons that isolate the application components from their operating environment can be viewed as a hardware *container* which provides activation (e.g writes), synchronization (e.g. clock) and storage (e.g. memories) services.

The **dispatching layer** contains a generic request router and a reply arbiter. The *request router* acts as an object adapter, which dispatches incoming invocations to the appropriate skeleton according to the operation identifier transferred in the GIOP request message header. This router is similar to a bus address decoder or a NoC router. It decodes the internal logical address of the request and routes it to the appropriate skeleton. The router shares the same design trade-offs as in buses and NoCs. This operation name is a string of ASCII characters decoded by an FSM which compares it with the manually generated arrays declared in a VHDL package. Due to the small number of component ports or skeletons, this router is implemented as a 1-to-N multiplexer which is not scalable. To increase its scalability, the request router can have a distributed implementation e.g. as a tree of 1-to-2 multiplexer. The *reply arbiter* selects a skeleton according to their priority among those which want to send a reply message. In this test case, only the `Control` interface skeletons of the Encoder and Interleaver components return a reply to the `getPort` operation.

The **messaging layer** implements the GIOP protocol state machine. It contains a request decoder,

[76] http://www.wireshark.org

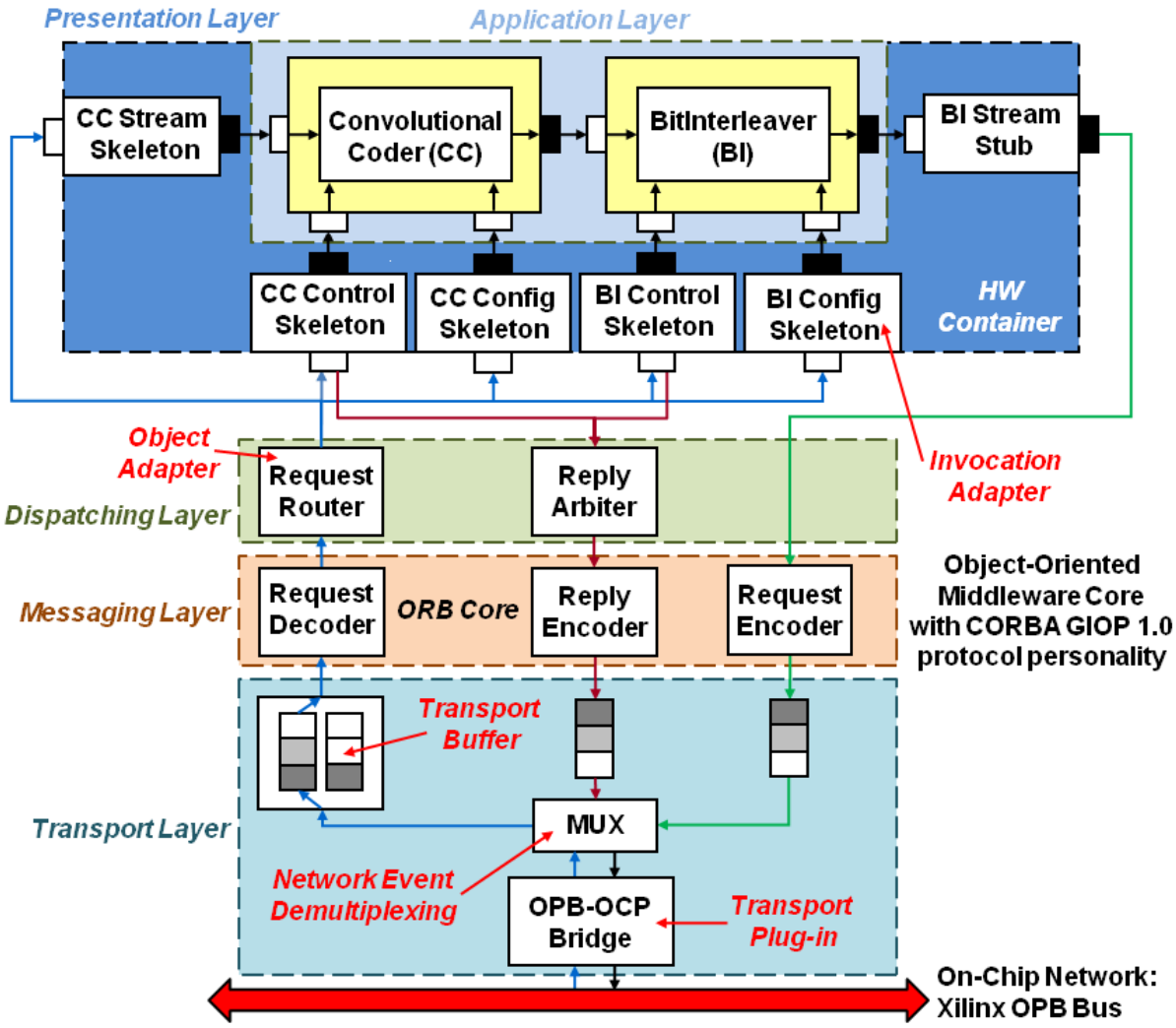


Figure 8.12: Hardware ORB with a GIOP 1.0 protocol personality

a reply encoder and a request encoder. The request decoder decodes GIOP request message of the `initialize`, `start`, `stop`, and `set_redundancy` and `configure` operations using a hierarchical FSM. A first FSM decodes the GIOP message header, then a second FSM continues the decoding depending on the message type (`Request` or `Reply`). This modular FSM reduces the required hardware resources by only implementing the needed types of messages and can be used for an optimized code generation. For instance, the following types of GIOP messages were not implemented: `LocateRequest`, `LocateReply`, `CancelRequest`, `MessageError` and `CloseConnection`. The request decoder writes on its output port the fields extracted from the message header. These fields are then used by the following ORB modules, the request router and the skeletons, to perform their processing based on the message body. To reduce the footprint of IDL `string`s while decoding standard GIOP messages, this module maps sequences of ASCII characters in the header into numerical identifiers. Such a mapping may be performed by IDL compilers inside the generated stubs at the expense of GIOP interoperability. The request encoder integrates a generic scheduler like in software middlewares to choose between the two available reception buffers. There are two reception buffers acting as two virtual input channels. The scheduling policy is the following: when a request is available, the request decoder selects a buffer according to its priority and reads a predefined number of message words per buffer to be decoded as long as there are enough data available otherwise it chooses another channel buffer. These virtual channels allow the efficient sharing of the available bandwidth between the virtual channels. A priority is assigned to each FIFO to avoid *priority inversion*. The decoding is performed separately for each virtual channel. The FSM decoding state is saved and restored during virtual channel switch to allow the decoder to alternatively decode one message from each FIFO buffer. The GIOP request decoder is thus generic and independent from the number of transport buffers. The request and reply encoders perform the reverse steps of the request decoder. They produce the message header based on the fields it receives from the stubs/skeletons. The *reply encoder* prepends a GIOP message header and a reply header to the `pushPacket` message payload provided by the `Control` skeletons and writes the resulting message to a transport buffer. The *request encoder* prepends a GIOP message header and a request header to the `getPort` message payload provided by the `Stream` stubs and writes the resulting message to a transport buffer. The request encoder uses the numerical identifier generated by the request decoder and received from the stubs to produce the IDL strings contained in the outcoming request message.

The **transport layer** includes transport buffers used to send and receive GIOP messages across the network, here the Xilinx OPB bus for validation purpose. It contains a transport plug-in, which is a an OPB-to-OCP bridge. The transport buffers are *active* FIFOs. The receiving buffers send write requests to the request decoder when data from a message are received. The request decoder may immediately accept the write request or delay its response to process a message from another buffer. In the same way, the sending buffers send read requests to the reply and request encoders which are free to accept or not. The active FIFOs reduce the latency required between a read or write request from the messaging layer and the response from passive FIFOs. In the current implementation, there are two input buffers for incoming request messages, an output buffer for reply messages and an output buffer for outcoming request messages. The number and depth of each buffer is chosen according to our particular scenario and may be adapted for other use cases. A multiplexer is used to demultiplex incoming messages to the input buffers and outcoming messages from the output buffers.

The transport buffers were implemented as FIFO memories, but they also may be implemented as addressable memories using RAMs to bypass useless data from GIOP messages such as padding, requesting principal and service context list during their decoding instead of reading these data from the FIFO buffer.

GIOP message fields	OCP signals
object key incl. object id	MAddr(31 : 8)
operation name	MAddr(7 : 0)
request id	MReqInfo(31 : 0)
body length	MReqInfo(47 : 32)
response expected	MReqInfo(48 : 48)
little endian	MReqInfo(49 : 49)

Table 8.1: Mapping of semantics between GIOP message fields and OCP signals

Mapping GIOP message fields to OCP signals As depicted in table 8.1, the message fields written by the request decoder and read by the request and reply encoder have been mapped onto OCP signals to provide these information in one clock cycle at the expense of the routing resources.

The `request id` field is generated by the request encoder. The body length parameter is used to transfer information between layers and does not directly represent a GIOP message field.

Limitations of GIOP As previously discussed in the section concerning middlewares, the CORBA General Inter-ORB Protocol (GIOP) was not designed for highly embedded and hard real-time application domains. For small messages, a GIOP message contains more information relative to the RMI protocol itself than to the operation parameters. For instance, a GIOP request message for the `void start()` operation used by the SCA CF requires at least 48 bytes[77] i.e. 12 writes on an on-chip or/and off-chip 32bit bus, whereas the same functionality can be achieved by a single write in the control register of an IP core.

The footprint and relative complexity of the GIOP protocol requires more development time and hardware resources. Indeed, the GIOP decoder requires 21% of the total number of LUTs available in the VirtexIV FPGA.

Modem Hardware Abstraction Layer (MHAL) message-oriented protocol personality

The overhead of the GIOP protocol for real-time and embedded platforms necessitates lightweight messaging protocols to more efficiently use network and area resources. As our work took place in the SCA context, we naturally choose the SCA MHAL protocol [JTR07], which was recently specified by the SCA in order to evaluate its performances compared to the GIOP protocol. The MHAL is a low-level message passing APIs to abstract hardware waveform components. The GIOP and MHAL protocols have different natures. The GIOP protocol is a Remote Method Invocation Protocol, while the MHAL protocol may be considered as a low-level message passing protocol.

The MHAL specification [JTR07] only defines a message header which is composed of a logical address called *Logical Destination* (**LD**) and a 16 bit message size, while there is no standard encoding for the payload apart from the MHAL RF Chain Coordinator API. The MHAL RF Chain Coordinator API specifies sink and source functions to control the RF resources of a radio channel. The message payloads for the RF Chain Coordinator consists of one or several commands which includes a command symbol and command parameters. The format, unit and valid value range of each parameter is also specified. The command symbols are predefined integer constants like the LD, while parameters are a variable integer multiple of 16 bits. Hence, there is no delimiters or padding required between commands. For MHAL message sources which need a response from the RF Chain Coordinator, the logical

[77] with an object key of one byte

GIOP concepts	MHAL concepts
Object Key incl. Object ID	Logical Destination (LD)
Operation name	Command symbol
Operation parameter	Command parameter
Request ID	Reply ID

Table 8.2: Mapping of semantics between GIOP and MHAL concepts

Index	Hex Data		Comments
0	14	00	Logical Destination (LD) = 0x14
2	02	00	Message size = 2 bytes
4	01	00	Operation ID = "start"

Table 8.3: Example of a MHAL message for the `void start()` operation in little endian

destination of the source is transferred as command parameter like in the `RFC_ConnectTxBlock` and `RFC_TxBusyStatusRequest` sink functions which respectively inform the data source whether the transmission (TX) is blocked or busy [JTR07].

The sending of a source LD as parameter allows the implementation of GIOP like request-reply protocols and publish-subscribe communications using the push-only MHAL communication service. We consider the MHAL as an Environment-Specific Inter-ORB Protocol (ESIOP) and propose a mapping of semantics between GIOP and MHAL concepts represented in figure 8.2.

Based in this mapping, we give an example of a MHAL message for the `void start()` operation from the SCA `Resource` interface in table 8.3. This message only requires 6 bytes compared to the 48 bytes requires for the same operation using the GIOP protocol.

The architecture for the encoding and decoding of MHAL messages is similar to the one of GIOP messages. We take the same architecture template. The encoding and decoding of the GIOP message header is replaced by those of the MHAL message header using the same message payload.

However, the MHAL protocol provides a push-only communications service in which data can be pushed i.e. written to a destination but cannot be pulled or read from this destination [JTR07], and is not a request-reply protocol like GIOP.

Hence, there is no distinction between the reply and request encoders of GIOP. The MHAL implementation is composed of a MHAL header decoder called *MHAL receiver* and a MHAL header encoder called *MHAL sender*.

The MHAL encoder and decoder implementation is simpler and uses less hardware resources than those in GIOP for two main reasons. The MHAL FSM requires much less states than the GIOP protocol machine and the encoding and decoding of strings are no longer necessary. The interesting point is that the change of message protocol is transparent for the hardware application components and the hardware stubs/skeletons hosted in the hardware container.

Mapping of semantics between MHAL and OCP A MHAL message header contains the logical destination and the size of the message. The first 16 bits of the payload consist of the operation identifier and the following 16 bits include the sender identifier to return the reply. To be coherent with the semantics mapping between GIOP and OCP, the MHAL fields with a similar meaning than the GIOP fields have been mapped on the same OCP signals as shown in figure 8.4. Since the MHAL specification

MHAL message fields	OCP signals
logical destination (LD)	MAddr(30 downto 16)
operation id	MAddr(15 downto 0)
logical origin	MReqInfo(14 downto 0)
body length	MReqInfo(30 downto 15)

Table 8.4: Mapping of semantics between GIOP message fields and OCP signals

recommends that FPGA and DSP uses Little-Endian to avoid marshalling, the endianness indication of MHAL messages is not required as contrast to GIOP messages.

Memory-mapped middleware

To compare with the GIOP and MHAL message passing based communication model, a traditional memory-mapped based communication model has also been implemented as presented in figure 8.13. Each application component property and operation parameter has been assigned a logical address relative to the physical address of the memory-mapped middleware on the OPB bus. The same middleware architecture template has been used to ease this comparison. Due to the memory-mapped model, the messaging layer does not contain any message header encoding/decoding logic. The request router has been replaced by a classical address decoder which routes incoming data to the appropriate interface skeleton. The set of application component and their stubs/skeletons forming the hardware container have been reused without any modification, while the same message content is transferred with the values of operation parameters. Hence, changing the transport protocol is transparent for hardware components. The skeletons perform the decoding of the message payload dispatched by the address decoder and store the values of operation parameters in registers. The same reply arbiter has been reused to select a single reply among those returned by the skeletons. The outcoming data from the replies and requests are separately stored in FIFOs. A software testbench hosted on the PowerPC embedded processor consults the register status to be informed when FIFOs can be read and checks their content against the reference results. Without surprise, this traditional approach is the simplest to implement and require the least amount of hardware resources compared to message passing middlewares.

8.3.4 Results on Virtex IV after Place&Route

Figure 8.14 presents the synthesis results for Virtex IV FPGA after Place&Route. The percentage of the hardware resources required from Xilinx ML405 board are respectively depicted for memory mapping and message passing using SCA MHAL and CORBA GIOP 1.0. These results are expressed in percentage of Slice Flip Flop, LUTs for logic and LUTs in total. The LUTs in total include the LUTs for logic, Dual Port RAM and shift registers. All FIFOs have the same size to better compare the middleware implementations and require 55% of the embedded RAM blocks (RAMB16s). Moreover, the application and presentation layers are reused. The change among the middleware implementations concern the dispatching and messaging layers, and the number of FIFOs which is 2, 3 and 4 for memory mapping, MHAL and GIOP. Hence, these results mainly compares the amount of resources required by the middleware FSM. They have been obtained with default parameters without any optimization for timing or size. The figure 8.14 shows the increasing consumption of hardware resources. Each middleware implementation adds more functionalities with a message header for MHAL, then a message type header with an out-of-order request-reply protocol for GIOP, that is why more and more resources are necessary. The MHAL implementation requires x1.2 more resources than memory mapping in number

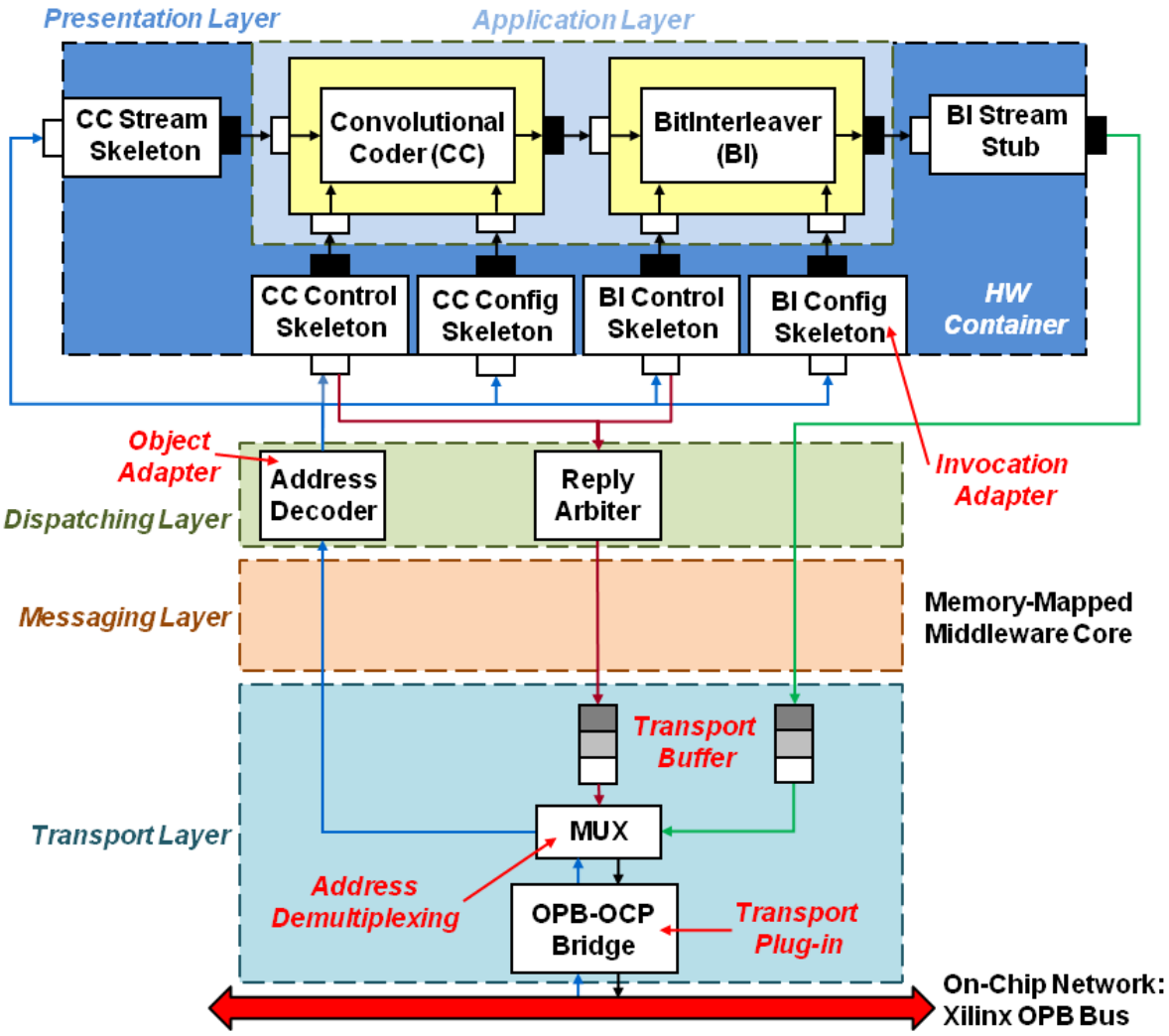


Figure 8.13: Memory-mapped middlewares

of slices, whereas GIOP needs x2.5. Moreover, GIOP consumes x2.15 resources in addition to MHAL. As previously mentioned, the overhead of GIOP is notably due to the relative complexity of its protocol and the decoding/encoding of data types like IDL `strings`.

As a conclusion, we recommend Memory Mapping for local communications within FPGAs and Message Passing for external communications outside FPGAs. As opposed to commercial hardware CORBA middlewares, we empirically demonstrated that the hardware implementation of GIOP requires much more hardware resources than a traditional memory mapping, while memory mapping can also be used to invoke the same functionalities. We also showed that MHAL messages can be considered as domain specific message protocol by mapping GIOP concepts to MHAL concepts. Such an ESIOP can be much more network and hardware efficient while being functionally equivalent

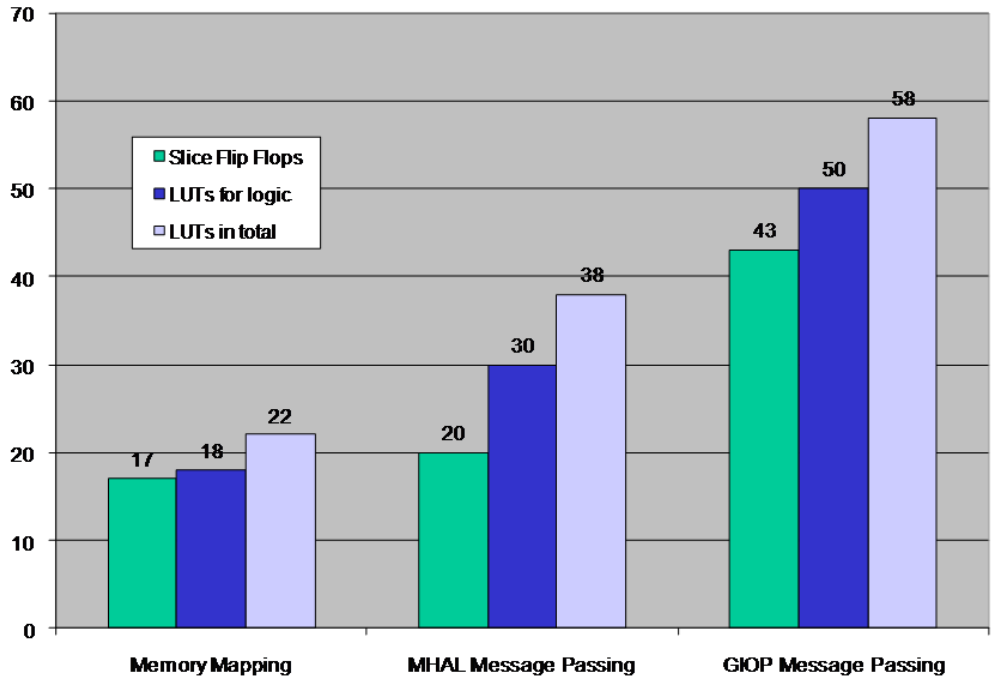


Figure 8.14: Results on Virtex IV after Place&Route

8.3.5 Results Analysis

The feasibility of the previously mentioned concepts have been demonstrated through the physical deployment of each of these types of middlewares on the Xilinx Virtex IV-based ML405 board. Different configurations have been tested:

- Both the convolutional encoder and the bitinterleaver in hardware with the GIOP implementation,
- the convolutional encoder in hardware and the bitinterleaver in software with the GIOP implementation,
- Both the convolutional encoder and the bitinterleaver in hardware with the MHAL implementation,
- Both the convolutional encoder and the bitinterleaver in hardware with the memory-mapped implementation.

All these different deployments have been transparent for the hardware SCA components thanks to the hardware stubs/skeletons acting as hardware proxies. The two first configurations demonstrate the transparent deployment of components in hardware and software. Moreover, the control of the dynamic and partial reconfiguration of a hardware module has been showed in [SMC+07] using the same board. Unfortunately, the integration of both approaches has not been performed for space limitations on the board and for time reasons. Nevertheless, we can envisage the dynamic partial reconfiguration of hardware SCA components under the control of a software SCA CoreFramework and their hardware/software migration according to some load conditions or waveform application modes.

8.4 Conclusion

In this chapter, we presented two modem applications on which we built our propositions and the experiments on which we validated part of our contributions. We built an executable Platform Specific Model in SystemC TLM of part of the DiMITRI MPSoC. We recognize the unifying concepts of component, container and connector at transactional and register transfer levels. The C reference model of the classical design flow 2.6 implements the business logic of a RTL IP core componentw, while its services are called by the enclosing container through register and bitfield callback functions using the SystemC Modeling Library (SCML) [SCM08]. These callbacks functions implement the technical logic like storage and synchronization). According to the taxonomy of connectors in 6.5.7, an OCP TL2 channel represents an arbitrator/distributor connector, while the OCP TL2-to-PV transactor is a conversion connector. A SystemC component at Programmer View level allows an earlier validation of the hardware/software interface and software drivers, which are error-prone code to develop.

We illustrated our IDL-to-SystemC TLM mapping proposition through the encapsulation of invocation messages in OCP TLM connectors. We present the refinement of a business component from a functional model in C to a SystemC component at Programmer View (PV) level, the refinement of a transactional connector from an OCP TL2 to an AHB bus functional model, and the refinement at different abstraction levels from SystemC FIFO channels to OCP TLM and OCP RTL. The introduction of SystemC TLM stubs/skeletons acting as connector fragments at distribution boundaries was transparent for hardware/software SystemC components.

We illustrate our IDL-to-VHDL mapping proposition with two hardware SCA components, a convolutional encoder and a bit interleaver. The RTL stubs/skeletons make the adaptation between the RTL interfaces derived from IDL interfaces and the custom RTL interfaces of the existing IPs, in a similar way to the software CORBA wrappers developed to reuse legacy code with automatically generated stubs/skeletons. The collection of stubs/skeletons that isolate the application components from their operating environment can be viewed as a hardware *container*, which provides activation (e.g writes), synchronization (e.g. clock) and storage (e.g. memories) services. The interesting point is that the change of message protocol is transparent for the hardware application components and the hardware stubs/skeletons hosted in the hardware container.

We refined a Coder and Interleaver SystemC components to hardware SCA components in VHDL using OCP RTL as a concrete mapping of our interface family. Our results showed that the overhead of the socket-based approach is quite negligible (17%) compared to a socket interface such as OCP, which only captures the communication needs of each IP core of overhead. Hence in our case, the wrapping consists in a direct mapping of signal names between the custom and OCP interfaces. We evaluated the important overhead (58%) of the encapsulation of the OCP components as hardware SCA Resource components with the standard encoding of IDL string proposed by commercial hardware ORB vendors. Hence, portability and interoperability of waveform components have a cost without any optimization.

From an industrial viewpoint, the maintenance of multiple models at different abstraction levels is

difficult to manage without dedicated standards like SPIRIT IP-XACT. Standard language mappings from OMG IDL/UML to the standard TLM interfaces may help to generate transactional connectors from one abstraction level to another, if not already in a library. Moreover, SystemC-based design flows require to train (dedicated) HW/SW engineers, to buy commercial tools like CoWare Platform Architect or to develop custom tools for entreprises, which reach a critical mass of complex projects like STMicroelectronics. For instance, hardware/software engineers at Thales Communications generally prefer C rather than SystemC : 1) to increase simulation speed via native execution (w/o ISSs) and direct plaform integrations, 2) to avoid learning another language, and 3) in reasons of the conceptual gap introduced by SystemC and its maturity. Since Thales products have long life cycles, SystemC waveform components and SDR platform models could be a good investment. Moreover, an MDE/MDA approach may allow one to define future mappings to new Platform Specific Models as the technology rapidly evolves. Even if HLS tools offer a direct refinement from C to HDL sometimes at the expense of some code rewriting, we argue that SystemC components may still be required since they provide higher simulation speed than RTL components.

Finally, three different configurations of the proposed hardware middleware architecture framework were implemented using memory mapping and message passing. We showed how a dedicated message passing protocol such as MHAL is more hardware efficient than a general purpose message protocol like CORBA GIOP. We recommend memory mapping for local communications within an FPGA, and message passing using GIOP or MHAL for external communications outside the FPGA.

Chapter 9

Conclusions and Perspectives

Contents

In this section, we recap the problems addressed in this work and summarize our contributions. We also present some limitations of this PhD thesis and draw some perspectives.

9.1 Problems

The SCA specifies an Operating Environment composed of a Core Framework, a CORBA-compliant middleware and a POSIX-compliant OS. The Core Framework is a set of UML and CORBA interfaces to load, deploy and run waveform applications. This OE targets GPPs, but not highly-constrained processing units such as DSPs, FPGAs and ASICs. The goal of this thesis was to apply some successful software engineering concepts used in the SCA down to hardware in FPGA to provide a common and unified approach of specification, design and implementation of hardware/software component-based SDR applications. Moreover, the main requirements of the SCA are portability, interoperability, reuse and reconfigurability of SCA components. Hence, we had also to address these requirements for hardware components in HDLs such as VHDL and SystemC RTL, and transactional components in SystemC TLM to raise the simulation speed. In addition, our second objective was to apply the Lightweight CORBA Component Model to VHDL/SystemC component to support another common hardware/software component model. The ultimate objective of this work is to share the same concepts between hardware and software engineering to define a common component-oriented system architecture and to raise the abstraction level of hardware components.

9.2 Synthesis

Usually, the integration of hardware modules relies on low-level hardware/software interfaces, which consists of proprietary bus interfaces, memory-mapped registers and interrupts to synchronize hardware/software modules. As a result, this integration is tedious and error-prone, and the reuse of HW/SW

modules is limited. Co-design methodologies have been developed to bridge the conceptual gap between hardware design and software design. To provide a common hardware/software design approach, the concepts of the object model has been applied to hardware design. In the object-oriented approach, an object is a self-contained and identifiable entity, which encloses its own state and behavior in a method-based interface, and communicates only through message passing. To apply the object model down to synthesizable hardware, the previous works propose a mapping between a method-based interface to a signal-based interface. Hardware object-oriented design allows to raise the abstraction level of hardware module interfaces, to reduce the conceptual and abstraction gap between interface specification and implementation, and to enforce a clean separation between computation and communication through remote method invocation. Existing works propose a mapping, which represents one solution in the mapping exploration space. Moreover, the object-oriented approach has some limitations: only one provided interface (the most derived), implicit dependencies, and no clear separation of concerns between functional and non functional logic like communication and deployment. The software component-oriented approach is an evolution from the object-oriented approach.

In software architecture, the component-oriented approach relies on four main standard models: **component model**s defining component structure and behavior, **connector model**s to specify component interactions, **container model** to mediate access to middleware services, and **packaging and deployment model**s to deploy distributed component-based applications.

A software component provides and requires services through multiple interfaces that allow an explicit specification of external dependencies and a separation of concerns between functional interfaces for business logic and non-functional interfaces for technical logic e.g. regarding configuration, deployment and activation, packaging, deployment (i.e. installation, connection), life cycle (i.e. initialization, configuration and activation), administration (e.g. monitoring and reconfiguration). A software component has logical ports that provide and require properties, methods and events. A software connector is a component which binds the required and provided ports of two or more components.

In the hardware component models, a hardware component has input, output, input/output RTL ports that provide and require one or several signal-based interfaces. The connection model of hardware components is based on standard bus interfaces and communication protocols. An emerging packaging model for hardware components is the IP-XACT specification, which defines XML schema files to describe the characteristics of IP core components such as hardware interfaces, registers and interconnection. The deployment model of hardware IP core components can be either statically deployed on an FPGA at reset-time or dynamically deployed at run-time.

To support the systematic refinement of object-oriented component interfaces in CORBA IDL or UML to system, software, and hardware component interfaces in C/C++, SystemC and VHDL, we propose to define new language mappings from CORBA IDL to SystemC TLM/RTL and VHDL.

9.3 Contributions

We investigate the hardware implementation of abstract object-oriented components. While most of the works on the application of the software concepts to hardware design are based on an object-oriented approach, we consider in contrast the application of the component-oriented software architecture. We argue that the component-oriented approach provides better unifying concepts for hardware/software embedded systems, than the object-oriented approach. We provide a state of the art on the application of the object, component and middleware concepts to hardware engineering.

Concerning SDR design methodology, we proposed to leverage the conceptual SDR design methodology based on MDA and the component/container approach, which existed at Thales Communications in 2005, with SystemC Transaction Level Modeling (TLM) to simulate the platform independent model

of waveform applications, and the platform specific models of waveform applications and SDR platforms. The component/container approach allows to transparently integrate functional models in C as SystemC components in virtual platforms.

The mapping from IDL3 to functional SystemC extends the IDL-to-C++ mapping with additional mapping rules to fit specific SystemC language constructs. A SystemC component is a `sc_module` component with required `sc_export` ports and provided `sc_port` ports associated to `sc_interface` interfaces.

The mapping from IDL3 to SystemC at Transactional Level is based on transactors we call *transactional connectors*, which adapt the mapping of the user-defined interfaces in IDL to standard TLM interfaces.

For the mapping from IDL3 to VHDL and SystemC RTL, we formulate industrial, user and implementation requirements for a standard IDL3-to-HDL mapping. The most original requirements concern internal and external concurrency, connectors, hardware interface and protocol framework, blocking and non-blocking synchronization, aligned/unaligned data types encoding rules, Time or Space Division Multiplexing (TDM/SDM), Extensible Transport Framework (ETF), Extensible Message Framework (EMF), and a subset of IDL data types for bit-accurate types in annexe. We tried to address most of these requirements in the IDL-to-HDL mapping proposition.

In our mapping proposition, a hardware object-oriented component (e.g. SCA, UML, CCM) is represented by a hardware module, whose implementation (e.g. VHDL `architecture` may be selected by a hardware configuration e.g. VHDL `configuration`. An abstract component port is mapped to a logical hardware component port, which is a set of RTL input and output ports.

A hardware connector may represent non-functional properties such as synchronization and communication such as flow control, clock domain crossing, interaction semantics, and protocols of point-to-point buses, shared buses or NoCs.

A hardware container provide all remaining non-functional services such as lifecycle and persistence to store component state. For instance, the lifecycle service may manage component creation/destruction e.g. with dynamic partial reconfiguration, component initialization e.g. with a asynchronous or synchronous reset, parameters configuration with register banks, and activation/deactivation e.g. with a clock or enable signal.

The portability requirement of the SCA is addressed with IDL language mappings. We propose to generalize and leverage the existing mapping propositions with an explicit and fine mapping configuration using a named family of hardware interfaces with well-defined semantics. Our mapping approach gives the user a level of control over the mapped hardware interface never reached by all other fixed mapping approaches and is similar to what is available for hardware designers in high-level synthesis e.g. from C-like languages. Our IDL-to-HDL mapping is the allocation of hardware resources such as memory elements (register, FIFO, RAM/ROM), hardware signals and protocols to software abstractions such as component, port, interface, operation and parameters and connectors. We also propose an application of the SCA Operating Environment for FPGAs to improve the SCA specification for hardware SCA components.

Concerning the mapping implementation, we manually applied it to four quite representative examples in the SDR domain: FIR filter, Transceiver, Convolutional Encoder and BitInterleaver. Moreover, we implemented part of this mapping in an IDL3-to-VHDL compiler prototype using an standard OMG Model-Driven Architecture approach.

To address the interoperability requirement of the SCA, we proposed a hardware middleware architecture framework and evaluated the overhead of the standard GIOP message protocol. We demonstrated that invocations of hardware operations can be performed with less communication overhead by considering the SCA MHAL message format as an Environment Specific Inter-ORB Protocol (ESIOP) for remote message passing outside FPGAs and classical memory-mapping for local communications out-

side FPGAs.

Results of these works have served as inputs to european projects, mainly from the European Defence Agency (EDA) such as RECOPS, WINTSEC, SPICES and ESSOR, which conversely contributed to improve our proposition.

The concepts of component-oriented hardware/software architecture are general enough to be not restricted to an application domain such as SDR and component models such as SCA and LwCCM. All along this thesis, we tried to fill the gap between industrial constraints and requirements such as industry standards from the OMG (UML, CORBA, MDA), OCP-IP, and standardization (OMG, ESSOR), and the state-of-art of the research in the domains such as computer science and electrical engineering.

9.4 Limitations

The validation of such complex language mappings requires the application of these mappings to various applications to refine the mapping requirements and improve its implementation. We notably chose a fine component granularity e.g. by considering a filter as one CCM or SCA component, which may be unrealistic. We applied the mapping on part of available application examples.

We focused on the specification of the IDL3-to-VHDL mapping. The implementation of our IDL3-to-VHDL compiler is still experimental, but remains innovative through the application of OMG MDA standards. In addition to OMG IDL3, this compiler may also support UML component diagrams as input. Such a compiler may increase designer productivity by reducing the gap between specification and implementation through code generation. Unfortunately, this productivity gain could not be measured in our examples.

Face to the number and complexity of standards, the relative maturity of tools, the required training of software, hardware and system engineers and the associated costs, a coherent HW/SW co-design flow is not easy to define and to apply within an enterprise or a set of specialized enterprise departments. The conceptual design flow we proposed was not applied to real designs, but it is based on current states of the art and trends. Moreover, the use of the MDA approach from a methodology point-of-view is conceptual, rather than a strict implementation of the OMG approach e.g. using UML and its various UML profiles such as MARTE, SysML, IP-XACT and PIM/PSMs for SDR.

Finally, we measured the increasing overhead in terms of area of three configurations of our middleware architecture framework, but we did not find the time to measure the corresponding overhead in terms of latency. The latency overhead would certainly also increase like the size overhead, since it also depends on message size and the complexity of their encoding and decoding.

9.5 Conclusions

After several decades of cross-fertilization between software and hardware engineering, one can observe a progressive convergence towards common component approach at system level. In this work, we attended to provide a unified object-oriented component-based approach for software components af functional level, hardware components at registertranfer level and system components at transactional level and register transfer level. Such a common approach may help software, hardware and system engineers to share a common and unified approach of specification, design and implementation of hardware/software component- embedded systems. We apply the portability and interoperability of software middlewares to provide a hardware/software middleware approach to mask the heterogeneity of languages, implementation, environment and abstraction level between hardware and software components.

Due to the complexity of the mapping space and our maturity, it may be better to define hardware application programming interfaces under the form of standardized hardware interfaces like the

Transceiver, to increase portability and reusability rather than defining mappings for business interfaces. This requires to find a boundary between business interfaces and standard platform services, and that SDR enterprises find a consensus on these standard interfaces.

In particular, the interfaces of radio services and radio devices may be standardized. An example is CORBA with business applications and standard CORBA services. However, interfaces of waveform components may be standardized like in the UML profile for SDR, but they are also part of business interfaces, know-how and innovation. As the SCA defines standard software interfaces, it may also define standard hardware interfaces other than low-level MHAL nodes.

It's only recently, in August 2009, that the JPEO JTRS annonced the development of a technology independent SCA specification in which CORBA will become optional [JTR09] ! This recent change is the result of a long maturation process. Even if CORBA becomes optional in the SCA, the needs to map abstract object-oriented components in technology/platform independent models to concrete components in technology/platform specific models will subsist.

9.6 Perspectives

Cooperations between our approach and third-party HLS tools may be envisaged to leverage the maturity of these tools, which are more and more recognized and used in the industry, to efficiently generate synthetizable HDL code from a high-level object-oriented component specification in UML or CORBA IDL. In particular, HLS tools may address the limitation of declarative interface definition languages like OMG IDL through the synthesis from behavioral/algorithmic specification in C/C++, domain-specific languages or UML action languages.

Existing component-oriented languages such as UML2, CORBA IDL3 and D&C XML descriptors could be extended with SCA and IP-XACT concepts to address the description, configuration and deployment of hardware/software application components onto hardware/software distributed embedded platforms. These languages could be improved by taking into account description of execution platforms such as available hardware resources and communication buses to generate optimized integration code. We defined different hardware interfaces and connector types to finely map the semantics of communications between components, but interaction semantics could be more formally specified using formal models of computation and communication, and executable meta-models of components and connectors. Transformation rules might guarantee the traceability of semantics between system specification and the resulting hardware, software and system components.

Despite numerous achievements in Model-Driven Engineering (MDE), Transaction-Level Modeling and High-Level Synthesis (HLS), the design and implementation of heterogeneous real-time distributed embedded systems remain complex and hard. A lot of work remains to do to seamlessly integrate these heterogeneous technologies and provide a unified approach to end-users. Numerous standards exist in the world that provide part of solutions to recurring problems to embedded system design such as mapping applications to execution platforms. Some languages such as AADL and domain-specific ADLs might be better than the languages we used in this work for mapping specification and code generation.

We believe that Model-Driven Engineering (MDE), model transformations, interface mapping and refinement at different abstraction levels and integration of such heterogeneous technologies are good candidates to reach such ambitious objectives. Finally, we believe that the common component-oriented and middleware approach we proposed for hardware/software embedded and real-time systems is promising and only at its beginning. Further investigations are still required, but such an approach could be standardized in the future at the OMG.

Appendix A

CORBA IDL3 to VHDL and SystemC RTL Language Mappings

Contents

In this annexe chapter, we propose the specification of language mappings from OMG IDL3 to VHDL and SystemC RTL. We improved a first proposition made by Hugues Balp and Christophe Jouvray in the scope of the ITEA SPICES project. Our improvements notably include the definition of common standard interfaces and protocols, the illustration of the mapping space exploration with these interfaces,

A.1 Naming Convention

To enhance readability, a clock signal is named `clk` and a reset signal `rst`. Moreover, the name of a signal active on low level ends with the suffix `_n`. Note that VHDL is a case-insensitive language.

A.2 Common Standard Interfaces and Protocols

Defining a mapping from IDLs to HDLs is a high-level interface synthesis problem. This mapping requires to be flexible enough in reason of the inherent hardware trade-off between timing, area and power consumption.

To support portability and interoperability of hardware components and propose a flexible and extensible mapping, we propose common hardware interfaces and protocols to provide the building blocks of a mapping framework. Such a framework will allow to fit the mapping to the application requirements.

The purpose of this mapping framework is to explicitly associate with one or more operation parameters the required hardware interface and protocol that best correspond to the implicit communication model of the parameter (control or data, continuous or buffered stream, with or without control flow, memory-mapped access and so on). We propose a bottom-up approach. For each defined hardware interface, we present an example in IDL and how this interface is related to other proposed or standard interfaces. These interfaces are defined from the client's or user's point of view. Server interface signals other than the clock and reset signals have complementary direction.

These interfaces present the mapping framework approach, but can not be considered as exhaustive. New interfaces could be added to address unforeseen needs. The names of the presented signals may be changed by the user. The new names correspond to alias for the real name of the standard interfaces signals. The only important point is to respect the standard interface protocol.

The hardware interface generated from an IDL interface is synchronous point-to-point interfaces to enable transparent access to hardware components. Thanks to a point-to-point interface, client and server hardware components have the same interface in a local direct connection as well as in a remote ORB mediated connection thanks to hardware proxies.

We define a clock signal named `clk` and an active low asynchronous reset signal named `rst_n`. All signals in an hardware interface other than the clock and the reset signals are unidirectional and are synchronous on the rising edge of the clock to simplify hardware implementation.

The standard family of hardware interfaces must include from simple data interface for registered attributes to addressable blocking interface for sequential synchronous invocations. Each successive interface adds semantics: blocking or non-blocking, push semantics for write or pull for read, with or without flow control, addressable or not. These interfaces can be easily mapped onto socket protocol like OCP. Instead of a raw set of OCP interface and protocol parameters and some OCP subsets, these named interfaces explicitly specify the invocation semantics and ease mapping configurability.

A.2.1 Control Interface

Note that all hardware interfaces contains in their module definition (e.g. VHDL entity) a `clk` and `rst_n` signals not represented for space reason.

```
1 entity non_blocking_control_if is
    port (
      push :in std_logic; -- ...
); end;
```

Listing A.1: Non-blocking control interface

For instance, this interface could correspond to:

```
interface non_blocking_control_if {
  oneway void push();
};
```

The push signal is a control signal used to inform hardware component of an event (operation call, notification). The interaction is non-blocking, no response is returned to the caller. This interface corresponds to a "send-and-forget" [SKH98] interaction semantics and to mgc_in_wire [Gra07]. In OCP, `clk` corresponds to Clk and `push` to MCmd(0)/WR. Other base names for push could be put, request, write, select and enable.

```
entity blocking_control_if is
  port (
    push : in  std_logic;
    ack  : out std_logic; -- ...
); end;
```

Listing A.2: Blocking control interface

For instance, this interface could correspond to:

```
interface blocking_control_if {
  void push();
};
```

The `ack` signal is an acknowledge signal used to inform the caller that the request is accepted and can be processed. Other base names for `ack` may be response, ready, grant, accept, ok, clear. This interface corresponds to a "strobe-based" [SKH98] interface.In OCP, `ack` can be mapped to `SCmdAccept`.

A.2.2 Data Interface

```
entity data_if is
  generic (
     DATA_WIDTH: integer:=8; -- ...
  );
  port (
    data : in std_logic_vector(DATA_WIDTH-1 downto 0); -- ...
); end;
```

Listing A.3: Data interface

For instance, this could correspond to:

```
interface data_if {
  void push(in octet data);
};
```

In OCP, data correspond to `MData` and `DATA_WIDTH` to `data_width` in OCP configuration file

A.2.3 Non-Blocking Push Interface

```
entity non_blocking_push_if is
  generic (
     PUSH_DATA_WIDTH: integer:=8; -- ...
  );
  port (
    push      : in std_logic;
    push_data : in std_logic_vector(PUSH_DATA_WIDTH-1 downto 0); -- ...
); end;
```

For instance, this could correspond to:

```
interface non_blocking_push_if {
oneway void push(in octet data);
};
```

The data signal is a data bus used to transport PUSH_DATA_WIDTH bits of user-defined data to the hardware component. The data is valid when push is active. The callee can store data in an internal register or process the data during strobe activation. Synonyms for push include `write`, `data_valid`, `enable` or `data_strobe` (**DS**). In OCP, `clk` corresponds to `Clk`, `push` is `MCmd(0)` e.g. WR, `push_data` is `MData` and `PUSH_DATA_WIDTH` is `data_wdth`. This interface can correspond to the Worker Streaming Interface in [NK07] and strobe-based interface in [SKH98].

A.2.4 Blocking Push Interface

```
entity blocking_push_if is
  generic (
     PUSH_DATA_WIDTH: integer:=8;
  );
  port (
    push      : in  std_logic;
    push_data : in  std_logic_vector(PUSH_DATA_WIDTH-1 downto 0);
    ack       : out std_logic; -- ...
); end;
```

For instance, this could correspond to:

```
interface blocking_push_if {
  void push(in octet data);
};
```

This interface corresponds to a "handshaking-based" [SKH98] interaction semantics and its implementation could be mgc_out_stdreg_en in [Gra07]. The `ack` signal is an acknowledge signal used to inform the caller that the request is accepted and can be processed. In OCP, `clk` corresponds to `Clk`, push to `MCmd(0)` e.g. WR, `push_data` to `MData`, `PUSH_DATA_WIDTH` to `data_wdth`. The data signal is a data bus used to transport `PUSH_DATA_WIDTH` bits of user-defined data to the hardware component.

A.2.5 Non-Blocking Pull Interface

```
entity non_blocking_pull_data_if is
  generic (
    PULL_DATA_WIDTH: integer:=8; -- ...
  );
  port (
    pull      : in  std_logic;
    pull_data : out std_logic_vector(PULL_DATA_WIDTH-1 downto 0); -- ...
); end;
```

For instance, this could correspond to:

```
interface non_blocking_pull_if {
  oneway void pull(in octet data);
};
```

For instance in OCP, `clk` is `Clk`, pull is `MCmd(1)` i.e. RD, `pull_data` is `SData` and `PULL_DATA_WIDTH` is `data_wdth` in OCP configuration file. A synonym for pull is read. Other names for `pull_data` could be `out_data`, `dout`, `data_out`.

A.2.6 Blocking Pull Interface

```
entity non_blocking_pull_data_if is
  generic (
    PULL_DATA_WIDTH: integer:=8; -- ...
  );
  port (
    pull      : in  std_logic;
    pull_data : out std_logic_vector(PULL_DATA_WIDTH-1 downto 0);
    ack       : out std_logic; -- ...
); end;
```

For instance, this could correspond to:

```
interface blocking_pull_if {
  void pull(in octet data);
};
```

This interface corresponds to mgc_wire_en [Gra07]. For instance in OCP, `clk` is `Clk`, pull is `MCmd(1)` i.e. RD, `pull_data` is `SData`, `ack` is `SCmdAccept` and `PULL_DATA_WIDTH` is `data_wdth` in OCP configuration file. A synonym for pull is read. Other names for `pull_data` could be `out_data`, `dout`, `data_out`.

A.2.7 Flow Control Blocking Push Interface

```
entity blocking_push_if is
  generic (
     PUSH_DATA_WIDTH: integer:=8;
     BUFFER_SIZE    : integer:=4;
     ALMOST_FULL_THRESHOLD: integer:=2; -- ...
  );
  port (
    push        : in  std_logic;
    push_data   : in  std_logic_vector(PUSH_DATA_WIDTH-1 downto 0);
    full        : out std_logic;
    almost_full : out std_logic; -- ...
); end;
```

For instance, this could correspond to:

```
interface blocking_push_if {
  void push(in octet data);
};
```

This interface corresponds to a "handshaking-based" [SKH98] interaction semantics and its implementation could be mgc_out_stdreg_en. In OCP, `full`/`almost_full` signals correspond to `SRespInfo` or `SFlag`.

A.2.8 Flow Control Blocking Pull Interface

```
entity non_blocking_pull_data_if is
  generic (
     PUll_DATA_WIDTH: integer:=8; -- ...
  );
  port (
    clk      : in  std_logic;
    pull  : in  std_logic;
    pull_data  : out std_logic_vector(PUll_DATA_WIDTH-1 downto 0);
    empty  : out std_logic;
    almost_empty : out std_logic; -- ...
); end;
```

For instance, this could correspond to:

```
interface non_blocking_pull_if {
  void pull(in octet data);
};
```

In OCP, `empty` and `almost_empty` correspond to `SRespInfo` or `SFlag`.

A.2.9 Non-Blocking FIFO Interface

```
entity non_blocking_fifo_if is
  generic (
     DATA_WIDTH        : positive:=8; -- FIFO data width
     DEPTH_WIDTH        : positive:=4; -- bits for coding fifo depth
     ALMOST_FULL_THRESHOLD : natural;
     ALMOST_EMPTY_THRESHOLD: natural;     -- ...
  );
  port (
    -- Push request
    push  : in  std_logic; -- write request
    push_data  : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    almost_full  : out std_logic;
    full  : out std_logic;
    -- Pull request
    pull  : in  std_logic;
    pull_data  : out std_logic_vector(DATA_WIDTH-1 downto 0);
    empty  : out std_logic;
    almost_empty : out std_logic;  -- ...
  ); end;
```

The meta-data required to choose and configure the hardware interface could be explicitly described by pragma directive like in high-level synthesis tool. However, a separate mapping configuration file is preferable, because it don't require IDL modifications. Here is an example:

```
#pragma non_blocking_fifo
interface non_blocking_fifo_if {
    enum PushFIFOStatusType {
      ALMOST_FULL,
      FULL
    };
    enum PullFIFOStatusType {
      ALMOST_EMPTY,
      EMPTY
    };
    exception PushFIFOException {
      PushFIFOStatusType errorNumber;
    };
    exception PullFIFOException {
      PullFIFOStatusType errorNumber;
    };
    #pragma depth_width 4
    #pragma almost_full_threshold  3
    #pragma almost_empty_threshold 1
    void push(in octet data) raises(PushFIFOException);
    void pull(out octet data) raises(PullFIFOException);
};
```

A.2.10 Non-Blocking Push Addressable Interface

```
entity non_blocking_push_mem_if is
  generic (
     PUSH_ADDR_WIDTH: integer:=8;
     PUSH_DATA_WIDTH: integer:=8;     -- ...
  );
  port (
    push      : in std_logic;
    push_addr : in std_logic_vector(PUSH_ADDR_WIDTH-1 downto 0);
    push_data : in std_logic_vector(PUSH_DATA_WIDTH-1 downto 0); -- ...
); end;
```

This interface can correspond to the Worker Memory Interface [NK07]. It could be used for method invocations or attributes mapped in RAM/ROM.

For instance, this could correspond to:

```
interface non_blocking_push_mem_if {
  oneway void push(in sequence<octet, 256> data);
};
```

A.2.11 Blocking Addressable Push Interface

```
entity non_blocking_push_mem_if is
  generic (
     PUSH_ADDR_WIDTH: integer:=8;
```

```
     PUSH_DATA_WIDTH: integer:=8; -- ...
  );
  port (
    clk   : in  std_logic;
    push   : in  std_logic;
    push_addr : in  std_logic_vector(PUSH_ADDR_WIDTH-1 downto 0);
    push_data : in  std_logic_vector(PUSH_DATA_WIDTH-1 downto 0);
    ack   : out std_logic; -- ...
); end;
```

For instance, this could correspond to:

```
interface blocking_push_mem_if {
  void push(in sequence<octet, 256> data);
};
```

This interface corresponds to the write-only OCP subset.

A.2.12 Non-Blocking Addressable Pull Interface

```
entity non_blocking_pull_mem_if is
  generic (
     PULL_ADDR_WIDTH: integer:=8;
     PULL_DATA_WIDTH: integer:=8; -- ...
  );
  port (
    clk   : in  std_logic;
    pull      : in  std_logic;
    pull_addr : in  std_logic_vector(PULL_ADDR_WIDTH-1 downto 0);
    pull_data : out std_logic_vector(PULL_DATA_WIDTH-1 downto 0); -- ...
); end;
```

For instance, this could correspond to:

```
interface non_blocking_pull_mem_if {
  void pull(out sequence<octet, 256> data);
};
```

This interface corresponds to the read-only OCP subset.

A.2.13 Non-Blocking Single-Port Memory Interface (or Addressable)

```
entity twoway_if is
  generic (
     ADDR_WIDTH: integer:=8;
     DATA_WIDTH: integer:=8; -- ...
  );
  port (
    clk       : in  std_logic;
    addr      : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
```

```
    -- Push request
    push      : in  std_logic;
    push_data : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    -- Pull request
    pull      : in  std_logic;
    pull_data : out std_logic_vector(DATA_WIDTH-1 downto 0); -- ...
); end;
```

For instance, this could correspond to:

```
interface non_blocking_single_port_mem_if {
  void process( in  sequence<octet, 4> data_in,
                out sequence<octet, 4> data_out);
};
```

A.2.14 Non-Blocking Dual-Port Memory Interface (or Addressable)

```
entity non_blocking_dual_port_mem_if is
  generic(
     ADDR_WIDTH: integer:=8; -- width of the address bus
     DATA_WIDTH: integer:=8; -- width of the data bus
  );
  port(
    push_clk  : in  std_logic; -- input data clock
    pull_clk  : in  std_logic; -- output data clock
    -- Push request
    push      : in  std_logic; -- write request
    push_addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
    push_data : in  std_logic_vector(DATA_WIDTH-1 downto 0);
        -- Pull request
    pull      : in  std_logic;
    pull_addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
    pull_data : out std_logic_vector(DATA_WIDTH-1 downto 0));
end;
```

For instance, this could correspond to:

```
interface non_blocking_dual_port_mem_if {
  void process(inout sequence<octet, 256> data_in);
};
```

A.2.15 Generic Message Communication Interface

```
entity non_blocking_dual_port_mem_if is
  generic(
     ADDR_WIDTH: integer:=8; -- width of address bus
     DATA_WIDTH: integer:=8; -- width of data bus
  );
```

```
  port(
    clk       : in  std_logic;
    -- Push request
    push : in  std_logic; -- write request
    pull : in  std_logic; -- read request
    addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
    push_data : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    pull_data : out std_logic_vector(DATA_WIDTH-1 downto 0));
  end;
```

For instance, this could correspond to:

```
interface transport_if {
  void pushMessage(in sequence<octet> request_message);
  void pullMessage(in sequence<octet> reply_message);
};
```

This interface may represent the interface of a bus or NoC connector.

A.2.16 Optional Signals

Optional signals can be added when required. For instance, all these interfaces may have an optional `enable` signal to activate or not the component, a *clear* signal to perform a synchronous reset, etc.

A.3 Constant

Constant identifiers are mapped to constants in VHDL and SystemC. For example:

```
// OMG IDL3
const short my_long_value = 1357;

-- VHDL RTL
constant my_short_value : CORBA_short := "0000010101001101";

// SystemC RTL
const CORBA_short my_long_value = "0000010101001101";
```

A.4 Basic Data Types

The IDL-to-HDL language mapping uses a subset of all data types defined by the CORBA specification [OMG08b].

All integer basic data types are supported by the IDL to HDL mapping The following data types are not deliberately supported by this mapping: wide char, string, wide string. These data types are not required for embedded and real-type systems. They may be replaced by sequence of octets.

IDL basic data types are mapped to synthesizable logic vectors as shown for VHDL in listing A.4 and for SystemC RTL in listing A.5.

```
package CORBA is
  subtype CORBA_boolean is std_logic; -- (=1 if TRUE, 0 if FALSE )
  subtype CORBA_octet is std_logic_vector(   7 downto 0 );
  subtype CORBA_short is std_logic_vector( 15 downto 0 );
  subtype CORBA_long is std_logic_vector( 31 downto 0 );
  subtype CORBA_long_long is std_logic_vector( 63 downto 0 );
  subtype CORBA_unsigned_short is std_logic_vector( 15 downto 0 );
  subtype CORBA_unsigned_long is std_logic_vector( 31 downto 0 );
  subtype CORBA_unsigned_long_long is std_logic_vector( 63 downto 0 );
  -- new data types
  subtype CORBA_small is std_logic_vector( 7 downto 0 );
  subtype CORBA_int2 is std_logic_vector(  1 downto 0 ); -- and so on
  subtype CORBA_bit is std_logic;
  subtype CORBA_exception is std_logic;
end package CORBA;
```

Listing A.4: Mapping of IDL basic data types to VHDL

As VHDL does not support namespaces, the `CORBA_` prefix is added to the mapped IDL data types as in the C language mapping. Four new IDL data types are introduced. The `small` data type is a 8 bit integer type proposed to replace the opaque `octet` data type and the `short` type when 16 bit are not required. The `intX` data type denotes an integer of X bits to support integers with a various number of bits. For instance, CORBA_int3 can be defined as: `subtype CORBA_int3 is std_logic_vector( 2 downto 0 );`. On GPPs and DSPs, `intX` types may be interpreted as a vector of useful bits in a fixed data type. The remaining bits of the data type are non-significant and simply ignored. As in SystemC, these types may be supported by a dedicated library able to properly marshalled and demarshalled their value. Another notation could be int<bit_width>. We also introduce a logic vector in such as CORBA_lv(2 downto 0); The `bit` data type only correspond to a standard logic value. The `exception` logic signal is used to indicate that an exception has occurred. An `exc_id` logic vector will be defined in each hardware interface to indicate an exception number, whose size is the log_2 of the maximum number of throwable exceptions.

```
namespace CORBA {
  typedef  sc_logic     boolean;      //  (=1 if TRUE, 0 if FALSE )
  typedef  sc_lv< 8 >   octet;
  typedef  sc_lv< 16 >  short;
  typedef  sc_lv< 32 >  long;
  typedef  sc_lv< 64 >  long_long;
  typedef  sc_lv< 16 >  unsigned_short;
  typedef  sc_lv< 32 >  unsigned_long;
  typedef  sc_lv< 64 >  unsigned_long_long;
  // new data types
  typedef  sc_lv< 8 >   small;
  typedef  sc_lv< 2 >   int2;
  typedef  sc_logic     exception;
  typedef  sc_logic     bit;
};
```

Listing A.5: Mapping of IDL basic data types to SystemC

A.5 Constructed Data Types

The following constructed data types are supported by the IDL-to-HDL mapping: enumeration, fixed-length structure types, fixed-length union types, fixed-length array of IDL basic types, fixed- or variable-length sequence of IDL basic types.

A.5.1 Enumeration

The HDL mapping of an OMG IDL `enum` type is a set of VHDL signals encoding the enum values. For example:

```
// OMG IDL3
enum etat1 { idle, active, waiting, end };
```

The mapping in VHDL depends on a chosen enumeration encoding:
1) **symbolic encoding** which is defined by the synthesis tool e.g.

```
-- VHDL
package etat1 is
  type etat1 is (idle, active, waiting, end);
end etat1;

// SystemC RTL
enum etat1 { idle, active, waiting, end };
```

2) **binary encoding** which uses $\lceil log_2(n) \rceil + 1$ registers for `n` states. The enum values are encoded in increasing order in a logic vector e.g. "00" for idle, "01" for active "10" for waiting and "11" for end.

```
-- VHDL
package etat1 is
  subtype etat1_enum is std_logic_vector( 1 downto 0 );
end etat1;

// SystemC RTL
typedef sc_lv< 2 > etat1_enum;
```

3) **one hot encoding** which uses `n` registers for `n` states. This encoding is recommended for FPGAs.

```
-- VHDL
package etat1 is
  subtype etat1_enum is std_logic_vector( 3 downto 0 );
end etat1;

// SystemC RTL
typedef sc_lv< 4 > etat1_enum;
```

The enum values are encoded as follows: "0001" for idle, "0010" for active, "0100" for waiting and "1000" for end.

For example suppose the operation `operation1` defined in interface `intf1` as follows:

```
// OMG IDL3
interface intf1 {
  void operation1( in etat1 a1 );
};
```

If two non-blocking push interfaces are chosen, this is equivalent to the following declarations in VHDL:

```
-- VHDL RTL
use work.etat1.all;
entity intf1 is port (
  operation1_push_in         : in  CORBA_bit;
  operation1_push_data_in_a1 : in  etat1_enum;
  operation1_push_out        : out CORBA_bit;    -- ...
  );
end intf1;

// SystemC RTL
SC_MODULE( intf1 ) {
  sc_in< CORBA::bit >  operation1_push_in;
  sc_in< etat1_enum >  operation1_push_data_in_a1;
  sc_in< CORBA::bit >  operation1_push_out;
  // ...
};
```

A.5.2 Array

An IDL `array` can be mapped to registers, FIFO or internal/external RAM/ROM. If the VHDL `array` keyword is used, the synthesis tool may infer internal registers or RAMs. We propose a mapping for arrays independent from the chosen hardware implementation. For each named array type in OMG IDL, mapping to software languages defines a slice. A slice of an array is another array with all the dimensions of the original except the first. With the proposed mapping, slices are not required.

```
// OMG IDL3
typedef long array1[ 4 ][ 5 ];

-- VHDL RTL
package array1 is
  use corba.CORBA.all;
  type array1_0_0  is CORBA_long;
  -- . . .
  type array1_3_4  is CORBA_long;
end array1;

// SystemC RTL
namespace array1 {
  typedef CORBA::long[ 4 ][ 5 ] array;
  typedef CORBA::long[ 5 ]      array_slice;
};
```

Array used as attributes

For example, suppose the array attribute a1 defined in "intf1" as follows:

```
// IDL
interface intf1 {
  attribute array1 a1;
};
```

In SystemC, the array is a private class member:

```
// SystemC RTL
SC_MODULE( intf1 ) {
private:
  array       a1;
  array_slice a1_slice;  // ...
};
```

Array mapped to register

Since concurrent access to array members may be required, dedicated data bus signals are necessary for the user interface. The mapping to VHDL is the following:

```
-- VHDL RTL
entity intf1 is
port (
  data_in_a1_0_0  : array1_0_0; -- ...
  data_in_a1_3_4  : array1_3_4;
);
```

Array mapped to internal or external RAM/ROM

If the mapping is based on a single memory port, array items can not be read and write at the same time, therefore `push_in/out` and `pull_in/out` signals are merged into the same shared `push_pull_in/out` signal.

```
-- VHDL RTL
entity intf1 is
generic (
  A1_ADDR_WIDTH          : integer:=5;
  A1_PUSH_DATA_WIDTH_IN  : integer:=32;
  A1_PUSH_DATA_WIDTH_OUT : integer:=32; -- ...
);
port (
  a1_push_pull_in       : in  CORBA_bit;
  a1_addr               : in  CORBA_lv(A1_ADDR_WIDTH -1 downto 0);
  a1_push_data_in_item : in  CORBA_lv(A1_PUSH_DATA_WIDTH_IN-1 downto 0);
  a1_push_data_out_item : out CORBA_lv(A1_PUSH_DATA_WIDTH_OUT -1 downto 0);
  a1_push_pull_out : out CORBA_bit; -- ...
);
```

Array used as operation parameters

In case an array is passed as an operation parameter, the mapping of the array type to VHDL is applied on the corresponding parameter.

For example suppose the operation `operation1` defined in interface `intf1` as follows:

```
// OMG IDL3
interface intf1 {
  void operation1( in array1 a1 );
};
```

If two non-blocking push interfaces are chosen, this is equivalent to the following declarations in HDLs:

```
-- VHDL RTL
entity intf1 is
generic (
  PUSH_DATA_WIDTH_IN : integer:=32; -- ...
);
port (
  push_in               : in  CORBA_bit;
  push_data_in_a1_item : in  CORBA_lv(PUSH_DATA_WIDTH_IN-1 downto 0);
  push_out     : out CORBA_bit; -- ...
);

// SystemC RTL
#define PUSH_DATA_WIDTH_IN 8
SC_MODULE( intf1 ) {
  sc_in< CORBA_< CORBA::bit >                     push_in;
  sc_in < CORBA_< CORBA::lv<PUSH_DATA_WIDTH_IN> > push_data_in_a1_item;
  sc_out< CORBA_< CORBA::bit >                    push_out; -- ...
};
```

A.5.3 Data Structure

OMG IDL `struct` can be mapped to registers, FIFOs or internal/external RAM/ROM. This mapping proposition supports fixed-size structures of simple and/or structured types. Each field of the struct is mapped according to the mapping rule of its type.

For example suppose the following IDL structure:

```
// OMG IDL3
struct struct1 {
 octet x;
 short y;
 long  z;
};
struct struct2 {
 octet   a;
 struct1 b;
};
```

This is equivalent to the following types in VHDL:

```
-- VHDL RTL
package struct1 is
  use corba.CORBA.all;
  subtype struct1_x is CORBA_octet;
  subtype struct1_y is CORBA_short;
  subtype struct1_z is CORBA_long;
end struct1;

package struct2 is
  use corba.CORBA.all;
  use work.struct1.all;
  subtype struct2_a   is CORBA_octet;
  subtype struct2_b_x is struct1_x;
  subtype struct2_b_y is struct1_y;
  subtype struct2_b_z is struct1_z;
end struct2;

// SystemC RTL
using namespace CORBA;
namespace struct1 {
  typedef CORBA_octet x;
  typedef CORBA_short y;
  typedef CORBA_long  z;
};

namespace struct2 {
  typedef CORBA_octet a;
  typedef struct1::x  b_x;
  typedef struct1::y  b_y;
  typedef struct1::z  b_z;
};
```

Struct used as attributes

For example, suppose the struct attribute `s1` defined in `intf1` interface as follows:

```
// IDL
interface intf1 {
  attribute struct2 s1;
};
```

In SystemC, the array is a private class member:

```
 // SystemC RTL
SC_MODULE( intf1 ) {
private:
  struct2 s1;
```

```
  // ...
};
```

Struct mapped to registers

Since concurrent accesses to struct members may be required, dedicated data bus signals are necessary for the user interface. The mapping to VHDL is the following:

```
-- VHDL RTL
entity intf1 is
port (
  data_in_s1_a   : struct2_a;
  data_in_s1_b_x : struct2_b_x;
  data_in_s1_b_y : struct2_b_y;
  data_in_s1_b_z : struct2_b_z;   -- ...
);
```

Struct mapped to internal or external RAM/ROM

The data address offset, data layout and endianness must be explicitly defined in the mapping configuration file. For instance, with the previous example:

```
-- VHDL RTL
entity intf1 is
generic (
  S1_ADDR_WIDTH          : integer:=6;
  S1_PUSH_DATA_WIDTH_IN  : integer:=32;
  S1_PUSH_DATA_WIDTH_OUT : integer:=32; -- ...
);
port (
  s1_push_pull_in       : in  CORBA_bit;
  s1_addr               : in  CORBA_lv(S1_ADDR_WIDTH -1 downto 0);
  s1_push_data_in_item : in  CORBA_lv(S1_PUSH_DATA_WIDTH_IN-1 downto 0);
  s1_push_data_out_item : out CORBA_lv(S1_PUSH_DATA_WIDTH_OUT -1 downto 0);
  s1_push_pull_out      : out CORBA_bit; -- ...
);
```

Struct used as operation parameters

In case the structure is passed as parameter in an operation, the mapping of the struct parameter to VHDL result to as many signals as the number of fields of the structure.

For example suppose the operation `operation1` defined in interface `intf1` as follows:

```
// IDL
interface intf1 {
  void operation1( in struct1 a1 );
};
```

If two non-blocking push interfaces are chosen, this is equivalent to the following entity in VHDL:

```
-- VHDL RTL
  use work.struct1.all;
  entity intf1 is port (
    operation1_push_in           : in  CORBA_bit;
    operation1_push_data_in_a1_x : in  struct1_x;
    operation1_push_data_in_a1_y : in  struct1_y;
    operation1_push_data_in_a1_z : in  struct1_z;
    operation1_push_out          : out CORBA_bit; -- ...
  );
  end intf1;
```

This is equivalent in SystemC to:

```
// SystemC RTL
  SC_MODULE( intf1 ) {
    sc_in < CORBA_< CORBA::bit > operation1_push_in;
    sc_in< struct1::x >          operation1_push_data_in_a1_x;
    sc_in< struct1::y >          operation1_push_data_in_a1_y;
    sc_in< struct1::z >          operation1_push_data_in_a1_z;
    sc_out< CORBA_< CORBA::bit > operation1_push_out; // ...
  };
};
```

A.5.4 Union

The `union` type is not really required, nevertheless a mapping is proposed for completeness. This IDL-to-HDL mapping supports fixed-size `union`s of simple types. Each field of the `union` is mapped according to the mapping rule of its data type.

Consider the following OMG IDL declaration:

```
 // IDL
 union union1 switch( octet ) {
  case 1: octet x;
  case 2: short y;
  case 3: long  z;
  default: unsigned short d;
 };
```

This is mapped to the following HDL codes:

```
-- VHDL RTL
package union1 is
  use corba.CORBA.all;
  -- union1_width_ is equals to the maximum width of included types
  constant union1_width : integer := 32;  -- size of a CORBA_long
  -- union discriminator
  subtype union1_d    is CORBA_octet;
  -- union subtypes
  subtype union1_u    is std_logic_vector( union1_width - 1 downto 0 );
```

```
  subtype union1_u_x  is CORBA_octet;
  subtype union1_u_y  is CORBA_short;
  subtype union1_u_z  is CORBA_long;
  subtype union1_u_d  is CORBA_unsigned_short;
end union1;

// SystemC RTL
using namespace CORBA;
namespace union1 {
  -- union_width_ is equals to the maximum width of included types
  const short union1_width = 32; // size of a CORBA_long
  -- union discriminator
  typedef CORBA_octet           d;
  -- union sub-types
  typedef sc_lv< union1_width > u;
  typedef CORBA_octet           u_x;
  typedef CORBA_short         u_y;
  typedef CORBA_long         u_z;
  typedef CORBA_unsigned_short  u_d;
};
```

The discriminator is referred to as <union_name>_d_, while the `union` is referred to as <union_name>_u_.

If a `union` type is passed as parameter of an operation, it is mapped in VHDL to a `std_logic_vector` whose width equals to the maximum width of the types of the union and the discriminator signal. For example, suppose the operation `operation1` defined in interface `intf1` as follows:

```
// IDL
interface intf1 {
  void operation1( in union1 a1 );
};
```

If two non-blocking push interfaces are chosen, this is equivalent to the following entity in VHDL:

```
-- VHDL RTL
use work.union1.all;
entity intf1 is port (
  operation1_push_in           : in  CORBA_bit;
  operation1_push_data_in_a1_u : in union1_u;
  operation1_push_data_in_a1_d : in union1_d;
  operation1_push_out          : out CORBA_bit; -- ...
);
end intf1;

// SystemC RTL
SC_MODULE( comp1 ) {
    // facet1_operation9
    sc_in < CORBA_< CORBA::bit > operation1_push;
    sc_in< union1::u >           operation1_push_data_in_a1_u;
```

```
    sc_in< union1::d >           operation1_push_data_in_a1_d;
    sc_out< CORBA_< CORBA::bit > operation1_push_out; // ...
  };
```

Some aliases could be defined to access the union elements as illustrated below :

```
architecture intf1_impl of intf1 is
  -- Access to union1_x
  alias operation1_a1_u_x_1 : union1_u_x is operation1_a1_u_(  7 downto  0 );
  alias operation1_a1_u_x_2 : union1_u_x is operation1_a1_u_( 15 downto  8 );
  alias operation1_a1_u_x_3 : union1_u_x is operation1_a1_u_( 23 downto 16 );
  alias operation1_a1_u_x_4 : union1_u_x is operation1_a1_u_( 31 downto 24 );
  -- Access to union1_y
  alias operation1_a1_u_y_1 : union1_u_y is operation1_a1_u_( 15 downto  0 );
  alias operation1_a1_u_y_2 : union1_u_y is operation1_a1_u_( 31 downto 16 );
  -- Access to union1_z
  alias operation1_a1_u_z_1 : union1_u_z is operation1_a1_u_( 31 downto  0 );
begin
  -- One of the union1 type is used depending on the discriminator
end intf1_impl;
```

A.5.5 Sequence

An IDL `sequence` is a bounded or unbounded arrays whose elements have the same type. A `sequence` can be mapped to a register, FIFO or in internal/external RAM/ROM. Consider the following OMG IDL declaration:

```
// IDL
typedef sequence<long> sequence1;

-- VHDL RTL
package sequence1 is
  use corba.CORBA.all;
  constant SEQUENCE1_LENGTH_WIDTH : integer := 16;
  constant SEQUENCE1_ITEM_WIDTH   : integer := 8;
  subtype sequence1_length   is CORBA_lv(SEQUENCE1_LENGTH_WIDTH-1 downto 0);
  subtype sequence1_sequence is CORBA_lv(SEQUENCE1_ITEM_WIDTH-1 downto 0);
end sequence1;

// SystemC RTL
namespace sequence1 {
  const int SEQUENCE1_LENGTH_WIDTH = 16;
  const int SEQUENCE1_ITEM_WIDTH   = 8;

  typedef CORBA::lv<SEQUENCE1_LENGTH_WIDTH>length;
  typedef CORBA::lv<SEQUENCE1_ITEM_WIDTH>sequence;
};
```

If a `sequence` is passed as operation parameter, the mapping of the sequence type to VHDL is applied on the corresponding parameter

For example suppose the operation `operation1` defined in interface `intf1` as follows:

```
// IDL
interface intf1 {
  void operation1( in sequence1 a1 );   // ...
};
```

If two non-blocking push interfaces are chosen, this is equivalent to the following entity in VHDL:

```
-- VHDL RTL
use work.sequence1.all;
entity intf1 is port (
  operation1_push_in                : in  CORBA_bit;
  operation1_push_data_in_a1_length : in  sequence1_length;
  operation1_push_data_in_a1_item : in  sequence1_item;
  operation1_push_out               : out CORBA_bit;  -- ...
);
end intf1;

// SystemC RTL
SC_MODULE( intf1 ) {
  sc_in < CORBA_< CORBA::bit > operation1_push_in;
  sc_out< CORBA_< CORBA::bit > operation1_push_out;
  sc_in< sequence1::length >   operation1_push_data_in_a1_length;
  sc_in< sequence1::item >     operation1_push_data_in_a1_item;  // ...
};
```

Sequence elements are transmitted sequentially through the `operation1_a1_sequence` signal. The `operation1_a1_length_` signal indicates the number of elements that are still to be transmitted.

When the *operation1_a1_length_* signal equals zero, then no more useful bits are available in the sequence. In this case, there is no distinction between bounded and unbounded sequences.

For unbounded sequences, a packet-oriented interface may be chosen when the receiver does not need to know the real length of a sequence for streaming interfaces. A _end of packet signal and an optinal _start signal replace the _length signal in a similar way to OCP and Avalon.

```
-- VHDL RTL
use work.sequence1.all;
entity intf1 is port (
  operation1_push_in                : in  CORBA_bit;
  operation1_push_data_in_a1_length : in  sequence1_length;
  operation1_push_data_in_a1_item : in  sequence1_item;
  operation1_push_out               : out CORBA_bit; -- ...
);
end intf1;

// SystemC RTL
SC_MODULE( intf1 ) {
```

```
  sc_in < CORBA_< CORBA::bit > operation1_push_in;
  sc_out< CORBA_< CORBA::bit > operation1_push_out;
  sc_in< sequence1::length >   operation1_push_data_in_a1_length;
  sc_in< sequence1::item >     operation1_push_data_in_a1_item;  // ...
};
```

A.5.6 Any

Any are not really required in hardware, nevertheless a mapping is proposed for completeness and backward compatibility with the JTRS SCA interfaces. Any is mapped in HDLs as a sequence of bits and could be mapped in register or RAM/ROM like sequence. CORBA 2.3 defines 32 typecodes. The HDL mapping of the any type is defined as follows:

```
-- VHDL RTL
package CORBA is
  constant CORBA_TYPECODE_WIDTH : integer := 5;
  constant ANY_LENGTH_WIDTH     : integer := 16;
  constant ANY_ITEM_WIDTH       : integer := 8;
  subtype CORBA_TypeCode        is CORBA_lv(CORBA_TYPECODE_WIDTH-1 downto 0);
  subtype CORBA_any_typecode    is CORBA_TypeCode;
  subtype CORBA_any_length      is CORBA_lv(ANY_LENGTH_WIDTH-1 downto 0);
  subtype CORBA_any_sequence    is CORBA_lv(ANY_ITEM_WIDTH-1 downto 0);
end CORBA;

// SystemC RTL
namespace CORBA {
  // ...
  const int CORBA_TYPECODE_WIDTH = 5;
  const int ANY_LENGTH_WIDTH  = 16;
  const int ANY_ITEM_WIDTH  = 8;
  typedef TypeCode CORBA_any_typecode;
  typedef bv< ANY_LENGTH_WIDTH > CORBA_any_length;
  typedef bv< ANY_ITEM_WIDTH >   CORBA_any_sequence;
};
```

If a any is passed as operation parameter, its mapping rules are applied on the corresponding parameter. For example, suppose the operation operation1 defined in the interface intf1 as follows:

```
// IDL
interface intf1 {
  void operation1( in any a1 );   // ...
};
```

If two non-blocking push interfaces are chosen, this is equivalent to the following entity in VHDL:

```
-- VHDL RTL
use corba.CORBA.all;
entity intf1 is port (
  operation1_push_in                : in  CORBA_bit;
```

```
  operation1_push_data_in_a1_typecode : in  CORBA_any_typecode;
  operation1_push_data_in_a1_length : in  CORBA_any_length;
  operation1_push_data_in_a1_sequence : in  CORBA_any_sequence;
  operation1_push_out     : out CORBA_bit;
);
end comp1;

// SystemC RTL
SC_MODULE( comp1 ) {
  sc_in < CORBA_< CORBA::bit >          operation1_push_in;
  sc_in < CORBA_< CORBA::any_typecode>  operation1_in_a1_typecode;
  sc_in < CORBA_< CORBA::any_length >   operation1_in_a1_length;
  sc_in < CORBA_< CORBA::any_sequence > operation1_in_a1_sequence;
  sc_out< CORBA_< CORBA::bit >          operation1_push_out;    // ...
};
```

The `operation1_push_data_in_a1_typecode` denotes the type code of the data contained in the `any`. The `any` value is transferred sequentially through the `operation1_a1_any_` signal. The `operation1_a1_length` signal indicates the number of data that are still to be transferred.

A.5.7 Exception

IDL exceptions allows developers to indicate to users that an error has occurred and that the interface contract is broken. CORBA distinguishes two kinds of exceptions. *Standard exceptions* are defined by the CORBA specification itself, while *user exceptions* are specified by users in IDL.

Standard Exception

CORBA 2.3 defines 33 standard exceptions. This information can be used to specify the maximum width of the signal that will carry the exception code. For instance:

```
-- VHDL RTL
package CORBA is
  constant CORBA_STANDARD_EXCEPTION_WIDTH : integer := 6;
end CORBA;
```

User Exception

The IDL to VHDL mapping supports fixed-size exceptions of simple and/or structured types. Each field of the exception is mapped according to the mapping rule of its associated type. For example suppose the following IDL exception

```
// IDL
exception exception1 {
  long dummy;
};
```

This is mapped to the following HDL declarations:

```
-- VHDL RTL
package exception1 is
  use corba.CORBA.all;
  subtype exception1_dummy is CORBA_long;
end exception1;

// SystemC RTL
namespace exception1 {
  typedef CORBA_long dummy;
};
```

If an operation `operation1` from the interface `intf1` can raise the previous exception:

```
// IDL
interface intf1 {
  exception exception1 {
    long id;
  };
  void operation1() raises( exception1 );
};
```

In HDLs, this is mapped to:

```
use work.exception1.all;
entity intf1 is port (
  operation1_push_in    : in  CORBA_bit;
  operation1_push_out   : in  CORBA_bit;
  operation1_exception1 : out exception1_dummy; -- ...
);
end intf1;

// SystemC RTL
SC_MODULE( intf1 ) {
  sc_in < CORBA::bit >        operation1_push_in;
  sc_out< CORBA::bit>         operation1_push_out;
  sc_out< exception1::dummy > operation1_exception1; // ...
};
```

If the `operation1_exception` signal is null, no exception has been raised. Otherwise, it contains the unique identifier of the raised exception.

A.5.8 Strings

An IDL `string` corresponds an octet sequence of ASCII characters. This type is not particularly useful for the DRE systems. Nevertheless, it may be required for backward compatibility with JTRS SCA interfaces. `String` are mapped to numeric identifiers.

A.5.9 Float

Float integers could be temporarily mapped into logic vectors, however a recent Accellera standard called VHDL-2006-D3.0 already provides VHDL packages for synthesizable floats. Hence, we recommend a direct mapping from IDL floats to VHDL floats. This standard is ready for industry adoption. Synthesis tools will certainly support it when hardware middlewares will be available and used.

A.5.10 Fixed

Fixed point integers could be temporarily mapped into bitvectors, however a recent Accellera standard called VHDL-2006-D3.0 already provides a VHDL package called `ieee.fixed_pkg.all` with the synthesizable `ufixed` and `sfixed` fixed point types. Hence, we recommend a direct mapping from IDL floats to VHDL floats. This standard is ready for industry adoption. Synthesis tools will certainly support it when hardware middlewares will be available and used.

A.6 Attribute

In software language mappings, IDL `attribute`s are inferred as a pair of setter and getter operations to assign and retrieve the value of an `attribute`. Some possible mapping choices include defining two operations for each attribute, defining two operations to set or get any attribute, and defining operations to set or get groups of attributes, and so forth.

A.7 Scoped Name

As HDLs such as VHDL and Verilog do not support scoped names, hardware designers must use the global name for a type, constant, exception, or operation. The scope symbol `::` in OMG IDL is mapped to an underscore _ in VHDL.

A.8 Module

The module IDL keyword is mapped to package in VHDL and namespace in SystemC as follows:

```
// OMG IDL3
module module1 {
  // ...
};

-- VHDL RTL
package module1 is
  -- ...
end module1;

// SystemC RTL
namespace module1 {
  // ...
};
```

A.9 Interface

In the following, we present mapping configurations for some IDL interfaces based on the previous family of hardware interfaces. In other words, we try to explore the mapping exploration space and illustrate its inherent complexity.

A.9.1 Non-blocking control interface for non-reliable oneway operation

If the interface `intf1` is defined as:

```
// OMG IDL3
interface intf1 {
  // unreliable oneway
  oneway void operation1( );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is port (
  operation1_push_in : in  CORBA_bit;  -- ...
);
end intf1;

// SystemC RTL
SC_MODULE( intf1 ) {
  sc_in < CORBA::bit > operation1_push_in;  // ...
};
```

A.9.2 Non-blocking control interfaces for operations without parameters or reliable oneway operation

If the interface `intf1` is defined as:

```
// OMG IDL3
interface intf1 {
  void operation1( );
};
or
interface intf1 {
  // reliable oneway
  oneway void operation1( );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is port (
  operation1_push_in  : in  CORBA_bit;
  operation1_push_out : out CORBA_bit;  -- ...
```

```
);
end intf1;

// SystemC RTL
SC_MODULE( intf1 ) {
  sc_in < CORBA::bit > operation1_push_in;
  sc_out< CORBA::bit > operation1_push_out;  // ...
};
```

The `operation1_push_out` indicates the completion of the operation. This interface corresponds to the mapping proposed in [BRM+06] and the process-level handshake in [Gra].

A.9.3 Addressable non-blocking push interface without data in input and non-blocking control interface in output for several operations without parameters or reliable oneway operation

If the interface `intf1` is defined as:

```
// OMG IDL3
interface intf1 {
  void operation1( );
  void operation2( );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is
generic (
  OPERATION_WIDTH: integer:=1; -- ...
);
port (
  push_in  : in  CORBA_bit;
  addr     : in  CORBA_lv(OPERATION_WIDTH-1 downto 0);
  push_out : out CORBA_bit; -- ...
);
// SystemC RTL
#define OPERATION_WIDTH 1
SC_MODULE( intf1 ) {
  // operation1
  sc_in < CORBA::bit >                  push_in;
  sc_in< CORBA::bv< OPERATION_WIDTH > >addr;
  sc_out < CORBA::bit >                 push_out;
  // ...
};
```

The parameter `OPERATION_WIDTH` is the numbers of bits required to binary encode the operations i.e. $OPERATION\_WIDTH = \lceil log_2(\#operations) \rceil$.

A.9.4 Addressable non-blocking push interface for input and non-blocking control interface for output for several operations with input parameters

For an important number of operations and parameters, hardware I/O ports can be shared to save area. The width of the data bus for operation parameters have to be specified in a mapping configuration file. The parameters are serialized on the data bus according to some standard encoding rules.

If the interface `intf1` is defined as:

```
// OMG IDL3
interface intf1 {
  void operation1( );
  void operation2( in long a1 );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is
generic (
  OPERATION_WIDTH    : integer:=1;
  PUSH_DATA_WIDTH_IN : integer:=32; -- ...
);
port (
  push_in       : in  CORBA_bit;
  addr          : in  CORBA_lv(OPERATION_WIDTH-1 downto 0);
  push_data_in : in  CORBA_lv(PUSH_DATA_WIDTH_IN-1 downto 0);
  push_out      : out CORBA_bit;  -- ...
);

// SystemC RTL
#define OPERATION_WIDTH 4
#define PUSH_DATA_WIDTH_IN 32
SC_MODULE( intf1 ) {
  sc_in < CORBA::bit >                       push_in;
  sc_in< CORBA::bv< PUSH_ADDR_WIDTH > >    addr;
  sc_in< CORBA::bv< PUSH_DATA_ WIDTH_IN> > push_data_in;
  sc_out < CORBA::bit >                      push_out;  // ...
};
```

The `push_out` indicated the end of the operation and does not acknowledge input parameters reception.

A.9.5 Addressable non-blocking push interface in inputs and Non-blocking push interface in outputs for several operations with parameters

If the interface `intf1` is defined as:

```
// OMG IDL3
interface intf1 {
  void operation1( );
```

```
  void operation2( in  long a1 );
  void operation3( out octet a2 );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is
generic (
  OPERATION_WIDTH     : integer :=2;
  PUSH_DATA_WIDTH_IN  : integer :=32;
  PUSH_DATA_WIDTH_OUT : integer :=8; -- ...
);
port (
  -- input
  push_in       : in CORBA_bit;
  addr          : in CORBA_lv(OPERATION_WIDTH-1 downto 0);
  push_data_in  : in CORBA_lv(PUSH_DATA_ WIDTH_IN-1 downto 0);
  -- output
  push_out      : out CORBA_bit;
  push_data_out : out CORBA_lv(PUSH_DATA_WIDTH_OUT-1 downto 0);  -- ...
);

// SystemC RTL
#define OPERATION_WIDTH 2
#define PUSH_DATA_WIDTH_IN 32
#define PUSH_DATA_WIDTH_OUT 8
SC_MODULE( intf1 ) {
  // operation1
  sc_in< CORBA::bit >    push_in;
  sc_in< CORBA::bv< PUSH_ADDR_WIDTH >  >addr;
  sc_in< CORBA::bv< PUSH_DATA_WIDTH_IN > >  push_data_in;
  sc_out< CORBA::bit >                      push_out;
  sc_out< CORBA::bv< PUSH_DATA_WIDTH_IN > > push_data_out;
  // ...
};
```

A.9.6 Addressable blocking push interface in inputs and non-blocking push interface in outputs for several operations with parameters

This configuration could be used for Asynchronous Method Invocation (AMI) or when invocation reliability is required.

If the interface `intf1` is defined as:

```
// OMG IDL3
interface intf1 {
  void operation1( );
  void operation2( in  long a1 );
  void operation3( out octet a2 );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is
generic (
  OPERATION_WIDTH     : integer :=2;
  PUSH_DATA_WIDTH_IN  : integer :=32;
  PUSH_DATA_WIDTH_OUT : integer :=8; -- ...
  );
port (
  -- input
  push_in       : in  CORBA_bit;
  addr          : in  CORBA_lv(OPERATION_WIDTH-1 downto 0);
  push_data_in  : in  CORBA_lv(PUSH_DATA_ WIDTH_IN-1 downto 0);
  ack_in        : out CORBA_bit;
  -- output
  push_out      : out CORBA_bit;
  push_data_out : out CORBA_lv(PUSH_DATA_WIDTH_OUT-1 downto 0); -- ...
);

// SystemC RTL
const int OPERATION_WIDTH     = 2;
const int PUSH_DATA_WIDTH_IN  = 32;
const int PUSH_DATA_WIDTH_OUT = 8;
SC_MODULE( intf1 ) {
  // operation1
  sc_in< CORBA::bit >                        push_in;
  sc_in< CORBA::bv< PUSH_ADDR_WIDTH >  >     addr;
  sc_in< CORBA::bv< PUSH_DATA_WIDTH_IN > >   push_data_in;
  sc_out< CORBA::bit >         ack_in;
  sc_out< CORBA::bit >         push_out;
  sc_out< CORBA::bv< PUSH_DATA_WIDTH_IN > > push_data_out; // ...
};
```

The `ack_in` signal acknowledges the reception of the input parameters.

A.9.7 Addressable non-blocking push interface in input and a non-blocking pull interface in output for several operations with parameters

If the interface `intf1` is defined as follows:

```
// OMG IDL3
interface intf1 {
  void operation1( );
  void operation2( in  long a1 );
  void operation3( out octet a2 );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is
generic (
  OPERATION_WIDTH     : integer := 2;
  PUSH_DATA_WIDTH_IN  : integer := 32;
  PULL_DATA_WIDTH_OUT : integer := 8; -- ...
);
port (
  -- input
  push_in       : in CORBA_bit;
  addr : in CORBA_lv(OPERATION_WIDTH-1 downto 0);
  push_data_in  : in CORBA_lv(PUSH_DATA_ WIDTH_IN-1 downto 0);
  -- output
  pull_out      : in CORBA_bit;
  pull_data_out : out CORBA_lv(PULL_DATA_WIDTH_OUT-1 downto 0);  -- ...
);

// SystemC RTL
const int OPERATION_WIDTH     = 2;
const int PUSH_DATA_WIDTH_IN  = 32;
const int PULL_DATA_WIDTH_OUT = 8;
SC_MODULE( intf1 ) {
  // operation1
  sc_in< CORBA::bit >                        push_in;
  sc_in< CORBA::bv< PUSH_ADDR_WIDTH >  >addr;
  sc_in< CORBA::bv< PUSH_DATA_WIDTH_IN > >  push_data_in;
  sc_out< CORBA::bit >        pull_out;
  sc_out< CORBA::bv< PULL_DATA_WIDTH_IN > > pull_data_out;
  // ...
};
```

A.9.8 Addressable non-blocking push interface in input and blocking pull interface in output for several operations with parameters

If the interface `intf1` is defined as follows:

```
// OMG IDL3
interface intf1 {
  void operation1( );
  void operation2( in  long a1 );
  void operation3( out octet a2 );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is
generic (
  OPERATION_WIDTH     : integer :=2;
```

```
  PUSH_DATA_WIDTH_IN  : integer :=32;
  PULL_DATA_WIDTH_OUT : integer :=8; -- ...
);
port (
  -- input
  push_in       : in CORBA_bit;
  addr          : in CORBA_lv(OPERATION_WIDTH-1 downto 0);
  push_data_in  : in CORBA_lv(PUSH_DATA_ WIDTH_IN-1 downto 0);
  -- output
  pull_out      : in CORBA_bit;
  pull_data_out : out CORBA_lv(PULL_DATA_WIDTH_OUT-1 downto 0);
  pull_ack      : out CORBA_bit;
  -- ...
);

// SystemC RTL
const int OPERATION_WIDTH     = 2;
const int PUSH_DATA_WIDTH_IN  = 32;
const int PULL_DATA_WIDTH_OUT = 8;
SC_MODULE( intf1 ) {
  // operation1
  sc_in< CORBA::bit >                         push_in;
  sc_in< CORBA::bv< PUSH_ADDR_WIDTH >  >      addr;
  sc_in< CORBA::bv< PUSH_DATA_WIDTH_IN > >    push_data_in;
  sc_in< CORBA::bit >                         pull_out;
  sc_out< CORBA::bv< PULL_DATA_WIDTH_IN > >   pull_data_out;
  sc_out< CORBA::bit >                        pull_ack; // ...
};
```

The `pull_ack` signal acknowledges the validity of the output parameters.

A.9.9 Addressable non-blocking pull interface in input and non-blocking push interface in output for several operations with parameters

```
// OMG IDL3
interface intf1 {
  void operation1( );
  void operation2( in  long a1 );
  void operation3( out octet a2 );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity intf1 is
generic (
  OPERATION_WIDTH     : integer :=2;
  PULL_DATA_WIDTH_IN  : integer :=32;
  PUSH_DATA_WIDTH_OUT : integer :=8; -- ...
```

```
);
port (
  -- input
  pull_out      : out CORBA_bit;
  addr          : in CORBA_lv(OPERATION_WIDTH-1 downto 0);
  pull_data_in  : in CORBA_lv(PULL_DATA_WIDTH_IN-1 downto 0);
  -- output
  push_out      : out CORBA_bit;
  push_data_out : out CORBA_lv(PUSH_DATA_WIDTH_OUT-1 downto 0);  -- ...
);

// SystemC RTL
const int OPERATION_WIDTH     = 2;
const int PUSH_DATA_WIDTH_IN  = 32;
const int PULL_DATA_WIDTH_OUT = 8;
SC_MODULE( intf1 ) {
  // operation1
  sc_out< CORBA::bit >                        pull_out;
  sc_in< CORBA::bv< PUSH_ADDR_WIDTH > >       addr;
  sc_in< CORBA::bv< PUSH_DATA_WIDTH_IN > >    push_data_in;
  sc_out< CORBA::bit >                        push_out;
  sc_out< CORBA::bv< PULL_DATA_WIDTH_OUT > > push_data_out;
  // ...
};
```

The `pull_out` serves to read input parameters, for instance in the consumer model of communication.

A.10 Operation Invocation

A.10.1 Phases

We distinguish two phases: request for operation call and input parameters transfer, and response for operation return, error/exception and output parameters transfer. Each phase should have a two-phase handshake. An acknowledge is useful in SW for synchronization, but needless most of the time in HW (e.g. no acknowledge for write to a configuration register). This acknowledge should be configure for each operation.

A.10.2 Mapping solutions

We identify two main mapping solutions: 1) explicit IDL-to-VHDL mapping configuration file such as OCP interface configuration files with which hardware interfaces exactly satisfy communication needs, but more tedious and possible incompatibilities to be handled by adapter connectors, 2) fixed hardware interface and protocol adapted to specific constraints. This second solution is suitable for the definition of a fixed API as we will see for the Transceiver component.

A.10.3 Operation parameter transfer

A virtual hardware input port is a a set of VHDL entity I/O ports that is associated with one or several input parameters It implements a standard hardware interface (syntax) and protocol (semantics) e.g. push or pull, blocking or non-blocking, with/without data or flow control and addressable interface. This mapping should be flexible and extensible, because the nature of operation parameters is totally implicit in IDL: control flow, data flow or a mix, continuous or discontinuous, with or without flow control etc. The association between parameters and input ports is explicitly specified in a mapping configuration file. The component-oriented nature of CORBA can be used to define dedicated IDL interfaces for control, configuration or data-oriented aspects. The nature of communication could be expressed by new IDL keywords to better characterize CORBA component ports such as `buffered`, `addressable`, `stream` and infer hardware interfaces based on register, fifo, and RAM. When several input parameters are associated to an input port, the way input parameters are transferred on hardware interface signals should be explicit. The data bus width is specified in the mapping configuration file. It is not required to be a power of two. The endianness is configured to determine if transfers are LSB or MSB first. The data layout of parameters transfer shall also be configured.

Different transfer modes can be distinguished in particular: *Serial* or *Time-Division Multiplexing* (**TDM**), where one parameter (chunk) is transferred at a time, and *Parallel* or *Spatial-Division Multiplexing* (**SDM**) where (one chunk of) each parameter is transferred at a time [Bje05]. Serial transfers require less hardware wires therefore less silicium area and power consumption, but are slower. Parallel transfers require more hardware wires therefore more silicium area and power consumption, but are faster.

Different encoding rules can be distinguished in particular: *with padding* where non-significant data are inserted to align parameters on address boundaries as in CORBA CDR [OMG08b], and *without padding* where no data are inserted to align parameters as in ICE [HS08] [Hen04]. As in ASN.1 Packed Encoding Rules (PER) [IT02] support both alternatives. Unaligned data are more compact but are slower for middlewares to encode and decode. Aligned data are easier to encode and decode but use more bandwidth.

We propose to support all these solutions to best fit each particular application requirements in DRE systems. Designers should indicate to IDL-to-HDL compilers which one they want to use during mapping configuration. For both transfer modes and encoding rules, reusable connectors can be envisaged to convert parameter data from one configuration to another.

For instance, consider the following interface `intf1`:

```
// OMG IDL3
interface intf1 {
  void operation1( in octet a1, in short a2, in long a3 );
};
or similarly:
typedef struct s { octet a1; short a2; long a3; } myStruct;
  void operation1( in myStruct s1);
};
```

Hence, we identify two main solutions to transfer parameters: 1) a dedicated signal for each parameter whose width correspond to the size of the parameter data type. An appropriate naming convention is required to avoid a name clash. This solution is user-friendly and performant, but requires a lot of area.

```
interface Example {
  exception Err1{};
```

```
  exception Err2{};
  boolean op1( in boolean p1, out short p2, inout long p3) raises(Err1, Err2);
};
```

If blocking push interfaces are used, the mapped hardware interface may be:

```
entity Example is
port (
  op1_push_in  : in  CORBA_bit; -- request
  op1_ack_in   : out CORBA_bit;
  op1_p1       : in  CORBA_boolean;
  op1_p3_in    : in  CORBA_long;
  op1_push_out : out CORBA_bit; -- response
  op1_ack_out  : in  CORBA_bit; -- optional
  op1_p2       : out CORBA_short;
  op1_p3_out   : out CORBA_long;
  op1_return   : out CORBA_boolean;
  op1_exc      : out exception;
  op1_exc_id   : out CORBA_lv(op1ExcDW-1 downto 0);
);
end entity Example;
```

2) shared wires for all parameters going in the same direction whose width is fixed by a constant (e.g. VHDL generic). This may generalized to all parameters of all operations of an interface. This approach is the most generic, area-efficient, but less user-friendly as parameters are serialized.

```
entity Example is
generic (natural op1InPrmDW := 1);
port (
  op1_push_in  : in  std_logic; -- request \\
  op1_acc_in   : out std_logic; -- optional}
  op1_in       : in  std_logic_vector(op1InPrmDW-1 downto 0);
  op1_push_out : out std_logic;  -- response
  op1_ack_out  : in  std_logic; -- optional
  op1_out      : out std_logic_vector(op1OutPrmDW-1 downto 0);
  op1_return   : out CORBA_boolean;
  op1_exc      : out exception;
  op1_exc_id   : out std_logic_vector(op1ExcDW-1 downto 0);
);
end entity Example;
```

The data width used to transfer parameters in parallel or serial may be chosen at instantiation time. The endianness and alignment rules - with or without padding - should be defined. The VHDL data type e.g. record inferred from UML/IDL definitions could not be used in entity/component interface, but within VHDL implementation (architecture) where these data types should be serialized in a generic way on the shared data bus (solution 2). Each approach has its strengths and weaknesses. It is a trade-off between flexibility, complexity to guaranty portability/interoperability and performance (area, timing, power consumption).

The transfer of parameters should be *Globally Serial Locally Parallel* (**GSLP**). Indeed, request messages are typically globally serialized on buses, while operation parameters may be locally transferred in parallel to maximize performance and avoid a bottleneck regardless of the used serial or parallel bus. Adapter connectors would be in charge of serialization/deserialization between the bus interface and the hardware invocation interface.

A.10.4 Operation-level concurrency

Operation-level concurrency should be defined in a similar way to UML concurrency constraint, [GMTB04] and [JS07] for more efficiency. Groups of sequential operations are inferred as shared invocation signals acting as a virtual address, while parallel operations are inferred as dedicated invocation signals to enable independent invocations. For instance, we consider that all operations of the SCA Resource interface are sequentially invocable to allows consistency between mono-threaded and multi-thread SCA CoreFramework implementations.

A.11 Object

We assume that the interconnection between client and server objects is entirely handled by the underlying middleware in a transparent manner from the user's point of view.

The configuration of the middleware for a given application is made from an OMG D&C component deployment plan. This deployment plan specifies all object instances, their locations and connections.

To identify object instances, a CORBA::Object is defined by a system unique identifier encoded in an enumerated type. The enumerate is fixed thanks to the D&C component deployment plan and takes into account all component instances.

A.12 Inheritance

OMG IDL supports inheritance of IDL interfaces. An child interface can inherit operations from other interfaces. For instance:

```
// OMG IDL3
interface intf1 {
  void operation1( );
};
interface intf2 : intf1 {
  void operation2( in long a2, out octet a3 );
};
```

If four non-blocking push interfaces are chosen with the data bus width corresponding to parameters size, this is equivalent to the following VHDL declarations:

```
-- VHDL RTL
entity intf2 is port (
  operation1_push_in          : in  CORBA_bit;
  operation1_push_out         : out CORBA_bit;
  operation2_push_in          : in  CORBA_bit;
  operation2_push_data_in_a2  : in  CORBA_long;
  operation2_push_data_out_a3 : out CORBA_octet;
```

```
  operation2_push_out         : out CORBA_bit; -- ...
);
end comp1;

// SystemC RTL
SC_MODULE( comp1 ) {
  sc_in< CORBA_< CORBA::bit >    operation1_push_in;
  sc_out< CORBA_< CORBA::bit >   operation1_push_out;
  sc_in < CORBA_< CORBA::bit >   operation2_push_in;
  sc_in < CORBA_< CORBA::long >  operation2_push_data_in_a2;
  sc_out< CORBA_< CORBA::octet > operation2_push_data_out_a3;
  sc_out< CORBA_< CORBA::bit >   operation2_push_out; -- ...
};
```

If an addressable non-blocking push interface is chosen for the inputs and a non-blocking push interface for the outputs with 8bits data bus width, this is equivalent to the following VHDL declarations:

```
-- VHDL RTL
entity intf2 is
generic (
  OPERATION_WIDTH   : integer := 1;
  PUSH_DATA_WIDTH_IN  : integer := 8;
  PUSH_DATA_WIDTH_OUT : integer := 8; -- ...
);
port (
  operation1_push_in          : in  CORBA_bit;
  operation1_addr             : in  CORBA_lv(OPERATION_WIDTH-1 downto 0);
  operation1_push_data_in_a2  : in  CORBA_lv(PUSH_DATA_WIDTH_IN-1
downto 0);
  operation1_push_data_out_a3 : out CORBA_lv(PUSH_DATA_WIDTH_OUT-1
downto 0);
  operation1_push_out         : out CORBA_bit;  -- ...
); end;

// SystemC RTL
const int OPERATION_WIDTH     = 1;
const int PUSH_DATA_WIDTH_IN  = 8;
const int PUSH_DATA_WIDTH_OUT = 8;
SC_MODULE( comp1 ) {
  sc_in< CORBA_< CORBA::bit >                       operation1_push_in;
  sc_in < CORBA_< CORBA::lv<PUSH_DATA_WIDTH_IN> >  operation1_push_data_in_a2;
  sc_out< CORBA_< CORBA::lv<PUSH_DATA_WIDTH_OUT> > operation1_push_data_out_a3;
  sc_out< CORBA_< CORBA::bit >                      operation1_push_out; -- ...
};
```

A.13 Interface Attribute

Consider the following attribute definitions in the interface `intf1`:

```
// OMG IDL3
interface intf1 {
  attribute short radius;
  readonly attribute long pos_x;
  readonly attribute long pos_y;  //....
};
```

An IDL compiler implicitly transforms this interface into an *equivalent* interface [OMG08b]:

```
// OMG IDL3
interface intf1 {
  short _get_radius();
  void  _set_radius( in short r );
  long  _get_pos_x();
  long  _get_pos_y();
};
```

We do not consider the leading underscore. The language mapping for attributes is then the same as for the mapping of the equivalent operations. For instance:

```
-- VHDL RTL
entity intf1 is port (
  get_pos_x           : in  CORBA_bit;
  get_pos_x_out_data  : out CORBA_long;
  get_pos_y           : in  CORBA_bit;
  get_pos_y_out_data  : out CORBA_long;
  get_radius          : in  CORBA_bit;
  get_radius_out_data : out CORBA_short;
  set_radius          : in  CORBA_bit;
  set_radius_r        : in  CORBA_short;
  set_exception       : out CORBA_exception; -- ...
);
end comp1;

// SystemC RTL
SC_MODULE( comp1 ) {
  // facet1 intf1 attributes
  sc_in< CORBA::bit >        get_pos_x;
  sc_out< CORBA::long >      get_pos_x_out_data;
  sc_in< CORBA::bit >        get_pos_y;
  sc_out< CORBA::long >      get_pos_y_out_data;
  sc_in< CORBA::bit >        get_radius;
  sc_out< CORBA::short >     get_radius_out_data;
  sc_in< CORBA::bit >        set_radius;
  sc_in< CORBA::short >      set_radius_r;
  sc_out< CORBA::exception > set_exception;
  // ...
 };
```

If an accessor signal fails to set the attribute value, then the entity should return a standard CORBA exceptions number as defined in [OMG08b] through the `set_exception` signal.

A.14 Component Feature

A.14.1 Component

The IDL3 component construct is mapped as an entity in VHDL and as a module in SystemC.

```
// OMG IDL3
component intf1 {
  // ...
};

-- VHDL RTL
entity intf1 is port (
  clk : in std_logic;
  rst : in std_logic;  -- ...
);
end intf1;

// SystemC RTL
SC_MODULE( intf1 ) {
  sc_in< sc_logic >  clk;
  sc_in< sc_logic >  rst;
};
```

As mentioned for objects in [Rad00], a hardware client component needs to identify the server component instance with which it wants to communicate. We propose to identify component instances by a system unique identifier encoded in an enumerated type as it appears in the D&C component deployment plan. This identifier is a static VHDL generic parameter which is configured by the IDL compiler during hardware component instantiation. During method invocation, the servant component identifier is transferred onto a signal called `component_id`. The width of this signal corresponds to the number of bits required to binary encode the total number of components in the system. This number is not expected to be huge. To simplify hardware component implementation, responses to method invocations are assumed to be strictly ordered.

```
// PIDL
#define component_nb_width 8
interface Component {
  unsigned int<component_nb_width> component_id;
};
-- VHDL RTL
entity hardware_client_component is
generic(
  COMPONENT_NB_WIDTH : natural := 8;
);
port(
  component_id : out std_logic_vector(COMPONENT_NB_WIDTH-1 downto 0);
);
end;
```

```
entity hardware_server_component is
generic(
  COMPONENT_NB_WIDTH:  natural:= 8;
);
port(
  component_id : in std_logic_vector(COMPONENT_NB_WIDTH-1 downto 0);
);
end;
```

A.14.2 Facet

In the CORBA component model, a facet denotes a provided component port. For instance, the facet called `facet1` is defined in the component `comp1` as follows:

```
// OMG IDL3
component comp1 {
  provides intf1 facet1;
};
```

In the HDL mapping, the name of the facet is used added at the beginning of the mapped hardware interface signals. For instance, if the interface `intf1` is defined as:

```
// OMG IDL3
interface intf1 {
  // unreliable oneway
  oneway void operation1( );
};
```

The mapped hardware interfaces are defined as:

```
-- VHDL RTL
entity comp1 is port (
  facet1_operation1_push_in : in CORBA_bit;  -- ...
);
end intf1;

// SystemC RTL
SC_MODULE( comp1 ) {
  sc_in < CORBA::bit > facet1_operation1_push_in;  // ...
};
```

A.14.3 Receptacle

In the CORBA component model, a receptacle denotes a uses component port. For instance, the receptacle called `recep1` is defined in the component `comp1` as follows.

```
// OMG IDL3
component comp1 {
  uses intf1 recep1;  //....
};
```

The mapping is the same as for the facets with the inverse signal direction for all interface signals other than clock and reset.

For instance with a non-blocking control interface for input (void) and a non-blocking control interface for output (void): one operation without parameters in the IDL interface:

```
-- VHDL RTL
entity comp1 is port (
  recep1_operation1_push_out : out CORBA_bit;
  recep1_operation1_push_in  : in  CORBA_bit; -- ...
);
end comp1;

// SystemC RTL
SC_MODULE( comp1 ) {
  // ...
  // recep1_operation1
  sc_out< CORBA_< CORBA::bit > recep1_operation1_push_out;
  sc_in < CORBA_< CORBA::bit > recep1_operation1_push_in;
};
```

A.15 Not Supported Features

The following CORBA features are not supported at the moment by our IDL-to-HDL mapping: Pseudo-objects, Portable Object Adapter, Dynamic Invocation and Skeleton Interfaces and Events. The functionalities of the Portable Object Adapter and its associated interfaces are delegated to the underlying middleware. The Dynamic Invocation and Skeleton Interfaces are beyond the scope of this preliminary IDL-to-HDL mapping. For events, the basic idea is to apply the mapping rules corresponding to the event data type with a "send-and-forget" interaction semantics [SKH98]. Wide string are deliberately not supported by this IDL-to-HDL mapping.

A.16 Mapping Summary

Table A.1 provides a summary of the proposed IDL3-to-VHDL mapping.

IDL3 constructs	**VHDL constructs**
`module M { ... };`	`package M is ... end package M;`
`interface I { ... };`	`entity I is ... end entity I;` `component I ... end component I;`
`struct S { ... };`	`type S is record ... end record S;`
`enum E { E1; E2 };`	`type E is ( E1; E2 );`
`typedef B D;`	`subtype D is B;`

`typedef sequence<I> ISeq;`	`type ISeq is record`
	
	`length: ushort;`
	`item : I;`
	`end record ISeq;`

`const ushort C = 1;`	`constant C : natural := 1;`
`exception Err;`	`constant ExcDW : natural := 3;` `constant Err: std_logic_vector (ExcDW-1 downto 0) := "000";`
`readonly attribute short ROA;` or `void getROA(in short ROA);`	`ROA: out CORBA_short;`
`writeonly`[78] `attribute long WOA;` or `void setA(in ulong A);`	`WOA: in CORBA_long;`
`attribute short A;`	`A_in: in CORBA_short;` `A_out: out CORBA_short;`

`component C {`
`provides I1 P1;`
`uses I2 P2;`
`};`
	`entity C is ... end entity C;`
	`component I ... end component I;`
	`P1_I1_signal1 : in T1; ...`
	`P2_I2_signal1 : out T2; ...`

`exception Err1{};`
`exception Err2{};`
`boolean op1(`
` in boolean p1,`
` out short p2,`
 	
` inout long p3)`
`raises(Err1, Err2);`
	`op1_rq: in std_logic; - request`
	`op1_rq_acc: out std_logic; - optional`
	`op1_p1: in CORBA_boolean;`
	`op1_p3_in: in CORBA_long;`
	or `op1_in: in std_logic_vector(op1InPrmDW-1 downto 0);`
	`op1_rp: out std_logic; - response`
	`op1_rp_acc: in std_logic; - optional`
	`op1_p2: out CORBA_short;`
	`op1_p3_out: out CORBA_long;`
	or `op1_out: out std_logic_vector(op1OutPrmDW-1 downto 0);`
	`op1_return: out CORBA_boolean;`
	`op1_exc: out exception;`
	`op1_exc_id: out std_logic_vector(op1ExcDW-1 downto 0);`

Table A.1: Summary of the proposed IDL3-to-VHDL mapping

Appendix B

Personal Bibliography

This section contains the list of the publications written during this PhD work.

1. Gailliard, G.; Balp, H.; Jouvray, C.; Verdier, F., "Towards a Common HW/SW Interface-Centric and Component-Oriented Specification and Design Methodology", in Proceedings of the Forum on Specification and Design Languages (FDL), pp. 31-36, 23-25 September 2008, Stuttgart, Germany.

2. Gailliard, G.; Balp, H.; Sarlotte, M.; Verdier, F., "Mapping Semantics of CORBA IDL and GIOP to Open Core Protocol for Portability and Interoperability of SDR Waveform Components", International Conference on Design and Test in Europe (DATE), pp. 330-335, 10-14 March 2008, Munich, Germany.

3. Manet, P.; Maufroid, D.; Tosi, L.; Gailliard, G.; Mulertt, O.; Di Ciano, M.; Legat, J.; Aulagnier, D.; Gamrat, C.; Liberati, R.; La Barba, V.; Cuvelier, P.; Rousseau, B.; and Gelineau, P. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. EURASIP Journal on Embedded Systems. 2008 , pp. 1-11., 3 November 2008.

4. Gailliard, G.; Nicollet, E.; Sarlotte, M.; Verdier, F., "Transaction Level modeling of SCA Compliant Software Defined Radio Waveforms and Platforms PIM/PSM", International Conference on Design and Test in Europe (DATE), pp. 1-6, 16-20 April 2007, Nice, France.

5. Gailliard, G.; Mercier, B.; Sarlotte, M.; Candaele, B.; Verdier, F., "Towards a SystemC TLM based Methodology for Platform Design and IP reuse: Application to Software Defined Radio", Second European Workshop on Reconfigurable Communication-Centric SoCs (RECOSOC), 3-5 July 2006, Montpellier, France.

This section contains the list of the project deliverables in which I participated presented during this PhD work.

1. WINTSEC (Wireless INTeroperability for SECurity), State on the art on hardware component models, October 2007.

2. Watine, V.; Seignole, V.; Balp, H. and G. Gailliard, "IDL-to-VHDL Mapping Initial Requirements", OMG Washington Technical Meeting, March 2008.

3. RECOPS (Reconfiguring Programmable Devices for Military Hardware Electronics) The RECOPS project is an EDA project no 05/102.016/010

4. SPICES (Support for Predictable Integration of mission Critical Embedded Systems) . SPICES is an EUREKA-ITEA project and runs from 01/09/2006 until 31/08/2009.

5. MOSART (Mapping Optimization for Scalable multi-core ARchiTecture) Project (IST-215244) Website (ITEA)

6. Nicollet, E.; Pothin, S.; Gailliard G.; Caron B.; Ruben, Transceiver Facility Specification, SDRF-08-S-0008-V1.0.0, January, 2009, SDR Forum.

This section contains the list of the presentations presented during this PhD work.

1. Gailliard, G., "Déclinaison matérielle du canevas logiciel SCA", Poster presented at Ecole d'hiver Francophone sur les Technologies de Conception des systèmes embarqués Hétérogènes (FETCH'07), 10-12 January 2007, Villard-de-Lans, France.

2. CNRS Groupement de Recherche (GDR) Information, Signal, Images, viSion (ISIS), C theme, "Outils, méthodes pour les Traitements Du Signal et de l'Image (TDSI)", September 2006

Appendix C

Résumé Etendu

Cette thèse s'intéresse à la déclinaison matérielle des concepts logiciels d'intergiciel et d'architecture logicielle à base de composants, conteneurs et connecteurs dans les réseaux de portes programmables in situ (*Field-Programmable Gate Array* - **FPGA**). Le domaine d'applications ciblé est la radio définie logiciellement ou plus généralement radio logicielle (*Software Defined Radio* (**SDR**)).

Une radio logicielle est une radio reconfigurable dont les fonctionnalités sont définies logiciellement au plus proche de l'antenne. Elle est capable d'exécuter un ensemble d'applications de forme d'onde sur la même plateforme radio. Une seule application est exécutée à la fois. Une application de forme d'onde est une application logicielle à base de composants. Ces composants implémentent notamment des algorithmes de traitement du signal, qui traitent les données reçues de l'antenne vers l'utilisateur final et vice versa.

Une plateforme radio est un ensemble de couches logicielles et matérielles fournissant les services requis par la couche applicative de forme d'onde.

Au delà de la vision idéale d'une radio entièrement logicielle, les traitements numériques du signal de certaines applications de formes d'onde requièrent néanmoins une forte puissance de calcul, que ne sont pas capables de fournir les processeurs classiques (ARM, PowerPC, etc). Ils ont donc besoin d'unités de traitements dédiées telles que les processeurs de traitements du signal (*Digital Signal Processors* - **DSPs**), les FPGAs et les circuits intégrés spécifiques à une application (*Application Specific Integrated Circuits* - **ASICs**).

Nous nous intéressons plus particulièrement aux radios logicielles conformes au canevas d'architecture logicielle appelé *Software Communications Architecture* (**SCA**). Le SCA est un standard de-facto de la radio logicielle militaire, mais il peut tout à fait être utilisé pour des équipements civils. De plus, les principaux besoins du SCA sont **portabilité** , **interoperabilite**, reutilisation et reconfiguration des composants SCA.

Les principaux besoins de la SDR et du SCA sont: la **portabilité** des applications de forme d'onde parmi différentes plateformes radios, l'**interopérabilité** des communications hétérogènes entre ses composants logiciels et matériels, leur **réutilisabilité** et la **reconfigurabilité** de la plateforme radio pour exécuter un ensemble de formes d'ondes, qui peuvent avoir des contraintes, notamment temps-réels, différentes.

Le SCA spécifie un environnement d'exploitation (ou *Operating Environment* **(OE**)) pour gérer les applications radios et offrir les services extra-fonctionnels de la plateforme radio. Cet OE contient un ensemble d'interfaces logicielles appelé *Core Framework*, un intergiciel conforme à *Common Objet Request Broker Architecture*) (**CORBA**) pour l'embarqué (*CORBAe* anciennement appelé *Minimum CORBA*) et un système d'exploitation (*Operating System* - **OS**) conforme avec les interfaces **POSIX** (*Portable Operating System Interface*). L'OE SCA cible les processeurs généraux, mais n'est pas adapté

pour les unités de traitement dédiées comme les DSPs, FPGAs et ASICs.

Les besoins de portabilité et de réutilisation des composants SCA requièrent que leurs interfaces abstraites définies à un niveau système soient indépendantes d'une implémentation logicielle ou matérielle et puissent être indifféremment traduites dans un langage de programmation logiciel tel que C/C++, un langage système tel que SystemC au niveau transaction , ou un langage de description matériel tel que VHDL ou SystemC au niveau registre (*Register Transfer Level* - **RTL**).

La modélisation au niveau transactionnel (*Transaction Level Modeling* - **TLM**) consiste à modéliser les communications d'un système à différents niveaux d'abstraction en utilisant des appels de fonctions, plutôt que des signaux matériels comme dans les simulateurs au niveau RTL, afin d'augmenter la vitesse de simulation de systèmes embarqués mêlant logiciel et matériel,

Le but ambitieux poursuivi par cette thèse est de décliner les concepts du génie logiciel, notamment utilisés par le SCA, aux FPGAs afin de proposer une approche commune de spécification et d'implémentation des applications de radio logicielle à base de composants logiciels/matériels. Ces concepts sont issus principalement de l'architecture logicielle à base de composants et des intergiciels.

Habituellement, l'intégration de modules matériels dans les FPGAs s'appuie sur des interfaces logicielles et matérielles bas niveau, composées d'interfaces de bus propriétaires, de registres mappés en mémoire et d'interruption pour la synchronisation entre les modules logiciels et matériels. Cette intégration est fastidieuse et source d'erreurs. De plus, la réutilisabilité de ces modules est limitée. Des méthodologies de conception conjointe pour le logiciel et le matériel ont donc été proposées pour réduire le fossé conceptuel entre conception logicielle et conception matérielle, et ainsi faciliter cette intégration.

Etant donné que le SCA est fondamentalement basé sur le modèle objet, nous nous sommes particulièrement intéressés par l'application des concepts du modèle objet à la conception matérielle.

Dans l'approche orientée objet, un objet est une entité identifiable, qui a un état et un comportement, qui lui sont propres. Ce comportement n'est accessible qu'au travers d'une interface à base de signatures d'opérations. L'appel d'une opération déclenche le comportement associé et est équivalent à l'envoi d'un message contenant l'identifiant et les paramètres de l'opération.

Pour appliquer le modèle objet à du matériel synthétisable dans les FPGAs, les travaux précédents proposent un mapping d'une interface logicielle à base d'opérations à une interface matérielle à base de signaux ou "fils". Cependant, ces travaux ne représentent qu'une solution dans l'espace d'exploration du mapping. De plus, l'approche objet a quelques limitations. Un objet ne présente pas plusieurs, mais une seule interface fournie, la plus dérivée, même s'il hérite de multiple interfaces. Les dépendances d'un objet sont implicites car aucune interface n'est requise, et il n'y a pas de séparation claire entre logique fonctionnelle et non-fonctionnelle telle que la communication entre objets et le déploiement d'objets.

Alors que la plupart des travaux sur l'application des concepts logiciels à la conception matérielle sont basés sur une approche orientée objet, nous allons plus loin en considérant l'application de l'architecture logicielle orientée composant, qui est un autre aspect du SCA.

L'approche orientée composant s'appuie sur quatre principaux modèles: les modèles de composants définissant la structure et le comportement des composants, les modèles de connecteurs pour spécifier les interactions entre les composants, les modèles de conteneurs pour accéder notamment aux services d'intergiciels, et les modèles de déploiement et d'archivage pour déployer les applications à base de composants distribués. Nous avons naturellement décliné ces concepts pour les composants matériels.

Nous avons investigué l'implémentation matérielle de composants abstraits orienté objet. Nous défendons l'idée que l'approche composant fournit de meilleurs concepts unifiant pour les systèmes embarqués mêlant logiciel et matériel, que l'approche objet seule. Nous proposons un état de l'art sur l'application des concepts d'objet, de composant et d'intergiciel à la conception matérielle.

Pour permettre le raffinement systématique d'interfaces de composants orientés objet définis dans le langage de définition d'interface (IDL) de l'intergiciel CORBA ou le langage de modélisation unifiée UML, en interfaces de composants systèmes, logiciels et matériels, nous proposons de définir de nou-

veaux mappings de langages de CORBA IDL vers SystemC au niveau transactionnel ou au niveau registre, et VHDL, lui aussi au niveau registre. L'exigence de portabilité du SCA est adressée par ces mappings de langages. Notre mapping d'IDL à SystemC au niveau fonctionnel considère surtout SystemC comme un langage et non comme une extension du C++ pour le mapping des concepts de composant et de connecteur. Pour le reste, il s'appuie sur la mapping au niveau RTL (e.g. types de données) et à défaut sur le mapping existant entre IDL et C++ par héritage des interfaces de la librairie SystemC.

Le mapping d'IDL à SystemC au niveau transactionnel s'appuie sur des adapteurs d'interfaces appelés transacteurs. Nous considérons ces transacteurs comme des connecteurs transactionnels, qui adaptent le mapping des interfaces utilisateurs en IDL/UML aux interfaces standards TLM 1.0/2.0.

Pour le mapping d'IDL à VHDL et SystemC RTL, nous avons formulé des exigences pour un mapping standard IDL et les langages de description matériel. Notre mapping IDL/HDL consiste en une allocation manuelle ou semi-automatique d'éléments mémoire (registre, FIFO, RAM/ROM), d'interfaces matérielles et de protocoles, à des abstractions logicielles telles que composant, port, interface, opérration, paramètre et connecteur. Cette approche permet un mapping générique, flexible, configurable, mais plus complexe à mettre en oeuvre. L'implémentation de composants SCA matériels montre la nécessité d'une implémentation optimisée d'un modèle de composant logiciel.

De plus, nous avons prototypé un compilateur IDL3-VHDL en utilisant les technologies d'ingénierie dirigée par les modèles en utilisant des transformations *Query View Transform* (**QVT**) du meta-modèle *Lightweight CORBA Component Model* (LwCCM) étendu pour la mapping RTL, vers un meta-modèle VHDL, et des transformations modèle à texte pour la génération de code VHDL avec *MOF2Text*.

Le besoin d'interopérabilité du SCA a été adressé en prototypant un intergiciel matériel utilisant de façon transparente le mapping mémoire et deux protocoles de messages: CORBA *General Inter-Object Request Broker Protocol* (**GIOP**) et le protocole *Modem Hardware Abstraction Layer* (**MHAL**) du SCA. Nous avons proposé un canevas d'architecture pour les intergiciels matériels en appliquant les patrons de conception de la littérature sur les intergiciels logicielles. Ce canevas est basé sur cinq couches. La couche applicative contient les composants applicatifs matériels. La couche présentation de données réalise l'encodage/décodage des corps de message. La couche de routage route les messages entrants et arbitre les messages sortants. La couche de messagerie traite de l'encodage/décodage des entêtes de message, de l'encapsulation des corps de messages, et de la machine d'état du protocole. La couche transport offre une interface matérielle de communication générique et des tampons mémoires pour les messages. Ce canevas d'architecture intègre aussi les concepts de personnalités applicatives et protocolaires. Une personnalité applicative correspond à un modèle de composants e.g. SCA ou CCM, tandis qu'une personnalité protocolaire correspond à un protocole e.g. SCA MHAL et CORBA GIOP. Les résultats expérimentaux montrent qu'un adressage mémoire classique et qu'un format de message spécifique à l'embarqué sont moins coûteux que GIOP, qui n'a pas été conçu pour l'embarqué temps-réel.

List of Figures

List of Tables

Listings

Acronyms

3G	Third generation of mobile telecommunications standards
4G	Fourth generation of mobile telecommunications standards
ADC	Analog-to-Digital Converter
AHB	AMBA High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
ANSA	Advanced Network Systems Architecture
ANSI	American National Standards Institute
AMI	Asynchronous Method Invocation
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
ASN.1	Abstract Syntax Notation One
BFM	Bus Functional Model
BSP	Board Support Package
CBSE	Component-Based Software Engineering
CCDR	Compact Common Data Representation
CCM	CORBA Component Model
CDR	Common Data Representation
CF	Core Framework
CLB	Configurable Logic Block
CORBA	Common ORB Architecture
COTS	Commercial Off-The-Shelf
DAC	Digital-to-Analog Converter
DDC	Digital Down Converter
DPR	Dynamic Partial Reconfiguration
DRE	Distributed Real-time Embedded
DSL	Domain-Specific Language
DSML	Domain-Specific Modeling Language
DSP	Digital Signal Processor or Digital Signal Processing
DTD	Document Type Definition
EDA	Electronic Design Automation
EJB	Enterprise JavaBean
EMF	Eclipse Modeling Framework
ESIOP	Environment Specific Inter-ORB Protocol
ESL	Electronic System Level
FE	Front-End
FEC	Forward Error Correction
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FSMD	Finite State Machine with Datapath

GIOP	General Inter-ORB Protocol
HDL	Hardware Description Language
ICAP	Internal Configuration Access Port
IDL	Interface Definition Language
IOP	Inter-ORB Protocol
IOR	Interoperable Object Reference
IP	Intellectual Property or Internet Protocol
IPC	Inter-Process Communication
IRQ	Interrupt Request
ISS	Instruction Set Simulator
JET	Java Emitter Templates
JPEO	Joint Program Executive Office
JTRS	Joint Tactical Radio System
LD	Logical Destination
LSB	Least Significant Bit
LUT	Look-Up Table
MAC	Medium Access Control
MDA	Model-Driven Architecture ®
MDE	Model-Driven Engineering
MHAL	Modem Hardware Abstraction Layer
MOF	Meta-Object Facility
MOM	Message-Oriented Middleware
MPI	Message Passing Interface
MPSoC	Multi-Processor System-on-Chip
MSB	Most Significant Bit
MUX	Multiplexers
NoC	Network-on-Chip
OCP	Open Core Protocol
OCP-IP	OCP International Partnership
OE	Operating Environment
OFDM	Orthogonal Frequency-Division Multiplexing
OMA	Object Management Architecture
OMG	Object Management Group
OPB	On-Chip Peripheral Bus
ORB	Object Request Broker
OSCI	Open SystemC Initiative
OSI	Open Systems Interconnection
P^2SRC	PIM & PSM for Software Radio Components
PA	Power Amplifier
PDU	Protocol Data Unit
PER	Packed Encoding Rules
PIM	Platform-Independent Model
PLB	Processor Local Bus
PSM	Platform-Specific Model
POSIX	Portable Operating System Interface (for Unix)
QoS	Quality of Service
QVT	Query / Views / Transformations
RAM	Random Access Memory
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTOS	Real-Time Operating System
SCA	Software Communications Architecture

SDM	Space Division Multiplexing
SDR	Software-Defined Radio
SDRP	Software-Defined Radio Profile
SLD	System-Level Design
SoC	System On Chip
SoPC	System On Programmable Chip
SPIRIT	Structure for Packaging, Integrating and Re-using IP within Tool flows
SysML	Systems Modeling Language
TDM	Time-Division Multiplexing
TLM	Transaction Level Modeling
UML	Unified Modeling Language
VC	Virtual Channel
VCI	Virtual Component Interface
VHDL	VHSIC Hardware Description Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Glossary

Abstract: an abstract class typically cannot be instantiated and delegates the implementation of methods to concrete classes.

Abstraction: an abstraction is a relative distinction made between relevant and less relevant details of a problem domain to enhance understanding and reduce complexity.

Access transparency: access transparency masks differences in data representation and invocation mechanisms to enable interworking between objects.

Aggregation: the aggregation relationship refers to a strong containment by value in UML.

Application Binary Interface (ABI): an ABI corresponds to the instructions set and calling convention of real and virtual machines.

Application Programming Interface (API): an API is a set of operations which are directly called by developers according to some use rules.

Asynchronous Call: asynchronous calls allow a caller to initiate an invocation that immediately returns and continues its processing, whereas the invocation is concurrently performed. Caller execution and callee invocation are thus decoupled, while the latency of communication and callee computation are masked.

Class: A class describes the common characteristics shared by a set of objects in term of structure and behavior [BME+07]. In programming languages, a class defines the *type* of an object.

Client: any entity that requires the services provided by another entity called the server.

Component: "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A component can be deployed independently and is subject to composition by third parties." [SP96]

Component Model: "A **component model** defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model." [Kra08] The component model imposes design constraints on component developers, and the component framework enforces these constraints in addition to providing useful services." [BBB+00]

Composition: the composition relationship refers to weak containment by reference in UML.

Connector: In software architecture, a software connector is a component which binds the required and provided ports of two or more components. It mediates component interactions and may deal with non-functional services such as communication, coordination, conversion, synchronization, arbitrating, routing, and scheduling. While components have ports, the interface access points of connectors are called roles.

Container: A container allows the isolation of a component from the technical details of its possibly numerous operating environments. Containers transparently integrate components into their execution environment. They use middleware services such as persistence, event notification, transactions, load-balancing and security as proxy of components. Containers also manage component life cycle such as creation/destruction using component factories, interconnection of component ports, configuration using configuration files typically in XML and activation e.g. using a timer service.

Encapsulation: encapsulation consists in hiding the internal view of an abstraction in order to better deal with its complexity [BME+07]. It denotes the separation of concerns between an interface and its implementation.

Exception: An *exception* is an user-defined or system error raised by a method in object-oriented languages.

Globally Asynchronous Locally Synchronous: systems on chip in which IP cores do not shared a global clock, but may share a local clock forming "synchronous islands".

Hierarchy establishes an ordering among a set of abstractions [BME+07]. The decomposition of a system into hierarchical layers of abstraction is a key concept to design complex systems.

Inheritance: the **inheritance** principle consists in deriving the declaration and possibly implementation of a *derived* or *child* class from a *base* or *parent* class. The child class includes by default the data and behavior of the parent class. The child class can be either based or replace (*override*) the behavior of the inherited behavior. The derived class may also declare additional attributes and methods.

Interoperability: Interoperability is the ability of applications to interoperate (i.e. understand each other) and perform system functionalities although they are executed using different operating environments.

IP core: an Intellectual Property (IP) core is a reusable hardware module in an ASIC or FPGA.

Location transparency provides a logical view of naming, independent of actual physical location.

Message: A message is equivalent to a method invocation. The transfer of control and data of a method invocation, which is implicit and local to a computer, can be formally translated into a remote and explicit transfer of a message between distributed computers and vice versa.

Meta-Model: A meta-model is a model of a modeling language. It specifies the concepts of the language used to define a model. A meta-model defines the structure, relationships, semantics and constraints of these concepts for a family of models conforming to this meta-model. Each model can be captured by a particular meta-model.

Meta Object Facilities (MOF): MOF is a meta-model and modeling language standardised by the OMG to describe OMG meta-models such as UML and CCM meta-models.

Method: A method describes the actions to perform depending on the type of invocation message received by the object. The term *method* originates from Smalltalk, whereas the terms *operation* and *member function* are respectively employed in Ada and C++ [BME+07]. In UML, a method is an operation implementation that specifies the algorithm associated with an operation.

Method Dispatching: *Method dispatching* is the process of selecting a method based on a request message. *Dynamic dispatching* refers to the process of determining at run-time which method to invoke.

Method Invocation: A *method invocation* requests the execution of a method.

Middleware: A middleware is communication infrastructure which aims at supporting transparent communications between heterogeneous systems regardless of their distribution, hardware computing architecture, programming language, operating system, transport layer and data representation.

Model of Computation (**MoC**): a model of computation is the formal abstraction of the execution in a computer. A model of computation defines the behavior and interaction semantics that govern the elements in different parts or at different levels of a system.

Modularity: modularity consists in breaking down a complex system into a set of simple and more manageable modules [BME+07].

Model: a model represents the key characteristics of a problem and provides a conceptual framework to reason about it. A model is a formal specification of the function, structure and/or behavior of an application or system.

Model-Driven Architecture (MDA): the Model-Driven Architecture (MDA) [OMG03] [MSUW04] is the Model-Driven Engineering approach standardized by the OMG. The MDA promotes the separation between the specification of system functionalities captured in a *Platform Independent Model* (**PIM**)

from the specification of their implementation on a particular platform defined in the *Platform Specific Model* (**PSM**).

Model transformation: model transformation consists in translating a source model M1 conforming to the meta-model MM1 into a model M2 conforming to the meta-model MM2 using mapping rules. Mapping rules are defined at the meta-model level and are not dedicated to a particular model.

Monitor: A monitor protects a shared resource by guarantying that the execution of all operations accessing the resource are mutually exclusive. Instead of managing explicitly critical sections of code using locks, monitor declarations are used by compiler to transparently introduce synchronization code.

Object: An object is a self-contained and identifiable entity, which encloses its own state and behavior and communicates only through message passing [Arm06] [BME+07]. The state of an object corresponds to the set of values of all static and dynamic properties or *attributes* of an object [BME+07]. The behavior of an object takes the form of a set of methods in the object class.

Open Core Protocol (**OCP**): OCP is a hardware interface specification maintained by the OCP-IP organization created in 2001 by members such as TI, MIPS, Sonics, Nokia and ST. OCP supports communication protocols ranging from simple acknowledge to multiple out-of-order pipelined concurrent block transfers. The OCP promotes a socket based layered architecture [OIb].

Operation: In UML and CORBA, an operation declares a service that can be provided by class instances.

Operation Call: An operation call is the invocation of a service.

Operation Signature: an operation signature contains the name of the operation, the type and name of its input, output, input-output parameters, the type of the return value and the name of errors or exceptions which may be raised.

Persistent object: a persistent object is saved before it is destroyed and restored in another class instance at its next creation or "reincarnation". Objects that are not persistent are called *transient* objects.

Platform-Based Design (PBD): Platform-Based Design consists in mapping an application model and a platform model successively through various abstraction levels. The result of a mapping of an application to a platform model at level N is the application model at the level N+1 that will be mapped on the platform model at level N+1.

Polymorphism: polymorphism is refers to "the ability to substitute objects of matching interface for one another at run-time" [GHJV95].

Portability: portability denotes the property of an application, which can be migrated or ported from one operating environment to another one with little efforts in terms of time and cost. There are usually two kinds of portability: portability at *source code* level via *Application Programming Interface* (**API**) and portability at *binary code* level via *Application Binary Interface* (**ABI**).

Programming Model: a programming model defines how parts of an application communicate with one another and coordinate their activities [CSG98]. The two main programming models which have been traditionally identified for parallel computing architectures are shared memory and message passing. Other programming models include data parallel processing i.e. Single Program Multiple Data (SPMD), dataflow processing and systolic processing [CSG98].

Proxy: middleware interface compilers automatically generate **proxies** a.k.a. *surrogates* to make local and remote communications indistinguishable from the viewpoint of client and server objects.

Push model: model of interaction in which the transfer of control and/or data is initiated by the sender e.g. writes from a master to a slave in on-chip bus.

Pull model: model of interaction in which the transfer of control and/or data is initiated by the receiver e.g. reads from a master to a slave after an interruption in on-chip bus.

Query / Views / Transformations (QVT): MOF QVT is a model transformation language standardized by the OMG [OMG08e].

Remote Method Invocation (RMI): A Remote Method Invocation is a method invocation which takes place between distributed objects. In distributed systems, the systematic translation between method invocation and message passing is the cornerstone of *Remote Method Invocation* (**RMI**) used in object-oriented middlewares such as Sun Java RMI, Microsoft DCOM or OMG CORBA.

Semaphore: A semaphore is a non-negative integer shared variable that can be initialized to a value and then can only be modified by two operations called *wait* and *send* or *P* and *V* by its inventor Dijkstra. These operations respectively acquire and release a lock and decremente and incremente the semaphore value. If the semaphore value becomes zero, then the calling thread will wait until it acquires the semaphore. The initialization value is 0 or 1 for binary semaphores to protect a single shared resource or N for counting semaphores to protect a pool of N resources.

Server: any entity that provides the services required by another entities called the clients.

Skeleton: A server-side proxy called *skeleton* decodes a request message payload into the input and input-output parameters of a local method invocation called on the object implementation and conversely encodes the output and input-output parameters and the return value into a reply message payload.

Software Component: "A **software component** is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard." [Kra08]

Software Component Infrastructure: "A *software component infrastructure* or *framework*, is a software system that provides the services needed to develop, deploy, manage and execute applications built using a specific component model implementation. It may itself be organized as a set of interacting software components." [Kra08]

Software Interface: The set of operations signatures specifies an **interface** between the outside view of a class provided by its declaration and the inside view incarnated by the object implementation.

Stub: A client-side proxy called *stub* encodes the input and input-output parameters of a remote method invocation into a request message payload and conversely decodes a reply message payload into the output and input-output parameters and the return value. The request message payload is encapsulated by the request message header of the RMI protocol and the lower protocol layer

Synchronous Call: In synchronous calls, the caller thread of control is blocked until a reply is returned, while asynchronous RPCs do not block the caller [ATK92]. Synchronous calls are space and time coupled with partial synchronization decoupling.

Transaction: A transaction is a sequence of operations, which is either totally executed or not at all.

XML Metadata Interchange (XMI): XMI is an OMG standard for storing or *serializing* models and meta-models as persistent XML files. XMI is a mapping from MOF to XML. For instance, it allows the exportation and importation of UML models from/to UML modeling tools.

Bibliography

[ABC+07] R. Ben Atitallah, P. Boulet, A. Cuccuru, J.-L. Dekeyser, A. Honoré, O. Labbani, S. Le Beux, P. Marquet, E. Piel, J. Taillard, and H. Yu. Gaspard2 UML profile documentation. Technical Report 0342, INRIA Futurs, Lille, France, September 2007.

[Acc04a] Accellera. Property Specification Language Reference Manual, v.1.1. `www.eda.org/sv`, June 2004.

[Acc04b] Accellera. SystemVerilog 3.1a Language Reference Manual (LRM). `www.eda.org/sv`, May 2004.

[AHMMB07] D. Akehurst, G. Howells, K. McDonald-Maier, and B. Bordbar. An Experiment in Using Model Driven Development : Compiling UML State Diagrams into VHDL. In *Forum on specification and design languages (FDL'07)*, pages 88–93, Septembre 2007.

[AKL+09] Denis Aulagnier, Ali Koudri, Stéphane Lecomte, Philippe Soulard, Joël Champeau, Jorgiano Vidal, Gilles Perrouin, and Pierre Leray. SoC/SoPC development using MDD and MARTE profile. In *Model Driven Engineering for Distributed Real-Time Embedded Systems*. Hermes, 2009.

[Alt] Altera. C-to-Hardware. `www.altera.com`.

[Alt07a] Altera. Avalon Memory-Mapped Interface Specification v3.3. `www.altera.org`, May 2007.

[Alt07b] Altera. Nios II C2H Compiler User Guide, v7.2, 2007.

[And00] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[AND08] ANDRES (Analysis and Design of run-time Reconfigurable, heterogeneous Systems) IST-FP6 Project, June 2006 - May 2009. `andres.offis.de`, June 2008.

[APS+07] E. Anderson, W. Peck, J. Stevens, J. Agron, F. Baijot, S. Warn, and D. Andrews. Supporting High Level Language Semantics within Hardware Resident Threads. *17th International Conference on Field Programmable Logic and Applications (FPL'07)*, pages 98–103, August 2007.

[AR98] P. J. Ashenden and M. Radetzki. A Comparison of SUAVE and Objective VHDL, Report for the IEEE DASC Object-Oriented VHDL Study Group, December 1998.

[ARM04] ARM. AMBA AXI Protocol Specification v1.0, March 2004.

[Arm06] D. J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, February 2006.

[ATK92] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 26(2):92–109, 1992.

[AWM98] P. J. Ashenden, P. Wilsey, and D. Martin. SUAVE: Object-Oriented and Genericity Extensions to VHDL for High-Level Modeling. In *In Proceedings of Forum on Specification and Design Languages (FDL'98)*, pages 109–118, September 1998.

[AWM99] P. J. Ashenden, P. Wilsey, and D. Martin. Principles for Language Extensions to VHDL to Support High-Level Modeling. *VLSI Design*, 1999.

[Bál02] Dušan Bálek. *Connectors in Software Architectures*. PhD thesis, Charles University, Czech Republic, 2002.

[Bal07] P. J. Balister. A Software Defined Radio Implemented using the OSSIE Core Framework Deployed on a TI OMAP Processor. Master's thesis, Virginia Polytechnic Institute and State University, Dec. 2007.

[BBB+00] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. C. Wallnau. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute (SEI) Carnegie Mellon University (CMU), May 2000.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proc. of the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM '06)*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[BCS02] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proc. of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga, Spain, 2002.

[BDGC08] P. Bomel, J.-P. Diguet, G. Gogniat, and J. Crenne. Bitstreams Repository Hierarchy for FPGA Partially Reconfigurable Systems. In *Parallel and Distributed Computing, 2008. ISPDC '08. International Symposium on*, pages 228–234, July 2008.

[BDR07] P.J. Balister, C. Dietrich, and J.H. Reed. Memory Usage of a Software Communications Architecture Waveform. In *SDR Forum Technical Conference*, Denver, CO, November 2007.

[Beu07] S. Le Beux. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. PhD thesis, University of Lille, Laboratoire d'Informatique Fondamentale de Lille (LIFL), France, December 2007.

[BFS04] T. Beierlein, D. Fröhlich, and B. Steinbach. Model-driven compilation of UML-models for reconfigurable architectures. In *In Proc. of the Second RTAS Workshop on Model-Driven Embedded Systems (MoDES'04)*, 2004.

[BGG+07] C. Brunzema, C. Grabbe, K. Grüttner, P. A. Hartmann, A. Herrholz, H. Kleen, F. Oppenheimer, A. Schallenberg, C. Stehno, and T. Schubert. *OSSS - A Library for Synthesisable System Level Models in SystemC(TM) - A tutorial for OSSS 2.0*, November 2007.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48, Aug. 2006.

[BHS07] F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-Oriented Software Architecture (POSA) Vol. 4: A Pattern Language for Distributed Computing*. Wiley, May 2007.

[Bic07] J. Bickle. IDL to HDL Mapping Proposal, OMG Burlingame Technical Meeting. `www.omg.org/docs/mars/07-12-20.pdf`, December 2007.

[BJ07] John Bard and Vincent J. Kovarik Jr., editors. *Software Defined Radio: The Software Communications Architecture*. Wiley, May 2007.

[Bje05] T. Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2005.

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[BKH+07] A.V. Brito, M. Kuhnle, M. Hubner, J. Becker, and E.U.K. Melcher. Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC. In *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, pages 35–40, March 2007.

[Blu] Bluespec. Bluespec SystemVerilog (BSV). `www.bluespec.com/products/bsv.htm`.

[BM02] L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, January 2002.

[BM06] T. Bjerregaard and S. Mahadevan. A survey of research and practices of Network-on-Chip. *ACM Comput. Surv.*, 38(1):1, 2006.

[BMB+04] S. Bailey, E. Marschner, J. Bhasker, J. Lewis, and P. Ashenden. Improving Design and Verification Productivity with VHDL-200x. In *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, Washington, DC, USA, 2004. IEEE Computer Society.

[BME+07] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston. *Object-Oriented Analysis and Design with Applications, 3rd Edition*. Addison Wesley, Redwood City, CA, USA, April 2007.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture (POSA) Vol. 1: A System of Patterns*. Wiley, July 1996.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[Bon06] L. Bondé. *Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP*. PhD thesis, University of Lille, Laboratoire d'Informatique Fondamentale de Lille (LIFL), France, December 2006.

[BP04] T. Bures and F. Plasil. *Software Engineering Research and Applications*, chapter Communication Style Driven Connector Configurations, pages 102–116. Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2004.

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide Second Edition*. Addison Wesley, May 2005.

[BRM+06] J. Barba, F. Rincón, F. Moya, F. J. Villanueva, D. Villa, and J. C. López. Lightweight Communication Infrastructure for IP Integration. In *IP-Based SoC Design Conference (IP-SoC)*, December 2006.

[BRM+07] J. Barba, F. Rincon, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, and J. C. Lopez. OOCE: Object-Oriented Communication Engine for SoC Design. In *Proc. of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD'07)*, pages 296–302, Washington, DC, USA, Aug. 2007. IEEE Computer Society.

[BS05] Bruno Bouyssounouse and Joseph Sifakis. *Embedded Systems Design*, volume 3436 of *Lecture Notes in Computer Science*, chapter Component Models and Integration Platforms: Landscape, pages 160–193. Springer-Verlag New York, LLC, March 2005.

[BW05] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, pages 262–286, 2005.

[BW07] A. Burns and A. Wellings. *Concurrent and Real-time Programming in Ada 2005*. Cambridge University Press, 2007.

[BWH+03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, April 2003.

[Cap03] L. F. Capretz. A brief history of the object-oriented approach. *SIGSOFT Softw. Eng. Notes*, 28(2):6, March 2003.

[CCB+06] P. Coussy, E. Casseau, P. Bomel, A. Baganne, and E. Martin. A formal method for hardware IP design and integration under I/O and timing constraints. *ACM Trans. Embed. Comput. Syst.*, 5(1):29–53, 2006.

[CCSV07] Ivica Crnkovic, Michel Chaudron, Séverine Sentilles, and Aneta Vulgarakis. A classification framework for component models. In *Proceedings of the 7th Conference on Software Engineering and Practice in Sweden*, October 2007.

[CDK01] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design, 3rd ed.* Addison-Wesley, Boston, MA, USA, 2001.

[Che76] P.P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[Chu06] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, April 2006.

[CLN+02] W. O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A.Jerraya, L. Gauthier, and M. Diaz-Nava. Multiprocessor SoC Platforms: A Component-Based Design Approach. *IEEE Design and Test of Computers*, 19(6):52–63, 2002.

[CM08] P. Coussy and A. Morawiec, editors. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[CSG98] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, August 1998.

[CSL+03] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, and J. Rabaey. *UML for Real: Design of Embedded Real-Time Systems*, chapter UML and platform-based design, pages 107–126. Kluwer Academic Publishers, 2003.

[CT05] F. P. Coyle and M. A. Thornton. From UML to HDL: a Model Driven Architectural Approach to Hardware-Software Co-Design. In *Information Systems: New Generations Conference (ISNG)*, pages 88–93, April 2005.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *In Proc. Of the 9th Symposium on Foundations of Software Engineering (FSE)*, 26(5):109–120, 2001.

[DBDM03] C. Dumoulin, P. Boulet, J.-L. Dekeyser, and P. Marquet. MDA for SoC design, intensive signal processing experiment. In *In Proc. of Forum on Specification and Design Languages (FDL'03)*, September 2003.

[DMM+05] J.-L. Dekeyser, P. Marquet, S. Meftali, C. Dumoulin, P. Boulet, and S. Niar. Why to do without Model Driven Architecture in embedded system codesign? In *First IEEE BENELUX/DSP Valley Signal Processing Symposium*, 2005.

[DMv03] R. Damaševičius, G. Majauskas, and V. Štuikys. Application of design patterns for hardware design. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 48–53, New York, NY, USA, 2003. ACM.

[DPML07] J.-P. Delahaye, J. Palicot, C. Moy, and P. Leray. Partial Reconfiguration of FPGAs for Dynamical Reconfiguration of a Software Radio Platform. *Mobile and Wireless Communications Summit, 2007. 16th IST*, pages 1–5, July 2007.

[DPSV06] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli. A Platform-Based Taxonomy for ESL Design. *IEEE Des. Test*, 23(5):359–374, 2006.

[DRB+07] J. Dondo, F. Rincon, J. Barba, F. Moya, F.J. Villanueva, D. Villa, and J.C. Lopez. Dynamic Reconfiguration Management Based on a Distributed Object Model. *in Proc. of 17th International Conference on Field Programmable Logic and Applications (FPL'07)*, pages 684–687, August 2007.

[DRS04] V. D'silva, S. Ramesh, and A. Sowmya. Bridge over troubled wrappers: automated interface synthesis. *In Proc. of the 17th International Conference on VLSI Design, (VLSI Design'04)*, pages 189–194, 2004.

[Dv04a] R. Damaševičius and V. Štuikys. Application of the object-oriented principles for hardware and embedded system design. *Integr. VLSI J.*, 38(2):309–339, 2004.

[Dv04b] R. Damaševičius and V. Štuikys. Application of UML for hardware design based on design process model. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 244–249, Piscataway, NJ, USA, 2004. IEEE Press.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[Emm00] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 117–129, New York, NY, USA, 2000. ACM.

[ESSL99] Eric Eide, James L. Simister, Tim Stack, and Jay Lepreau. Flexible IDL compilation for complex communication patterns. *Sci. Program.*, 7(3-4):275–287, 1999.

[FAM05] A. Foster and S. Aslam-Mir. Practical Experiences using the OMG's Extensible Transport Framework (ETF) under a Real-Time CORBA ORB to Implement QoS Sensitive Custom Transports for SDR. In *Proc. of the 2005 Software Defined Radio Technical Conference (SDR'05)*, Hyatt Regency - Orange County, California, November 2005.

[FDH08] Luc Fabresse, Christophe Dony, and Marianne Huchard. Foundations of a Simple and Unified Component-Oriented Language. *Journal of Computer Languages, Systems & Structures*, 34/2-3(2-3):130–149, 2008.

[FGL01] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 134–140, New York, NY, USA, 2001. ACM.

[For08] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.1. `www.mpi-forum.org`, Sept. 2008.

[Fos07] A. Foster. Using a Hardware ORB to Facilitate Seamless Communication with FPGAs in DRE Systems, OMG Real-time & Embedded Systems Workshop (RTESW). `www.omg.org/news/meetings/workshops/rt_2007.htm`, July 2007.

[Fou08] Eclipse Foundation. Java Emitter Templates (JET). `www.eclipse.org/modeling/m2t/?project=jet`, 2008.

[Fra08] The Fractal component model. `fractal.ow2.org`, 2008.

[Frö06] D. Fröhlich. *Object-Oriented Development for Reconfigurable Architectures*. PhD thesis, Hochschule Mittweida, TU Bergakademie Freiberg, Germany, August 2006.

[FSLM02] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think: A Software Framework for Component-based Operating System Kernels. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX Association.

[GBA+07] A. Goderis, C. Brooks, I. Altintas, E.A. Lee, and C. Goble. Heterogeneous Composition of Models of Computation. Technical Report UCB/EECS-2007-139, EECS Department, University of California, Berkeley, November 2007.

[GBG+06] K. Grüttner, C. Brunzema, C. Grabbe, T. Schubert, and F. Oppenheimer. OSSS-Channels: Modelling and Synthesis of Communication With SystemC. In *Forum on Specification & Design Languages*, September 2006.

[GBJV08] G. Gailliard, H. Balp, C. Jouvray, and F. Verdier. Towards a Common HW/SW Interface-Centric and Component-Oriented Specification and Design Methodology. In *In Proc. of Forum on Specification and Design Languages 2008 (FDL'08)*, Sept. 2008.

[GBSV08] G. Gailliard, H. Balp, M. Sarlotte, and F. Verdier. Mapping Semantics of CORBA IDL and GIOP to Open Core Protocol for Portability and Interoperability of SDR Waveform Components. *Design, Automation and Test in Europe (DATE '08)*, pages 330–335, March 2008.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[GCW+02] Thomas Genssler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, and Chris Stich. Components for embedded software: the PECOS approach. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26, New York, NY, USA, 2002. ACM.

[GDGN03] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. *In Proc. of 16th International Conference on VLSI Design*, pages 461–466, Jan. 2003.

[GGON07] K. Grüttner, C. Grabbe, F. Oppenheimer, and W. Nebel. Object Oriented Design and Synthesis of Communication in Hardware-Software Systems with OSSS. In *In Proceedings of SASIMI 2007, Hokkaido, Japan*, October 2007.

[Ghe06] Frank Ghenassia, editor. *Transaction-Level Modeling with Systemc: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag, 2006.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GHM03] M. Goudarzi, S. Hessabi, and A. Mycroft. Object-oriented ASIP Design and Synthesis. In *Proc. of Forum on Specification and Design Languages (FDL'03)*, September 2003.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™Language Specification Third Edition*. Addison-Wesley Professional, 2005.

[GK83] D.D. Gajski and R.H. Kuhn. New VLSI Tools. *Computer*, 16(12):11–14, Dec. 1983.

[GLMS02] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, May 2002.

[GLR06] D. Grisby, S.-L. Lo, and D. Riddoch. The omniORB version 4.1 User's Guide. `omniorb.sourceforge.net`, April 2006.

[GMN06] Z. Guo, A. Mitra, and W. Najjar. Automation of IP Core Interface Generation for Reconfigurable Computing. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug. 2006.

[GMS+06] G. Gailliard, B. Mercier, M. Sarlotte, B. Candaele, and F. Verdier. Towards a SystemC TLM based Methodology for Platform Design and IP reuse: Application to Software Defined Radio. In *Second European Workshop on Reconfigurable Communication-Centric SoCs (RECOSOC 2006)*, Montpellier, France, July 2006.

[GMTB04] S. Gerard, C. Mraidha, F. Terrier, and B. Baudry. A UML-based concept for high concurrency: the real-time object. *In Proc. of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 64–67, May 2004.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[GNSV07] G. Gailliard, E. Nicollet, M. Sarlotte, and F. Verdier. Transaction Level Modelling of SCA Compliant Software Defined Radio Waveforms and Platforms PIM/PSM. *Design, Automation & Test in Europe Conference (DATE '07)*, pages 1–6, April 2007.

[GO01] S. Gérard and I. Ober. Parallelism/Concurrency specification within UML, white paper, Workshop on Concurrency Issues in UML, The Unified Modeling Language. Modeling Languages, Concepts, and Tools UML 2001. `wooddes.intranet.gr/uml2001`, October 2001.

[GO03] E. Grimpe and F. Oppenheimer. Extending the SystemC Synthesis Subset by Object Oriented Features. In *CODES + ISSS 2003*. CODES + ISSS 2003, October 2003.

[Gou05] M. Goudarzi. *The ODYSSEY Methodology: ASIP-Based Design of Embedded Systems from Object-Oriented System-Level Models*. PhD thesis, Sharif University of Technology, Iran, June 2005.

[GP08] V. Giddings and D. Paniscotti. IDL to Hardware Response to Thales/Mercury Presentations, OMG Washington Technical Meeting. `www.omg.org/docs/mars/08-03-18.pdf`, March 2008.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, May 1983.

[Gra] Mentor Graphics. Catapult. `www.mentor.com`.

[Gra07] Mentor Graphics. Catapult®synthesis user's and reference manual, system-level and block-level products, release 2007a, May 2007.

[GRE95] GRETSI, Groupe d'Etudes du Traitement du Signal et des Images. *Array-OL : proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel*, September 1995.

[GRJ04] A. Grasset, F. Rousseau, and A. A. Jerraya. Network Interface Generation for MPSOC: From Communication Service Requirements to RTL Implementation. In *RSP '04: Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping*, pages 66–69, Washington, DC, USA, 2004. IEEE Computer Society.

[Gro01] On-Chip Bus Development Working Group. Virtual Component Interface Standard Version 2 (OCB 2 2.0). Technical report, VSI Alliance™, April 2001.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[GS98] Aniruddha S. Gokhale and Douglas C. Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7*, page 376, Washington, DC, USA, 1998. IEEE Computer Society.

[GS99] A.S. Gokhale and D.C. Schmidt. Optimizing a CORBA Internet Inter-ORB protocol (IIOP) engine for minimal footprint embedded multimedia systems. *Selected Areas in Communications, IEEE Journal on*, 17(9):1673–1706, Sep 1999.

[GT03] S. Gérard and F. Terrier. *UML for real: design of embedded real-time systems*, chapter UML for Real-Time: Which native concepts to use ?, pages 107–126. Kluwer Academic Publishers, 2003.

[GTF+02] E. Grimpe, B. Timmermann, T. Fandrey, R. Biniasch, and F. Oppenheimer. SystemC Object-Oriented Extensions and Synthesis Features. In *Tagungsband*. In Proc. of Forum on Specification and Design Languages (FDL'02), September 2002.

[GTS+05] C. Grassman, A. Troya, M. Sauermann, U. Ramacher, and M. Richter. Mapping waveforms to mobile parallel processor architectures. In *Proc. of the SDR Forum Technical Conference (SDR'05)*, November 2005.

[GZH07] M. Goudarzi, N. Mohammad Zadeh, and S. Hessabi. Using on-chip networks to implement polymorphism in the co-design of object-oriented embedded systems. *J. Comput. Syst. Sci.*, 73(8):1221–1231, 2007.

[HA04] J.C. Hoe and Arvind. Operation-centric hardware description and synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(9):1277–1288, Sept. 2004.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HC01] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[Hen04] M. Henning. A New Approach to Object-Oriented Middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[HK05] S. Hong and T. H. Kim. Resource conscious development of middleware for control environments: a case of CORBA-based middleware for the CAN bus systems. *Information and Software Technology*, 47(6):411 – 425, 2005.

[HLEJ06] Seongsoo Hong, Jaesoo Lee, Hyeonsang Eom, and Gwangil Jeon. The robot software communications architecture (RSCA): embedded middleware for networked service robots. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

[HLL+03] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, University of California at Berkeley, July 2003.

[HM07] M. Henning and M.Spruiell. Distributed Programming with ICE, revision 3.2. www.zeroc.com, March 2007.

[HO97] R. Helaihel and K. Olukotun. Java as a specification language for hardware-software systems. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 690–697, Washington, DC, USA, 1997. IEEE Computer Society.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hos98] Philipp Hoschka. Compact and efficient presentation conversion code. *IEEE/ACM Trans. Netw.*, 6(4):389–396, 1998.

[HP06] F. Humcke and D. Paniscotti. Integrated Circuit ORB (ICO) White Paper, v1.1. www.prismtech.com, 2006.

[HS08] M. Henning and M. Spruiell. Distributed Programming with Ice, Revision 3.3.0. zeroc.com, May 2008.

[Hug05] J. Hugues. *Architecture et Services des Intergiciels Temps Réel*. PhD thesis, University of Paris VI, Pierre et Marie-Curie, France, September 2005.

[Hum06] F. Humcke. Making FPGAs "First Class" SCA Citizens. In *Proc. of the SDR Forum Technical Conference (SDR'06)*, November 2006.

[Hum07] F. Humcke. Making FPGAs "First Class" SCA Citizens, OMG Software-Based Communications (SBC) Workshop. www.omg.org/news/meetings/workshops/sbc_2007.htm, March 2007.

[Hur98] Judith Hurwitz. Sorting out middleware. *DBMS*, 11(1):10–12, 1998.

[HVP+05] J. Hugues, T. Vergnaud, L. Pautet, Y. Thierry-Mieg, S. Baarir, and F. Kordon. On the Formal Verification of Middleware Behavioral Properties. *Electr. Notes Theor. Comput. Sci.*, 133:139–157, 2005.

[ICE01] ICECS 2001: the 8th IEEE International Conference on Electronics, Circuits and Systems. *Object-oriented high level synthesis based on SystemC*, September 2001.

[ICO08] ICODES (Interface and Communication based Design ofEmbedded Systems) IST-FP6 Project, August 2004-2007. icodes.offis.de, June 2008.

[IEE02] IEEE. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076(TM)-2002, May 2002.

[IEE04] IEEE. IEEE Standard for Information Technology- Standardized Application Environment Profile (AEP)-POSIX Realtime and Embedded Application Support, 2004.

[III00] J. Mitola III, editor. *Software Radio Architecture: Object-Oriented Approaches to Wireless Systems Engineering*. Wiley, November 2000.

[Ini08] Open SystemC Initiative. Systemc. `www.systemc.org`, June 2008.

[IT02] International Telecommunication Union (ITU)-T. Specification of Packed Encoding Rules (PER), ITU-T Recommendation X.691. `www.itu.int`, July 2002.

[Jac08] J. M. Jacob. No Processor Is an Island: Developing Multiple Processor Systems with the "New" CORBA. `www.dsp-fpga.com`, Feb. 2008.

[Jan03] A. Jantsch. NoCs: a new contract between hardware and software. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pages 10–16, Sept. 2003.

[JBP06] A. A. Jerraya, A. Bouchhima, and F. Pétrot. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 280–285, New York, NY, USA, 2006. ACM.

[JNH06] S. Jorg, M. Nickl, and G. Hirzinger. Flexible signal-oriented hardware abstraction for rapid prototyping of robotic systems. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3755–3760, Oct. 2006.

[JS07] S. Jagadish and R. K. Shyamasundar. UML-based Approach to Specify Secured, Fine-grained Concurrent Access to Shared Resources. *Journal of Object Technology*, 6(1):107–119, Jan.-Fev. 2007.

[JTR01] JPEO JTRS. Software Communications Architecture (SCA) Specification v2.2. `sca.jpeojtrs.mil`, November 2001.

[JTR02] JPEO JTRS. SCA Developer's Guide, Rev 1.1, June 2002.

[JTR04] JPEO JTRS. Specialized Hardware Supplement (SHS) to the Software Communication Architecture (SCA) Specification v3.0, August 2004.

[JTR05] JPEO JTRS. Extension for component portability for Specialized Hardware Processors (SHP) to the JTRS Software Communication Architecture (SCA) Specification, v3.1x, a.k.a. SCA Change Proposal (CP) 289. `www.mc.com/uploadedFiles/CP289_v1_0_TAG.pdf`, March 2005.

[JTR06] JPEO JTRS. Software Communications Architecture (SCA) Specification v2.2.2. `sca.jpeojtrs.mil`, May 2006.

[JTR07] JPEO JTRS. Modem Hardware Abstraction Layer (MHAL) API, Version 2.11.1. `sca.jpeojtrs.mil`, May 2007.

[JTR09] JPEO JTRS. News Release: JPEO JTRS initiates the development of a new Software Communications Architecture (SCA) Release. `sca.jpeojtrs.mil/_downloads/JPEO-NR-2009-005.pdf`, August 2009.

[JU08] J. M. Jacob and M. Uhm. CORBA for FPGAs: Tying together GPPs, DSPs, and FPGAs. `www.rtcmagazine.com`, February 2008.

[Kay93] A. C. Kay. The early history of smalltalk. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 69–95, New York, NY, USA, 1993. ACM.

[KB05] J. Kulp and M. Bicer. Integrating Specialized Hardware to JTRS/SCA Software Defined Radios. In *Milcom'05*, October 2005.

[KBG+08] L. Kriaa, A. Bouchhima, M. Gligor, A.-M. Fouillart, F. Pétrot, and A. A. Jerraya. Parallel Programming of Multi-processor SoC: A HW-SW Interface Perspective. *International Journal of Parallel Programming*, 36(1):68–92, 2008.

[KBY+06] Lobna Kriaa, Aimen Bouchhima, Wassim Youssef, Frederic Petrot, Anne-Marie Fouillart, and Ahmed Jerraya. Service Based Component Design Approach for Flexible Hardware/Software Interface Modeling. In *RSP '06: Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping*, pages 156–162, Washington, DC, USA, 2006. IEEE Computer Society.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002.

[Kel07] Stephen Kell. Rethinking software connectors. In *First International Workshop on Synthesis and Analysis of Component Connectors*, pages 7–19, September 2007.

[KGG06] Wolfgang Klingauf, Hagen Gädke, and Robert Günzel. TRAIN: a virtual transaction layer architecture for TLM-based HW/SW codesign of synthesizable MPSoC. In *in Proc. of the conference on Design, Automation and Test in Europe (DATE'06)*, pages 1318–1323, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[KJH+00a] K. Kim, G. Jeon, S. Hong, S. Kim, and T. Kim. Resource-Conscious Customization of CORBA for CAN-Based Distributed Embedded Systems. In *In Proc. of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00)*, page 34, Washington, DC, USA, 2000. IEEE Computer Society.

[KJH+00b] K. Kim, G. Jeon, S. Hong, T.-H. Kim, and S. Kim. Integrating Subscription-Based and Connection-Oriented Communications into the Embedded CORBA for the CAN Bus. In *In Proc. of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS'00)*, page 178, Washington, DC, USA, 2000. IEEE Computer Society.

[KKS02] R. Klefstad, A. S. Krishna, and D. C. Schmidt. Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, and Embedded CORBA Applications. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 549–567, London, UK, 2002. Springer-Verlag.

[KKSC04] A. S. Krishna, R. Klefstad, D. C. Schmidt, and A. Corsaro. *Middleware for Communications*, chapter Real-time CORBA Middleware. Wiley, 2004.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KMD05] A. Koudri, S. Meftali, and J.-L. Dekeyser. IP integration in embedded systems modeling. In *In 14th IP Based SoC Design Conference (IP-SoC'05)*, December 2005.

[KNRSV00] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[KOSK+01] T. Kuhn, T. Oppold, C. Schulz-Key, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai. Object oriented hardware synthesis and verification. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 189–194, New York, NY, USA, 2001. ACM.

[KOW+01] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai. A framework for object oriented hardware specification, verification, and synthesis. In *In Proc. of the 38th conference on Design automation (DAC '01)*, pages 413–418, New York, NY, USA, 2001. ACM.

[KR98] T. Kuhn and W. Rosenstiel. Java Based Modeling And Simulation Of Digital Systems On Register Transfer Level. In *In Proc. of Workshop on System Design Automation (SDA)*, March 1998.

[Kra08] S. Krakowiak. Middleware Architecture with Patterns and Frameworks. `sardes.inrialpes.fr/~krakowia/MW-Book`, April 2008.

[KRK99] T. Kuhn, W. Rosenstiel, and U. Kebschull. Description and simulation of hardware/software systems with Java. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 790–793, New York, NY, USA, 1999. ACM.

[KRS03] R. Klefstad, S. Rao, and D. C. Schmidt. Design and Performance of a Dynamically Configurable, Messaging Protocols Framework for Real-Time CORBA. In *In Proc. of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[KSKR00] T. Kuhn, C. Schulz-Key, and W. Rosenstiel. Object oriented hardware specification with java. In *Proceedings of the ninth workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'2000)*, 2000.

[Lah04] Vesa Lahtinen. *Design and Analysis of Interconnection Architecture in On-Chip Digital Systems*. PhD thesis, Tampere University, 2004.

[Lea99] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley, Boston, MA, USA, 1999.

[Lew07] J. Lewis. What's Next in VHDL. `www.accellera.org`, April 2007.

[LJB05] S. Lankes, A. Jabs, and T. Bemmerl. Design and performance of a CAN-based connection-oriented protocol for real-time CORBA. *J. Syst. Softw.*, 77(1):37–45, 2005.

[LKPH06] Jaesoo Lee, Saehwa Kim, Jiyong Park, and Seongsoo Hong. Q-SCA: Incorporating QoS support into Software Communications Architecture for SDR waveform processing. *Real-Time Syst.*, 34(1):19–35, 2006.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[LN04] Edward A. Lee and Stephen Neuendorffer. *Actor-oriented models for codesign: balancing re-use and performance*, pages 33–56. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[LN07] F. Lafaye and E. Nicollet. A DSP Micro-Framework (DSP μF) for OMG SWRadio specification extension. *In Proc. of the SDR 07 Technical Conference*, pages 1–6, Nov. 2007.

[LPHH05] Jaesoo Lee, Jiyong Park, Seunghyun Han, and Seongsoo Hong. Extending Software Communications Architecture for QoS support in SDR signal processing. In *RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 117–122, Washington, DC, USA, 2005. IEEE Computer Society.

[LTG97] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 70–75, New York, NY, USA, 1997. ACM.

[LU95] R.I. Lackey and D.W. Upmal. Speakeasy: the military software radio. *Communications Magazine, IEEE*, 33(5):56–61, May 1995.

[LvdADtH05] A. Lachlan, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. *In Proc. of the 7th International Symposium on Distributed Objects and Applications (DOA), On the Move to Meaningful Internet Systems*, volume 3761 of *Lecture Notes in Computer Science (LNCS)*, chapter On the Notion of Coupling in Communication Middleware, pages 1015–1033. Springer Berlin / Heidelberg, Agia Napa, Cyprus., Oct. 2005.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, Oct. 2007.

[LY99] T. Lindholm and F. Yellin. *The Java™Language Specification Second Edition*. Addison-Wesley, 1999.

[Man89] Architecture Projects Management. The Advanced Network Systems Architecture (ANSA) Reference Manual, release 1.1, Castle Hill, Cambridge, UK, July 1989.

[Mar05] Philippe Martin. Design of a virtual component neutral network-on-chip transaction layer. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 336–337, 2005.

[Mat08] Mathworks. `www.mathworks.com`, 2008.

[MB02] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley, May 2002.

[MC99] W. E. McUmber and B. H. C. Cheng. UML-Based Analysis of Embedded Systems Using a Mapping to VHDL. In *in the 4th IEEE International Symposium on High-Assurance Systems Engineering (HASE '99)*, pages 56–63, Washington, DC, USA, 1999. IEEE Computer Society.

[McH94] C. McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, University of Dublin, Trinity College, October 1994.

[McH07] C. McHale. CORBA Explained Simply. www.CiaranMcHale.com, February 2007.

[McI68] M. D. McIlroy. *Software Engineering: Report on a conference by the NATO Science Committee*, chapter Mass produced software components, pages 138–150. Scientific Affairs Division, NATO, 1968.

[MDD+03] A. D. McKinnon, K. E. Dorow, T. R. Damania, O. Haugan, W. E. Lawrence, D. E. Bakken, and J. C. Shovic. A Configurable Middleware Framework with Multiple Quality of Service Properties for Small Embedded Systems. In *In Proc. of the Second IEEE International Symposium on Network Computing and Applications (NCA'03)*, page 197, Washington, DC, USA, 2003. IEEE Computer Society.

[Mey92] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[Mic97] Sun Microsystems. Javabeans specification v1.0.1-a. `java.sun.com/javase`, August 1997.

[Mic02] Sun Microsystems. Java Message Service (JMS), Version 1.1, JSR 914. `java.sun.com/products/ejb/docs.html`, April 2002.

[Mic06] Sun Microsystems. Enterprise $javabeans^{TM}$ 3.0, java specification request (jsr) 220. `java.sun.com/products/ejb/docs.html`, May 2006.

[Mic08] Sun Microsystems. Java remote method invocation. `java.sun.com/javase/6/docs/platform/rmi`, November 2008.

[MMP00] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM.

[MMT+08] Philippe Manet, Daniel Maufroid, Leonardo Tosi, Gregory Gailliard, Olivier Mulertt, Marco Di Ciano, Jean-Didier Legat, Denis Aulagnier, Christian Gamrat, Raffaele Liberati, Vincenzo La Barba, Pol Cuvelier, Bertrand Rousseau, and Paul Gelineau. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. *EURASIP J. Embedded Syst.*, 2008:1–11, 2008.

[MNT+04] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The nostrum backbone-a communication protocol stack for networks on chip. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 693–696, 2004.

[MNTJ04] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 890–895, Feb. 2004.

[Moc08] Mocca Project, the Mocca-compiler for run-time reconfigurable architectures. `www.htwm.de/~lec`, 2008.

[MRC+00] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: an IDL for hardware programming. In *In Proc. of the 4th conference on Symposium on Operating System Design & Implementation (OSDI'00)*, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.

[MS98] G. De Micheli and J. Smith. Automated composition of hardware components. *DAC*, 00:14–19, 1998.

[MS04] G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274, New York, NY, USA, 2004. ACM.

[MSUW04] S. J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, Boston, March 2004.

[MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[MVS+05] P. Marchal, D. Verkest, A. Shickova, F. Catthoor, F. Robert, and A. Leroy. Spatial division multiplexing: a novel approach for guaranteed throughput on NoCs. *Hardware/-Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/I-FIP International Conference on*, pages 81–86, Sept. 2005.

[MVV+07] F. Moya, D. Villa, F. J. Villanueva, J. Barba, F. Rincón, and J. C. López. Embedding standard distributed object-oriented middlewares in wireless sensor networks. *Wireless Communications and Mobile Computing*, pages 1–10, 2007.

[ND81] K. Nygaard and O.-J. Dahl. The development of the simula languages. In *History of programming languages I*, pages 439–480, New York, NY, USA, 1981. ACM.

[NH07] Bernhard Niemann and Christian Haubelt. Towards a Unified Execution Model for Transactions in TLM. *In Proc. of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'07)*, pages 103–112, June 2007.

[NK07] J. Noseworthy and J. Kulp. Standard Interfaces for FPGA Components. In *Milcom'07*, pages 1–5, October 2007.

[NP08] E. Nicollet and L. Pucker. Standardizing Transceiver APIs for Software Defined and Cognitive Radio, RF Design magazine. `www.rfdesign.com`, February 2008.

[NP09] E. Nicollet and S. Pothin. Transceiver Facility Specification, SDRF-08-S-0008-V1.0.0, January 2009.

[NST] Ltd. NEC System Technologies. Cyberworkbench. `www.necst.co.jp/product/cwb/english`.

[OIa] OCP-IP. Socket-centric ip core interface maximizes ip applications. `www.ocpip.org/socket/whitepapers`.

[OIb] OCP-IP. The Importance of Sockets in SOC Design. `www.ocpip.org/socket/whitepapers`.

[OI04] OCP-IP. The Open SystemC Initiative And The Open Core Protocol International Partnership Join Forces to Target TLM Layer Standardization. www.ocpip.org, May 2004.

[OI06] OCP-IP. Open core protocol specification, v2.2.1. www.ocpip.org, January 2006.

[OI07] OCP-IP. A systemc™ocp transaction level communication channel, v2.2. www.ocpip.org, February 2007.

[OIS08] Objective interface systems website. www.ois.com, June 2008.

[OKS+00] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. In *IFIP/ACM International Conference on Distributed systems platforms, Middleware'00*, pages 372–395, Secaucus, NJ, USA, 2000. Springer-Verlag New York,.

[OMG95] OMG. Object Management Architecture (OMA) Guide, Third Edition, Juin 1995.

[OMG99] OMG. IDL to C Language Mapping, v.1.2, July 1999.

[OMG01] OMG. IDL to Ada Language Mapping, v.1.2, October 2001.

[OMG02] OMG. UML Profile for CORBA Specification, Version 1.0, April 2002.

[OMG03] OMG. MDA Guide V1.0.1, June 2003.

[OMG04] OMG. Extensible Transport Framework (ETF), OMG Final Adopted Specification, March 2004.

[OMG05a] OMG. Meta Object Facility (MOF) Specification, v 1.4.1, May 2005.

[OMG05b] OMG. Real-time CORBA Specification v.1.2, January 2005.

[OMG06a] OMG. CORBA Component Model (CCM) Specification, Version 4.0, April 2006.

[OMG06b] OMG. Deployment and Configuration of Component-based Distributed Applications Specification, Version 4.0, April 2006.

[OMG06c] OMG. Meta Object Facility (MOF) Core Specification, v2.0, January 2006.

[OMG06d] OMG. Object Constraint Language, v.2.0, May 2006.

[OMG06e] OMG. UML Profile for SoC Specification, v1.0.1, August 2006.

[OMG07a] OMG. Data Distribution Service for Real-time Systems, v1.2, Jan. 2007.

[OMG07b] OMG. Infrastructure UML modeling, v.2.1.1, February 2007.

[OMG07c] OMG. MOF 2.0/XMI Mapping, Version 2.1.1, Dec. 2007.

[OMG07d] OMG. Superstructure UML modeling, v.2.1.1, February 2007.

[OMG07e] OMG. UML Profile for Software Radio (a.k.a. PIM & PSM for Software Radio Components), March 2007.

[OMG08a] OMG. Common Object Request Broker Architecture (CORBA) for embedded (CORBAe), Version 1.0 - Beta 2, Janvier 2008.

[OMG08b] OMG. Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Janvier 2008.

[OMG08c] OMG. IDL to C++ Language Mapping, v.1.2, January 2008.

[OMG08d] OMG. IDL to Java Language Mapping, v.1.3, January 2008.

[OMG08e] OMG. MOF 2.0 Query/View/Transformation (QVT), v1.0, Apr. 2008.

[OMG08f] OMG. OMG Systems Modeling Language (SysML), v1.1, Nov. 2008.

[OMG08g] OMG. UML profile for CORBA and CORBA Components Specification, Version 1.0, April 2008.

[OMG08h] OMG. UML profile for Modeling and Analyzis of Real-Time and Embedded Systems (MARTE), Beta 1, August 2008.

[oOSI04] Synthesis Working Group of Open SystemC Initiative. SystemC Synthesizable Subset, Draft.1.1.18. `www.systemc.org`, December 2004.

[Ope02] OpenCores.org. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores specification, revision B.3. `www.opencores.org`, September 2002.

[OPM94] Sean O'Malley, Todd Proebsting, and Allen Brady Montz. USC: a universal stub compiler. *SIGCOMM Comput. Commun. Rev.*, 24(4):295–306, 1994.

[Opp05] F. Oppenheimer. *OOCOSIM - An Object-Oriented Co-design Method for Embedded HW/SW Systems*. PhD thesis, University of Oldenburg, Germany, February 2005.

[Org98] International Standards Organization. Information Technology - Open Distributed Processing - Reference Model: Overview, ISO/IEC 10746-1:1998(E), December 1998.

[OS99] I. Ober and I. Stan. On the Concurrent Object Model of UML. In *Proc. of the 5th International Euro-Par Conference on Parallel Processing (Euro-Par '99)*, pages 1377–1384, London, UK, 1999. Springer-Verlag.

[OS07] D. R. Oldham and M. C. Scardelletti. JTRS/SCA and Custom/SDR Waveform Comparison. *IEEE Military Communications Conference (MILCOM'07)*, pages 1–5, Oct. 2007.

[OS09] OMG and SDRForum. PIM and PSM for Smart Antenna, Version 1.0, Jan. 2009.

[OSC09] OSCI. TLM-2.0 Language Reference Manual. `www.systemc.org`, Jul. 2009.

[Pap92] M. Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*. PhD thesis, University of Geneva, January 1992.

[PB07] D. Paniscotti and J. Bickle. SDR Signal Processing Distributive-Development Approaches. In *In Proc. of the SDR Forum Technical Conference (SDR'07)*, November 2007.

[Per97] D. E. Perry. Software architecture and its relevance to software engineering, invited talk. In *Second International Conference on Coordination Models and Languages (Coord'97)*, Berlin, Germany, September 1997.

[PH04] L. Pucker and J. Holt. Extending the SCA core framework inside the modem architecture of a software defined radio. *IEEE Radio Communications*, 42:21–25, March 2004.

[PH07] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface, 3th edition*. Morgan Kaufmann, San Francisco, CA, USA, 2007.

[Pin04] Hennadiy Pinus. Middleware: Past and present a comparison. `www.st.informatik.tu-darmstadt.de`, June 2004.

[PPB02] P. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A System-Level Exploration Platform for Network Processors. *IEEE Des. Test*, 19(6):17–26, 2002.

[PPB+04] P. G. Paulin, C. Pilkington, E. Bensoudane, M. Langevin, and D. Lyonnard. Application of a Multi-Processor SoC Platform to High-Speed Packet Forwarding. In *In Proc. of the conference on Design, Automation and Test in Europe (DATE '04)*, page 30058, Washington, DC, USA, 2004. IEEE Computer Society.

[PPL+06a] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benny, D. Lyonnard, B. Lavigueur, and D. Lo. Distributed object models for multi-processor SoC's, with application to low-power multimedia wireless systems. In *DATE*, pages 482–487, 2006.

[PPL+06b] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, and G. Nicolescu. Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(7):667–680, July 2006.

[PPL+06c] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, and G. Nicolescu. *Embedded Systems Handbook*, chapter A Multiprocessor SoC Platform and Tools for Communications Applications. CRC Press, 2006.

[Pri08] Prismtech website. `www.prismtech.com`, June 2008.

[PRP05] A. Puder, K. Römer, and F. Pilhofer. *Distributed Systems Architecture: A Middleware Approach*. Morgan Kaufmann, San Francisco, CA, USA, November 2005.

[PRSV98] R. Passerone, J.A. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. *In Proc. of the 36th Design Automation Conference (DAC'98)*, pages 8–13, Jun 1998.

[PS98] Irfan Pyarali and Douglas C. Schmidt. An overview of the CORBA portable object adapter. *StandardView*, 6(1):30–43, 1998.

[PSG+99] Irfan Pyarali, Douglas Schmidt, Aniruddha Gokhale, Nanbor Wang, and Vishal Kachroo. Applying Optimization Principle Patterns to Real-time ORBs. In *Proc. 5 th USENIX Conference on O-O Technologies and Systems (COOTS'99*, 1999.

[PT05] D. Pellerin and S. Thibault. *Practical FPGA Programming in C*. Prentice Hall PTR, April 2005.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. *Software Engineering, IEEE Transactions on*, 28(11):1056–1076, Nov 2002.

[QS05] Yang Qu and Juha-Pekka Soininen. SystemC-based Design Methodology for Reconfigurable System-on-Chip. In *DSD '05: Proceedings of the 8th Euromicro Conference on Digital System Design*, pages 364–371, Washington, DC, USA, 2005. IEEE Computer Society.

[QSC04] J. Quévremont, M. Sarlotte, and B. Candaele. Development process of a DRM digital broadcast SoC receiver platform. *Annales des Télécommunications*, Sept-Oct 2004.

[Rad00] M. Radetzki. *Synthesis of Digital Circuits from Object-Oriented Specifications*. PhD thesis, University of Oldenburg, Germany, April 2000.

[Rat] Rational. `www.ibm.com/software/rational`.

[RBK00] A. Rajawat, M. Balakrishnan, and A. Kumar. Interface synthesis: issues and approaches. *In Proc. of the 13th International Conference on VLSI Design (VLSI Design'00)*, pages 92–97, 2000.

[RCP+01] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. A. Najjar, and W. Böhm. An automated process for compiling dataflow graphs into reconfigurable hardware. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):130–139, 2001.

[RDP+05] A. Radulescu, J. Dielissen, S.G. Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):4–17, Jan. 2005.

[Rec07] W3C Recommendation. SOAP Version 1.2 specification. `www.w3.org/TR/soap12`, April 2007.

[RGR+03] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. *IEEE Proceedings Computers and Digital Techniques*, 150(5):294–302, Sept. 2003.

[RIJ04] J. Rumbaugh and G. Booch I. Jacobson. *The Unified Modeling Language Reference Manual Second Edition*. Addison Wesley, July 2004.

[Rit98] David Ritter. The middleware muddle. *SIGMOD Rec.*, 27(4):86–93, 1998.

[RKC01] Manuel Roman, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001.

[RL02] John Reekie and Edward A. Lee. Lightweight Component Models for Embedded Systems. Technical Report UCB ERL M02/30, Electronics Research Laboratory, University of California at Berkeley, October 2002.

[RMB+05] F. Rincón, F. Moya, J. Barba, D. Villa, F. J. Villanueva, and J. C. López. A New Model for NoC-based Distributed Heterogeneous System Design. In *Parallel Computing, ParCo'05*, pages 777–784, September 2005.

[RMBL05] F. Rincón, Francisco Moya, Jesús Barba, and Juan Carlos López. Model Reuse through Hardware Design Patterns. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 324–329, Washington, DC, USA, 2005. IEEE Computer Society.

[RMKC00] Manuel Roman, M. Dennis Mickunas, Fabio Kon, and Roy Campbell. LegORB and ubiquitous CORBA. In *in Proc. IFIP/ACM Middleware'2000 Workshop on Reflective Middleware (RM2000*, pages 1–2, 2000.

[RRS+05] Sylvain Robert, Ansgar Radermacher, Vincent Seignole, Sébastien Gérard, Virginie Watine, and François Terrier. The CORBA connector model. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, pages 76–82, New York, NY, USA, 2005. ACM.

[RSB+06] M. Rieder, R. Steiner, C. Berthouzoz, F. Corthay, and T. Sterren. *Rapid Integration of Software Engineering Techniques*, volume 3943 of *Lecture Notes in Computer Science*, chapter Synthesized UML, a Practical Approach to Map UML to VHDL„ pages 203–217. Springer Berlin / Heidelberg, May 2006.

[RSH01] Hans Reiser, Martin Steckermeier, and Franz J. Hauck. IDLflex: a flexible and generic compiler for CORBA IDL. In *In Proc. of the Net.ObjectDays*, pages 151–160, Erfurt, Germany, Sep. 2001.

[RSPF05] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. Transaction Level Modelling in SystemC, OSCI whitepaper. `www.systemc.org`, April 2005.

[RSRB05] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A SoC Design Methodology Involving a UML 2.0 Profile for SystemC. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 704–709, Washington, DC, USA, 2005. IEEE Computer Society.

[RSV97] J. A. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 178–183, New York, NY, USA, 1997. ACM.

[RTZ] RT Zen. `doc.ece.uci.edu/rtzen`.

[Rum87] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 466–481, New York, NY, USA, October 1987. ACM.

[RZP+05] Krishna Raman, Yue Zhang, Mark Panahi, Juan A. Colmenares, Raymond Klefstad, and Trevor Harmon. RTZen: Highly Predictable, Real-Time Java Middleware for Distributed and Embedded Systems. In *Middleware*, pages 225–248, 2005.

[S.06] Rouxel S. *Modélisation et Caractérisation d'une Plate-Forme SOC Hétérogène : Application à la Radio Logicielle*. PhD thesis, University of South Brittany, France, December 2006.

[SB03] D. C. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In *In Proc. of the 25th International Conference on Software Engineering (ICSE '03)*, pages 694–704, Washington, DC, USA, 2003. IEEE Computer Society.

[SBF04] B. Steinbach, T. Beierlein, and D. Fröhlich. *Languages for system specification: selected contributions from FDL'03*, chapter UML-based co-design for run-time reconfigurable architectures, pages 5–19. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[SC99] D.C. Schmidt and C. Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine*, 37(4):54–63, Apr 1999.

[SCG05] V. Subramonian and C. Christopher Gill. *Embedded Systems Handbook*, chapter Middleware Design and Implementation for Networked Embedded Systems, pages pp. 1–17. CRC Press, Florida, richard zurawski edition, 2005.

[Sch94] D. C. Schmidt. The adaptive communication environment: Object-oriented network programming components for developing client/server applications. In *12th Sun Users Group Conference*, June 1994.

[Sch98] D. C. Schmidt. Evaluating architectures for multithreaded object request brokers. *Commun. ACM Special Issue on CORBA*, 41(10):54–60, 1998.

[Sch99] G. Schumacher. *Object-oriented hardware specification and design with a language extension to VHDL*. PhD thesis, University of Oldenburg, Germany, April 1999.

[Sch06] D.C. Schmidt. Model-driven engineering. *Computer*, 39(2):25–31, February 2006.

[SCM08] CoWare SCML (SystemC Modeling Library) source code kit. `www.coware.com/solutions/scml_kit.php`, 2008.

[SDG+07] Venkita Subramonian, Gan Deng, Christopher Gill, Jaiganesh Balasubramanian, Liang-Jui Shen, William Otte, Douglas C. Schmidt, Aniruddha Gokhale, and Nanbor Wang. The design and performance of component middleware for QoS-enabled deployment and configuration of DRE systems. *J. Syst. Softw.*, 80(5):668–677, 2007.

[Sel04] B. Selic. *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, chapter On the Semantic Foundations of Standard UML 2.0, pages 181–199. Springer, December 2004.

[SFB05] B. Steinbach, D. Fröhlich, and T. Beierlein. *UML for SOC Design*, chapter Hardware/Software Codesign of Reconfigurable Architectures Using UML, pages 89–117. Springer, 2005.

[SG07a] D. Schreiner and K. M. Göschka. Explicit connectors in component based software engineering for distributed embedded systems. In *SOFSEM*, pages 923–934, 2007.

[SG07b] D. Schreiner and K. M. Göschka. Synthesizing communication middleware from explicit connectors in component based distributed architectures. In *Software Composition*, pages 160–167, March 2007.

[Sha86] M. Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proceedings of the 6th Int. Conf. on Distributed Systems (ICDCS)*, pages 198–204, Cambridge MA (USA), May 1986.

[Sif05] Joseph Sifakis. A framework for component-based construction extended abstract. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 293–300, Washington, DC, USA, 2005. IEEE Computer Society.

[SK97] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, April 1997.

[SKH98] B. Svantesson, S. Kumar, and A. Hemani. A methodology and algorithms for efficient interprocess communication synthesis from system description in SDL. In *In Proc. of the Eleventh International Conference on VLSI Design*, pages 78–84, Erfurt, Germany, Jan. 1998.

[SKH05] E. Salminen, A. Kulmala, and T.D. Hamalainen. HIBI-based multiprocessor SoC on FPGA. *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, 4:3351–3354, May 2005.

[SKH07] E. Salminen, A. Kulmala, and T.D. Hamalainen. On network-on-chip comparison. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 503–510, Aug. 2007.

[SKKR01] C. Schulz-Key, T. Kuhn, and W. Rosenstiel. A Framework for System-Level Partitioning of Object-Oriented Specifications. In *Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI)*, 2001.

[SKWS+04] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented modeling and synthesis of SystemC specifications. In *In Proc. of the 2004 conference on Asia South Pacific design automation (ASP-DAC '04)*, pages 238–243, Piscataway, NJ, USA, 2004. IEEE Press.

[SLM98] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4):294–324, April 1998.

[SMC+07] M. Sarlotte, D. Maufroid, R. Chau, B. Counil, and P. Gelineau. Partial reconfiguration concept in a SCA approach. In *In Proc. of the SDR Forum Technical Conference (SDR'07)*, November 2007.

[Smi98] Douglas J. Smith. *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog*. Doone Publications, 1998. Foreword By-Zamfirescu, Alex.

[Sol] Agility Design Solutions. DK Design Suite. `www.agilityds.com`.

[SON06] A. Schallenberg, F. Oppenheimer, and W. Nebel. OSSS+R: Modelling and Simulating Self-Reconfigurable Systems. In IEEE, editor, *In Proc. of International Conference on Field Programmable Logic and Applications (FPL'06)*, pages 177–182, August 2006.

[SP96] C. Szyperski and C. Pfister. First International Workshop on Component-Oriented Programming WCOP'96, Workshop Report, July 1996.

[SPI08] SPIRIT. IP-XACT v1.4: a specification for XML meta-data and tool interfaces. www.spiritconsortium.org, March 2008.

[SR04] T. Schattkowsky and A. Rettberg. UML for FPGA Synthesis. In *Proc. of the UML for SoC Design Workshop*, June 2004.

[SS01] S. Sendall and A. Strohmeier. *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, chapter Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML, pages 391–405. Springer, January 2001.

[SSC98] A. Singhai, A. Sane, and R.H. Campbell. Quarterware for middleware. *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 192–201, May 1998.

[SSM+01] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design. In *In Proceedings of the 38th Design Automation Conference, DAC '01*, June 2001.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture (POSA) Vol. 2: Patterns for Concurrent and Networked Objects*. Wiley, July 2000.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[SV01] Douglas C. Schmidt and Steve Vinoski. Using Standard C++ in the OMG C++ Mapping. *Dr. Dobb's Journal*, avril 2001.

[SV02] A. Sangiovanni-Vincentelli. Defining Platform-based Design. *EEDesign of EETimes*, February 2002.

[SV07] A. L. Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[SVL01] D. Truscan S. Virtanen and J. Lilius. SystemC Based Object Oriented System Design. In *In Proc. of Forum on Specification and Design Languages 2002 (FDL'01)*, September 2001.

[SXG+04] V. Subramonian, Guoliang Xing, C. Gill, Chenyang Lu, and R. Cytron. Middleware specialization for memory-constrained networked embedded systems. *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 306–313, May 2004.

[Syn] Synfora. Pico express. www.synfora.com.

[Sys] Forte Design Systems. Cynthesizer. www.forteds.com.

[Sys08] Objective Interface Systems. ORB*express*®FPGA Datasheet. www.ois.com, June 2008.

[Szy02] C. Szyperski. *Component Software - Beyond Object-Oriented Programming, 2nd ed.* Addison-Wesley, 2002.

[Tan96] A. S. Tanenbaum. *Computer Networks, 3rd ed.* Prentice Hall, 1996.

[TAO08] The ACE ORB (TAO). `www.cs.wustl.edu/~schmidt/TAO.html`, 2008.

[TBR07] T. Tsou, P.J. Balister, and J.H. Reed. Latency Profiling for SCA Software Radio. In *SDR Forum Technical Conference*, Denver, CO, November 2007.

[Tec] Impulse Accelerated Technologies. ImpulseC. `www.impulsec.com/C_to_fpga.htm`.

[THA05] T.Kogel, A. Haverinen, and J. Altis. OCP TLM for Architectural Modelling, OCP-IP white-paper. `www.ocpip.org`, 2005.

[Tho97] J. Thompson. Avoiding a middleware muddle. *Software, IEEE*, 14(6):92–95, Nov/Dec 1997.

[Tor92] F.M. Torre. Speakeasy - a new direction in tactical communications for the 21st century. In *Proc. of the Tactical Communications Conference*, volume 1, pages 139–142 vol.1, Apr 1992.

[Tut02] W. H.W. Tuttlebee, editor. *Software Defined Radio: Origins, Drivers and International Perspectives*. Wiley, January 2002.

[UoT08] Germany University of Tuebingen. OASE (Objektorientierter hArdware/Software Entwurf, Object oriented design of hardware/software systems) Project. `www-ti.informatik.uni-tuebingen.de/~oase`, June 2008.

[vdWdKH+04] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217, New York, NY, USA, 2004. ACM.

[vdWdKH+06] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. *Embedded Systems Handbook*, chapter Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach. CRC Press, 2006.

[VHPK04] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: A Schizophrenic Middleware to Build Versatile Reliable Distributed Applications. In *Reliable Software Technologies - Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science (LNCS)*, pages 106 — 119, June 2004.

[Vin03] S. Vinoski. It's just a mapping problem. *Internet Computing, IEEE*, 7(3):88–90, May-June 2003.

[VKZ04] M. Voelter, M. Kircher, , and U. Zdun. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. Wiley, October 2004.

[vOvdLKM00] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, Mar 2000.

[WAU+08] S.K. Wood, D.H. Akehurst, O. Uzenkov, W.G.J. Howells, and K.D. McDonald-Maier. A Model-Driven Development Approach to Mapping UML State Diagrams to Synthesizable VHDL. *Computers, IEEE Transactions on*, 57(10):1357–1371, Oct. 2008.

[WQ05] A. J. A. Wang and K. Qian. *Component-Oriented Programming*. John Wiley & Sons, May 2005.

[WSBG08] V. Watine, V. Seignole, H. Balp, and G. Gailliard. IDL-to-VHDL Mapping Initial Requirements, OMG Washington Technical Meeting. `www.omg.org/docs/mars/08-03-17.pdf`, March 2008.

[WT08] LIFL Laboratory West Team, DaRT Project. GASPARD2 (Graphical Array Specification for Parallel and Distributed Computing) MPSoC co-modeling tool. `www2.lifl.fr/west/gaspard/`, 2008.

[WX07] Y. F. Wu and Y. Xu. Model-Driven SoC/SoPC Design via UML to Impulse C. In *Proceedings of the UML for SoC Design Workshop*, June 2007.

[WZZ+06] Ying Wang, Xue-Gong Zhou, Bo Zhou, Liang Liang, and Cheng-Lian Peng. A MDA based SoC Modeling Approach using UML and SystemC. In *Computer and Information Technology, 2006. CIT '06. The Sixth IEEE International Conference on*, pages 245–245, Sept. 2006.

[Xil07] Xilinx. Virtex-II Pro and Virtex-II Pro X FPGA User Guide v.4.2. `www.xilinx.com`, Nov. 2007.

[YMS+98] J. S. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A. R. Newton. Design and specification of embedded systems in Java using successive, formal refinement. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 70–75, New York, NY, USA, 1998. ACM.

[Zim80] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, Apr 1980.

[ZRM06] Y. Zhang, J. Roivainen, and A. Mammela. Clock-Gating in FPGAs: A Novel and Comparative Evaluation. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 584–590, 2006.

Abstract

This thesis deals with the hardware application of the software concepts of middleware and software architecture based on components, containers and connectors within *Field-Programmable Gate Arrays* (**FPGAs**). The target application domain is *Software Defined Radio* (**SDR**) compliant with the *Software Communications Architecture* (**SCA**). With the SCA, software radio applications are broken into functional waveform components to be deployed on heterogeneous and distributed hardware/software radio platforms. These components provide and require abstract software interfaces described using operation signatures in the *Unified Modeling Language* (**UML**) and/or the *Interface Definition Language* (**IDL**) of the *Common Object Request Broker Architecture* (**CORBA**) middleware, both standardized by an international software industry consortium called *Object Management Group* (**OMG**). The portability and reusability needs of these business components require that their abstract interfaces defined at a system level are independent of a software or hardware implementation and can be indifferently translated into a software programming language like C/C++, a system language like SystemC at transaction level (*Transaction Level Modeling* - **TLM**), or a hardware description language like VHDL or SystemC at *Register Transfer Level* (**RTL**). The interoperability need of SDR components requires transparent communications regardless of their hardware/software implementation and their distribution. These first needs were addressed by formalizing mapping rules between abstract components in OMG IDL3 or UML2, signal-based hardware components described in VHDL or SystemC RTL, and system components in SystemC TLM. The second requirement was addressed by prototyping a hardware middleware using transparently memory mapping and two message protocols: CORBA *General Inter-Object Request Broker Protocol* (**GIOP**) and SCA *Modem Hardware Abstraction Layer* (**MHAL**).

Résumé

Cette thèse s'intéresse à la déclinaison matérielle des concepts logiciels d'intergiciel et d'architecture logicielle à base de composants, conteneurs et connecteurs dans les réseaux de portes programmables in situ (*Field-Programmable Gate Array* - **FPGA**). Le domaine d'applications ciblé est la radio définie logiciellement (*Software Defined Radio* (**SDR**)) conforme au standard *Software Communications Architecture*) (SCA). Avec le SCA, les applications radio sont décomposées en composants fonctionnels, qui sont déployés sur des plateformes radios hétérogènes et distribuées. Ces composants fournissent et requièrent des interfaces logicielles abstraites décrites sous forme de signatures d'opérations dans le langage de modélisation unifié appelé *Unified Modeling Language* (**UML**) et/ou le langage de définition d'interface (*Interface Definition Language* - **IDL**) de l'intergiciel **CORBA** (*Common Object Request Broker Architecture*) standardisé par un consortium industriel appelé *Object Management Group* (**OMG**). Les besoins de portabilité et de réutilisation de ces composants requièrent que leurs interfaces abstraites définies au niveau système soient indépendantes d'une implémentation logicielle ou matérielle et puissent être indifféremment traduites dans un langage de programmation logiciel tel que C/C++, un langage système tel que SystemC au niveau transaction (*Transaction Level Modeling* - **TLM**), ou un langage de description matériel tel que VHDL ou SystemC au niveau registre (*Register Transfer Level* - (**RTL**)). Le besoin d'interopérabilité de ces composants requière des communications transparentes quelques soient leur implémentation logicielle ou matérielle et leur distribution. Ces premiers besoins ont été adressés en formalisant des règles de mise en correspondance entre des composants abstraits en OMG IDL3 ou UML2, des composants matériels à base de signaux en VHDL ou SystemC RTL, et des composants systèmes en SystemC TLM. Le deuxième besoin a été adressé en prototypant un intergiciel matériel utilisant de façon transparente le mapping mémoire et deux protocoles messages: CORBA *General Inter-Object Request Broker Protocol* (**GIOP**) et SCA *Modem Hardware Abstraction Layer* (**MHAL**).