University of Paderborn
Heinz Nixdorf Institute
Fürstenallee 11
33102 Paderborn

Modeling and Automated Synthesis of Reconfigurable Interfaces

Dissertation

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering and Mathematics
of the
University of Paderborn
in partial fulfillment of the requirements for the
degree of *Dr. rer. nat.*

Stefan Ihmor

Paderborn, Germany
September 14, 2006

Supervisors:

1. Prof. Dr. Wolfram Hardt, Chemnitz University of Technology
2. Prof. Dr. Franz J. Rammig, University of Paderborn
3. Prof. Dr. Daniel D. Gajski, University of California, Irvine

Date of public examination: 22 November, 2006

Acknowledgements

This work was carried out at the Department of Computer Science, Electrical Engineering and Mathematics of the University of Paderborn from 2002 to 2006. Initially starting my Ph.D. in the Informatik und Prozesslabor (IPL), I finished this thesis at the Heinz Nixdorf Institute (HNI), an interdisciplinary center of research and technology of the University of Paderborn.

First of all, I would like to express my gratitude to my supervisors, Professor Rammig and Professor Hardt (Chemnitz University of Technology). I consider it a great accomplishment to have completed this dissertation under their direction. I was very fortunate to work as a member of both chairs. Therefore, I thank Professor Hardt and Professor Rammig for their helpful advice, their excellent support and their constant guidance during this work.

Within the working groups of both professors, I had the opportunity to work on research projects, to supervise student projects, and to advise several students on their bachelor and master theses. All of this, including my involvement in the lectures of Prof. Rammig and Prof. Hardt, has given to me a detailed insight into the wide field of embedded systems.

In the same way I want to thank my colleagues Klaus Danne, Florian Dittmann, Marcelo Götz, Tales Heimfarth, Peter Janacik, Christophe Bobda, Thomas Lehmann, Achim Rettberg, and Mauro Zanella for their cooperation in different projects. Their constructive criticism improved the quality of the thesis considerably and brought novel ideas into my work. Furthermore, I am particulary thankful to my colleagues Markus Visarius and André Meisel from the Chemnitz University of Technology for their cooperation and interesting discussions.

I also thank my working students Michel Camel Kouamo Sime, Bertrand Gnokam Defo and all Bachelor and Master Students who implemented a considerable part of my concepts. Thanks also to Sheila Fleißner who spent a lot of time to revise the language of the manuscript.

Finally, I would like to express my gratefulness to my dear parents Anna and Gerhard. Their care and never ending support were a great motivation for me.

Paderborn, September 2006

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

1.1 Motivation and Challenges

Most embedded systems consist of distributed but highly interconnected applications. Due to the growing complexity and number of interacting components, the design of communication has become a central aspect along with the system functionality itself. Hence, many resources are invested to design and implement reliable and efficient communication systems.

Abstraction in the form of system-level design is one way to cope with the complexity of designing such systems. Model-driven development and hardware/software codesign are also well-known methodologies of abstraction. As the major drawback, abstraction involves the *Design Gap* that characterizes the distance between the concept and the final implementation of a design. A common technique to overcome the Design Gap is to sequentially derive a more precise implementation from the abstract model in an iterative process. Therefore, a particular implementation is reapplied as input for the next iteration, also denoted as the synthesis step. In many cases, EDA (Electronic Design Automation) tools combine particular synthesis steps into an integrated design flow.

The automation of synthesis steps is an important feature of a design flow. It allows us to evaluate even complex models and speeds up the synthesis process. Additionally, the design flow becomes less error prone and it helps to create implementations for platforms with limited resources and specific characteristics. Furthermore, automation facilitates to enforce compliance with design rules or coding styles and thus, simplifies the reuse and maintainability of the created code. Automated testing and verification allows us to increase the code quality and is essential during the development of safety critical systems.

The increasing complexity and the shorter time-to-market of embedded systems have forced hardware designers to consider reuse of Intellectual Properties (IPs). The composition of IPs is one of the most central but also error-prone parts during system integration. Since most IPs are provided by different vendors, they comprise incompatible interfaces and protocols. In this case, the designer has to create an adapter or wrapper for the integrated IP. A full

understanding of all interfaces is required for this process, which is complicated especially for hard IPs where the designer has to rely on informal descriptions. We can observe a similar development for reusable software components, called COTS (Commercial Off-The-Shelf).

An outstanding class of computing systems is denoted by the term *reconfigurable systems*. Beginning with first approaches in the 60's [111, 110], reconfigurable computing became very popular, especially due to the enormous technical improvements within the last decade. Specific hardware architectures have been developed like Field Programmable Gate Arrays (FPGAs), Programmable Logic Arrays (PLAs), and Complex Programmable Logic Devices (CPLDs). These platforms offer the ability to configure the implemented hardware circuit offline (*configuration, offline* or *static reconfiguration*). Nevertheless, some FPGAs even provide dynamic reconfiguration during runtime (*online* or *runtime reconfiguration*).

Next to the implementation of the applications, the inter-module communication represents a critical aspect in reconfigurable computing. In particular, module interdependencies in real-time environments lead to massive reconfiguration efforts. The runtime reconfiguration of modules has to be handled, while the synchronization and inter-module communication of the remaining system has to be kept running to avoid communication gaps.

We classify relevant methodologies that explicitly handle interfaces as follows:

Interface-Aware (System-Level) Design Flows

This class copes with interfaces as part of integrated design flows on a high-level of abstraction. In most cases, the interfaces are derived as a consequence of a successive refinement process. On the one hand, default-interfaces with standardized protocols are applied; on the other hand, the interfaces result from user-specific component libraries. If generic interfaces are used, the designer parameterizes them during the steps of the refinement process.

Interfaces in Reconfigurable Systems

Since powerful reconfigurable architectures are available, various approaches have been developed to cope with reconfigurable computing. Even nowadays, technical platform details set up a significant barrier in the design process of reconfigurable systems. This is especially true for the interfaces within reconfigurable systems, which require a substantiated knowledge about hardware programming. Approaches that focus on the communication in reconfigurable systems concentrate on restricted sets of components, comprising homogenous interfaces that consist of compliant protocols between the exchanged modules.

Dedicated Interface Synthesis Approaches

The approaches within this class are designed to synthesize dedicated protocol adapters for components with incompatible protocols. Protocol state machines or regular expressions are applied to model the interfaces that have to be interconnected. Specific algorithms create the adapter modules based on the modeled interfaces in a completely or partially automated way. Until now, the reconfiguration aspect was not considered by approaches of this class.

We can find various representatives for the three classes in literature. The most relevant approaches will be presented in Chapter 3. However, we lack an appropriate methodology that provides an integrated design flow that specializes in the synthesis of dedicated protocol adaptation exploiting reconfigurable computing paradigms like runtime reconfiguration.

1.2 Aim of the Thesis

The aim of this thesis is to provide a methodology for the communication-based design of runtime reconfigurable embedded systems. The center stage of this methodology is an integrated design flow that consolidates interface synthesis techniques with reconfigurable computing concepts as presented in Figure 1.1. The major objective of the presented design flow is the automated generation of an interface adapter module that interconnects tasks and media comprising incompatible interfaces.

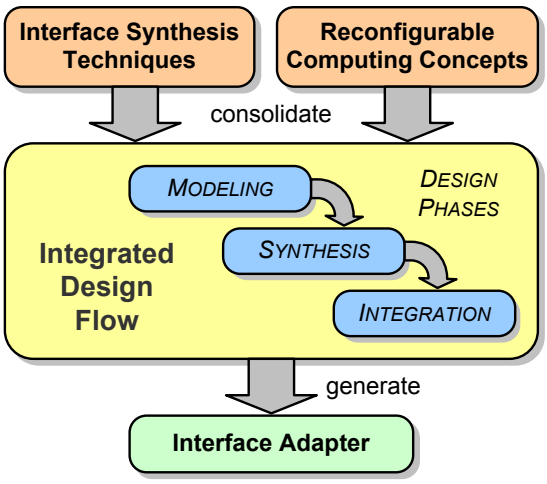


Figure 1.1: Integrated Design Flow: Overview

Integrated Design Flow

The designated fields of application regarding our approach are the reconfigurable computing domain, the rapid system prototyping of embedded systems, and the IP-based System-on-Chip design. An analysis of these domains clarifies the intention of our methodology and helps us to derive important requirements for the design flow.

A fundamental limitation of IP-based design is that, apart from an initial configuration, the IPs may not be modified anymore during the integration process. In most cases, an alteration of an IP would imply a breach of warranty, and, in the case of hard IPs, a modification would not even be technically feasible. For this reason, the integration and flexible reuse of IPs has to be considered by our design flow.

Rapid system prototyping demands a high degree of flexibility in the composition of heterogeneous components and the possibility to create solutions very quickly. Therefore, our design flow has to provide a maximum degree of adaptability in combination with an efficient generation process of the interface adapter modules.

Another aim of this thesis is to close the gap between hardware and software. Therefore, our design methodology covers the scope of hardware as well as software interfaces. However, in this thesis we restrict ourselves to software interfaces in the form of memory mapped I/O. Message based communication like function calls cannot be expressed in our models. In this way, we can treat hardware and software interfaces in consistent manner.

The interface adapter itself can be implemented either in hardware or software. This decision is a challenge in the field of hardware/software codesign and depends, for example, on the available execution platforms. In this thesis, we concentrate our synthesis efforts on the creation of the hardware realization, which allows us to employ hardware paradigms like parallelism and analysis techniques, like a cycle accurate schedulability analysis.

To interconnect tasks with media, we model the interfaces of tasks and media in a unified manner. Restricting ourselves to the observable behavior of an interface, we do not consider the internal nature of tasks and media and therefore, treat them as a black box model. In this way, also the inner functionality of an IP, which represents a task or medium, remains a black box and does not have to be understood in order to be integrated.

To cope with the complexity of all theses requirements, the design flow is structured into three consecutive design phases as presented in Figure 1.1. Within the modeling phase, we define the communication infrastructure that comprehends a description of each interacting component. Our model has to provide all the information that is required for the synthesis phase. The particular synthesis steps create an implementation of the adapter module in a defined target language in an automated way. Finally, this implementation has to be integrated into the ongoing system design process.

Automated Interface Synthesis

It is a primary goal of our design flow to generate transparent interface adapter modules. Transparency is fundamental for the protocol adaptation of tasks that are not aware of an interconnecting interface. Another requirement is to support synchronous and asynchronous I/O. As usual for interfaces, the adapter module must not loose or unpredictably tamper with the transferred data. To cope with fail critical systems, the interface has to provide strategies to detect and handle communication errors. Furthermore, we have to derive a maximum of information from the modeling phase as input for the synthesis process to optimize the degree of automation. Thereby, we avoid the creation of product automata to create resource efficient implementations. In our case, the well-known data processing concept *Input–Processing–Output* provides the basic principle for the modular structure of our interface adapter. The output's level of abstraction depends on the deployed low-level synthesis flow that we apply for the integration of the generated adapter module.

Runtime Reconfigurable Protocol Adapter

Runtime reconfiguration becomes critical whenever it affects the interaction of components. This is especially true in the case of real-time systems, which demand a deterministic behavior during runtime. For this purpose, existing approaches evade disturbing effects that result from the reconfiguration of interacting components. A well-known problem, for example, is the absence of tasks due to partial runtime reconfiguration. In general, the remaining system does not consider the reconfiguration of particular components and thus, assumes the complete behavior to be present all the time. To guarantee a continuous and thereby deterministic system behavior anyway, the model and the implementation of our interface adapter have to support the reconfiguration of particular tasks. Therefore, our approach deploys and extends existing reconfigurable computing techniques. One way to improve the interface adapter module is to perform a reconfigured execution. In this case, we apply reconfiguration techniques to optimize the area consumption, which is required to execute the hardware implementation on a runtime reconfigurable platform.

1.3 Contribution of the Thesis

Our contribution to the field of communication-based design is the Interface Synthesis (IFS) approach that combines the modeling and automated synthesis of runtime reconfigurable interfaces. We developed the *Interface Synthesis Design Flow* to create interface adapter modules, called *Interface Block* (IFB), to interconnect tasks and media comprising interfaces with incompatible protocols.

Interface Synthesis Design Flow and Interface Block

The Interface Synthesis Design Flow is structured as presented in Figure 1.1 and starts with a UML 2.0 based modeling. To provide a standardized and intuitive graphical user interface (GUI) we developed a UML2.0 profile, customized for the Interface Synthesis Design Flow. We will demonstrate that this is an adequate way for the model-based design of real-time interfaces in embedded systems.

We developed the IFS System Architecture to design complex communication scenarios. The System Architecture describes a hierarchical structure composed of architectural components (system, board, chip) and interacting communication components (task, medium). To perform an automated synthesis we require information about the target platform and the communication components that we are going to interconnect. Our design methodology distinguishes between statically determinate interface descriptions (IFD) of communication components, and a mapping of the exchanged data (IFD-Mapping) that has to be defined each time we create an IFB. In this way, we avoid defining a global data semantics for IFDs and earn a maximum degree of freedom in interconnecting heterogeneous components.

The modeling phase is followed by an automated interface synthesis that results in a target language independent IFB model. A subsequent code generation creates an executable IFB implementation that can be integrated into a standard design flow. Based on the modular design of the IFB we developed a hardware template on the RT (Register Transfer) level that allows us to create runtime reconfigurable adapter modules. Due to the specific architecture of the IFB Hardware Template, which defines a precise construction pattern for the hardware implementation, we can perform a cycle-accurate evaluation of the IFB. With this information, we can accomplish a precise schedulability analysis of the protocol adaptation even for complex scenarios including several communication partners.

To guarantee predictability during runtime reconfiguration, our approach employs reconfigurable computing techniques. Thus, whenever a connected task is exchanged, the affected parts of the IFB are reconfigured as well, and a predefined behavior is executed during the reconfiguration process. In combination with a reconfiguration control unit (RCU) the IFB provides a basis technology to compose embedded systems that allow deterministic reconfiguration of tasks and media at runtime.

EDA Tool: IFS-Editor

To evaluate our Interface Synthesis methodology, we have developed an EDA tool called IFS-Editor. This Java-tool provides complete functionality to generate runtime reconfigurable IFBs. Modeling, synthesis, and code generation, are particular design phases that have been implemented as functional entities which are closely linked. Interactive wizards guide the designer through the synthesis and the code generation phases.

The design entry can be performed in accordance to the defined UML 2.0 profile by the UML case tool *Fujaba* (From UML to Java and back again [69]). A model transformation allows automatic transferring of a design from Fujaba to the IFS-Editor [36].

Currently, the code generation has been developed for the hardware description language VHDL. This allows us to create the synthesizable VHDL code of an IFB. Finally, in the integration phase, the created IFB VHDL code can be integrated into the system design. Standardized tools for the low-level hardware synthesis, for example Xilinx ISE, or compiler (like the GNU C-compiler (gcc)), are supposed to be used to create the final configuration bit-streams or the executable files, respectively.

1.4 Organization of the Work

This thesis is organized as follows:

Chapter 2 describes a communication framework for embedded systems and defines our terminology in the context of the IFS methodology. We present our interface description format and define our System Architecture model. Starting from this, we deliver scenarios for the adaptation of interfaces and discuss the role of reconfiguration.

Chapter 3 provides a survey of related work. We discuss related approaches and highlight analogies and differences to the presented one. In accordance with the motivation, we distinguish between interface-aware (system-level) design flows, dedicated interface synthesis approaches, and interfaces in reconfigurable systems.

Chapter 4 presents the Interface Synthesis methodology in the form of an integrated design flow. We introduce the concepts of the Interface Synthesis Design Flow and define our modeling concept. Furthermore, we explain the structural composition of the Interface Block and describe the functionality of its components. Afterwards, we present the methodology and the realization of the runtime reconfigurable interface block.

Chapter 5 refers to the details of the Interface Synthesis Design Flow. Here, we discuss the consecutive design steps: modeling, synthesis, code generation, and code integration. To provide an abstract and intuitive modeling of the System Architecture, we define a specific UML 2.0 profile and the related model transformation. Afterwards, we focus on the developed IFB synthesis algorithms and the code generation techniques.

Chapter 6 specializes on the hardware implementation of the Interface Block. We explain the IFB Hardware Template and perform a cycle accurate timing analysis for the schedulability analysis of multiple adapted protocols. Furthermore, we deliver two optimization approaches, which minimize the latency and the required chip area.

Chapter 7 discusses the implementation of the Interface Synthesis Design Flow in the form of the IFS-Editor. This Java tool represents an adequate EDA tool that implements all major aspects of the Interface Synthesis Design Flow. We provide a representative case-study to evaluate the IFB, and compare our results with related work.

Chapter 8 summarizes the presented work. It concludes this thesis and gives a brief outlook on future work in the field of the Interface Synthesis Design Flow and the automated synthesis of Interface Blocks.

CHAPTER 2

Communication Framework for Embedded Systems

Developing an integrated design flow that provides a high-level modeling and an automated synthesis of interface adapter modules is not an easy task. To make this feasible, we require a *framework* that characterizes our approach precisely and identifies its intention. To introduce into the Interface Synthesis methodology we begin this chapter by systematically developing this communication framework. Within the framework, we define our terminology, specify dedicated fields of application, and describe the challenges that our methodology copes with. Furthermore, we discuss the relevance of the system architecture, with a focus on the various interfaces. To exemplify the derived concepts, we present the Virtex-II Pro FPGA. The chapter ends with a discussion about the role of reconfiguration in embedded real-time systems and introduces the term "communication gap" as the main challenge for our approach which results from runtime reconfiguration.

2.1 Framework

Terms like *interface* and *synthesis* have several meanings throughout the computer science domain. Thus, before presenting our Interface Synthesis approach, we have to outline a framework that places our work into correct context. First of all, our approach focuses on the embedded systems domain. While small platforms comprise few resources and a tiny functionality, modern embedded systems constitute an increasing part of our technological environment. We can find them in the form of highly integrated System-on-Chip solutions as well as in the form of extensive architectures, as known from automobiles or aircrafts. Most of the complex embedded systems are distributed and consist of several heterogeneous software and hardware platforms.

Based on the heterogenity of complex embedded systems, we can find multifaceted kinds of interfaces. Our approach handles the adaptation of such interfaces through a wide spectrum of embedded systems. We can find a high variety in embedded systems since their composition mainly depends on the desired functionality. To better confine our framework, we will now classify the terms tasks and media.

2.1.1 Tasks & Media

In contrast to general-purpose computers, embedded systems are subject to specific conditions and, in general, perform a pre-defined function. This function can be partitioned into units, called tasks. Media are used to interconnect tasks but do not provide any computing power. Tasks and media can be implemented in either software or hardware. Examples for tasks and media are given in Tabular 2.1.

	Hardware	Software
Tasks	- Fast Fourier Transformation - Distance Sensor	- Electronic Stability Program - Park Distance Control
Media	- Firewire, CAN, RS232 - AMBA	- SOAP - TMO

Table 2.1: Examples for hardware and software

The Fast Fourier Transformation (FFT) is a typical representative of a hardware task. The FFT algorithm does not change during runtime and is available as efficient hardware implementation, which grants a high performance. In our methodology, dedicated hardware components like a distance sensor could also be treated as a hardware task, as long as it comprises a digital interface. Applications like the Electronic Stability Program (ESP) or the Park Distance Control (PDC) are exemplary software tasks, which usually communicate via function calls or memory mapped I/O to interact in a distributed environment.

Familiar examples for media in the form of hardware are Firewire, CAN, or RS232, which can be found in embedded real-time systems. The Advanced Microprocessor Bus Architecture (AMBA) is a well-known candidate for the interconnection of hardware tasks in the System on-Chip design. In contrast, SOAP (originally Simple Object Access Protocol) or TMO (Time-triggered Message-triggered Object) are representatives for media in software-based systems. SOAP, in combination with TCP/IP based web-services as well as TMO denote a service-oriented middleware that establishes the communication between software tasks.

In our terminology, a medium neither specifies a communication channel in the meaning of communications engineering nor a “primitive wire” between two tasks. Comparable to tasks, each medium consists of interfaces comprising a self-contained communication protocol.

2.1.2 System Composition

We distinguish between two kinds of execution platforms in embedded systems: software execution platforms like processors and hardware devices, which can be either dedicated or programmable. The selection of particular execution platforms or in general, the *allocation* of architectural components as well as the *binding* of tasks and media to these components is a well understood problem [232]. Therefore, we treat the allocation and binding process as completed for our approach. The decision whether tasks or media are implemented in hardware or in software is not considered in this work either, as is challenge of the *hardware/software codesign* domain [179]. Thus, for our approach the system composition has to be completed, except for the adaptation of the remaining incompatible interfaces.

2.1.3 Hardware & Software Interfaces

If we partition an embedded system into its hardware and software components, we can identify up to three kinds of interfaces:

- Hardware/Hardware interfaces
- Software/Software interfaces
- Hardware/Software interfaces

Hardware/Hardware Interfaces

Typical hardware/hardware interfaces are related to hardware execution platforms and can be found in the hardware development process as, for example, in the IP-based design. Here, a designer has to connect the interfaces of particular IPs with other IPs or with a full-custom design. In the case of VHDL, interfaces are represented by entities, which are modeled in the form of port descriptions.

Example-Code 2.1: Exemplary VHDL Entity

```
01  ----------------------------------------------------------------
02  -- Entity: Hardware interface for a fully interlocked protocol
03  ----------------------------------------------------------------
04  ENTITY hw_interface IS
05   PORT( DATA : INOUT std_logic_vector(7 downto 0);
06         ACK  : IN    std_logic;
07         RDY  : OUT   std_logic );
08  END hw_Interface;
```

Example-Code 2.1 presents the interface (VHDL uses the keyword *entity* for this purpose) of a fully interlocked protocol which includes a bidirectional `data` word, as well as a `ready` and an `acknowledge` signal of the standard logic type (a multi value logic defined by the IEEE). Each signal represents an electrical signal. A digital logic applies the values *logic one* and *logic zero* to a signal. When a design is mapped to the target technology, the logical values are replaced by technology-dependent voltage levels.

Software/Software Interfaces

Basically, software communicates in the form of function calls. One special development is the memory mapped I/O, which is very common in embedded systems. In this case, the software applications communicate via read()- and write()-functions on a shared piece of memory that is mapped into the present address space. Therefore, the software has to be aware of the exact position (address) and the meaning of the exchanged bits inside the shared memory. Beginning with these two versions, we can find software/software interfaces modeled in the form of an Application Programmer Interface (API) or a memory map.

The example depicted in Figure 2.1 specifies a memory map in accordance to Example-Code 2.1. The memory defines the bit fields that are required for a fully interlocked protocol, including an eight bit data word, as well as the two `ready` and `acknowledge` bits. The given shared memory is mapped into a 32-bit address space and holds the 2^8 bits starting from the base address 0x1234FF00. The address of each bit is determined by summing up the *base address* and the particular *offset*. Therefore, the address of the `acknowledge` bit (Ack) in our example is 0x1234FF01.

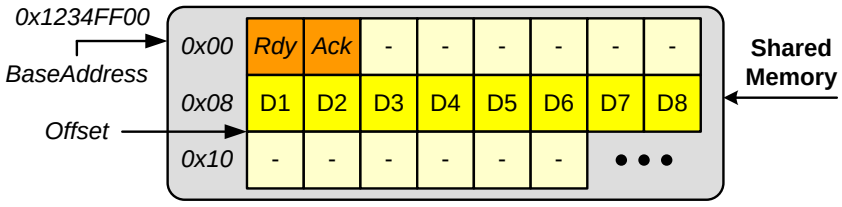


Figure 2.1: Specification of a memory map

Hardware/Software Interfaces & Memory Mapped I/O

From a technical point of view, the implementation of hardware/software interfaces is based on memory mapped I/O [144]. To exchange information between hardware and software we utilize a piece of shared memory, which can be available in the form of dedicated registers to the point of distributed RAM (Random Access Memory). Shared memory is addressed implicitly or explicitly. The status registers of a Central Processing Unit (CPU) [145], for example, are implicitly addressed; while the memory space of FireWire (IEEE 1394) expects the address to be given in addition to the data [48].

To characterize hardware/software interfaces more in more detail, we distinguish between two possible realizations for memory mapped I/O. As depicted in the middle of Figure 2.2, we assume a hardware device (bottom) sharing a piece of memory with a software process (top). The shared memory can be closer to the software as shown on the left side or be a part of the hardware design as presented on the right.

On the left side of Figure 2.2, a RAM is employed as shared memory for the memory mapped I/O. In this case, the software running on the processor can easily access the RAM by read()- and write()-functions [199]. A hardware circuit, which is synthesized into the FPGA, requires a separate memory interface to access the "external" RAM. Usually, this memory interface is realized by a memory bus IP, which is additionally integrated into the hardware design. In the other case, the memory is a part of the hardware design, as presented on the right side of Figure 2.2. Here, the memory is a set of dedicated registers (register file), which have been synthesized for the memory mapped I/O. While the hardware is directly interconnected to

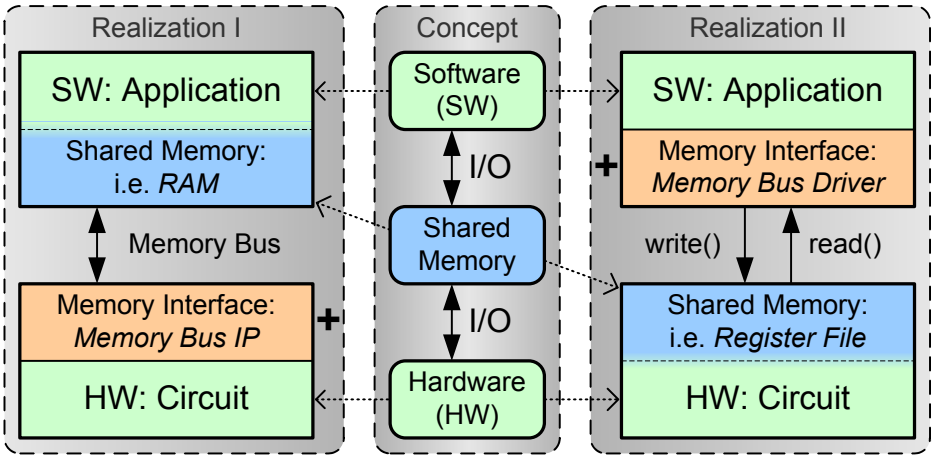


Figure 2.2: The hardware/software interface based on memory mapped I/O

the synthesized registers, the software running on the processor cannot directly access the shared memory. Therefore, we have to integrate a memory bus driver that handles the data transmission between the processor and the FPGA.

In general, our Interface Synthesis methodology can be applied to all three kinds of interfaces (Hw/Hw, Sw/Sw and Hw/Sw). As we concentrate on embedded systems and focus on the development of a hardware implementation of the Interface Block, we do not consider API-based communication of software tasks and media. Existing tools like the Common Object Request Broker Architecture (CORBA) provide a well-established technology that already handles this sector and so we are not going to re-implement it. Therefore, every time we mention software interfaces, we implicitly refer to memory mapped I/O.

2.1.4 Modeling Interfaces

As we will see, our understanding of tasks and media allows us to model hardware and software interfaces in a unified manner including port descriptions and memory maps. Restricting ourselves to the *observable behavior* of interfaces, we do not consider the internal nature of tasks and media and therefore, treat them as a black box. In this way, the inner functionality of an IP, which also represents a task or medium, remains a black box and does not have to be understood in order to be integrated. Therefore, from the communication point of view we can even use the item task and medium as synonyms.

The term *observable behavior* denotes the entire information about the I/O that crosses the border of a component through a particular interface. In the introduction, we had stated that our primary goal is the generation of *transparent* interface adapter modules. Therefore, it is obviously not only a restriction, but also a necessity to refer to the observable behavior in our interface model since a transparent interface adapter is never allowed to access task-internal signals. In our methodology, interfaces are defined by three aspects:

- Physical interface structure (Topology)
- Electrical properties (Nature)
- Communication protocol (Behavior)

A division of topology and behavior is already known from hardware description languages like VHDL [173, 180] and Verilog [223]. It allows the separated development and the reuse of particular code-fragments. Furthermore, we can find a comparable tripartition in existing meta-languages that are used to describe hardware IPs, for example, the VSIA format [131, 132]. Our interface description format (*IFD-Format*) actually became part of the IPQ format [239, 238], which is a standardized European format for the retrieval of IPs as part of the IP based design process [241, 240].

The presented tripartition is adequate to model reconfigurable distributed embedded systems. When tasks, which can be resident only within chips, are exchanged, the structure of all connected chip interfaces remains unchanged, although the behavior which is performed on these interfaces may vary. A good example is the Spyder FPGA-Board [51] that owns two expansion-headers, directly connected to the FPGA. Although the functionality performed on the I/O pins may vary, the structure of this chip interface is unchangeable.

Physical Interface Structure (Topology)

As we have seen in the previous example, the physical structure of an interface is fixed and thus does not change over time. In contrast, the behavior (communication protocol) that is performed by an interface can vary in programmable architectures. This is true for signal-based I/O as well as for memory mapped I/O. We apply a compact notation using set theory to define our models. To refer to objects inside these models we assign unique identifiers. Each identifier consists of a *name*, a *comment* and a locally *unique integer value*.

Definition 2.1 *Unique identifier to characterize objects*

$$\text{ID} := (\text{Name, Comment, ID}) \qquad \text{with ID} \in \mathbb{N} \wedge \forall x_i, x_j \in ID : x_i = x_j \Leftrightarrow i \equiv j$$

To model the physical structure of *signal-based I/O* we define a *signal-based interface* (SBIF). It comprises a number of *ports*, which in turn provide a number of *pins*. Each pin belongs to exactly one port. To provide the maximum degree of freedom, each pin owns a *Direction* {input, output, bidirectional} and a *Data-Type* {bit, std_logic, GND, VCC, other}.

Definition 2.2 *Signal-Based Interface (SBIF)*

$$\begin{aligned}
&\text{SBIF} := (\text{Ports, Pins}) \\
&\textit{port} \in \text{Ports} = (\text{ID, PinList}) && : \text{Define } \textit{port} \\
&\textit{pin} \in \text{Pins} \quad = (\text{ID, } \textit{Direction}\text{, } \textit{Data-Type}) \in \text{PinList} && : \text{Define } \textit{pin} \\
&\forall\, i, j \mid i \neq j : \text{PinList}_i \bigcap \text{PinList}_j = \varnothing && : \text{Unique } \textit{pin} \text{ assignment}
\end{aligned}$$

Like signal-based interfaces, *shared memory* consists of a number of *registers*, which comprise a number of *bits* that provide an ID and a *Direction* {read, write, read/write}. Each register is parameterized by a *base address*, the *accessible data size* $(1, ..., \text{n bits})$, the *minimum allocation size* and the *maximum allocation size*. The accessible data size is a memory specific value and defines the amount of bits that can be written or read at the same time. Minimum- and maximum allocation size specify a number of "words" in the dimension of the accessible data size. To get the minimum (maximum) number of bits that can be accessed simultaneously we multiply the minimum (maximum) allocation size by the accessible data size. The total register-size results from the number of bits modeled inside the bit-list.

Definition 2.3 *Memory Mapped Interface (MMIF)*

$$\begin{aligned}
&\text{MMIF} := (\text{Registers, Bits}) \\
&\textit{register} \in \text{Registers} = (\text{ID, BaseAddress, AccessibleDataSize,} \\
&\qquad\qquad \text{MinAllocSize, MaxAllocSize, BitList}) && : \text{Define } \textit{register} \\
&\textit{bit} \in \text{Bits} \qquad = (\text{ID, } \textit{Direction}) \in \text{BitList} && : \text{Define } \textit{bit} \\
&\forall\, i, j \mid i \neq j : \text{BitList}_i \bigcap \text{BitList}_j = \varnothing && : \text{Unique } \textit{bit} \text{ assignment}
\end{aligned}$$

When we introduced hardware and software interfaces, we presented examples for both kind of interfaces. The interface given in Example-Code 2.1 depicts an SBIF containing the three ports: DATA, ACK and RDY. The DATA-port consists of eight bidirectional pins of the type std_logic. The following two ports (ACK and RDY) are 1-bit std_logic signals which are incoming and outgoing, respectively. The MMIF presented in Figure 2.1 defines exactly one register. The register starts at base Address 0x1234FF00 and its accessible data size is 1 bit. To access the Ack- and Rdy-bit the minimum allocation size is 1, where the maximum allocation size is 8 (for the 8-bit Data bit-field). To allow also read and write operations, the direction has to be read/write.

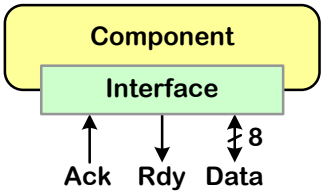


Figure 2.3: Interface topology for a fully interlocked handshake

To handle signal based I/O and memory mapped I/O *consistently* we define the *interface topology* that unifies shared properties of SBIF and MMIF. The topology considers the width (number of I/O signals) and the direction of I/O signals. Here, we apply the directions of a SBIF (input, output ...); the directions of a MMIF are mapped correspondingly. As depicted in Figure 2.3 the two delivered examples describe a *topologically equivalent interface*.

Definition 2.4 *Topology of an Interface*

Topology := (IOPorts, IOSignals)

$ioport \in$ IOPorts	$=$ (ID $\times$ IOSignalList)	: Define *ioport*
$iosignal \in$ IOSignals	$=$ (ID $\times$ Direction) $\in$ IOSignalList	: Define *iosignal*
$\forall\, i, j \mid i \neq j$: IOSignalList$_i \bigcap$ IOSignalList$_j$	$= \varnothing$	: Unique *iosignal* assignment

To interconnect interfaces, we have to ensure their *connectivity*. This means that all of the considered topology-parameters have to match except for the direction, which has to be inverted. A significant characteristic of each physical interface are its electrical properties.

Electrical Properties (Nature)

Electrical properties describe the characteristic *nature* of an interface in the form of its electronic parameters. These parameters could be used to retrieve specific devices or to create dedicated circuits related to the given electrical parameters. Until now, our approach does not handle this low-level adaptation of interfaces.

Otherwise, the electrical properties can be used to verify the electrical *compatibility* of two interfaces, which is a *necessary criterion* to read and write signals properly and thus, to guarantee a correct protocol adaptation.

Some examples for electrical properties, which are part of our interface description, are given afterwards. We distinguish between properties that carry a single value out of an enumeration and those, which define an interval:

- Enumeration: IO-Technology-Type $\in$ {TTL, Open Collector, LVDS, ... }
- Interval: Voltage $\in$ [V$_{Min}$, V$_{Max}$]
- Interval: Jitter $\in$ [J$_{Min}$, J$_{Max}$]

Two interfaces IF_A and IF_B are electrical compatible if the electrical properties P_{el} of two interfaces IF_A and IF_B are equal in the case of a single value:

$$P_{el}(IF_A) = P_{el}(IF_B)$$

or they share at least one common value in the case of an interval:

$$P_{el}(IF_A) \bigcap P_{el}(IF_B) \neq \varnothing$$

Communication Protocol (Behavior)

Next to the physical structure and the electrical properties, we have to describe the dynamic behavior of interfaces. Typically, the behavior of an interface is denoted as *communication protocol* or simply as *protocol*. An adequate model for us to describe protocols has to be applicable to our topology model and allows us to specify asynchronous and synchronous protocols, the values of all I/O signals over time, and the timing (for timed protocols).

As presented in Figure 2.4, which depicts a protocol including a fully interlocked handshake, *signal waveforms* provide a graphical method to represent communication protocols. They provide the advantage of describing the behavior of multiple signals over time. The particular values that a signal can take are represented by defined waveforms as visualized in Figure 2.4 (*logic one* → '1', *logic zero* → '0', *unknown* → 'X' and *high impedance* → 'Z'). The four values are a subset of the IEEE standard IEEE.STD_LOGIC_1164 [174]. A disadvantage of signal waveforms is that we can visualize only particular *traces* (or *scenarios*) [116].

To overcome this drawback, we model the entire communication protocol of (at least) two interacting component interfaces as a finite state machine (FSM) that represents the *complete waveform diagram*, considering all possible traces (see Figure 2.5). We call this automata the *waveform state machine* (*WSM*). Relating to the WSM, a signal waveform diagram is nothing more than the visualization of a particular path through the WSM [146]. As demonstrated in Figure 2.4, we can model the state machine of our exemplary protocol by a sequence of five states, which would be repeated infinitely. We define the *WSM* as follows:

Definition 2.5 *Waveform State Machine* (*WSM*)

$Protocol := (S, TC, PP, \delta, \lambda, s_0)$

S	:	A finite number of states
TC	:	A set of *transition conditions*
PP	:	A set of *protocol pins*
δ	$: S \times TC \rightarrow S$	The *state transition function*
λ	$: S \rightarrow (Val \mapsto PP)$	The *(Moore) output function*
s_0	$\in S$	The *start state*

Similar to a FSM, the WSM owns a finite number of states S. $s_0 \in S$ is the start state, which means that the *current state* of the automata is initially set to this state. As depicted in Figure 2.5, the WSM implies all states of the interacting protocol state machines, which cause *observable effects* on the shared signals. In our *protocol synthesis*, we use the WSM as input to *construct* a *state machine* inside the interface adapter which interacts with the task. In Section 5.2.5 we define the detailed functionality of our protocol synthesis algorithm.

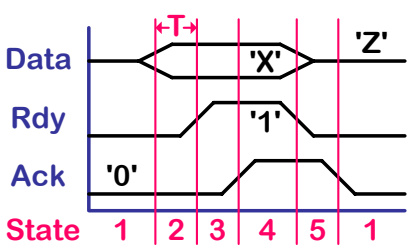


Figure 2.4: Signal waveform annotated with FSM values (states + timing)

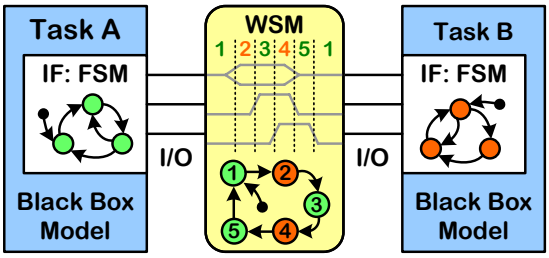


Figure 2.5: The shared communication protocol modeled as WSM

The state transition function $\delta : \mathit{current\,state} \times \mathit{transition\,condition} \rightarrow \mathit{next\,state}$ allows us to define synchronous as well as asynchronous transitions. The decision for one of the two kinds depends on the signals that are triggered by the transition conditions (TC), which represent the *dependencies* within the waveform diagram. We distinguish between *signal based* dependencies and *time* dependencies. Under certain conditions, we can identify the signal based dependencies automatically as far as they result from observable signal events. We discuss these conditions in Section 5.2.5. As time is a non observable parameter, we have to model it explicitly. Therefore, the TC of the WSM specify the timing in the case of timed protocols. We distinguish between three types of timing:

- clocks,
- timer (= deadlines) and
- periodic time schedules.

Clocks are used to model periodic signal events with a constant frequency. *Timer* and *deadlines*, which are in fact the same, describe time intervals, which exceed after the *TimerOrDeadline* has been started. We have seen an example of a timer in Figure 2.4. Here the time T specifies that amount of time, which the data signals require to become stable. Real-time systems often use predefined time schedules to define a global timing as with TTP [108, 167]. Particular events are modeled within a hyper-period that is executed repeatedly. All events, which are related to a transition condition, have to be fulfilled to let the transition fire. Including the given timing model, we define transition conditions as:

Definition 2.6 *Transition Condition (TC)*

TC := [*Cond* { (and | or | nand | nor | xor) *Cond*) }] (Extended Backus-Naur form)
Cond ∈ {*Clock*'event, *TimerOrDeadline* exceeded, *Event* ∈ *Periodic Schedule* arrived}

The output function $\lambda : S \rightarrow (Val \mapsto PP)$ defines Moore-outputs, which assign a value Val to a protocol pin PP depending on the current state. *Protocol pins* are virtual I/O signals, which are required to describe protocols independently of a particular topology. Protocol pins and its allowed values are defined as follows:

Definition 2.7 *Protocol Pin* (*PP*)

PP := (ID, *Direction*, *Data-Type*, *Behavior*)
Direction = {incoming, outgoing, bidirectional}
Data-Type = {bit, std_logic, other}
Behavior = {control, data, data+control}

Definition 2.8 *Signal Value* (*Val*) assigned to PP

Val := (*Value*, *Direction*, *UseCase*)
Value = {logic one, logic zero, unknown, high impedance}
Direction = {incoming, outgoing}
UseCase = {control, data}

Each PP owns the parameters direction, data-type, and behavior, which define the general characteristics of the virtual I/O signal. Each value, which is assigned to a PP, specifies the status for exactly one particular state and has to match the general PP parameters. Thus, a bidirectional data+control PP can hold incoming data in one state and outgoing control information in another state. However, we would not be allowed to assign outgoing data to an incoming control PP or high impedance to a PP of the type bit.

The application of protocol pins allows a separated development and a more comfortable reuse of topologies and protocols. However, this separation needs the mapping of protocols to topologies, the so-called *protocol-map*. Each entry of the protocol-map allocates exactly one matching I/O signal of the physical structure for every protocol pin.

Definition 2.9 *Protocol-Map*

$$
\begin{array}{ll}
\textit{Protocol-Map} := (\textit{Topology}, \textit{Protocol}, \gamma) & \\
\textit{Topology} : & \text{Given topology} \\
\textit{Protocol} : & \text{Given protocol} \\
\gamma \quad : pp \in PP \rightarrow x \in \{Pins, Bits\} & \text{PP } \textit{mapping function} \\
\forall i, j, m, n \mid i = j : pp_m \rightarrow x_i \wedge pp_n \rightarrow x_j \Rightarrow pp_m \equiv pp_n & \text{Max. one PP per Bit/Pin}
\end{array}
$$

Each pin or bit within the topology may be allocated by a maximum of one protocol pin. If there are not enough pins or bits, the protocol cannot be executed on this topology. Unused pins or registers do not cause problems except that they are wasted resources. We know this effect from scalable protocols like the RS232. In the simplest version, this protocol uses only two pins of a Sub-D connector. The most complex version requires all nine pins.

We may couple two protocols if they are *compliant* to each other (see Figure 2.6). This is a metric, different from the functional equality. On one hand, the two state machines may never halt (end state) or run into a deadlock. On the other hand, the modeled times have to match and the transferred data has to be semantically compatible. With our current model, we cannot determine automatically whether two interfaces are compliant, or not.

Including the given definitions, we are now able to model an interface description as:

Definition 2.10 *Interface Description (IFD)*

$$\textit{IFD} := (\textit{Topology}, \textit{Nature}, \textit{Protocol}, \textit{ProtocolMap})$$

Based on these IFDs our interface synthesis algorithm will automatically synthesize the protocol state machines inside the IFB, which interact with the connected tasks and media. This level of abstraction appears to be rather deep for the description of interfaces in a high-level synthesis approach. Nevertheless, we require a bit and cycle accurate behavior model for our IFB synthesis, which is applicable to signal-based as well as to memory mapped I/O.

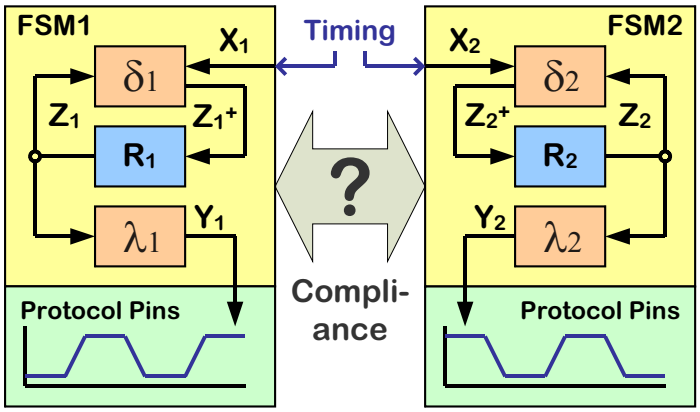


Figure 2.6: Compliance of two waveform state machines

2.1.5 Scenarios for Interface Adaptation

Based on the *similarity criteria* that we introduced for the topology, the nature, and the protocols we now can discuss characteristic scenarios for the interconnection of interfaces.

I) No compatibility (electrical properties): At our level of abstraction, the (electrical) compatibility of interfaces is a necessary criteria for the interface adaptation. Without it, we must not interconnect physical signals, or we cannot guarantee the correct handling of the electrical signal values.

II) Compatibility, connectivity (topology) and compliance (protocol) satisfied: The given interfaces are fully matched and can be directly interconnected.

III) Compatibility and compliance satisfied, but no connectivity: In this special case, we hold compliant protocols that are performed on physically different interfaces. As each protocol requires a number of I/O signals, the compliant protocols must possess a common subset of PPs. Therefore, we just have to interconnect this subset of I/O signals from each interface with each other to adapt the heterogeneous interfaces.

IV) Compatibility but no compliance: Independent of the connectivity, this very common scenario describes the *classical case of interface adaptation*. Two heterogeneous interfaces, which cannot be directly interconnected, have to be adapted. This scenario also defines the **field of application** for our **Interface Synthesis approach**.

2.2 System Architecture

An important aspect of an embedded system is the underlying system architecture. The hardware and software execution platforms of distributed embedded systems are closely coupled to execute the desired functionality. As one part of this functionality, the system has to provide the computing power to perform and the resources to implement the protocol adaptation. In the case of developing the functionality of a complete system within a single component we are talking about *System-on-Chip* (SoC) *Design*. Otherwise, we have to handle distributed architectures consisting of multiple interconnected components.

2.2.1 The IFS System Architecture Model

The *IFS System Architecture* model has been developed to design complex communication scenarios. As depicted in Figure 2.7, the System Architecture defines a hierarchical structure composed of the *system components*: system, board, chip, task, and medium. To decide between passive and active elements, we distinguish between *architecture components* (system, board, and chip) and *communication components* (task, medium). The main objective of the System Architecture is to model the available communication infrastructure.

The *communication infrastructure* allows us to place interacting communication components not only in one component but to distribute them over several interconnected components. Therefore, they have to be connected via interfaces of the architecture components. Compatible communication components may directly communicate via the present interconnections while incompatible tasks are interconnected by Interface Blocks.

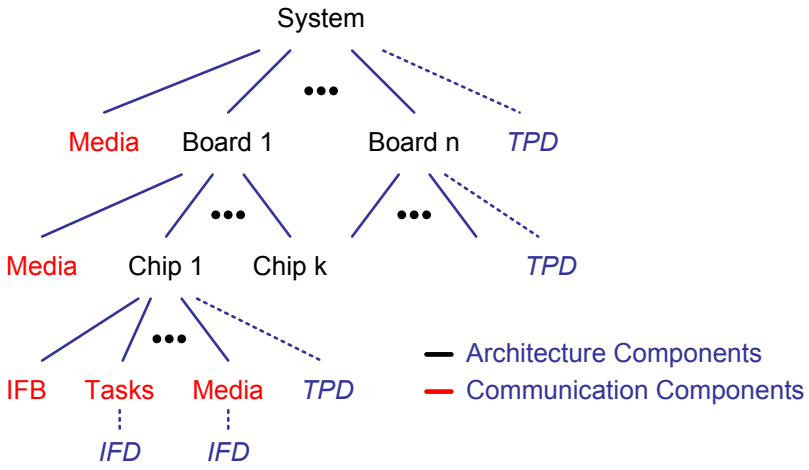


Figure 2.7: IFS System Architecture

To get an idea how a realistic system could look like, Figure 2.8 depicts an exemplary System Architecture composed of the Board `B1` and the two chips `FPGA1` and `FPGA2`. Each chip implements two tasks. Further on, there exists a medium on the system-level (`RS232`) and on the board-level (`USB`). `IFB1` adapts these two media and is placed on `FPGA1`. `IFB2` connects `Task3` and `Task4` while it is executed on `FPGA2`. In this way, `IFB1` represents the adaptation of two incompatible media, while `IFB2` is a typical representative for the IP-based SoC design. Both IFBs acquire their clocks from the related chip (`CLKs`). A common reset (`Reset`) is available from the board level. The edges between the interfaces (blue squares: $\texttt{I}_x$) represent physically existing interconnections between the component interfaces and are modeled by the so-called interface-map.

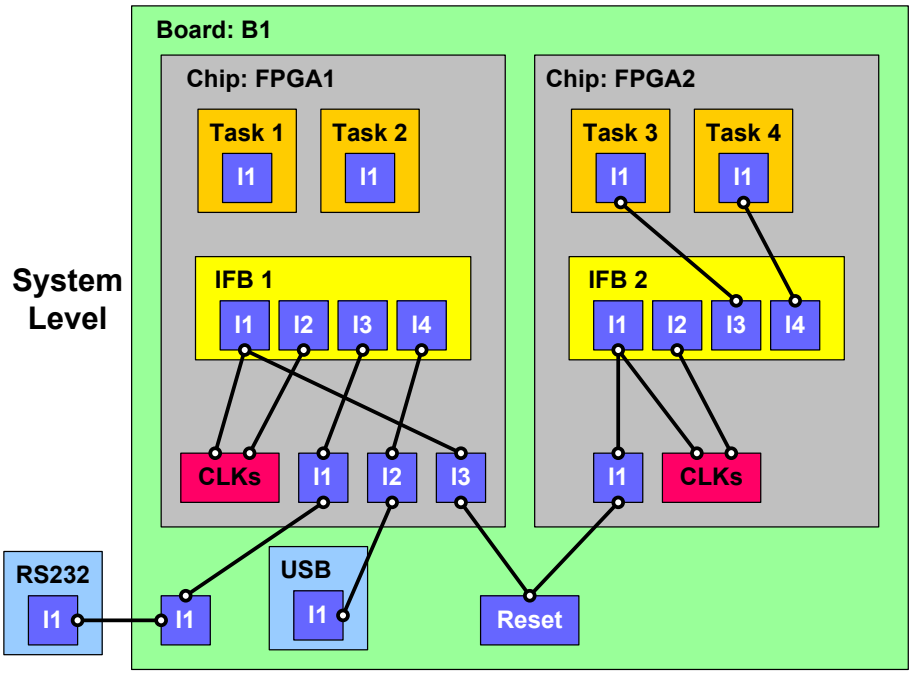


Figure 2.8: Exemplary System Architecture comprising two Interface Blocks

As depicted in Figure 2.7, each communication component possesses an IFD. To model the communication infrastructure within the System Architecture each architecture component provides an interface-map and a description of its private interfaces, which consists of the interface topology and its electrical properties. A protocol is not included here, since the behavior of an architecture component interface is only determined by the interconnected communication components. An interface-map is defined as follows:

Definition 2.11 *Interface-Map*

$$
\begin{array}{ll}
\textit{Interface-Map} := (\textit{Topology}_1,\ \textit{Topology}_2,\ \textit{SigMap}) & \\
t_1 \in \textit{Topology}_1 : & \text{Topology No.1} \\
t_2 \in \textit{Topology}_2 : & \text{Topology No.2} \\
\textit{SigMap} \ni \sigma = (\,\text{iosignal} \in \text{IOSignals} \in t_1,\ \text{iosignal} \in \text{IOSignals} \in t_2\,) & \textit{Signal mapping}
\end{array}
$$

Each entry $\sigma \in \textit{SigMap}$ of an *interface-map* links two IOSignals of the given topologies. In contrast to the protocol-map, we can connect one and the same IOSignal to multiple signals. A protocol-map is available in each architecture component. Thus, we may select the interfaces of the current architecture component, the interfaces of the contained communication components and, except for the system, the available parent interfaces to create an entry in the map. In this way, we can compose architectures with realistic interconnections.

2.2.2 Modeling the IFB Target-Platform

Chips are extraordinary architecture components, as only chips possess the *resources* to implement tasks. By contrast, media are self-contained communication components that can exist on the system, board and chip level. Next to the tasks, the Interface Blocks also have to be implemented on the chip level (see Figure 2.7). Even if there are only media to be adapted, the related IFB has to be placed inside a chip as presented in Figure 2.8. If there are multiple IFBs in the System Architecture, the process of creating and mapping IFBs to chips offers the chance to optimize the total communication costs.

Next to the IFDs of the adapted tasks, the information about the platform that executes the created IFB is an important input for our synthesis algorithm. The hardware version of the IFB, for example, is a *synchronous design*, which requires a clock and a reset signal that have to be provided by the related chip. We define our target platform description as:

Definition 2.12 *Target-Platform Description (TPD)*

$$
\begin{array}{ll}
AC = \{\,\text{System, Board, Chip}\,\} \times \text{TPD} : & \text{Architecture Component} \\
TPD = \begin{cases} \textit{Topology},\ \textit{Nature},\ \textit{InterfaceMap},\ \textit{Resources} & : AC \ni ac = \textit{Chip} \\ \textit{Topology},\ \textit{Nature},\ \textit{InterfaceMap} & : \textit{else} \end{cases} &
\end{array}
$$

The *resource* attribute is present only at the chip level. It comprises information about the available *implementation resources* and *clock networks* of the target platform. A *clock* is characterized by its *frequency* and the type of its *clock-network*, which specifies the technical distribution of a clock signal inside the hardware platform.

To perform the protocol adaptation for a given protocol frequency f_{prot}, the selected IFB clock frequency f_{IFB} that triggers the synchronous IFB has to satisfy: $f_{IFB} \geqslant CPB \cdot f_{prot}$. The Cycles-Per-Bit ($CPB$) value defines the maximum number of clock-cycles, which are required to process a particular bit inside the IFB. The CPB value depends on the IFB

implementation and the kind and number of processed protocols. In Section 6.3 we discuss the detailed IFB execution and provide a formalization for the real-time analysis after we introduced the IFB hardware circuit.

Furthermore, the target platform has to provide sufficient implementation resources to place the IFB implementation. Software execution platforms possess resources in the form of *memory*; while hardware resources are measured in *gate equivalents*, or in the case of FPGAs in *complex logic blocks* (CLB). The size of an IFB implementation depends, among others, on the low-level synthesis process, which allows us to derive the final size.

In this thesis, we focus on one target architecture, namely the Virtex-II Pro from Xilinx, which is an FPGA with up to four embedded PowerPCs. It consists of the typical FPGA building blocks like processing elements and programmable I/O as presented in Figure 2.9. Furthermore, we can find dedicated high speed I/Os and Block RAM (BRAM), which are also shared with the embedded processors. In this way, the Virtex-II Pro chip provides a very compact architecture that is well equipped to implement real-time capable embedded systems in the form of the presented System Architecture.

2.2.3 Hardware Execution Platform

An FPGA is an array of processing elements, called configurable logic blocks (CLBs) used to build combinatorial and synchronous logic designs. The CLBs are connected via an array of programmable interconnection elements. This routing structure consists of horizontal and vertical wires to interconnect the inputs and outputs of the CLBs as presented in Figure 2.10. Additional switch matrices (SM) allow the programming of the interconnections based on programmable multiplexers. In this way, the switch matrices can interconnect the vertical and horizontal lines, thus making a routing possible on the FPGA.

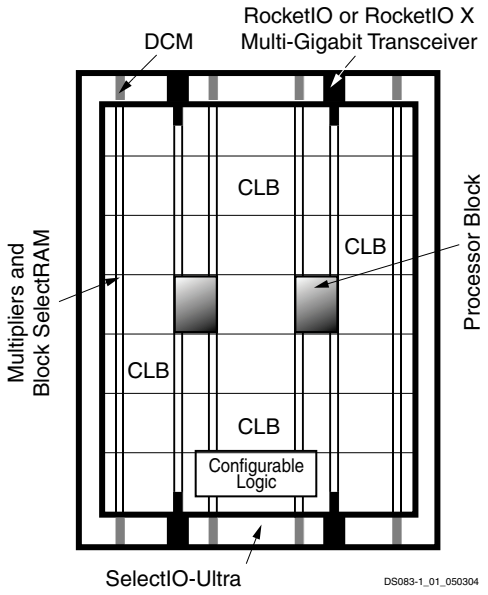


Figure 2.9: Virtex-II Pro – Generic Architecture Overview [152]

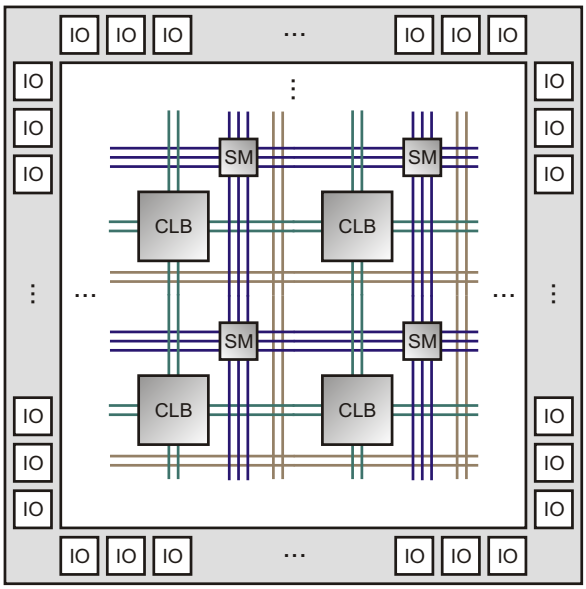


Figure 2.10: Classical FPGA Structure [124]

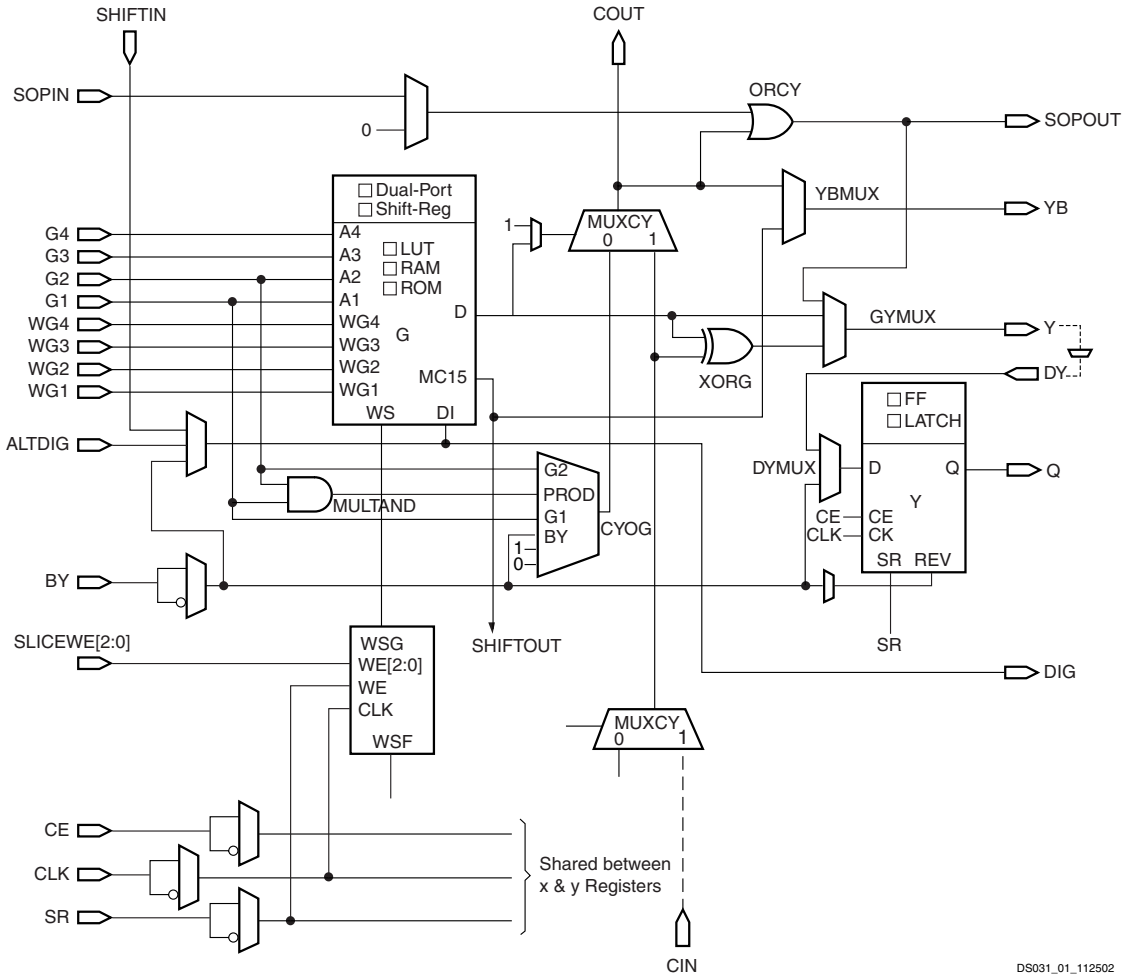


Figure 2.11: Virtex-II Pro Slice (Top Half) [152]

A CLB element comprises four similar slices, with fast local feedback within the CLB. The four slices are split in two columns of two slices with two independent carry logic chains and one common shift chain. Figure 2.11 depicts the upper half of a particular slice in detail, which is constructed from look up tables (LUT), flip-flops (FF) and multiplexers (MUX).

The LUTs represent the basic computing elements inside an FPGA. An n-input LUT is an n-address memory used to store the 2^n possible values of an n-input boolean function. Thus, it is possible to implement any function of n variables. Therefore, the functional characteristics are computed for each combination of the n variables and stored in the LUT. During runtime, the actual input variables address the LUT that delivers the stored function value as output.

Flip-flops store values temporally and are synthesized to registers. The values, remembered within this memory, can originate from LUT-outputs or an external source. The Multiplexers within the CLBs are required to connect LUT-outputs or CLB-input signals with the internal flip-flops or the CLB-output.

2.2.4 Software Execution Platform

Inside the Virtex-II Pro, there are up to four PowerPCs 405 (PPC405) cores embedded into the FPGA as software execution platforms. The PPC405 core is a 32-bit Harvard architecture processor. Figure 2.12 illustrates its functional blocks:

- Fetch & Decode unit
- Execution unit
- Memory Management unit
- Cache units
- Timers
- Debug logic unit

The processor provides a five stage pipeline consisting of a fetch, decode, execute, write-back, and load write-back stage. With this manageable architecture the PPC405 is a typical processor for embedded systems.

The two cache units can be disabled, which is demanded to perform real-time processing. To guarantee a predictable execution of software we would have to perform a worst case execution time analysis (WCET-Analysis), and in the case of multiple threads we require a schedulability analysis. Further on, we have to consider the communication with peripheral units; for example, the communication with synthesized hardware units within the FPGA.

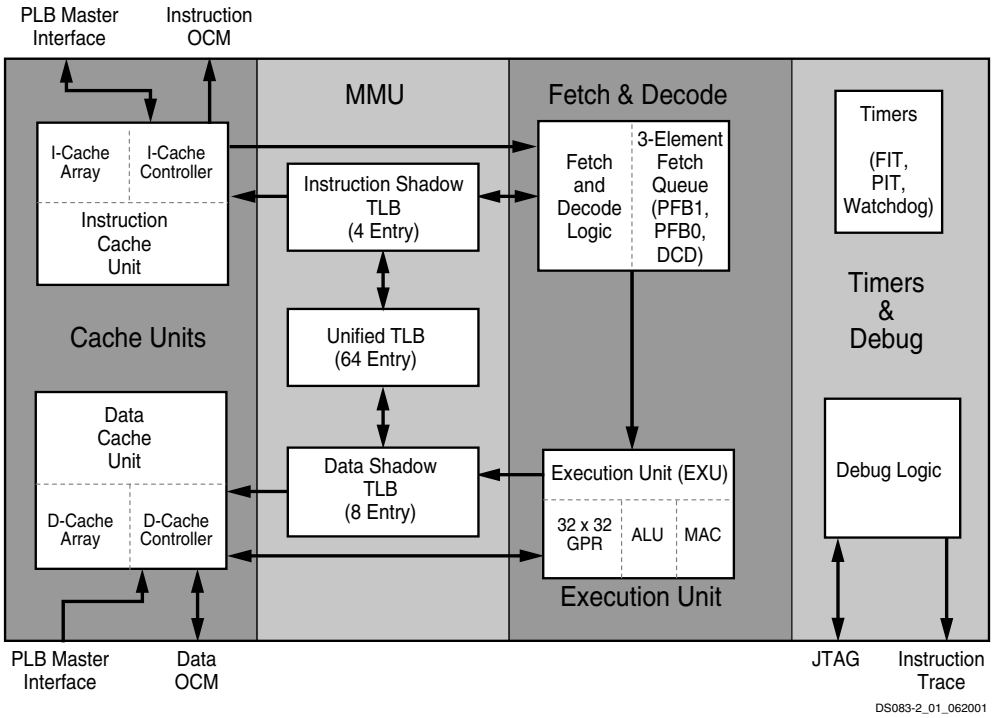


Figure 2.12: Virtex-II Pro Embedded PPC 405 Core Block Diagramm [152]

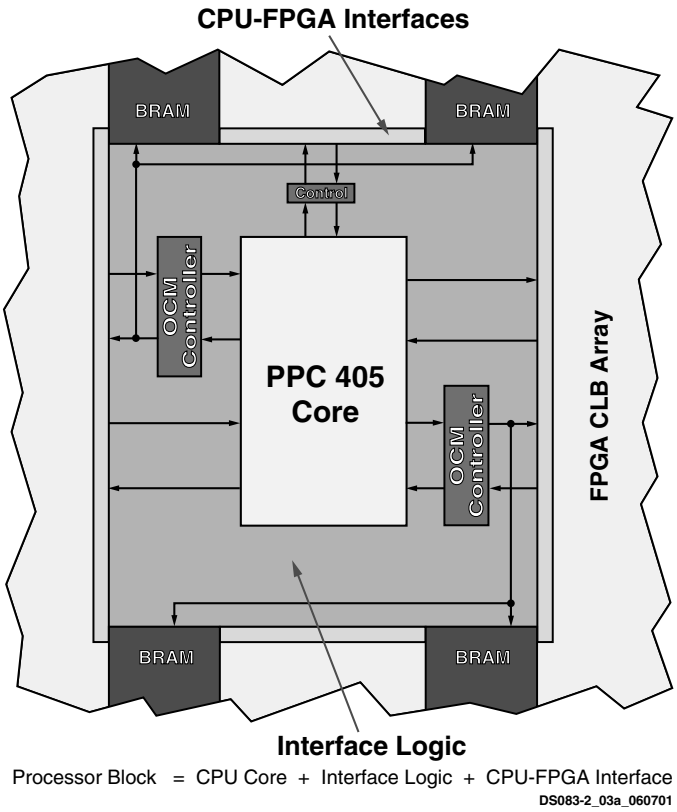


Figure 2.13: Virtex-II Pro Processor Block Architecture [152]

2.2.5 The Hardware/Software Interface

To interact with its environment, the PPC405 is surrounded by an interface logic, which enables the processor to access peripheral units, external I/O interfaces, and the user defined hardware inside the FPGA. All signals which cross the border of the interface logic have to be routed through the FPGA. Figure 2.13 illustrates the components of the interface logic:

- On-Chip Memory (OCM) controllers and interfaces
- Clock/control interface logic
- CPU-FPGA interfaces

The OCM controller allows the CPU and the FPGA to access the shared memory; here, in the form of BRAM. Clocks are distributed via the Clock/control interface logic. To connect the processor block with its environment, the Virtex-II Pro provides three different bus systems, which are closely coupled together (see Figure 2.14):

- Processor Local Bus (PLB)
- On-Chip Peripheral Bus (OPB)
- Device Control Register (DCR) bus

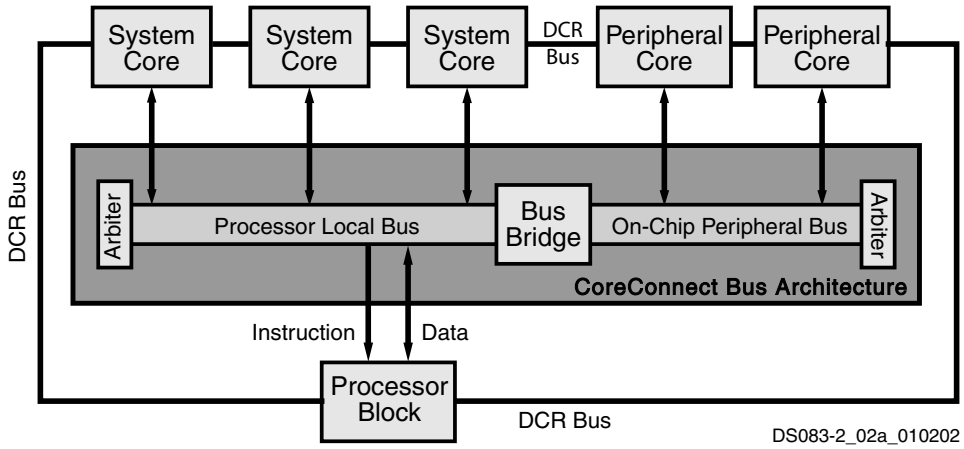


Figure 2.14: Virtex-II Pro CoreConnect Block Diagramm [152]

The processor local bus is a high-bandwidth, low-latency bus to connect high-performance peripherals. An on-chip peripheral bus, which is bridged to the PLB, connects slower peripheral cores, which reduces traffic on the PLB, resulting in greater overall system performance. PLB and OPB are elements of the standardized high-performance CoreConnect Bus Architecture. The device control register bus is designed to transfer data between the CPU's general purpose registers and the DCR slave logic's device control registers. The DCR bus removes configuration registers from the memory address map, reduces loading, and improves bandwidth of the processor local bus [150].

Memory Mapped I/O

Based on the available technology provided by the Virtex-II Pro, we can implement both versions of memory mapped I/O between hardware (FPGA) and software (PPC405). On one hand, the Virtex-II Pro offers Block RAM and on the other hand we can synthesize dedicated registers inside the FPGA. A drawback is that we always have to consider the previously presented interface logic.

Figure 2.15 presents an exemplary HW/SW interface deploying synthesized registers as shared memory. An application that is executed on the Power PC accesses these registers via the OPB bus. Therefore, the created OPB driver provides objects that can be written or read by the application. An OPB IFIP (OPB interface IP) connects the OPB with the synthesized data registers. The hardware circuit inside the FPGA can directly access the shared registers. The presented HW/SW interface has been studied in [31].

A basic requirement of the Interface Synthesis approach is the ability to reconfigure the target platform. The Virtex-II Pro allows the runtime reconfiguration of the hardware (FPGA) and the software (program, executed in the PPC405) side. With the possibility to implement tasks in hardware and software, including the option to establish a memory mapped I/O in the two presented variants, the Virtex-II Pro is the ideal platform to demonstrate our Interface Synthesis methodology.

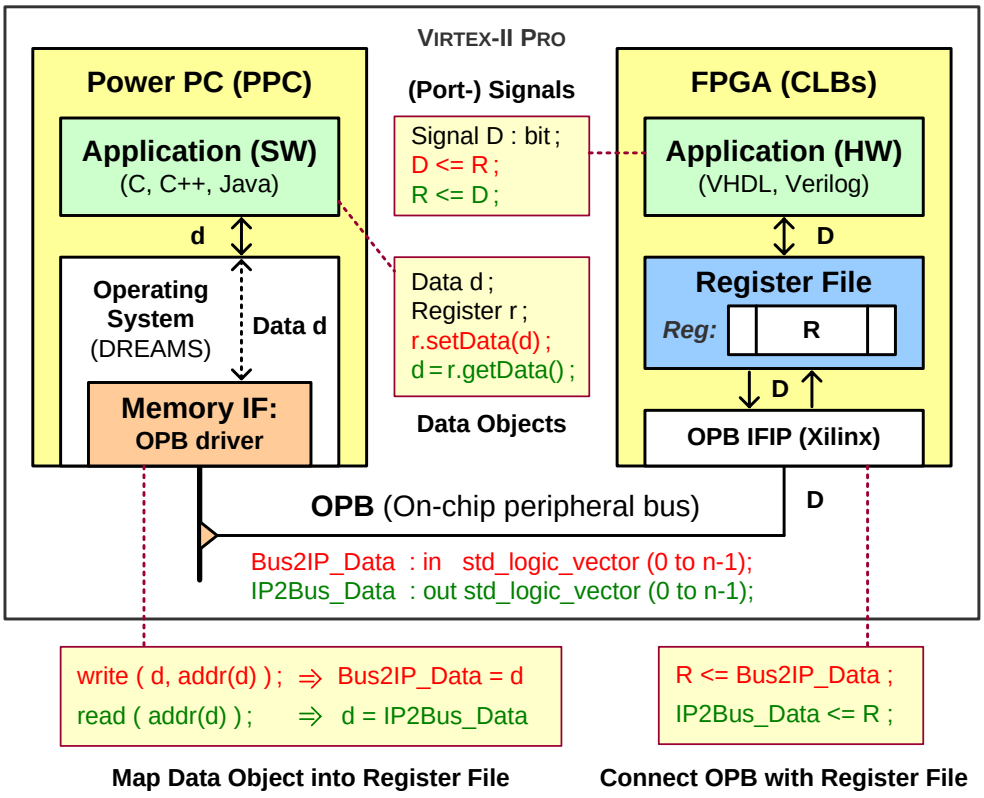


Figure 2.15: Virtex-II Pro Hardware/Software Interface

2.3 The Role of Reconfiguration

Reconfigurable systems may fill the gap between application-specific platforms based on custom hardware functions, and software programmable systems based on traditional microprocessors [79]. The resulting is one system, which can provide a higher performance by implementing dedicated functions in hardware, and still be flexible based on reprogrammable hardware like FPGAs and/or using a microprocessor (hybrid architectures).

The inherent flexibility of reconfigurable computing has the potential to sustain a new generation of electronic devices that are able to self-modify themselves according to user's needs. A cell phone, for instance, could be reconfigured to perform the functions of a PDA, MP3 player, digital camera, navigation system, and game pad, among others. Note that this is a scenario completely different from nowadays' multipurpose devices, which require specific circuitry to be integrated for each function they are supposed to perform. A reconfigurable cell phone would instead reprogram its hardware and software building blocks on-the-fly, just as the user selects a different function. If the device were to function as an MP3 player, reprogramming would give rise to a specific audio decoder. While performing a game pad, reprogramming would probably build a kind of graphics processing unit (GPU). In this example, reconfiguration is directly dictated by the distinct applications, which explicitly activate the reprogramming of building blocks as they are activated [242, 40].

Another reconfigurable computing scenario that has been consistently explored regards the implicit reconfiguration of hardware components without direct intervention by applications. This perspective of reconfigurable computing is based on the constant monitoring of hardware operational conditions, so as to initiate a pre-programmed reconfiguration whenever the associated conditions are observed. A good example of this kind of reconfiguration is a processor that is able to instantiate additional functional units as it detects overload situations. For instance, an application that runs into a heavy floating-point operations cycle would induce the processor to instantiate an additional floating-point unit (FPU) along with the structures needed to operate both units in parallel [54]. On a higher level of abstraction, we can employ this ability to reconfigure as a basic function of the operating system (OS). In doing so, we are able to improve the Quality of Service (QoS) for a dynamically changing set of applications [190, 189]. Furthermore, reconfigurable OS enable us to provide application specific hardware acceleration by migrating OS services into hardware [187, 85, 62, 140, 42].

In both scenarios, the main benefit associated with reconfigurable computing techniques is the possibility of reusing hardware and software components for multiple purposes, eliminating undesirable resource replications and allowing the system to go with requirements that were not initially taken into consideration. Eliminating replicated components directly improves metrics such as size and power consumption, while increasing reusability and flexibility directly affect non-recurring engineering costs. Nonetheless, in order to achieve these benefits, it is not enough that designers base their projects on FPGAs that support partial reconfiguration; the reconfiguration process has to take into account not only the exchanged applications but also the communication with the interacting components. This is especially critical in real-time systems, which have to guarantee a deterministic behavior at all times, including the reconfiguration process. The blackout time in which tasks are not available due to their reconfiguration is denoted as *communication gap*. It is a major challenge in our approach to avoid reconfiguration based communication gaps.

2.4 Summary

In this chapter, we have defined a framework to state out precisely the intention of our Interface Synthesis methodology. We introduced the terms *interface description* (IFD) and *target platform description* (TPD) as an essential input for the automated interface synthesis process. While an IFD specifies an interface by its *topology*, *electrical properties*, and *protocol*, the TPD delivers the resource information about the platform that executes the IFB.

Further on, we defined the three interface criteria *compatibility*, *connectivity*, and *compliance* that we applied to various scenarios to find out the field of application for an Interface Block and to address the challenges that our methodology copes with.

Afterwards, we have presented the *IFS System Architecture*. This model allows us to specify the *communication infrastructure* of complex communication scenarios, which are composed of hierarchical system components. Thereby, we distinguish the system components between passive architecture components and interacting communication components.

Finally, we discussed the role of reconfiguration in embedded real-time systems. Based on this, we introduced the term *communication gap* as the main challenge for our approach resulting from runtime reconfiguration.

CHAPTER 3

Background & Related Work

This chapter presents relevant *related work* and discusses the current *state-of-the-art*. For a better understanding, we deliver some additional background information. In conformance with Chapter 1 we subdivide the related work into the three domains:

- Interface-Aware (System-Level) Design Flows
- Interfaces in Reconfigurable Systems
- Dedicated Interface Synthesis Approaches

To *characterize* and *classify* these approaches we require some kind of metrics. This is especially important for contrasting our integrated design flow with existing ones. We can find an excellent introduction for system-level design in [93][1], including the aspects *Levels of Abstraction*, *Y-Chart*, and *Intellectual Properties*. In the next section, we deliver an updated excerpt of this work that reflects the most relevant aspects related to this thesis. In addition to the *Y-Chart*, we present the *P-Chart* which is an extension of the Y-Chart and considers multiple aspects of the design process. This introduction into aspects of the system-level design allows us to discuss the *advantages* and *disadvantages* of particular approaches and provides the basis for the *comparison* to our Interface Synthesis approach.

3.1 System-Level Design

System-level design (SLD) copes with the increasing complexity of embedded systems by raising the level of abstraction. In contrast to behavioral synthesis, which deals with the implementation of algorithms in application-specific hardware (ASIC design), system-level design focuses on the problem of mapping an abstract specification model of an entire system onto a target architecture (e. g. SoC design). In many cases, the design of large embedded systems considers mechanical as well as electrical aspects. In this thesis, we focus on the electronic system design (ESD) which refers to the electronic part of the SLD process.

[1] Ph.D. thesis written by Rainer Dömer, now Assistant Professor at the University of California, Irvine

Compared to the high cost of developing dedicated hardware, a software implementation is inexpensive. In addition, software can easily be modified if requirements change or new features need to be added. However, a software implementation may not be possible due to performance constraints. A task of the system-level design is to trade-off an inexpensive and flexible software solution versus a high-speed hardware implementation. Therefore, a major challenge of system-level design is the HW/SW codesign. Codesign is defined as the design of systems involving both hardware and software. The main task of codesign is the partitioning of a single system specification into hardware and software parts. Then, depending on whether a specific component is to be implemented in software or hardware, standard software technologies and established hardware design methods, respectively, are used for the final implementation of the component.

The system design flow usually starts from a formal, abstract specification of the intended design. After the specification has been validated for functional correctness, it is refined by a sequence of refinement steps, which eventually map the initial specification onto a selected target architecture. The interface synthesis process, which is presented in this thesis, can be seen as a specific refinement step.

A very important issue in the system-level design is the reuse of predesigned, complex components, often referred to as Intellectual Property (IP). In fact, the reuse of IP is the main key to cope with the complexity involved with SoC design. In contrast to redesigning a system completely from scratch, the use and integration of complex components, which are predesigned (possibly by somebody else) and well tested, drastically reduces the design complexity. Thus, reuse of IP saves a great amount of design and testing time and, hence, allows a shorter time-to-market.

While the idea of IP reuse promises great benefits for system design, there are also problems to be solved. In order to allow easy and seamless integration in a new system, IP components need to be portable to different technologies and must provide standard or flexible interfaces. Good documentation about the IPs functionality, its requirements with respect to the environment, and its performance and other metrics, are required as well.

The reuse of IP has to be an integral part of the system design methodology. The selection, easy insertion, and replacement of IP components ("plug-and-play") in the system must be directly supported by the design models, the tools and the languages being used throughout the design process. These and other issues involved with the reuse of IP are addressed in more detail in Section 3.1.3.

3.1.1 Levels of Abstraction

In computer science, a well known solution for dealing with complexity is to exploit hierarchy and to move to higher levels of abstraction. This effectively reduces the complexity in terms of the number of objects to be handled at one time.

Figure 3.1 illustrates this for digital systems. An embedded system, which at the lowest level consists of 10ths of millions of transistors, typically reduces to only thousands of components at the register-transfer level (RTL). Furthermore, RTL components are grouped together at the algorithm level. Finally, at the highest, the so-called system level, the one system is composed of only few components that include microprocessors, special-purpose hardware, memories, and buses.

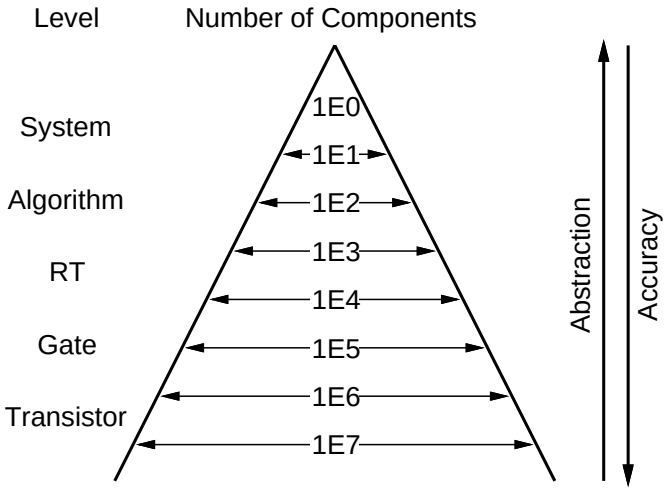


Figure 3.1: Abstraction versus complexity

From Figure 3.1, it is obvious that a complex embedded system is easier to deal with at the abstract system level than at the detailed gate or transistor level.

The level of abstraction is a trade-off with the level of accuracy. A high abstraction level implies low accuracy, and vice versa. The design process of a new system usually starts from a highly abstract specification model and ends with a highly accurate implementation model which reflects the real system with all its details.

The advantage of such a top-down approach is that all necessary design decisions can be made at an abstraction level where all irrelevant details are left out of the model. This allows the design tasks to work with a system model in minimum complexity.

The concepts of abstraction and hierarchy are closely related. In digital systems, hierarchy is inherent in the structure of a system. Every system is composed of a set of components, and each component is a (sub-) system that, again, is composed of (sub-) components. In other words, the terms system and component are recursively defined.

In order to break the recursion in this definition and to clearly identify the system and its components, it is necessary to name the current abstraction level. The abstraction level defines the type of the components used and, thus, also determines the system. For example, at the gate level, the components are logic gates and the system is the composition of such gates. One level below, at the transistor level, a single gate can represent an entire system that is composed of a set of transistors.

It should be pointed out that the term system, in general, refers to different things in different contexts. For example, a modern aircraft can be viewed as one single system or as a collection of thousands of systems. Within this work, unless stated otherwise, the term system refers to a digital, embedded system, which can be implemented by use of application-specific hardware and software running on one or multiple execution platforms.

Please note that this definition of a system is consistent with the term System-on-Chip. It is also well defined with respect to the abstraction level for the SoC design, the system level. A precise definition of system-level design will be given in the following section by use of the Y-Chart and the P-Chart, which is a derived version of the Y-Chart.

3.1.2 Y-Chart and P-Chart

The Y-Chart [206, 117], shown in Figure 3.2, is a conceptual framework which coordinates abstraction levels in different domains. This can be used to compare and classify different design tools and design methodologies. The Y-Chart distinguishes three domains represented by three axes.

A typical design process starts from the behavioral domain, which specifies the pure behavior of the system without any implementation details, for example in form of program functions or mathematical equations. The design is then mapped onto an architecture in the structural domain. The structural architecture is composed of components, for example logic gates or RT components, depending on the level of abstraction. Finally, an implementation of the design is manufactured in the physical domain.

The level of abstraction, as introduced in Section 3.1.1, is orthogonal to the domains. Starting from the center of the chart, the abstraction level, indicated by the dashed, concentric circles, increases from the transistor level up to the system level.

The Y-Chart allows us to illustrate design flows and design tasks as paths on the chart. For example, a complete system design flow starts on the behavioral axis at the system level. After step-wise refinement towards the center of the chart and mapping onto a structural and physical implementation, it finally ends on the physical axis at the transistor level.

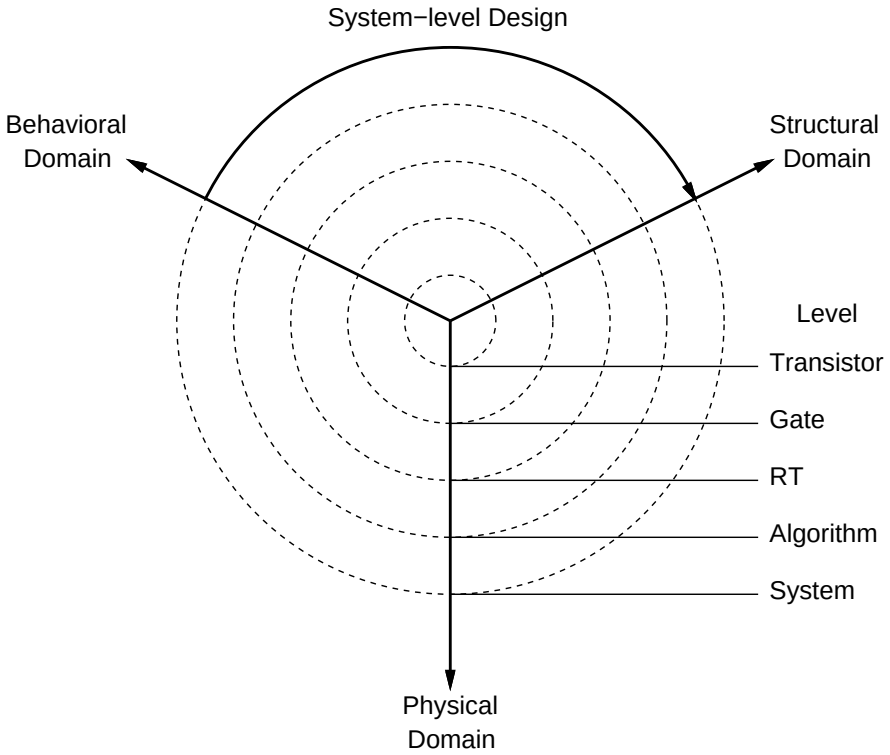


Figure 3.2: System-level design in the Y-Chart

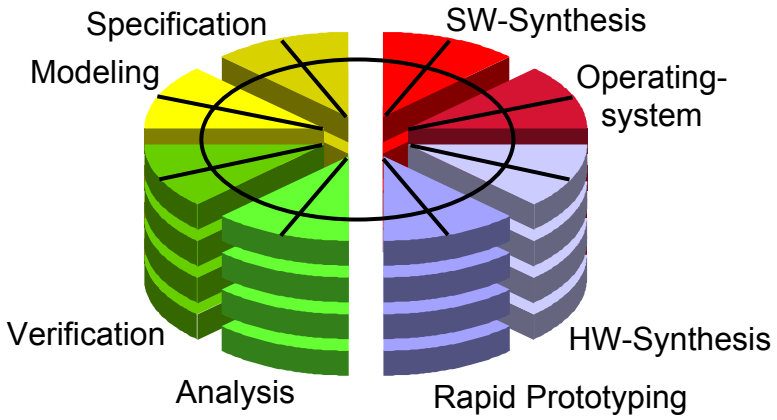


Figure 3.3: System-level design in the P-Chart

On the Y-Chart, synthesis is represented by an arc from the behavioral to the structural axis. The definition of system-level design is indicated by the arrow in Figure 1.2. The task of system-level design is to synthesize a structural system architecture from a behavioral system specification. As another example, high-level synthesis (HLS) is represented by an arc from the behavioral to the structural axis on the RT level.

Furthermore, the tasks of refinement and optimization can be demonstrated on the Y-Chart as well. Refinement is represented by an arrow on the behavioral axis from a high to a lower abstraction level. On the other hand, optimization can be represented as an arrow at any point in the chart, which points back to its starting point. Thus, such optimization is a task that is performed in-place and can occur at any level in any domain.

The P-Chart [136, 138, 139, 203] as depicted in Figure 3.3 is an extension of the Y-Chart, that explicitly models eight *orthogonal design dimensions*, namely SW-Synthesis, HW-Synthesis, Operating System, Rapid Prototyping, Specification, Modeling, Analysis, and Verification. Furthermore, the P-Chart model distinguishes between *level of abstraction* and the *design view*. The stack of the layered pie-elements models the different abstraction levels while the separation within each element describes the design view.

3.1.3 Intellectual Property and IP-Based Design

As stated earlier, the reuse of Intellectual Properties (IPs) is a key issue in SoC design. This section elaborates on IP components and the benefits, problems, and requirements with IP reuse and IP-based design.

IP Components

At the system level, predesigned components are frequently called IPs. IP components are independent processing elements, in other words, they have their own flow of control and interact with the other system components via their IP interfaces. Unlike full-custom components, which are synthesized from scratch specifically for one application, IP components are selected from an IP library and are fixed or allow only limited customization.

Nowadays, IP components include memories, processors, and industry standard circuits. Memory IPs, like RAM and ROM blocks, can usually be customized in size, whereas processor IPs typically come as fixed cores. Processor IPs include embedded micro controllers, general-purpose processors, and digital signal processors. Special-purpose IPs implement industry standards, for example, encoding and decoding algorithms like MPEG, JPEG, etc., or communication devices like PCI or VME bus interfaces.

IP components can be categorized into hard, firm, and soft IPs. Hard IP components are developed by use of a standard design process and are fully implemented in a specific technology. In particular, for hard IPs, there is a physical representation of the layout, for example, in the form of a GDS-II file. Since hard IPs are fully implemented, their performance characteristics and other metrics are very accurate and predictable. However, hard IPs are inflexible and limited to a specific target technology. Soft IP components, in contrast, are very flexible IPs that come typically in form of synthesizable RTL code. Usually, soft IPs can be parameterized or are user-configurable in terms of data size, features, etc. Since soft IPs are synthesizable, they can be implemented in any target technology as well. However, the implementation metrics of soft IPs are not as predictable as for hard IPs, because the final implementation has yet to be synthesized. Firm IPs exist between hard and soft IPs. We can find firm IPs, for example, in the form of EDIF-netlists. On one hand, this representation allows us to integrate the IP into the design flow on a higher level of abstraction than hard IPs. On the other hand, this format protects the IP still quite well, as modifications on netlists (gate level) are complicated.

IP components can also be classified into internal and external IPs. Since the process of developing the system is decoupled from the development of the IP components, these tasks can be performed independently by separate design teams in possibly different companies. Internal IPs are developed inside the same company which builds the system. Typical internal IPs include legacy designs which can be reused from former products that have been proven to be successful. External IP components are developed and provided by IP providers outside the company building the system. While the system house, also-called IP integrator, can focus on the problem of the system specification, integration, and implementation, IP vendors develop and offer the required IP components. With this approach, the system house benefits from a large library of optimized, well-tested, and well-documented components, which are available when needed. IP providers, on the other hand, take advantage of their expertise in specialized design areas without the need to build and sell complete systems.

IP Reuse

The reuse of predesigned components is well known in the EDA. For example, at the RT level, reuse includes the instantiation of components from the RTL library, such as registers, multipliers, arithmetic-logic units (ALU), etc. Reusing components drastically reduces the time and the cost of the design because the reused components are already designed, optimized, and tested. However, in order to exploit these benefits, several problems have to be overcome.

The main two problems involved with design reuse are component matching and component integration. First, the task of matching is to find a corresponding counterpart in the component library for a part of the design specification. A component can only be used in the implementation if it matches the functionality and meets the constraints in the specification. Then, the task of component selection is to choose one component from the set of matching

components that best meets the design goals. Typical design goals are minimal cost or best performance. Finally, when a suitable component is chosen, it must be integrated with the rest of the design. The task of integration is to ensure that the component is properly connected and controlled so that it cooperates with the other system components and works with the right data at the right time.

One way to select applicable IPs is to provide the system design and the IP with meta-data that can be evaluated automatically. Such meta-data offers information about the structure and the functionality of IPs. In the last years XML based description languages have emerged that allow us to employ web services, for example the web based retrieval of IPs. One format that has been especially developed for this purpose is the IPQ-Format.

The integration of IPs includes similar problems. Instead of communicating through plain ports, IP components usually interact via non-trivial interfaces by use of possibly complex communication protocols. Hence, IP integration typically requires interface synthesis and protocol translation to be performed. The diversity of external or even internal IPs is one of the main challenges for the automated interface synthesis that is addicted to the integration of incompatible IPs. The automated adaptation of interfaces always requires a minimum amount of information, modeled in a formal and computer readable format.

While the matching, selection, and integration of IP components are tasks performed by system integrators, IP providers have to deal with the task of IP protection, which is discussed in the following section.

IP Protection

Since business of IP vendors depends on selling their intellectual property to other companies, IP providers have to protect their IP from being copied, modified, or reverse engineered. IP protection addresses the security issues for external IPs. In general, IP components are covered by a copyright and can be further protected by legal contracts and non-disclosure or non-distribution agreements. However, it is usually very difficult to detect and to prove that an IP is used without permission. Therefore, technical measures are taken in addition to legal guarantees.

For hard IPs, protection can be easily achieved by keeping the final implementation with the IP provider. Instead of the real implementation, the system integrator is supplied with simulation models and estimation data of the IP. With these models, the system can be developed without the need for the real IP. Typically, the deliverables for a hard IP include simulation and timing models at different levels of abstraction, performance, power, and other metrics, a floor plan model, and comprehensive documentation about the functionality and interface specification of the IP.

For soft and firm IPs, a different approach is necessary. Since the final implementation will be synthesized by the system integrator, the complete, synthesizable model must be made available. In order to still hide the implementation or algorithm details, the IP can be provided in a pre-compiled format without a source code. This is basically the same, well-known idea used in the software business to protect proprietary code from being reverse engineered. In general we can say that the more a particular IP is protected, the more complicated becomes its integration, as the form of documentation becomes more abstract and cannot be evaluated automatically by synthesis tools.

3.2 Interface-Aware (System-Level) Design Flows

In this section, we discuss relevant (system-level) design flows that explicitly support the modeling of communication infrastructures and the description of interfaces and/or cope with IP-Based design. Thereby, the interface representation is of special interest for us, as it constitutes the basis for each interface synthesis flow. For this reason, we examine state-of-the-art interface and IP descriptions, which are standardized and automatically evaluable before we proceed with the design flows.

3.2.1 Interface and IP Descriptions

To perform an automated interface synthesis we require a precise and formal specification of the interfaces that we are going to adapt. In Chapter 2, we defined our way to model hardware and software interfaces. Other formal languages with respect to interfaces are:

HW, SW & HW/SW languages

- VHDL [174, 171], Verilog [223]
- C, C++, Handle-C [52]
- Esterel [57], LOTOS [250, 162], Estelle [249]
- ASN.1 [105, 80]

System-Level languages

- System Verilog [228, 151]
- SystemC [53, 78], SpecC [98, 72]
- SDL [83, 88], SLDL [213]
- UML 2.0 [192, 100]

ASN.1 [80]

All these languages are able to model interfaces, but they were not designed to describe IPs. The Abstract Syntax Notation One (ASN.1), for example, is an industry standard in telecommunications and computer networking. It was first standardized in 1984 by the CCITT (International Telegraph and Telephone Consultative Committee, now called ITU-T, International Telecommunication Union - Telecommunication Standardization Sector) under the name "X.409 Recommendation". A little later, ISO (International Organization for Standardization) chose to adopt this notation and split this recommendation into two separate documents: the abstract syntax (ASN.1) and the encoding rules (BER).

ASN.1 is a flexible notation that describes data structures for representing, encoding, transmitting, and decoding data. It provides a set of formal rules for describing the structure of objects that are independent of machine-specific encoding techniques and is a precise, formal notation that removes ambiguities [105]. ASN.1 was designed to specify data protocols in an open system interconnection (OSI) environment. Standardized *XML Encoding Rules* (XER) allow ASN.1 specifications to be used as XML schema against which XML documents can be validated. *Fast Web Services* specify ASN.1 specific SOAP messages, which allow ASN.1 to interact with distributed web services.

In the case of IPs the interface information is usually delivered as *meta-data* encapsulated in the IP description. Thereby, the interface description has to be independent of the particular IP type and may not contravene the IP protection rules. Text documents, which recently exist in different styles, e.g. Word documents, are useless for this task, since they cannot be evaluated automatically. Afterwards, we present four recent standards, which are well established and developed further with the support of the EDA industry.

VSI Alliance, VCT 2 & SLIF [47]

The VSI Alliance was formed in 1996 by a group of major EDA and Semiconductor companies. Currently the alliance incorporates about 200 members including companies, individuals, and organizations. The primary vision was to accelerate SoC development dramatically by specifying open standards that facilitate the mix and match of virtual components from multiple sources [45]. VSIA expanded that vision to meet the growing needs of the IC Industry by including software and hardware IP for System-on-a-Chip (SoC) Design [46].

Primary Goals: IP Based Design
- Minimize Design Time
- Maximize IP Reuse
- Optimize System Level

Advanced Goals: System Based Design
- Minimize System Design
- Maximize Software reuse
- Optimize HW/SW tradeoff

One condition to achieve these goals is an improved interaction of the IP-creator and the -integrator. Thereby, the IP creator is reliable for the IP authoring that means to create predictable and pre-verified *Virtual Components*[2] (VC), which are ready for the System-Chip integration. The main improvement of the design efficiency results from designing with blocks of 100,000 transistors. In order not to loose this efficiency gain, IP integrators are not allowed to modify VCs. VSIA offers several specifications, standards, and other technical documents for different areas of handling VCs/IPs. One of these documents specifies a system level design model taxonomy that defines general modeling concepts for system models, architecture models as well as hardware and software models [46].

The *Virtual Component Attributes (VCA) With Formats for Profiling, Selection, and Transfer Standard 2* (VCT 2) [131] defines the IP meta-data including IP attributes, the data types and structure of IP Properties, and a functional taxonomy. The *Virtual Component Transfer Specification* (VCT) [132] handles the transfer of the IP contend.

In conformance to VCT 2 the *System-Level Interface Behavioral Documentation Standard* (SLD 1 1.0) [130] defines data types, attributes, and transactions on ports. This description of VC System-Level Interfaces (SLIF) can be applied to compose systems of VCs but it cannot be employed for the automated synthesis of interface adapters.

SPIRIT [81]

SPIRIT is an emerging standard, which is still under development [181, 182]. EDA companies like Arm, Cadence, Mentor Graphics, Philips, ST and Synopsys are driving forces of this IP description. SPIRIT intends to create two standards:

IP meta-data description: The meta-data standard will create a common way to describe IPs, compatible with automated integration techniques and enabling integrators to use IPs from multiple sources with SPIRIT-compliant tools.

IP tool integration API: The tool integration API is going to provide a standard method for linking tools into an IP framework, enabling a more flexible, optimized development environment. SPIRIT-compliant tools will be able to interpret, configure, integrate, and manipulate IP blocks that comply with the proposed IP meta-data description.

[2] In VSIA IPs are also-called Virtual Components

SPIRIT is fully architecture-agnostic, both for interconnect - and virtual components. It can describe SoC designs and is a common standard for IP description that tools from multiple vendors can interpret. A SPIRIT description can be used by IP catalogs to display and exchange IP integration requirements. Thereby, it is fully design-language neutral.

Compared to VSIA's VCT standard, SPIRIT is a design deliverable that relates design intent to design implementation, while VCT is a mechanism for bundling and exchanging packages of design deliverables.

In spite of these features, SPIRIT does not expose proprietary data about an IP, as it encapsulates only such data that many IP suppliers provide openly in technical reference manuals today. SPIRIT 1.0 does not describe the internal architecture of an IP. The extensibility mechanism in SPIRIT provides a consistent way for users to add their requirements to the basic SPIRIT 1.0 descriptions.

CWL

To improve the reusability of design resources in the large scale integration (LSI) system design process, an IP interface description language called Component Wrapper Language (CWL) has been jointly developed in cooperation with Hitachi, Fujitsu Laboratories, and Fujitsu in 2002 [114].

One feature of this language is the capability to write formal and compact descriptions of signal changes at input/output ports for IP. This enables the following three primary goals of the Component Wrapper Language:

Removal of ambiguity: Correct conveyance of interface specifications

Facilitation of design: Automation of connection between IPs

Facilitation of verification: Support of IP design verification and connection of IPs

The advanced goals of the CWL are to increase the IP reusability and accelerate the system LSI design process by distributing interface specification descriptions in the CWL and the IPs in individual packages.

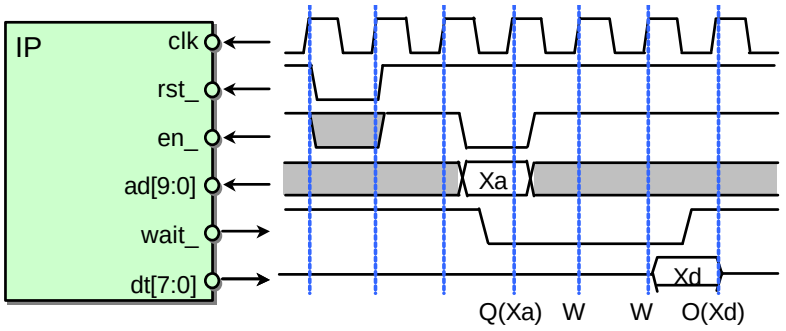


Figure 3.4: Interface representation as Timing Diagram

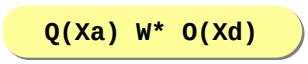


Figure 3.5: IP representation with CWL [114]

The Component Wrapper Language is a language used to correctly define the interface specifications of the target IP. Such interface specifications include specifications of logical signal changes as well as structural specifications, such as I/O pin information.

The key to this language is its method of describing these specifications, such as signal-waveform change specifications as presented in Figure 3.4. The waveform representation method in the Component Wrapper Language is based on regular expressions (see Figure 3.5), which are suitable for compactly representing individual sets of value sequences, such as waveform specifications. To be precise, the waveform pattern for "one clock cycle" is used instead of "one character" [115].

Timing diagrams (also-called signal waveform diagrams) are used to represent waveforms in specifications. As mentioned in Chapter 2, one drawback of this method is that only one sample representation of waveform specifications can be shown in a diagram. It is certainly helpful for understanding the specifications, but many timing diagrams have to be described to cover all cases.

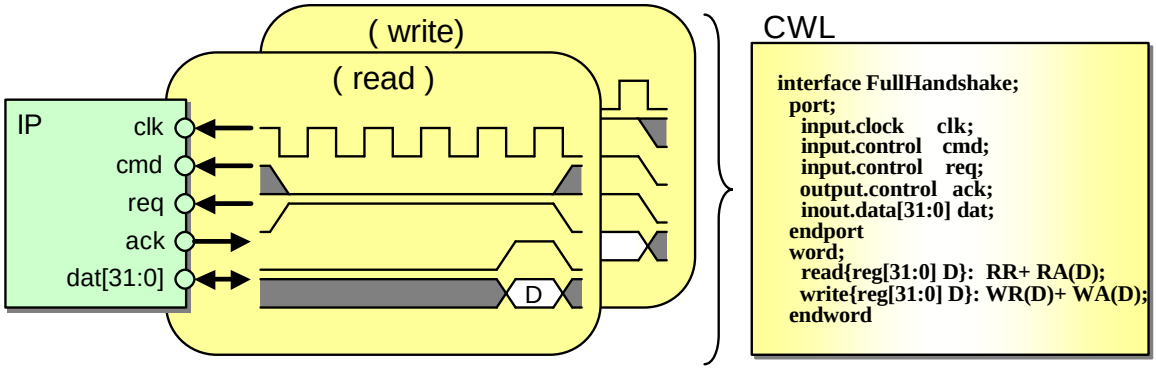


Figure 3.6: CWL example [114]

Writing descriptions of all cases is difficult, and doing so would needlessly increase the quantity of specifications. Moreover, computers cannot directly process timing diagrams written on paper, and manual work is thus necessary to use verification tools. This leads to a possibility of human error in addition to extra labor-hours of design work. Therefore, CWL descriptions allow the definition of ports, protocols, and transactions [116]. The example depicted in Figure 3.6 provides the following four parts:

Port declaration: This part defines the structural I/O information of the target IP.

Signal value naming: This part names the waveform pattern for one cycle. By labeling and using this waveform pattern, it becomes the minimum unit of representation.

Pattern definition for each function: This part defines the pattern of each transaction. The definition of pattern can also be represented hierarchically. The primary transaction operations are "read" and "write".

Overall definition: This part describes the operation of the complete target IP.

IPQ

The IPQ Format is intended to express IP Content and IP Characterization data and offers a unified IP description format. It is designed to support IP-based design services, including web-services. Consequently, the IPQ Format has to be powerful enough to comprehend all data that has to be exchanged between the IP integrator and different IP related services. This data spans mainly two areas: the comprehensive description of IPs and the input data for and output data from the utilized IP services. According to this division, the IPQ Format is constructed as defined in Figure 3.7. The whole IPQ Format has been implemented using XML Schema [20, 19, 22, 17].

The *IP Data Format* is responsible for the specification of the IP. On one hand, it must be capable of describing IPs precisely, since most IP services need certain parts of the IP Content. On the other hand, some services might only need meta-data about specific IP properties such as its operating frequency or whether it meets certain design guidelines. Accordingly, IPQ distinguishes between the content and the characterization of an IP.

The *IP Content* covers all deliverables of the design itself (HDL-files, layouts, test-scripts, manuals, data sheets, detailed validation documentation, etc.). For the development of the IP Content section, the Philips Semiconductors CoReUse standards [226] played an important role. The content data items are inserted into a skeleton of XML tags. Next to this structure, the XML skeleton conveys some semantics to the whole IP Content portion.

The *IP Characterization* describes the IP in a way that allows us to assess its characteristics, quality, suitability, and reusability. It consists of a set of IP attributes (power consumption, throughput, gate count, area, etc.). The attributes and their hierarchy are compatible with the VCT 2 standard. Aside from the VSIA attributes, the IP Characterization contains so-called quality criteria. These are based on a set of quality metrics from the Open-MORE Assessment Program [160, 159, 158, 128] which basically indicates whether an IP meets certain rules and guidelines like verification and validation strategies, testability, etc.

In contrast to other formats used, e. g., by D & R [90, 135] and VCX [112, 122], the IP Data Format integrates the IP Characterization and the IP Content portion. An IPQ toolbox [18, 23] has been implemented that exploits this close integration to address many different

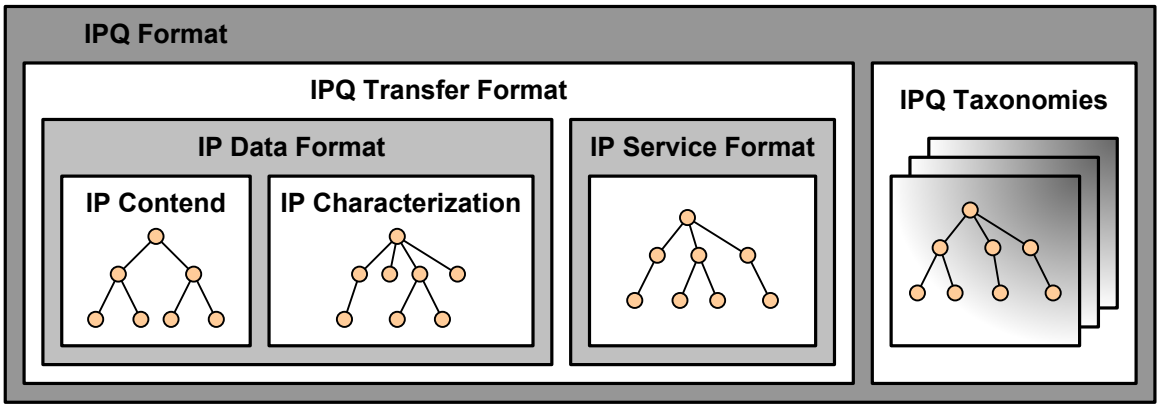


Figure 3.7: Architecture of the IPQ-Format [239]

IP related web services [24], including retrieval services [21, 16] as well as processing services. As the IPQ Format allows the modeling of dependencies between different IP attributes, it can also accommodate parametrizable IPs.

However, IP specification data is not sufficient to invoke external tools. Typically, tools need additional information like environment settings, control files, and other configuration data. Furthermore, the data they return is very diverse and cannot be expressed with a limited set of XML tags like the one the IP Data Format provides. Therefore, the necessary additional information is captured in the *IP Service Format*. This format consists of two portions, the service invocation and the service result portion.

The *IPQ Taxonomy* is a tree-like classification scheme. It turned out that taxonomies are very helpful in the IP context, because they allow the classification of IPs in various scenarios. In an IP taxonomy tree, every IP is assigned to a node (i. e., category). Depending on the type of taxonomy, an IP´s assignment to a particular taxonomy node describes the IP´s function, architecture, or application domain, etc. A functional class taxonomy like the one defined in [169], e. g., identifies IP functions.

Additional information about the IPQ Format can be found in [134, 89, 164, 204, 215].

IFD-Format

Our interface description (IFD) model that we presented in Chapter 2 has been developed as part of the IPQ Format [20, 22]. In this context, it covers the specification and modeling of IP interfaces. The XML representation of an IFD, namely the *IFD-Format* [29], defines an XML sub-scheme of the IP characterization.

The IFD-Format has been developed based on VSIA's VCT 2 and is compliant with this important industry standard. To perform an automated interface synthesis including IPs, it was necessary to enhance the available information. Although the specification of the interface structure is mainly covered by the VCT 2, it has been completely reorganized, as this information is distributed through several topics of the VCT 2. By contrast, the IFD-Format presents an *interface-centric* representation.

The behavioral interface description has been developed independently of any VSIA standard, since this aspect is not covered appropriately. Similar to the CWL, timing diagrams have been selected as an origin to model communication protocols. While the CWL uses

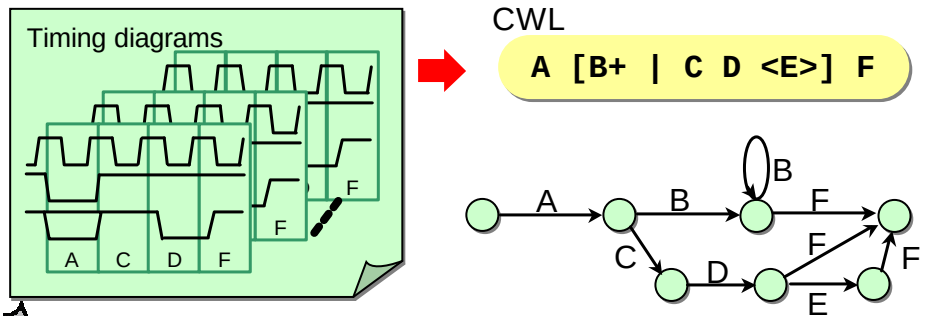


Figure 3.8: CWL regular expression visualized as state machine [114]

regular expressions to model the waveforms, the IFD-Format is based on state machines. As regular expressions are equivalent to FSM [146, 188], each CWL expression can be formulated and visualized as state machine as depicted in Figure 3.8. Nevertheless, the CWL FSM and the state machines used in the IFD are different, but they can be converted into one another. We discuss this aspect in Section 3.4.

Due to the fact that the IFD is part of the IPQ Format, IPQ IPs can be directly handled by the interface synthesis approach presented in this thesis. Therefore, the Interface Synthesis methodology proposes the IPQ Format as the recommended IP description format.

3.2.2 Design Flows

The (system-level) design flows that we discuss afterwards explicitly consider the interfacing of components. Although some of the given examples are scientific approaches, all of them are well considered by the EDA industry.

UML 2.0 Based Design Flows

UML provides the basis for several design flows. Tools like the Enterprise Architect[3], Together[4], and Rational Rose[5] already support the UML 2.0 standard or are going towards it. An advantage of UML 2.0 is its extensibility, thus designers can define UML profiles consisting of stereotypes and tagged values [192]. Several approaches for real-time extensions to the UML 2.0 have been proposed to support hard real-time [123, 216, 100, 99, 125].

ARTiSAN: ARTiSAN Software is a provider of UML tools used for the modeling of real-time embedded systems, and of software engineering for designing mission-critical and safety-critical applications [233]. Next to UML, ARTiSAN offers an implementation of SysML [142, 141]. In their system design process, ARTiSAN breaks a system up into different areas of functionality [235]. One of these areas is the "physical view". By applying UML stereotypes to elements on an assembly diagram, the system architecture can be described. The busses and interconnects, software and hardware interfaces, and the other integrated systems can all be shown in this type of diagram. A second way to model interfaces is by modeling sequences of interactions in Interaction Diagrams. To also capture IP specification and IP integration ARTiSAN presented a combination of UML and Autosar [234].

Rhapsody: I-Logix offers Collaborative Model-Driven Development (MDD) solutions for systems design through software development focused on real-time embedded applications [148]. Each design is iteratively analyzed, validated, and tested throughout the development process. In the presented MDD approach IP reuse is realized by inlining available legacy code [149]. In order to facilitate the reuse of IP captured as models in a UML based CASE tool, Rhapsody provides an XMI interface. Rhapsody's XMI interface allows the transfer of IP from a 3rd party tool into Rhapsody without any loss of information. System architectures can be modeled with Rhapsody comparable to ARTiSAN [186, 102, 103]. The intermodule communication is modeled in the form of sequence diagrams. In [101] I-Logix introduces a methodology to model the requirements of communication systems and how to design protocols. In addition, Rhapsody provides a CORBA interface implemented in C++ [104].

[3] Sparx Systems, Online at http://www.sparxsystems.com.au/ [229]
[4] Borland, Online at http://www.togethersoft.com/ [64]
[5] IBM, Online at http://www.rational.com/ [202]

Autosar

The AUTOSAR (AUTomotive Open System Architecture) partnership was launched by the BMW Group, Bosch, Continental, DaimlerChrysler, Siemens VDO and Volkswagen with the aim to manage the growing automotive electric/electronic (E/E) complexity and thus, to establish an open standard for the development of automotive functionality [49]. AUTOSAR focuses on the integration of SW modules based on standardized interfaces and APIs [209, 210] to provide an increased scalability and flexibility in the design process. While the currently available concept allows the integration of functional modules from multiple suppliers, it does not handle the adaptation of heterogeneous components.

SpecC

A scientific approach, which offers a system-level design flow that deals with communication synthesis, is the SpecC language [121, 97]. The SpecC methodology offers a complete design flow from an abstract model to a concrete implementation on Register Transfer level. On the highest level, synthesizable behaviors (see behavior B in Figure 3.9) are connected by channels. This leads to a strict separation of computation and communication. To do so, the SpecC methodology expects a SpecC conform description of the interfaces. Therefore, incompatible tasks, e. g. IPs, can be connected to the remaining design by wrapper channels and adapter modules. Both, *wrapper channels* and *adapter modules* are related to a transducer T. The in-lining of the module into the transducer T results in a synthesizable behavior B as presented in Figure 3.9. The creation of the wrappers or adapter modules is based on the use of high-level communication functions (v_1, v_2). Although a complete design methodology is covered by the SpecC approach, the automated generation of the wrapper channels or the adapter modules is not considered.

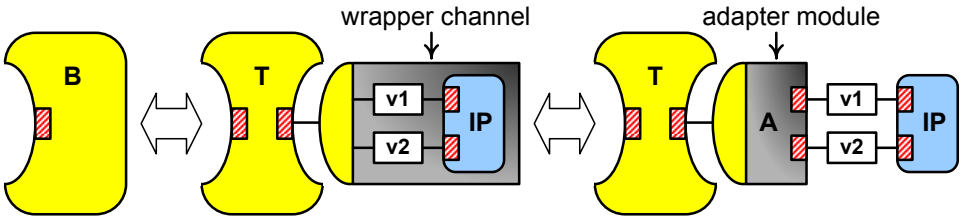


Figure 3.9: IP integration using wrapper channels & adapter modules [121].

A closer look shows that a SpecC adapter module behaves quite similarly to the functionality on an IFB. Both modules adapt two incompatible interfaces, even though the two approaches use a different methodology. Due to the high similarity, an automated way of generating a SpecC adapter module is conceivable.

Conclusion

The UML based design flows concentrate on the development of software for embedded real-time systems. Hardware is not generally considered while UML allows a very abstract and intuitive modeling. Each presented flow provides a concept for handling IPs and is able to create the required interface code. Nevertheless, only Autosar and SpecC focuses on HDL code generation. All of the design flows have in common that they do not cope with runtime reconfiguration. Further approaches with an emphasis on interface adaptation and/or interface synthesis are discussed in Section 3.4.

3.3 Reconfigurable Systems

In this section, we concentrate on reconfigurable architectures to analyze communication techniques that are available in reconfigurable designs. We focus on those hardware platforms, which are potential execution platforms for the hardware realization of the Interface Block. To handle on-chip reconfiguration we consider partially reconfigurable devices, e. g., the Virtex FPGA series from Xilinx Inc., next to those architectures that can be only completely reconfigured. Hereby, we are especially interested in communication techniques that support runtime reconfigurable designs, for example, on-chip busses.

In general, the reconfiguration of software and hardware execution platforms means to switch between a number of predesigned *configuration contexts*. A typical scenario for software reconfiguration is to replace parts of the executed program code. The reconfiguration process of hardware is more complex. In dependency of the architectural granularity – in general we distinguish between fine and coarse-grained architectures – a set of adjustable hardware elements has to be configured.

3.3.1 The FPGA – A Reconfigurable Hardware Platform

In the case of an FPGA, which is a fine grained architecture, a configuration bit-stream is written into the configuration memory (SRAM) via a unique reconfiguration port to determine the behavior of the particular Complex Logic Blocks (see Figure 2.10).

The configuration of FPGAs is technically restricted to a *column-wise* programming style. Thereby, each column is a combination of two adjacent slices as presented in Figure 2.11. Each fragment of a partitioned task, which can span a number of adjacent columns, is called *module*. A partitioned design is then distributed through the design space in the dimensions of space and time. At the least, this means a mapping of FPGA columns according to a given time slot. *Spatial-* and *partial reprogramming* methodologies cope with this problem.

There are two different ways of reprogramming an FPGA. The first approach, referred to as *complete reconfiguration*, stops the total device in execution and updates the portion of the configuration memory in the design which has to be changed. To interrupt the execution, all clocks are frozen. No computation or communication will be performed in any way until the clocks are reactivated again. The second approach, called *partial* or *runtime reconfiguration*, does not influence the functionality of unchanged sections of the FPGA in any way. It just reconfigures the affected frames by updating the dedicated configuration memory.

A challenge for each partially reconfigurable hardware device is the (technical) ability to disconnect the reconfigured circuitry from the remaining one. Therefore, whenever we are involved in partially reconfigurable designs we have to consider three aspects:

1. Modules that are mapped to identical columns have to share the same I/O resources.
2. The electrical interruption of I/O signals may not disturb the integrity of signal values during the reconfiguration process.
3. Single failures or even the complete breakdown of the ongoing communication protocol in the form of a deadlock or a lifelock that occurs from the absence of reconfigured communication partners has to be prevented.

The Xilinx Bus Macro

Partial configuration demands that resources, used by a reconfigurable module, cannot exist outside of a dedicated frame boundary. Moreover, the routing that is used to connect signals crossing reconfigurable module boundaries cannot change when a module is reconfigured. Therefore, all modules which are mapped to the same columns have to share the same I/O signals. To make this technically possible, we include fixed points into our design at the intermediate layer between two modules. As long as not the complete routing information is known during the generation of new modules, we have to refer to these fixed points to interconnect various modules.

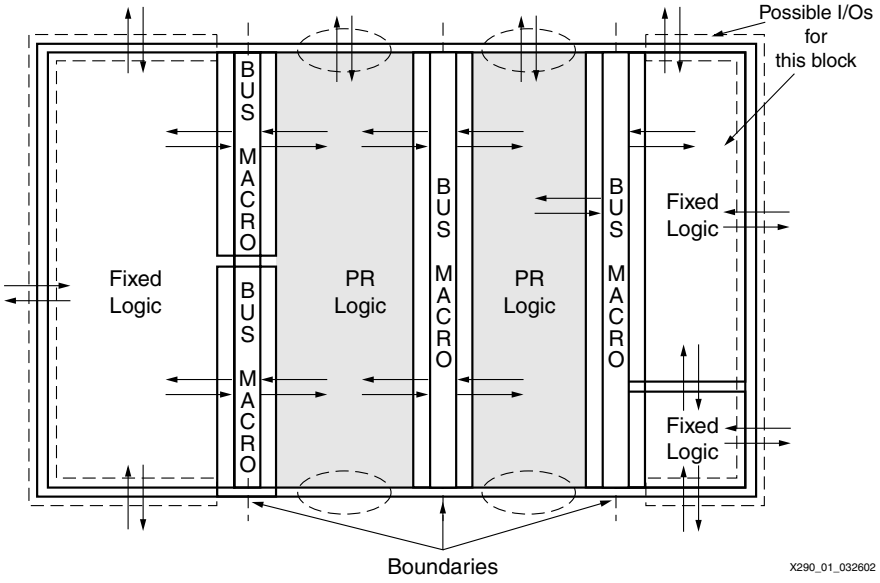


Figure 3.10: Xilinx Bus Macros [170]

The *TBUF* bus [170, 237], which was presented by Xilinx, is an example how a predictable control of routing resources of signals that cross module boundaries can be realized. It is also known as Xilinx *Bus Macro* and can be applied to Virtex FPGAs as depicted in Figure 3.10. Using these Bus Macros, the required fixed points are automatically included by the partial design flow to perform an exact placement and routing of the inter-module signals.

Up to four bits of data can be shared per macro, and the user has to instantiate as many of these macros as needed to cover all the cross-module signals in his design. The final size of a bus macro is determined by the number of connected signals. In Figure 3.10 some bus macros of different sizes are placed between two reconfigurable modules and two fixed modules on the borders of the design.

Electrical Signal Integrity

The implementation of the Bus Macros is based on tri-state drivers. Due to this, a reconfiguration will always lead to the state of high impedance on the dedicated tri-states, and thus keep the signal integrity of those signals, which are created by the remaining circuitry. Developments that are more recent apply specific gates to assume the tri-state functionality, which is cheaper by the meaning of hardware costs [236].

Our approach does not replace the necessity for such fixed points between interconnected modules when they are reconfigured dynamically. Quite the contrary, to cut an established connection properly at runtime, a tri-state buffer has to be included when using FPGAs. However, our approach includes automatically all of these fixed points inside the runtime reconfigurable IFB, and thus they become transparent for the designer.

Reconfiguration Based Communication Gap

Nevertheless, these fixed points bring some restrictions along. In real-time environments, it is necessary to provide a predictable behavior at runtime for all tasks of the design. This implies the computational as well as the communication aspect. During the reconfiguration process of one module, the other modules will remain active, but the communication might get stuck because of the missing signal source or destination. We call such a break in the communication protocol a *reconfiguration based communication gap*. Although the insertion of fixed points allows us to set all remaining communication lines to high impedance, a complex behavior cannot be specified. This definitely violates the requirements of many real-time protocols.

3.3.2 Communication in Reconfigurable Architectures

In the literature, we can find three classes of communication architectures that support the inter-module communication and avoid the overhead of a full reconfiguration:

Communication via third (Processor or dedicated I/O module) This approach has been proposed by Brebner and Walder et al. The communication is performed via a third (CPU in [67] and dedicated I/O module in [225, 247]), which offers an I/O buffer for the modules. To communicate, the sender has to copy the message into its send-buffer inside the CPU. Periodically, the CPU transfers all incoming messages into the receive-buffer of the destination modules. This procedure increases the communication costs and allows only a sequentialized communication through both buffers.

On-Chip Busses In bus based communication architectures [147, 67] all modules interact via a bus system. This requires an arbitration of the bus to guarantee an exclusive access of the device. The serialization of the communication that results from the bus arbitration is one of the major drawbacks, as modules possibly have to wait for their bus grant for a long time. Nevertheless, busses represent a cheap communication architecture.

On-Chip Networks Networks on a Chip (NoC) consist of a set of communication nodes, which are embedded in a network infrastructure and communicate by sending messages [84, 56, 143]. In [176, 177] the inter-module communication has been realized by NoC with nodes of a static size, whereas [60, 61] support nodes of a dynamic size as well.

To implement the bus and NoC based approaches, the FPGA is divided into fix slots of a non-variable size (*slotted architecture*), which of course, prevent the free reconfiguration of the device. The division into slots can lead to a fragmentation of the FPGA as some modules require less size that one slot provides, while others require the resources of several slots. The slotted FPGA architecture is explained in detail next.

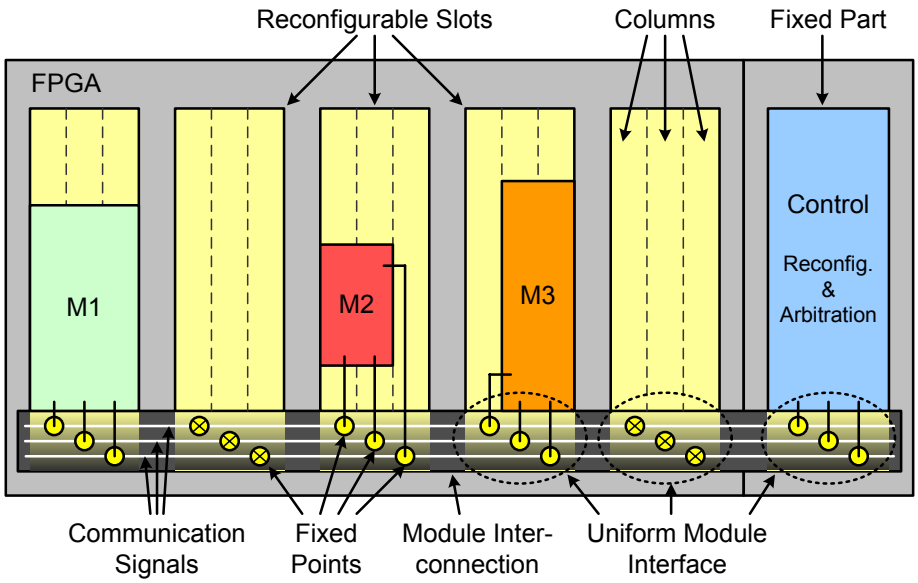


Figure 3.11: Slotted FPGA design

Slotted FPGA Architecture

In many cases, existing architectures using partial reconfiguration employ the presented technical features to compose *slot-based* FPGA designs as depicted in Figure 3.11. A slotted FPGA design consists of a set of *homogenous slots*, which are runtime reconfigurable and optional *fixed parts*. The fixed parts usually implement the FPGA internal control in the form of FSMs which are used to arbitrate the SoC bus, to control the (self-) reconfiguration, or to perform other module independent computation. Each reconfigurable slot provides a dynamic container that accepts modules which have been created for this type of slot. Next to the dimension – in our case the number of columns – it is necessary that each slot and thus, each module, consists of a uniform module interface, including a compliant communication protocol, to fulfill the first condition for module reconfiguration. The required fixed points have to be part of this interface. The published approaches of this domain employ either standardized or proprietary bus or network interfaces. Two examples for such systems are given afterwards.

Example 1 : Raptor 2000

The Raptor 2000 is an FPGA board that can hold up to six Xilinx Virtex FPGAs [157]. The approaches presented in [129, 156, 68] utilize the AMBA bus as standard SoC-bus to interconnect the particular slots. Therefore, each module has to implement the AMBA bus interface. The AMBA bus arbitration is implemented in a fixed FPGA area.

Example 2 : DyNoC

The DyNoC architecture presented by [60, 61] proposes a Dynamical-Network-on-Chip, consisting of several reconfigurable nodes, which are connected by a network routing structure. The routing network is able to guide a communication packet to the correct receiver node. DyNoC works with an proprietary network interface protocol. Therefore, each reconfigurable node has to satisfy the DyNoC interface standard.

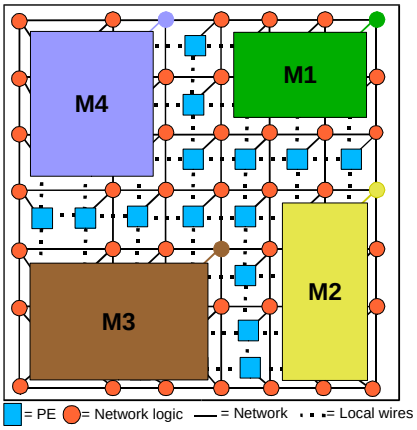


Figure 3.12: DyNoC structure [60]

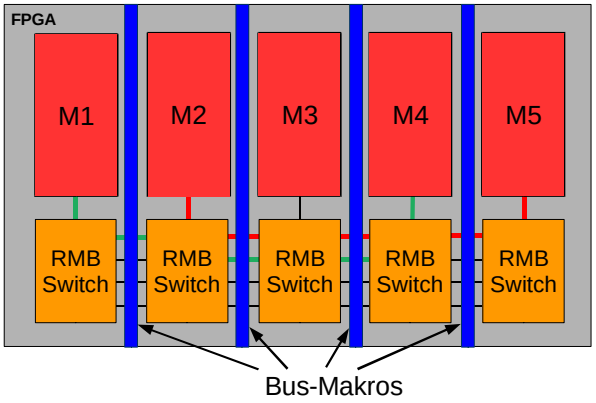


Figure 3.13: DyNoC implementation [60]

Figure 3.12 depicts a configuration geometry with a two-dimensional module placement and a communication network realized as mesh. Each module (M1 to M4) may occupy several Processing Elements (PE). The communication is carried out via the edges of the meshed network structure while the routing takes place in the network nodes (network logic) [63].

A possible implementation of the DyNoC architecture on a Xilinx FPGA is presented in Figure 3.13 which has been created by mapping the two-dimensional meshed network to a one dimensional slotted architecture (as depicted in Figure 3.11). A cost analysis of the required chip area in dependency of the bus width, including the amount modules and the number of segments is given in [41].

3.3.3 How to Avoid the Communication Gap?

Even if the electrical and the signal routing problems are solved by the synergy of modern technology and EDA tools, the challenge of the *reconfiguration based communication gap* remains unsolved in general. However, in real-time systems we require deterministic behavior. Therefore, the published approaches have developed strategies to avoid the reconfiguration based communication gap. We can classify these strategies into four categories:

Reconfiguration Time Hiding

Designs of this category do not access affected signals during the reconfiguration process. We can often find this solution in embedded controller implementations, since they own such a low sample frequency that the reconfiguration can be performed completely between two samples [87, 86]. In this way, the reconfiguration time is hidden by the sampling rate.

Another strategy to hide the reconfiguration time results from applying the concept of pipelining [5, 212, 211, 71]. If the reconfiguration phase becomes an independent stage inside the pipelined execution, it can be completely hidden, as long as the reconfiguration time is not dominating and thus, does not determine the pipeline clock. Signals related to the exchanged stage, which is actually "processed" in the reconfiguration stage, may not be accessed.

Context Switching

In this case, the architecture is not really reconfigured! Both the old and the new contexts are already physically established and a reconfiguration in this category just means switching between both contexts, which could be done by a simple multiplexer. Since the switching process is performed within a single clock cycle, we do not obtain any communication gap.

Redundancy

Redundant systems offer the opportunity to reconfigure one instance while the other instance keeps running. After the reconfiguration is finished, the instances have to be switched, as discussed previously. This category allows us to apply the complete reconfiguration, which is important when FPGAs of vendors other than Xilinx (e. g., Altera or Atmel) are used.

It is a necessary condition that the redundancy has to be greater than or equal to the quotient of the reconfiguration time to the computation time. If, for example, the reconfiguration time would be 3 ns and the computation would add up to only 1 ns, the design requires at least a three times redundant architecture if all devices work fully parallel and in a cyclic manner. Another drawback of redundant systems is that the current state and the computed results of all modules, even those inside unchanged slots, have to be transferred from the current device to that one which continues with the computation.

Reconfiguration Time Agnostic

Approaches that belong to this category do not care for the reconfiguration time. The applications of this class are not applicable to real-time or fail-critical systems, since we have to assume a break of the communication protocol, e. g., a deadlock in the protocol FSM or the loss or falsification of data, which could cause the complete system to crash.

Conclusion

We have seen that we do not have to care explicitly for electrical effects in matters of reconfiguration. The device vendors offer solutions to handle the interconnection of communication signals, for example, the Bus Macros in the case of Xilinx Virtex FPGAs. Further on, the mapping and routing for sharing I/O signals between several modules is supported by the synthesis tools.

We have presented four strategies on how actual designs cope with the communication gap. To ignore the reconfiguration time in a reconfigurable embedded real-time system is completely out of the question. In special scenarios, the reconfiguration time can be hidden, for example, as additional stage in pipelined architectures. If there is enough area available on the device, all contexts can be implemented in parallel and a simple reconfiguration-context switch may solve the problem. The same is true if we possess sufficient redundancy in our architecture and the device synchronization is no handicap. In whichever scenario the particular methodologies may perform well, neither of the approaches offers a solution to treat the reconfiguration based communication gap in general.

Based on the partial reconfigurability, our approach allows us to execute an a-priori defined behavior during the reconfiguration of interconnected blocks. In this way the design works deterministically all the time. The IFB architecture and the reconfiguration process that we developed to cope with this challenge is precisely explained in Chapter 6.

3.4 Dedicated Interface Synthesis Approaches

The third section of background information and related work is dedicated to the synthesis of interface adapters. This field of research is embedded in the *protocol-engineering* domain [172, 161] as depicted in Figure 3.14. Protocols are either modeled in the form of *service specifications* or directly as *protocol specifications*. To process a service specification, it has to be transformed into a protocol specification by a protocol synthesis. The final protocol implementation (executable protocol) is then created by a further synthesis step. Ideally, this final implementation step is performed fully automated. Protocol-engineering employs four kinds of operations on protocol specifications:

Validation / Verification Protocol *validation* - and *verification* methods are used to ensure the functional correctness of protocol specifications and allow us to proof if all timing constraints are met. Further on, it helps to find deadlocks and synchronization faults. While validation methods are based on tests – in [30] we present a probability-based testing method – verification utilizes formal methods, i. e., model checking.

Conversion / Internetworking The aspect *conversion / internetworking* handles the interconnection of interacting communication partners for SoC as well as for heterogeneous distributed environments. Whenever we have to cope with the *adaptation* of *incompatible protocols*, we apply *interface synthesis* techniques for the protocol conversion.

Performance Analysis Based on the *performance analysis* we are able to measure values like throughput, redundancy, latency, response time, failure sensitivity, etc. These metrics make protocols comparable between each other, enable us to fine tune, and select them for dedicated fields of applications.

Conformance Testing In addition to the other services, the *conformance testing* is applied to proof if a generated protocol has been created consistent with the protocol specification. Equivalence checking is one formal method to proof the conformance of protocol implementation and protocol specification.

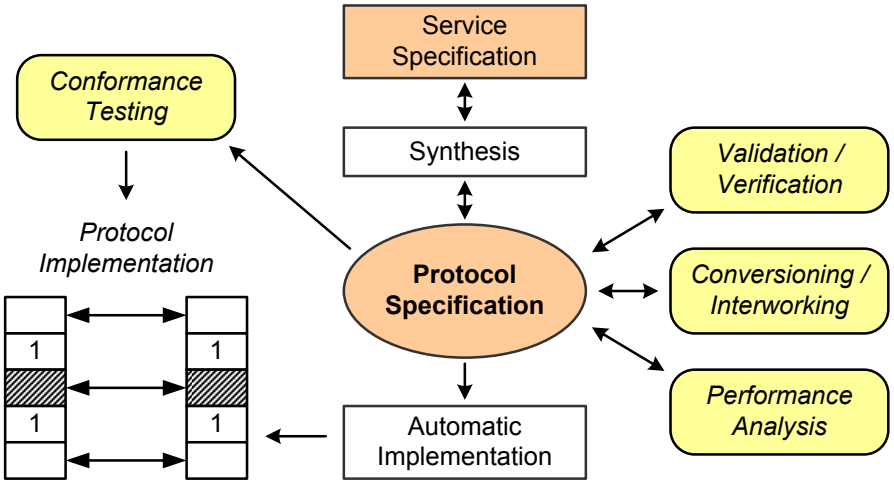


Figure 3.14: Survey of Protocol-Engineering [172]

This thesis is related to the protocol-engineering operation *Conversion / Internetworking* in combination with the automatic implementation of the interface adapter implementation. Thereby, the interface descriptions of the adapted components, which are by our definition an interface model including the communication protocol, have to be available as input for our synthesis algorithms.

It is important to note, that we strictly differentiate between the terms *interface adapter* and *interface*, even though the term interface is often used as synonym for interface adapter in the literature. An interface is an integral part of a communication component while an interface adapter is a self-containing entity that interconnects interfaces. Due to the ambiguity of the term interface, existing *interface synthesis* approaches cover the construction of component interfaces as well as the synthesis of self-containing protocol adapters.

Systematic Protocol Construction vs. Protocol Wrapping/Adaptation

The systematic construction or modification of (component) interfaces is one possibility for composing interacting communication systems. Therefore, partially modeled protocols are extended or additional interfaces are created, from the information provided by a number of completely designed protocols, to integrate further components. The protocol translation is directly integrated into the interfaces and becomes an inextricable part of the protocol. This kind of communication synthesis is applicable only to user-defined designs, since the interfaces and therewith the components have to be modified. To create or extend interfaces in this way, we require an insight into the component's *white box* model.

In the case of IP based design, interfaces are fixed and cannot (or may not) be modified. Here, the available interface information is applied to create *wrapper* or *adapter modules* that interoperate between the interacting components. The operation of adapter or wrapper modules is performed transparently for the interconnected interfaces and gets by with the information of the component's *black box* model.

On one hand, it is more efficient to implement the protocol translation directly inside the interfaces by means of implementation resources and latency, as no additional translation entity has to be added. On the other hand, the transparency of the adapter/wrapper solutions does not necessitate the intervention into components. To construct interacting interfaces we can either utilize *analytic* or *synthetic* protocol synthesis steps [200]; while the creation of adapter modules is restricted to analytical methods, which create the protocol translation by an analysis of the information delivered by the interconnected interfaces.

Protocol Translation Paradigms

Independent of the realization as stand-alone adapter or the integrated version, the protocol translation paradigm plays an important role. We distinguish between two methodologies: the one is based on *product automata* while the other applies the *Input - Processing - Output* (I - P - O) concept. Approaches related to the first class construct an optimized translation automaton that is a minimized version of the product automaton. The second class subdivides the process of protocol translation into three consecutive processing steps and thus avoids the complexity problem of a state explosion that the product automata implicates. Therefore, the separated processing steps have to communicate with each other to synchronize and exchange data, which also implies an overhead.

Hardware vs. Software Interface Adaptation

Analog to the description of interfaces we have to distinguish between interface synthesis approaches for hardware, software, and hardware/software interfaces. The adaptation of software is usually performed by specific middleware solutions that establish the interaction of communication APIs represented by functions or public object methods.

On the hardware side, dedicated translation circuits implementing control and data path are created. A common technique of adapting software APIs to hardware interfaces is to synthesize customized drivers that become part of the communication middleware or are directly integrated into the operating system. Next to API based communication, memory mapped I/O has to be considered for Hw/Sw interface as presented in Chapter 2.

3.4.1 Interface Synthesis for Communication APIs (SW/SW)

The API based interface synthesis is not the focus of this work, therefore we won't go into detail here. Just to convey the idea of the most relevant approach in the object-oriented domain, we introduce CORBA.

CORBA

The Common Object Request Broker Architecture (CORBA) is an open distributed object computing infrastructure standardized by the Object Management Group (OMG). It defines APIs, protocols, and object/service information models to enable heterogeneous applications written in various languages, and running on various platforms to interoperate.

The OMG defines interfaces as follows [191]: "An interface is a description of a set of possible operations that a client may request of an object, through that interface. It provides a syntactic description of how a service, provided by an object supporting this interface, is accessed via this set of operations. An object satisfies an interface if it provides its service through the operations of the interface according to the specification of the operations".

In a general sense CORBA "wraps" code written in some language into a bundle containing additional information on the capabilities of the code inside, and how to call it. The resulting wrapped objects can then be called from other programs (or CORBA objects) over the network. In this sense, CORBA can be considered as a machine-readable documentation format, similar to a header file but with considerably more information.

CORBA uses an interface definition language, the OMG IDL, to specify object interfaces [191]. It then specifies a "mapping" from the IDL to specific implementation languages like C++ or Java. This mapping precisely describes how the CORBA data types are to be used in both client and server implementations. Standard mappings exist for Ada, C, C++, Lisp, Smalltalk, Java, and Python.

To communicate, CORBA applies the abstract General Inter-ORB Protocol (GIOP) [196]. The IIOP (Internet Inter-Orb Protocol), for example, is a concrete implementation of the GIOP for TCP/IP. To connect to CORBA each object has to implement the GIOP itself. Therefore, a fully transparent adaptation of software components is not possible. For this reason, CORBA cannot be directly used to adapt COTS (Commercial Off-The-Shelf), which do not already implement the CORBA standard.

3.4.2 Systematic Protocol Construction Approaches (HW/HW)

Early work on the analysis and synthesis of protocols has been published by [251]. From the mid 80th to the mid 90th the interface synthesis domain was very popular and thus, many publications can be found in literature related to this period.

Interface Synthesis Classification Tree

Probert and Saleh classify relevant interface synthesis approaches for the systematic protocol construction published in this period in [200] and [207]. One result of this work is the *interface synthesis classification tree* (depicted in Figure 3.15) that distinguishes between *Service Oriented* and *Non-Service Oriented* methodologies.

Chu

A service oriented method that synthesizes protocol specifications from service specifications was presented by Chu et al. [77]. The services are modeled as FSMs that describe the interleaving of the interactions at two service access points (SAP). A synthesis algorithm transforms this specification into two closely synchronized protocol machines, which interact via a reliable communication channel.

Merlin, v. Bochmann

Another service oriented approach modeling services as interaction of SAPs was proposed by v. Bochmann et al. [245, 243]. The required service specifications are described by a subset of LOTOS operations [231]. A syntactic tree representation of this specification is automatically processed into protocol entities using a set of production rules. A mixed method described by v. Bochmann and Merlin in [183], starts from a given service definition and the specification of all but one of the protocol entities, then synthesizes the missing entity. Therefore, a tightly coupled interaction model is used that defines the sending and the subsequent receiving of a message as an atomic operation without a delay. More recent work of v. Bochmann handles the synthesis of complete communication systems [244].

Probert, Saleh

In [208] a set of distributed services specified by FSMs is mapped to a number of given SAPs. With the help of transition synthesis rules, FSM based protocol specifications are composed of message transmissions and receptions. These rules are service primitives customized by the direction, the temporal order, and the structure of the given service specification.

Non-service oriented methods start from partial or complete protocol descriptions as mentioned before and perform an automated or interactive synthesis based on FSMs.

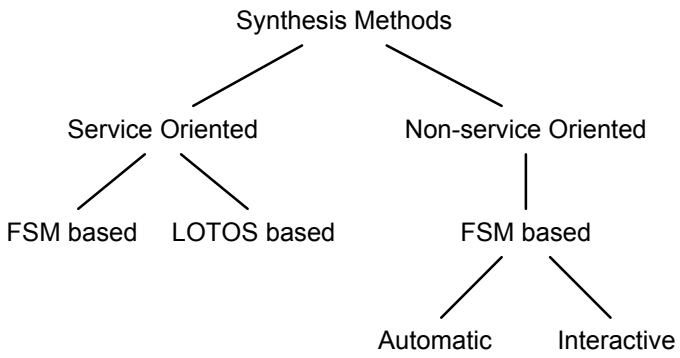


Figure 3.15: Interface synthesis classification tree [200]

Zhang Zhang et al. propose an interactive synthesis method to create interfaces that communicate via FIFO channels [252]. Based on a set of production rules, the synthesis algorithm creates the global state transition graph of the protocol to be synthesized and then produces the protocol by decomposing this graph into two communicating FSMs.

Zafiropulo Similarly, Zafiropulo et al. [251] present an interface synthesis based on production rules using FIFO channels. Every time a designer adds transitions to one protocol, a tracking algorithm extends the interacting FSM by a complementary transition. After each design step, the application of the production rules is evaluated. In this way, properties like deadlocks, incompleteness, and the ambiguity of particular states can be detected.

Gouda Another approach with an integrated protocol translation and including a FIFO channel has been published by Gouda et al. in [127]. Two interacting FSMs M' and N' are created by an algorithm called "machine synthesis algorithm" from the incomplete FSM specification M. A second algorithm, the "channel capacity algorithm" determines the required size of the FIFO between M' and N'. Due to synchronization problems in states that write and read as well, data can be lost, as the FIFO content is discarded as part of the resynchronization.

Ramamoorthy A communication using two reliable unidirectional channels between a local given entity and an automatically generated peer entity has been proposed by Ramamoorthy et al. in [201]. Therefore, a Perti Net model of the local entity is translated into an FSM representation. Based on six transformation rules, the peer entity is constructed including a set of fixed communication patterns. To be able to apply Petri Net analysis techniques, the peer entity is then translated into a Petri Net representation.

Kakuda Kakuda et al. presented a component based synthesis algorithm using process interaction diagrams [154, 155]. In their concept, each component represents a protocol specification primitive consisting of a pair of fundamental process behavior diagrams. Inputs of the protocol synthesis are process interaction diagrams. Starting from these diagrams, the protocols are synthesized by combining the previously modeled components.

The presented approaches differ clearly from our methodology as we propose a protocol translation in the form of an interface adapter. We decided on the adapter solution to allow the direct adaptation of IPs and thus, to support a system composition exclusively based on IP components. Approaches that handle the translation based on wrapper or adapter modules are discussed afterwards.

3.4.3 Protocol Wrapping/Adaptation Approaches (HW/HW)

VSIA – OCB VCI Two approaches using wrappers for the adaptation of VCs and IPs are VSIA's OCB VCI standard and SpecC. VSIA specifies the adaptation of Virtual Component Interfaces (VCI) within the On-Chip Bus Virtual Component Interface (OCB VCI) standard [106]. To interconnect two virtual components one master wrapper and one slave wrapper are created for the VCIs. The data between the two wrapper modules is transmitted over an on-Chip bus (OCB) of the SoC integrator's choice. VCIs are classified into Peripheral VCIs (PVCI), Basic VCIs (BVCI), and Advanced VCIs (AVCI), which allow us to model interfaces of an increasing complexity. Protocols between initiator (master wrapper) and target (slave wrapper) are composed of simple request and response messages. The protocol translation is performed by wrapping both protocols to the shared bus protocol. A disadvantage of the OCB VCI is that it is not automated and the translation has to be implemented by hand.

Like OCB VCI, SpecC allows the integration of incompatible IPs, by the use of hand written wrappers [95, 96, 98], to translate the communication protocols specified by bus functions (as depicted in Figure 3.9). Alternatively, the protocol translation can be implemented by user defined adapter modules. The wrapper/adapter modules have to be written in the SpecC language and become part of the SpecC synthesis flow. An automated generation of the wrapper/adapter modules is not offered by the SpecC approach [218, 94]. Gajski, Dömer – SpecC

Another approach taken by Gajski et al. [185, 184] proposes to reduce protocol specifications to the combination of five basic operations (data read/write, control read/write, time delay). The protocol description is then broken into blocks (called relations) whose execution is guarded by a condition on one of the control wires or by a time delay. Finally, the relations of the two protocols are matched into sets that transfer the same amount of data. Although this algorithm is able to account for data width mismatch between the two modules, the procedural specification of the protocols makes it difficult to adapt different data sequencing, so that only the synchronization problem is solved. The concept was extended to IP based design in [217] and an automatic communication refinement for system level design in [39]. Gajski

A different work is that of Boriello [66, 65], who introduces the "event graph" to establish the correct synchronization of data sequencing. The limitation of this approach is that the two protocols should be made compatible by manually assigning labels to the data on both sides, since the specification of the protocols is given at a very low-level of abstraction using waveforms. Jane Sun [227] extends the approach by providing a library of components that frees the user from considering low-level details, but does not solve the mentioned problems. Boriello

Akella and Mc Millan presented an approach, which assumes a pair of interacting protocols to be given, modeled in the form of two deterministic finite state machines, while a third FSM represents the valid transfer of data [43]. The product machine is taken and pruned of the invalid/useless states. The limitation here is that no data width mismatch can be handled and that the designer must manually enter the intended behavior of the interface in the form of the third FSM (called the C-machine). Akella, Mc Millan

Similar to Akella and McMillan, Passerone developed a protocol adapter that automatically resolves the correspondence between pieces of data of two interacting components. The adapter can translate between different sequencing of the data. It is created in the form of a deterministic FSM in combination with a simple data path from regular expressions based on the automata theory presented in [146]. The synthesis algorithm of the communication protocol applies a grammar-based specification similar to the one presented in [188]. The effectiveness of protocol intensive specifications based on grammars has been shown in [214]. However, the approach considers only point-to-point connections and assumes the two communicating parties to be driven by the same clock (synchronous communication), a limitation that is not present in [185] and in [66]. Both sides apply the same data types without an understanding of the concrete data semantics. The translated data is interpreted as strings of bits, therefore the number of sent bits has to be identical to the number of bits received. Therefore, the context sensitive translation of data is impossible. Passerone

Müller and Siegmund presented a SystemC based specification for protocols [220]. From this specification a deterministic interface adapter FSM is generated [222, 221]. In order to avoid a consideration of the complete product FSM, an architecture template is introduced. For the resulting interface adapter FSM, VHDL code is generated as input for a low-level synthesis [219]. Müller, Siegmund

Sangiovanni Vincentelli

In [205] Interface Based design was proposed as a methodology that attempts to orthogonalize the communication and the behavior of IPs. Therefore, IP blocks are abstracted to a transition or messaging level. In this abstract view, complex blocks exchange messages using a robust user-defined type system rather than using complex signaling conventions based on conventional wires.

The abstracted communication leads to an improvement in the simulation performance comparable to the approach presented in [120, 218] by Dömer and Gajski. However, this abstract level is not sufficient for the automated implementation of the protocol translation. One solution to get to the final implementation is the stepwise refine of the protocols down to a pin- and cycle-accurate bus functional model.

De Micheli – POLARIS

The interface synthesis algorithm presented in [224] can be used to generate a cycle-accurate, synchronous interface between two hardware subsystems, given an HDL model of each subsystem. The algorithm has been implemented in the POLARIS hardware composition tool. An interface adapter provides a FIFO-In and a FIFO-Out queue for each connected component (data path) in combination with a state machine (control path), which is responsible for the arbitration of the message exchange. The translation is performed by a central state machine that is created by the synthesis algorithm analyzing the HDL input.

The presented work on POLARIS focuses on generating a low-level, synthesizable description of synchronous interface adapters between hardware components. Similarly, [76] describes a mechanism for creating the glue logic between two hardware components, but requires a functional description of component ports.

3.4.4 Adaptation of Hardware Software Interfaces (HW/SW)

Next to the presented approaches for hardware and software, the interface synthesis has become an important research area in the hardware/software codesign domain as well. One approach, specialized on programmable devices, has been published by Eisenring & Teich in [107]. Knudsen & Madsen developed a communication estimation model and show, by the use of this model, the importance of integrating communication protocol selection with hardware/software partitioning [163]. Another research aspect focuses on the optimization of high-level communication between subsystems given a set of communication constraints as published by Ernst et al. in [109].

Interface modeling languages, such as that developed by Öberg et al. [188], allow a designer to explore a interface design space and generate a synthesizable description of the interface. Rosenstiel et al. [204, 133] describe a methodology for modeling an interface with a behavioral description suitable for high-level synthesis. Other interface synthesis research, such as that performed in [119] by Glesner, has investigated specifying and scheduling communication between hardware and software subsystems.

Boriello – Chinook

The Chinook approach extends the commonly used concept of device drivers to include the bus interface as well [248]. The driver/interface is described in a timing diagram description that captures both the behavior and timing constraints for the interface. The description is synthesized into low-level software code that accesses the device via the ports of a micro controller together with the required glue-logic. The synthesis procedure tries to find the cheapest implementation (smallest amount of hardware) regarding both timing and resource constraints.

Ortega

In their later work, Ortega et al. expand the approach to communication synthesis for a distributed system [198, 197, 76, 75]. Input to the system is a behavioral description consisting of communication processes. The processes communicate with each other via messages sent through ports connected by channels. All communication is broadcast type that is one to many unidirectional.

For each output port, the designer selects a high-level abstract protocol, that is, blocking or non-blocking, along with a deadline constraint. For each input port, the designer selects the appropriate queuing semantics. The designer also defines a system architecture that consists of heterogeneous processors connected by standard bus protocols (for example, TTP [166, 165]). Processes and communication channels are then mapped onto the selected architecture. The system then generates the communication interfaces. If there is no direct connection between two communicating processes, intermediate hop processes are automatically inserted to route a message from one bus to another bus. For each processor, a customized real-time operating system is generated that includes a real-time scheduler, a message routing process, and device drivers.

O'Nils

A further approach for the hardware/software interface synthesis was presented in [193] that ranges from the specification to implementation and the validation of hardware/software interface protocols. O'Nils models the information that is required for the automated interface synthesis in the form of grammar-based protocol specifications, information about the processor, and information related to the operating system. A synthesis tool allows us to generate device driver functions, a combination of device driver functions, and a DMA controller or simulation models. The approach includes a design space exploration with the combination of a processor, operating systems, and protocols.

Nebel, Oppenheimer

A methodology published by Oppenheimer & Nebel allows the automated synthesis of shared memory for the communication of hardware and software via memory mapped I/O [194]. As input for the synthesis algorithm, it is necessary to specify the communication registers in every detail. The approach proposes to model hardware/software interfaces in an XML-based interface description language called COMIX [195]. The abstract and target language independent modeling technique based on COMIX is intended to improve the designer's productivity and the systems reliability through reuse and automatically generated target code. A tool architecture named COHSID has been implemented to generate software device drivers and hardware I/O components automatically from a COMIX specification.

CoWare

CoWare is a commercialized codesign environment that addresses interface synthesis for hardware/software communication. CoWare targets at the simulation and the design of heterogeneous DSP systems. Supported input languages include SystemC, VHDL, Verilog, and C. In CoWare, interprocess communication is done with point-to-point communication channels.

The communication semantics is based on the concept of rendezvous channel communication via send/receive operations. Hardware/software communication channels are mapped onto a fixed architecture. This architecture is based on several library models. For software, the communication is captured as parameterized C functions that are mapped onto a software model, that is, they adapt to processor specific I/O handling, interrupt handling, etc. For hardware, a hardware interface cell is generated to connect via a handshake protocol to an I/O control unit. This I/O control unit is a link between the processor and the handshake protocol.

3.5 Summary

In the last section, we introduced the protocol-engineering domain. The interface synthesis is a sub-domain that covers the two aspects protocol conversion and automated synthesis of the protocol implementation. As stated in the introduction, these aspects are also major challenges of this thesis. Another technique of the protocol-engineering domain with a direct relation to our work is the conformance testing, which is used to validate the final protocol implementation. We do not have to cope with this challenge as we ensure the correctness of our implementation by a completely automated synthesis without any user interaction.

Furthermore, we introduced the Interface Synthesis Classification Tree. It distinguishes between service orient approaches, which have to be synthesized into a protocol description before they can be further processed, and the non-service oriented ones, which directly start from the protocol description. The approach presented in this thesis belongs to the second class, since the modeling phase provides the interface synthesis with a specific protocol description.

We have seen two alternatives on where to implement the protocol translation. On one hand, systematic protocol construction techniques are applied to extend component interfaces by the required communication parts; on the other hand, protocol wrapper or adapter modules offer a self-contained protocol translation. While the first method requires a direct intervention into the interacting components, the second method also supports black box models, and thus the integration of IPs. Nevertheless, the systematic protocol construction allows implementations that are more efficient by the meaning of implementation resources and latency compared to wrapper and adapter modules.

One metric for the classification of protocol adapters is the protocol translation paradigm, which can be based on I - P - O or an optimized version of the product automata. The second method is mostly applied in the case of FSM or grammar based approaches. Here, the main goal is to avoid a state explosion by creating a minimal state machine that translates between the communicating partners. The I - P - O concept avoids this problem by nature and is used in scenarios that are more complex. Therefore, the I - P - O scheme implies an additional implementation overhead concerning internal synchronization and communication.

We introduced several approaches related to hardware and/or software interfaces to summarize the most relevant related work and the current state of the art in interface synthesis. As we have seen, the adaptation of interfaces has been continuously explored during the last decades. By contrast, reconfiguration is a quite recent paradigm to the computing domain which has become very popular within the last years, and not only through the advances in reconfigurable platforms. Newly developed reconfigurable computing concepts need adequate methodologies to support the automated interface synthesis in reconfigurable embedded systems. This implies the automation to create executable code for reconfigurable architectures as well as the use of modern modeling languages like UML 2.0.

Obviously, the reconfiguration of system functionality requires the ability to adapt exchanged modules. Even if these modules are compatible by their interfaces, there remains the need for a deterministic behavior during the reconfiguration process in order to avoid communication gaps. We lack a combination of interface synthesis and reconfiguration concepts with an automated tool support. By our state of knowledge, there exists no approach in the literature, comparable to ours, that combines all three aspects.

3.5.1 Interface Synthesis Requirements Specification

As contribution to the state of the art, we propose an innovative methodology that allows us to perform a *deterministic* and *transparent reconfiguration* of tasks at *runtime* due to the *synergy* of *interface synthesis concepts* and *reconfiguration techniques*. As first approach in the *interface synthesis/reconfiguration* domain, we present an *integrated high level synthesis design flow* from *UML 2.0* down to *synthesizable VHDL*.

With respect to the delivered background information we can now *specify* the *requirements* for our Interface Synthesis Design Flow and the Interface Block:

Design Flow Requirements

- Integrated design flow to create the Interface Block
  - Provide an abstract and intuitive modeling
  - Maximize the degree of automation
- Combine interface synthesis with runtime reconfiguration techniques

In this thesis, we present an *interface-centered design methodology* embedded into an integrated design flow, including the automated synthesis of the Interface Block. We selected UML 2.0 as the modeling language to provide an abstract and intuitive interface to the designer. As we propose a scenario-based protocol translation, a mapping of the adapted interfaces has to be provided for each synthesis process. Aside from this user interaction, the interface synthesis process works completely automated.

Interface Synthesis Requirements

- Interconnect multiple hardware/software tasks/media with incompatible protocols
- Interface adaptation
  - Self-containing and transparent protocol adapter
  - I - P - O based protocol translation
  - Protocol adapter implementable in hardware and software
  - Handle synchronous and asynchronous communication
- Scenario-based protocol translation

While many approaches are restricted to point-to-point and master/slave communication, our approach supports the adaptation of multiple components including duplex traffic. Depending on the execution platform and the applied code generation, synchronous and/or asynchronous communication can be processed, while both variants can be expressed in our models. The self-containing Interface Block works completely transparent and requires no interaction with component internal signals.

Our interface adapter model applies the I - P - O paradigm for the protocol translation. To process the transmitted data, a set of interacting FSMs is generated from the given protocol description and the current data mapping. Thereby, our protocol description is comparable to the one presented in [217] by Gajski et al., which also applies the I - P - O scheme but provides no data transformation; the data is just passed through in a customized queue.

Next to signal based HW/HW interfaces, the presented approach is allows the adaptation of SW/SW and HW/SW interfaces based on memory mapped I/O. Considering function-based communication including the creation of dedicated device drivers is out of the scope of this work, but is part of our ongoing research.

We distinguish between the IFB model and a specific IFB implementation. The IFB model has been designed to be as general as possible to support hardware, as well as software interfaces. The IFB implementation presented in this thesis proposes a hardware construction pattern on the RT-level (Register Transfer level) – the IFB Hardware Template – optimized for the adaptation of several hardware interfaces, including memory mapped I/O. As the implementation of an IFB varies due to the applied programming paradigms, the precise evaluation of an IFB is highly dependable on the concrete implementation model.

Reconfiguration Requirements

- Support *coarse-grained* reconfiguration of the IFB to
  - Allow the predictable and transparent reconfiguration of tasks and media
  - Guarantee deterministic behavior during reconfiguration
  - Bridge communication gaps
- Allow *fine-grained* reconfiguration of the IFB (reconfigured IFB execution) to
  - Optimize the required area of the IFB hardware implementation

To exploit the capacity of runtime reconfiguration we developed a course-grained and a fine-grained IFB reconfiguration. The course-grained method allows the predictable and transparent reconfiguration of tasks and media at runtime. This is done by executing a predefined behavior in relation to the currently reconfigured tasks or media. To perform a *transparent reconfiguration* means to exchange affected components without informing or interrupting the unaffected ones. With the help of the coarse-grained reconfiguration, our Interface Block allows us to handle communication gaps and thus, to make general use of runtime reconfiguration in real-time and fail-critical embedded systems.

In the case of the fine-grained IFB reconfiguration, we employ runtime reconfiguration to perform a reconfigured execution of an Interface Block. The I - P - O scheme naturally leads to a pipelined data processing. Depending on the interacting protocols, some "stages" of this pipeline – which can be of the type input, processing, or output – can be idle. In our approach, only those stages, which currently have to process data, are downloaded to the reconfigurable hardware device (FPGA) by replacing unused stages on the chip. In this way, reconfiguration helps to save implementation resources in the form of required chip area and thus reduces the costs of our Interface Block.

The following Chapter 4 presents our methodology that we developed to fulfill the previously mentioned requirements. Afterwards, we explain the realization of the integrated design flow (Chapter 5) and the hardware implementation of the Interface Block (Chapter 6) in detail.

CHAPTER 4

Interface Synthesis Methodology

"*Divide et impera*" (*lat.*, *in English*: divide and conquer) is a sentence formulated by the French king Lui XI, or perhaps even by Juilius Caesar. At that time, it described a military strategy to govern occupied territory. The sentence also represents a well-known strategy to solve problems in the computer science domain (e. g., in *divide-and-conquer* algorithms). In the figurative sense, it means to divide a problem into manageable subproblems, which can be solved more efficiently than the problem as a whole. Thereby, the solutions of the particular subproblems are joined to solve the original problem itself. This strategy can be reapplied recursively down to the level of atomic subproblems.

In Chapter 2 we have defined a communication framework to characterize our approach and to introduce the fields of application of an IFB. In the following Chapter 3 we offered further background information including system-level design, abstraction levels, IP-based design, dynamic reconfiguration, and interface synthesis. Altogether, this information provides the basis for our Interface Synthesis methodology.

In the first part of this chapter, we apply the divide-and-conquer principle to refine the *Interface Synthesis Design Flow*, which is a central aspect of the Interface Synthesis methodology, successively into manageable portions. We explain the partitioning of the design flow into design phases and design steps. In doing so, we deliver an abstract survey of the particular phases and figure out their functionality as well as their interaction. We take a closer look at the syntax and semantics of protocols and discuss our *IFS modeling concept* with respect to XML, Jave and UML. A detailed explanation of the Interface Synthesis Design Flow is subject of Chapter 5.

In the second part of this chapter, we introduce our protocol adapter, the *Interface Block* (IFB), which is automatically generated by the Interface Synthesis Design Flow. We present the IFB Macro-Structure that describes the internal structure of an IFB and explain its functionality as basis for the Chapters 5 and 6. Based on this, we discuss the pipelined data processing inside the IFB. Another important aspect is the IFB reconfiguration. Finally, we highlight the relation of the IFB to the ISO/OSI model and introduce an environment for real-time communication.

4.1 Interface Synthesis Design Flow

The *Interface Synthesis Design Flow* is an integrated design flow, which was developed to create transparent adapter modules between the interfaces of interacting tasks [1] comprising incompatible protocols. The resulting adapter modules are called *Interface Block* (*IFB*). The Interface Block model was designed to support the interconnection of software interfaces as well as hardware interfaces.

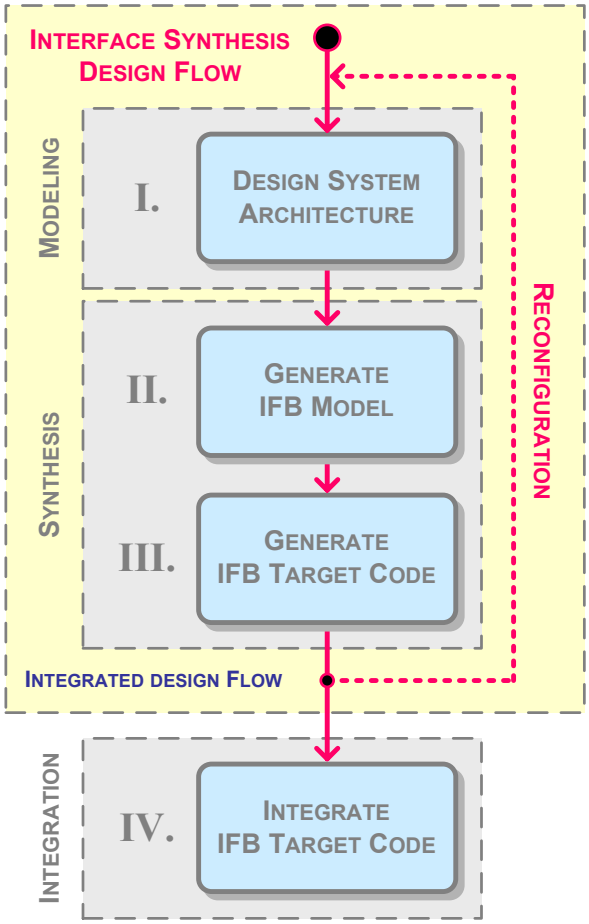


Figure 4.1: The Interface Synthesis Design Flow

As presented in Figure 4.1, the Interface Synthesis Design Flow implies all necessary design steps, beginning with an abstract modeling, to create the executable IFB code. Therefore, the particular design steps of the Interface Synthesis Design Flow (colored in blue) have to be passed through sequentially. Basically, the flow divides into a *modeling phase* and a *synthesis phase*. The modeling phase consists of only one particular design step, whereas the synthesis phase is split into two successive design steps. As shown on the bottom of Figure 4.1, the Interface Synthesis Design Flow is followed by a completing *integration phase*.

[1] As stated in Chapter 2 we use the term "task" as synonym for "task and medium" since all propositions are true for media and tasks. If this is not true in a special case, we will explicitly mention this.

4.1.1 Modeling Phase

The *modeling phase* comprises only one design step, entitled "Design System Architecture". It describes an interactive process in which a designer specifies the input for the IFB synthesis and is based on our *modeling concept* that we introduce in the next section.

As presented in Section 2.2 the IFS System Architecture has been developed to model complex communication scenarios optimized for the embedded systems domain. We employ the System Architecture to describe the *communication infrastructure*, which is composed of architecture components and communication components. The components themselves are treated as black box, only accessible by their interfaces. Based on the *interface descriptions* (*IFDs*) and the *target platform descriptions* (*TPDs*), which are related to the communication components and the architecture components, respectively, the System Architecture covers nearly all information that is required for the automated synthesis of the IFB.

Protocol-Syntax & Protocol-Semantics

The missing information refers to the mapping of the user-data that is transmitted via the IFB. If we remember the definition of an IFD, given in Chapter 2, we are able to model protocols as part of the IFD. This means that we can precisely specify states, state transitions, and the value of protocol-pins in particular states. This allows us to model the (packet-) structure of all data and control bits inside a protocol. We call this the *protocol syntax*.

Definition 2.8 in Chapter 2 considers *SignalValue*, *UseCase* and *Direction* to be assigned to a protocol-pin (e. g., incoming data, value 'unknown' or outgoing control, value '1'). Although we are able to model the protocol syntax, we cannot model the *semantics* of particular bits within the IFD. This aspect differentiates our approach from many others. The decision to dispense with global semantics for single bits is based on the following reasons:

Approaches that utilize global semantics to map data are reduced to designated scenarios. Given a scenario that defines a data word *'a'* inside the two protocols P_1 and P_2, the data word *'a'* can be mapped by its name between P_1 and P_2. In this case, the data word *'a'* is restricted to the current scenario and may own a completely different meaning from a data word *'a'* in another scenario using the protocols P_3 and P_4. If it is necessary, for example, to interconnect P_1 and P_3, both specifications would have to be rewritten.

As we divide bits into control- and data bits, one could imagine finding a more general description for the control bits. For example, a start bit or a CRC field (Cyclic Redundancy Check) could be labeled in such a way that they were recognized correctly in many different use-cases. Defining such a characterization for the data bits, which represent the *information* of a protocol, would lead to an explosion of the name space. In all those cases, where the meaning of the data bits depends on a particular combination of two interfaces, a static description becomes useless.

Let us demonstrate the necessity of a scenario-based data mapping by an example: There are two toy-trains, one produced by Märklin, the other by Trix. Both train systems provide a serial interface (RS232) to control the trains. A microprocessor inside the train system interprets the incoming serial packages to decode the commands, like "stop train 2".

Although Märklin and Trix use the same protocol-syntax, the meaning and thus, the protocol semantics of the data bits inside the serial packages is completely different. This results not only from different bit positions inside the data packages, but also from the coding of the train-control information. On one hand, Trix submits the train address together with the selected command in a first package. A second package, immediately sent after the first one, contains the parameters for the selected command. On the other hand, Märklin applies coded commands, generated by summing up the train address and the current parameter. To use one controller for both toy-trains, an active data mapping (*protocol translation*) between the controller and at least one of the train systems is indispensable.

As we can see, the definition of a semantics for individual data words is useful only in particular use-cases, which share a common "vocabulary". In general, we cannot declare a global protocol-semantics. Thus, an automated protocol translation can be established at most for either Trix or Märklin. For this reason, it is completely satisfying if an IFD provides a static specification of the protocol-syntax, which is use-case independent. The use-case specific information is interactively added during the interface synthesis process.

To support more than the simple routing of data, we omitted traditional models for data semantics. Instead, we developed the *IFD-Mapping* to handle the protocol information that is modeled within the IFDs, which supports even complex mapping algorithms. Therefore, it employs a mapping language defined by grammar that we present in Chapter 5.

The Level of Abstraction & Hardware/Software Interfaces

The way to model interfaces and the protocol translation are characteristic aspects of each interface synthesis approach. IFDs and TPDs provide a bit - and cycle-accurate behavior model. This level of abstraction appears to be rather deep for a high level synthesis approach. Nevertheless, this detailed information is required to generate the behavior model of the IFB. The presented "low level" representation ensures the highest degree of freedom in combination with a maximum degree of optimization for the IFB synthesis process.

As another advantage of the low level IFD representation, we are able to reduce hardware and software interfaces to a common denominator. We exploited this fact when we explained the HW/SW interfaces based on memory mapped I/O in Chapter 2. Only due to the low level of abstraction concerning memory mapped I/O and signal-based communication we are able to provide a common interface topology for both kinds of communication.

It would be the job of an additional *abstraction layer* to allow more complex data types, like integer values or customized std_logic vectors, which are often used in hardware description languages (HDL) as VHDL. This abstraction layer would also have to provide the translation between an abstract data type and its bit-level representation in both directions.

The idea to apply complex data types is also the first step towards the integration of function based communication, since the parameters in function calls are either standard or abstract data types. Thus, each abstract data type (ADT) inside a function call can be expressed in a bit-accurate representation similar to hardware, if we posses the functionality to switch between the different abstraction levels. Next to this, the execution semantics of function calls has to be considered. We do not go into detail here, as this aspect is ongoing research.

The presented interface synthesis currently allows us to design and to create IFBs for HW/HW interfaces and HW/SW interfaces restricted to memory mapped I/O.

Reuse & IP Based Design

To ensure reliability and efficiency in the design and implementation of communication systems, the reuse of well-tested components is an important technique next to the use of standardized interfaces and communication protocols. For this reason, the integration and flexible reuse of IPs became an integral part of system design in the IFS approach. The Interface Synthesis Design Flow offers an automated interface synthesis for incompatible IPs, which no longer requires the full understanding of IP internals. Therefore, the inner functionality of an IP can be kept as a black box model.

The proposed separation of protocol syntax and semantics goes in accordance with the presented IP integration rules, as we require only use-case independent properties of the IP interfaces. Similar to user-defined tasks, the automated integration demands only the IFD. The information on how to map the information, which is exchanged through the IP interfaces, is added during the IP integration process. In this way, our approach supports the adaptation of incompatible IPs as part of the design process. Thus, the IFS-Flow is well suited for the rapid design and evaluation of communication system prototypes using IP-based design.

4.1.2 Synthesis Phase

In contrast to the modeling process, the synthesis process is mostly automated. To cope with the complexity of creating an IFB, we have broken up the synthesis phase into the two design steps "generate IFB model" and "generate IFB target code". In the first step, we create a target language independent intermediate representation of the IFB; in the second step, we generate a target language dependent implementation of the IFB.

Generation of the IFB model

To perform an automated interface synthesis, we have to gather the input for our synthesis algorithm. Therefore, we select the necessary information from the System Architecture, which has been composed in the modeling phase. This information comprises a number of use-case-independent interface descriptions and the target platform description of the chip which is intended to implement the IFB.

Target Platform: After generating the IFB, the target platform becomes the execution platform for the IFB implementation. Therefore, the *target platform description* (TPD) provides the information, which is required to create the IFB, in such a way that it can be directly executed on the target device or processed further in the integration phase.

Our synthesis approach considers the number of available resources and clock networks. For software, resources can be specified as amount of memory; the hardware side uses gates or gate equivalents as a common unit. The amount of resources allows our synthesis to determine if the implemented IFB can be placed on the target device. Actually, a dedicated IFB synthesis for specific resources is out of our scope. The available clock frequency is essential to estimate if an IFB satisfies the specified timings of the adapted protocols. In Chapter 2 we already mentioned this point when we introduced the TPD. We have to take into account that even if we can synthesize an IFB, which satisfies the given timing, it can be invalidated by a lower level synthesis process within the integration phase. Therefore, the *timing analysis* is an essential, but no satisfying criterion, to create an executable IFB.

Component Interfaces: Apart from the target platform description, we require an interface description (IFD) of each component interface which we are going to interconnect by the current IFB. As defined in our communication framework, an IFD is use-case-independent and thus, a changeless property of each component. It comprises the information about three aspects: physical structure, electrical properties, and communication protocol.

The communication protocol given inside an IFD describes the task's interface behavior from the viewpoint of the task. As defined in Section 2.1.4, we apply a *waveform state machine* to represent waveform diagrams including branches in the protocol. This state machine specifies all I/O signals of the interface, including their directions, modeled from the tasks point of view.

We employ the IFDs to construct *communication stubs* inside the Interface Block. A stub establishes the transparent communication with a task conform to its protocol, i. e., the task must not recognize that it interacts with the IFB. Therefore the stub pretends with the task, to be a compliant communication partner. To do so, the created stub operates "complemented" with the component's *protocol state machine*. A stub extracts the information that is assembled into the redundancy of the communication protocol by the component and vise versa. In this way, we obtain an amount of incoming and outgoing data.

To create a stub, the waveform state machine given in the IFD has to be transformed into a *protocol state machine*. This transformation is automatically performed in a process called *protocol synthesis*. The resulting FSM implements the opposite behavior of the task's interface. In contrast to the waveform state machine, the protocol state machine comprises only those states which are relevant for the local execution of the protocol. Incoming control signals are transformed to state transition conditions, while outgoing control signals remain part of the Moore output function. Incoming and outgoing data signals are distinguished to be read or written by the IFB, respectively. A more detailed explanation of the protocol synthesis is given in Section 5.2.5.

Nevertheless, the protocol transformation is only the first step within the protocol synthesis. The second one handles the creation of the data translation, based on the IFD-Mapping. This scenario related data mapping is created interactively based on the selected IFDs.

Data Mapping: To model the *protocol transformation* inside the IFB we have defined the *IFD-Mapping*, which has to be provided by the designer during the synthesis process. The mapping file can be either selected from a database or interactively entered by a designer. An IFD-Mapping comprises a set of rules, called *Mapping Functions* (f_{Map}), which describe the mapping of incoming to outgoing data. A single Mapping Function is of the form:

$$\textit{Mapping Function}: \textit{dataOut} <= f_{Map}(\textit{dataIn}_1, \ldots, \textit{dataIn}_k)$$

As usual for the I-P-O concept, outgoing data depends on incoming data, possibly even from multiple sources. Therefore, f_{Map} allows modeling of four *data processing operations*:

1) Assign constant values
2) Shuffle incoming data
3) Guarded data assignment & Boolean equations
4) Complex data processing including a data processing library → FSM

We support assigning constant values or an arbitrary mapping of incoming data bits to outgoing data bits (shuffle data). Further, our mapping allows guarded assignments by applying Boolean equations on incoming data bits. To also handle complex behavior, the designer can model FSMs to process incoming bits. We can encapsulate mapping functions in a library of reusable data processing functions to support the *reuse* of complex or frequently required data processing operations, like the evaluation of a CRC (Cyclic Redundancy Check) field. The functions of this library can then be accessed from other IDF-Mappings.

Based on the IFD-Mapping, the protocol synthesis generates a set of state machines, which process the data provided by the stubs, to perform the protocol translation. However, the IFD-Mapping is not only utilized to create the protocol transformation, it is also applied to customize the IFB internal control and data memory as well as optimizing the boxing and unpacking of data portions inside the stubs. Similar to the stubs and the data processing, the IFB control is implemented as interacting FSMs, which coordinate the IFB operations.

Since the definition of the IFD-Mapping is an essential part of the synthesis process, we developed an *automated* design flow. If we act on the assumption that we select a predefined IFD-Mapping from a library as input of the synthesis, we obtain a completely *automatic* interface synthesis process. The result of the first synthesis design step is a target language independent IFB model, which is still qualified to be implemented in software or hardware.

Code Generation

After creating the IFB model, we attach standard code generation techniques to create an implementation of the IFB. The result of the code generation varies depending on the selected target architecture. In software, the IFB could be compiled into a communication library, for example, a *dll* (Dynamic Link Library, Microsoft Windows). In the case of a hardware design, we have to create the IFB implementation on that abstraction layer which is demanded by the subsequent low-level synthesis process.

Until now, we have developed the IFB code generation for the hardware description language VHDL. The code generator is based on the code generation technique *frame processing*, which is in particular adequate for generating the IFB, due to the modular structure of the IFB and VHDL. With the help of the IFS-Editor we are able to create the VHDL code of particular IFBs, the "configware" [2] for single chips, and the code of complete systems, which can be employed for a simulation of the system's communication behavior.

The synthesis phase is followed by a design step named "Integrate IFB target code". This process includes all necessary actions to create an executable system implementation, using the created IFB code as a building block.

4.1.3 Integration Phase

The integration phase is not primary a part of the Interface Synthesis Design Flow itself. It rather concludes the flow and offers the design step to create the executable parts for the target platform. This could mean integrating the generated IFB code into an existent design or to create a new design including the IFB code. Therefore, it is essential to possess the implementation of the other communication components in a similar format as the generated IFB code. In this design phase, we resort to available tools, as compilers (software) and dedicated synthesis tools (hardware), e. g., *Xilinx ISE* in the case of Xilinx FPGAs.

[2] *Configware* is a software used to set up a (re-)configurable device. The term was formed by R. Hartenstein.

4.2 IFS Modeling Concept

To constitute a *formal basis* for the composition of System Architecture models, we defined the *IFS Modeling Concept* [1, 36]. Figure 4.2 illustrates this concept in the form of a circular chart that consists of three *sectors* and three *levels* (rings). The outer level represents the *domain specific modeling language* that is considered as standard for the particular sector. The *models*, which are expressed with the help of the according modeling language, are depicted in the middle level. Each model itself is again used as *meta-model* for the construction of the *model instances*, presented in the inner level. The modeling layer (middle level) is continuous, as identical information is specified in each sector, which permits a *transformation* between the different models. The same is true for the instance layer (inner level) as a special development of the models. In our modeling concept, we deal with the three sectors: XML, Java, and UML.

The *IFS-Format* has been developed to provide a *formal representation* of the IFS System Architecture. We modeled the IFS-Format as an XML-scheme. Therefore, the dedicated sector is marked with "XML". An instance of the IFS-Format is called *IFS-Instance* and represents a valid specification of the System Architecture.

To automate the interface synthesis process and to evaluate the developed concepts, we implemented an EDA tool, the IFS-Editor, in accordance to the IFS-Format. The sector named "Java" handles this tool written in Java, which facilitates editing and visualizing IFS-Instances. We implemented the IFS-Editor as M-V-C (Model - View - Controller) architecture, which means a separation of data structure, graphical user interface (GUI), and the applied algorithms. The object-oriented data structure of the data model (*IFS-Data-Structure*) implements precisely the defined sub-schemes of the IFS-Format (blue arrow). Thus, IFS-Instances can be loaded and stored (blue double-headed arrow). The representation of an IFS-Instance in Java is called *IFS-Object*. The automated interface synthesis is performed on these objects.

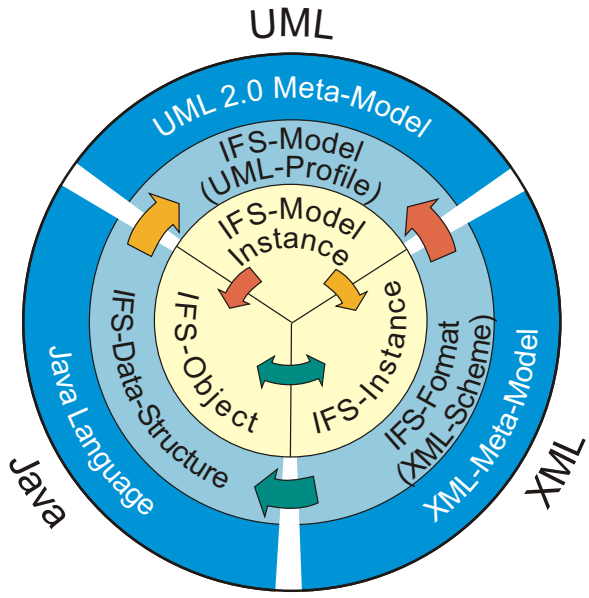


Figure 4.2: IFS *Modeling Concept*

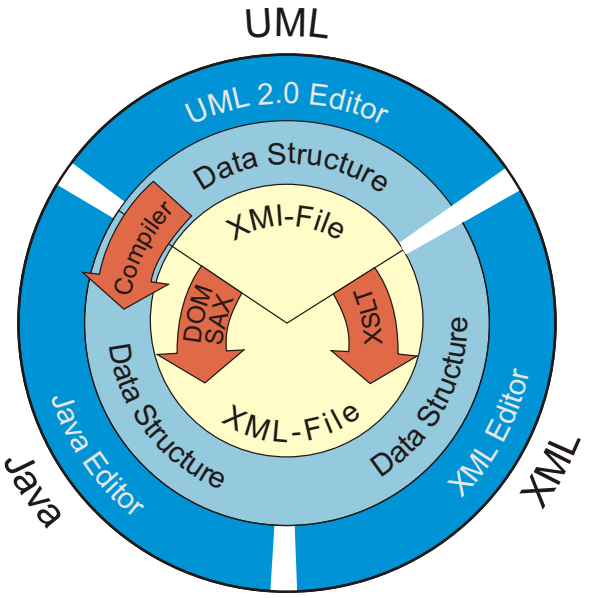


Figure 4.3: IFS model transformation

To provide an abstract and intuitive modeling for the designer, the IFS-Format has been transferred into UML. As depicted in Figure 4.2, we derived a UML2.0 profile from the IFS-Format (red arrow), namely the *IFS-Model*, which allows the model-based design of the System Architecture. Alternatively, we could have developed the IFS-Model from the IFS-Data-Structure with the help of reengineering techniques. An instance of the IFS-Model is named *IFS-Model-Instance*. The transformation of models and instances is guaranteed due to the equivalence of the information that is expressed on the modeling layer.

In the next section, we explain the IFS-Format in more detail and present the dependency of the XML and the Java sector. Afterwards, we discuss the trade-off between different solutions for the model transformation of the UML sector into the XML and the Java sector on the model and the instance level as presented in Figure 4.3.

4.2.1 The Interface Synthesis Format

We have deployed the *Interface Synthesis Format* (*IFS-Format*) as *meta-model* for the IFS System Architecture. On one hand, it defines our data exchange format; on the other hand, it has been used as a template for the implementation of the data structure in the IFS-Editor. Due to this consistence between exchange format and internal data structure, data can be easily imported and exported, as we will see in the next section.

As part of the IPQ Format, which has been introduced previously, the IFS-Format fulfills the role of describing interfaces of IPs within the IP characterization. In this way, our approach allows the seamless integration of "IPQ-IPs" into our design flow and therewith, the automated adaptation of these IPs.

Similar to IPQ, the *IFS-Format* has been modeled in the form of an XML scheme [246]. A detailed description of the IFS-Format is given in [29]. The XML scheme is composed of hierarchical sub-schemes as presented in Figure 4.4, which depicts the sub-scheme of the "task level". In accordance with the presented communication framework in Section 2.1.4 the task consists of an *identification*, a number of physical interface descriptions (*Interface*) and protocol descriptions (*Protocol*), which are mapped by protocol maps (*ProtocolMap*). Interface, Protocol, and ProtocolMap again are XML sub-schemes. Additionally, a *Version* attribute has been added to each sub-scheme for a version management of the created XML instances. *VHDLFile* is a task specific and optional tag that references the file that represents the implementation of the current task.

In its role as meta-model, the XML scheme defines precisely the structure of the creatable XML instances, which represent valid samples of the IFS System Architecture. Thereby, an XML instance may describe a particular component up to a complete system, including several architecture and communication components. In particular, the IFD and the TPD are the two instances required for the interface synthesis. Therefore, the IFD of an IPQ-IP is automatically derived from the IP's meta-data. A collection of IFDs in the form of IFS-Instances, for example, in a database, can be used as design library in the modeling phase of our Interface Synthesis Design Flow.

Since creating XML instances by hand is very error-prone and quite labor intensive, even with the help of XML editors, the IFS-Editor provides a graphical design front-end for modeling also complex communication infrastructures. The tool allows the graphical modeling of components in combination with specific masks for the component's parameters.

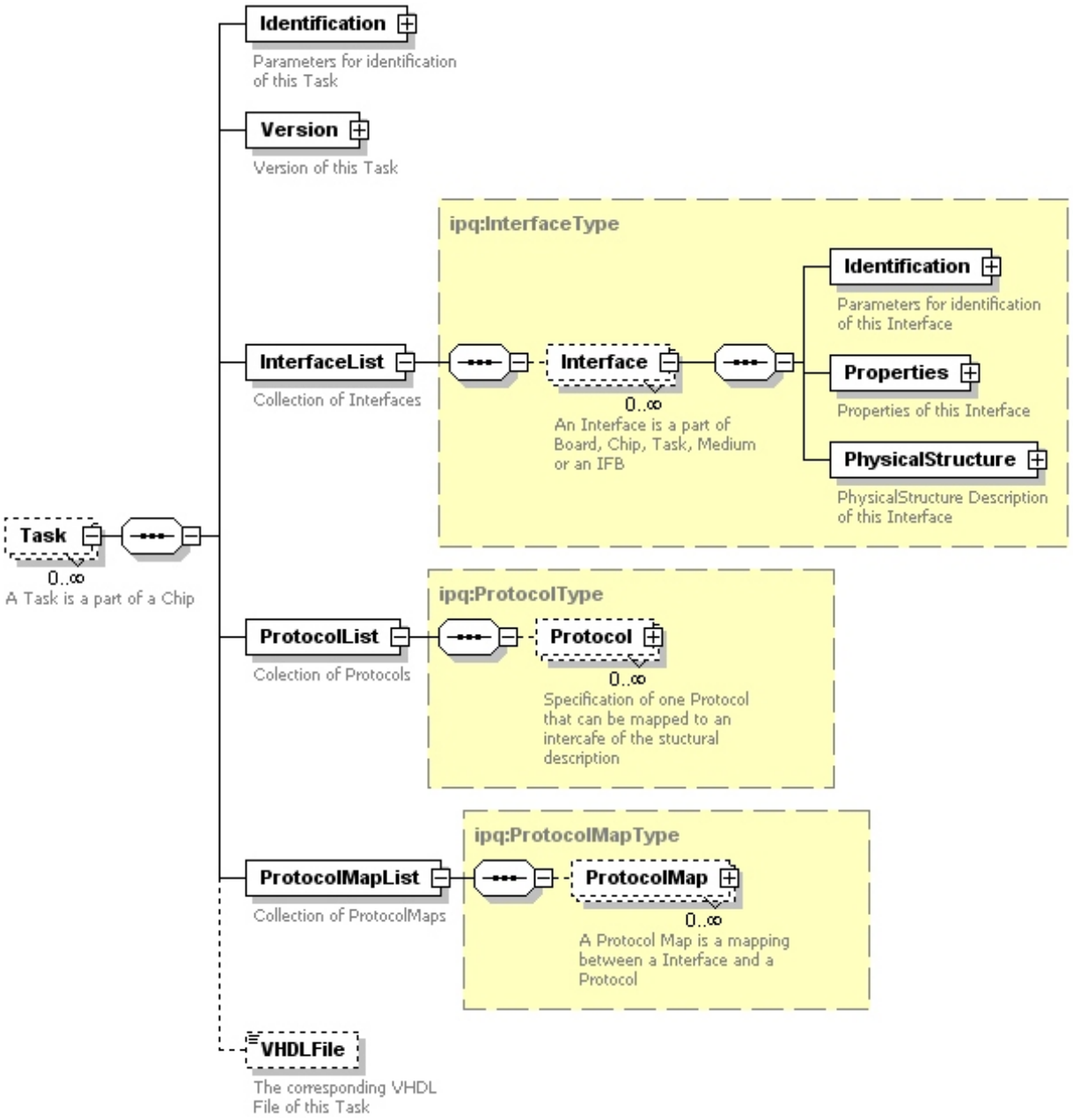


Figure 4.4: Excerpt of the XML based IFS-Format (XML-Scheme)

Each XML scheme carries information which is usually used by XML editors to validate the correctness of XML instances, or to facilitate the design process of the XML scheme or the XML instances. We employ this data inside the IFS-Editor to check the design entry and enhance the design process. Therefore, we acquire the following parameters from the XML scheme of the IFS-Format:

- Tag
- Default Value
- Unit
- Type
- Comment
- Enumeration

Example-Code 4.1: XML-Scheme – Tag, Type, Default Value, Comment, and Unit

```
<xs:element name="VLogicOne" default="3.3">
   <xs:annotation>
      <xs:documentation>Voltage related to logic one.</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:simpleContent>
         <xs:extension base="ipq:NR3..1.12ES..2">
            <xs:attribute name="unit" type="xs:string" fixed="V"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
</xs:element>
```

As presented in the XML Example-Code 4.1, which is an excerpt of the IFS-Format, we see the definition of the physical voltage level as one attribute of each physical interface. It specifies the value of the physical voltage that is related to binary value "logic one". Among others, we evaluate this attribute to check the connectivity of two interfaces.

The given example provides information about the XML element name (Tag = VLogicOne), its data type (Type = ipq:NR3..1.12ES..2), the default value (Default Value = 3.3), a comment (Documentation = Voltage related to logic one.), and the unit of the value (Unit = V).

Example-Code 4.2: XML-Scheme – Tag, Type, Default Value, Comment, Enumeration

```
<xs:element default="Input" name="Direction">
   <xs:annotation>
      <xs:documentation>Direction of ProtocolPin.</xs:documentation>
   </xs:annotation>
   <xs:simpleType>
      <xs:restriction base="ipq:M..32">
         <xs:enumeration value="Input"></xs:enumeration>
         <xs:enumeration value="Output"></xs:enumeration>
         <xs:enumeration value="Bidirectional"></xs:enumeration>
         <xs:enumeration value="Special(GND,VCC)"></xs:enumeration>
         <xs:enumeration value="Other"></xs:enumeration>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

XML Example-Code 4.2 depicts the direction attribute of the ProtocolPins. The values Tag, Type, and Comment are defined similar to Example-Code 4.1. In addition, an enumeration specifies a set of predefined values (Enumeration = Input, Output ...), which can be selected in the IFS-Editor instead of typing in a value. The attribute Default Value defines the initial selection. Therefore, this attribute has to be a member of the enumeration.

Based on VSIA, IPQ defines a number of standard data types, which are employed in the IFS-Editor to perform a consistency check of the inserted data. Within the IFS-Editor, tags and units are visualized in the interactive masks while comments are displayed as tool-tip. Each text field inside the masks, where the designer can enter a value, is initialized with its default value to avoid invalid or empty input data.

The combination of IFS-Format and IFS-Editor is a synergy of the two sectors XML and Java on the model layer. The interaction on the instance level is given in the next section.

4.2.2 Interaction of XML and Java

IFS-Instances are used as data exchange format for the IFS. Therefore, it is necessary to import (load) and to export (store) the XML instances using the IFS-Editor. The close interaction of the Java tool and XML on the instance level is visualized in Figure 4.5. To simplify the loading and storing of IFS-Instances we developed a data structure that directly works on the XML instances.

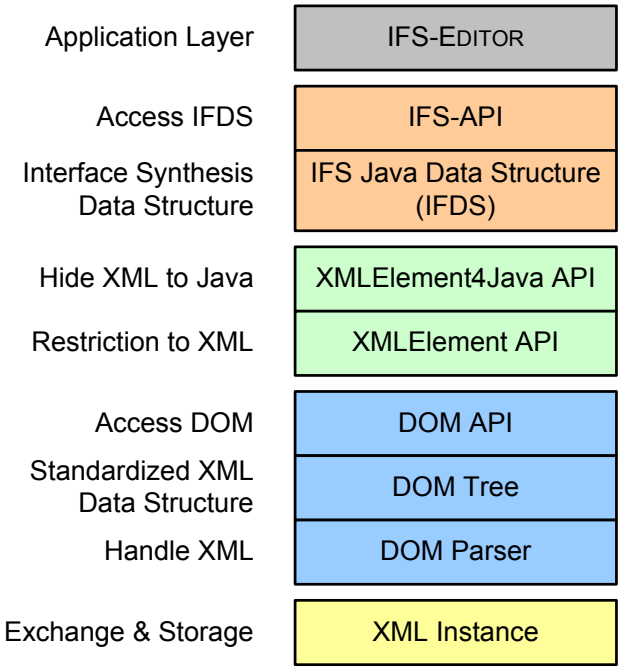


Figure 4.5: Interaction of XML and Java on the instance level

We attach the standardized *Document Object Model* (DOM) to cope with XML instances (read & write XML). The data structure of DOM is a rather complex tree of uniform element nodes without a semantics for the particular node.

Based on the DOM API we apply the so-called *XMLElement*-API, which restricts the general DOM view to an XML-wise programming model[3]. With the help of this API, the DOM tree nodes can be traversed as XML elements.

On top of the XMLElement-API, we put the *XMLElement4Java*-API to hide the usage of XML and DOM completely from the upper layers. The defined programming interface allows us to establish a truly object-oriented realization of the IFS-Data-Structure. The center of the XMLElement4Java-API are the two data types *X4J* and *X4JVector* that define methods to get and set data without dealing any knowledge of the lower layer implementation.

With the help of these two data types, the IFS-Objects of the IFS-Data-Structure (IFDS) have been implemented. The IFS-Editor and the synthesis algorithms work on these objects using the visible object methods.

[3]This API has been developed as part of the IPQ project

The method `setDirection("Input")` of a ProtocolPin, for example, calls the `setValue()` method of the X4J attribute *direction* inside the IFS-Object ProtocolPin. In a first step, the `setValue()` method checks the passed value `"Input"` against the type and the enumeration, which are defined in the XML scheme for this attribute. If this check passes successfully, the function call is delegated to the XMLElement-API and finally to the DOM-API. Here, DOM updates the dedicated node in its tree to store the new value.

To save an IFS-Instance we can simply export the DOM model as XML instance. Loading an XML instance is more complicated, as we have to construct the IFDs layer additionally after importing the DOM model.

4.2.3 UML 2.0 and its Interaction with XML and Java

In a second step we extended our modeling concept to the Unified Modeling Language (UML) due to is high familiarity with many designers and the possibility of defining specific profiles. There exists many popular UML CASE-tools (Computer-Aided-Software-Engineering tools) that we can employ to model the System Architecture. With the help of UML, we provide an intuitive and abstract way to model our communication infrastructures.

Because of the expressiveness of UML 2.0, it is possible to model the complete information covered by the IFS-Format. Nevertheless, the modeling and subsequent transformation of the IFS-Models into the IFS-Format is only feasible if every part of the IFS-Model owns a precise semantics relating to the IFS-Format. For this reason, we defined a UML 2.0 profile of the IFS-Format that comprises a specific semantics and certain syntax restrictions.

The UML 2.0 Profile

We analyzed the IFS-Format and UML 2.0 to define an intuitive *UML 2.0 profile*. For that purpose, we selected a set of UML diagrams and defined a precise semantics. The UML 2.0 profile consists of *component diagrams*, *class diagrams*, and *protocol state machines*.

We apply component diagrams to describe the structural aspects and connections of the System Architecture in the form of a component hierarchy. Class diagrams specify the static parameters of the System Architecture; for example, the electrical properties of the interfaces. These parameters are modeled in the form of class attributes and data types, which are linked to the component instances inside the component diagrams.

UML ports provide the connections between the components. Protocol state machines that are related to the UML ports are utilized to describe the behavioral aspect. As usual for extensions of the meta-model, all added classes are defined as stereotypes.

The UML 2.0 profile defines our *IFS-Model* which is textual and syntactically restricted to the IFS-Format. The IFS-Model samples as meta-model for the creation of the *IFS-Model-Instances* as depicted in Figure 4.2. To show the proof of concept, the hitherto developed UML 2.0 model is restricted to the required bit abstraction level. In Section 5.1, we discuss the UML 2.0 profile in detail.

In order to integrate the UML 2.0 profile into the Interface Synthesis Design Flow and thus make it a useable modeling front-end, the transformation of the IFS-Model into either the IFS-Format or the IFS-Data-Structure has to be achieved.

Concepts for the Model Transformation

As presented in Figure 4.3, there exists two alternatives for the transformation of the IFS-Model. The first way is to export the UML design as XMI (XML Meta-Data Interchange) and use this file for the model transformation. There are well-established techniques to convert XMI files like XSLT, DOM and SAX. All these approaches require the model to be exported in one of the available XMI formats. Apart from the restriction to one XMI version, a lot of efficiency is lost due to the export and adjacent transformation. Furthermore, the specification of complex XSLT scripts is very labor-intensive and error-prone.

Alternatively, we can translate an IFS-Model-Instance based on the data structure of the used UML editor (red arrow in Figure 4.3). This is much faster but requires knowledge about and access to the internal data structure of the CASE-tool, which is usually not available for the commercial tools.

Nevertheless, there exists one CASE-tool, namely *Fujaba* (From UML to Java and back again [69]), which has been developed at the University of Paderborn. This tool fulfills our needs concerning UML and grants us the required access to the internal data structure. Therefore, we decided on the direct translation of the data structure based on Fujaba for the model transformation.

Implementation of the Model Transformation

The model transformation is divided into the coupling of the IFS-Editor and Fujaba, and the model translation itself. To transform the Fujaba data structure we implemented a translator using standard compiler techniques. Input of the translation is the Fujaba data structure of the created IFS-Model-Instance. A parser traverses this tree structure with a single recursive descend and creates the IFS-Data-Structure with the help of pattern matching.

The definition and the implementation of the UML 2.0 profile and the model transformation was presented in [1, 36]. Furthermore, this work demonstrates how to model a communication system based on our UML 2.0 profile in Fujaba by an extensive example.

The next section introduces our interface adapter module, the Interface Block.

4.3 Concepts of the Interface Block

The Interface Synthesis Design Flow has been developed to generate an interface adapter module, called *Interface Block* (*IFB*). In this section, we introduce the *IFB Macro-Structure*, which depicts the structural composition of the *IFB model* based on communicating FSMs. With the help of the IFB Macro-Structure, we explain the functionality of individual IFB components and discuss the realization of the Input - Processing - Output concept.

Furthermore, we present the modifications and extensions towards the reconfigurable IFB and discuss the IFB as a fail-safe interface adapter. Afterwards, we demonstrate the relationship to the ISO/OSI model and introduce an environment for real-time communication.

On one hand, we apply the IFB Macro-Structure as a specification for the IFB model synthesis, presented in Chapter 5. On the other hand, we refine the IFB Macro-Structure to the IFB Hardware Template for the IFB realization in hardware as given in Chapter 6.

4.3.1 IFB Macro-Structure

To explain the construction of the IFB Macro-Structure, we begin with the elementary case: the protocol translation for one sender and one receiver. Within the context of my Diploma Thesis [7, 3], I developed the basic concepts of the IFB, which worked as a peer-to-peer adapter module supporting simplex communication [2]. Figure 4.6 illustrates the hierarchical structural of this basic IFB in a tree-like representation.

In accordance to the Input - Processing - Output concept, the IFB Macro-Structure is divided into an *incoming protocol handler* (PH_{IN}), an *outgoing protocol handler* (PH_{OUT}), and the *sequence handler* (*SH*). Each handler comprises a set of *operation modes*. The PH-Modes accomplish the role of the stubs while the SH-Modes perform the protocol translation. It is essential to *separate* PH and SH to avoid the construction of the *product automata*. Furthermore, the structuring into independent modes is crucial for the *reconfigurable* IFB.

A *control unit* (*CU*) performs the internal supervision of the Interface Block. Therefore, the CU provides *control signals* to the PH-Modes and the SH-Modes and evaluates their *status signals* (vertical communication). The processed data signals in combination with the required handshake signals are communicated in the horizontal direction. This partitioning of signals is further on referred to as *orthogonal communication structure* [14, 13, 12].

To transfer data through the IFB, the PH_{IN} reads incoming data, which is then processed by the related SH-Mode, and finally sent to the receiver with the help of the PH_{OUT}. For these purposes, the PH-Modes, the SH-Modes, and the CU are implemented as *interacting state machines*. As mentioned before, it is an advantage of the I - P - O concept to avoid the creation of complex product automata. We reproduce this effect by subdividing the IFB into the PH, the SH, and the CU. However, this division accounts for the introduction of additional states into the FSMs, which are required for the synchronization and the *fully interlocked handshaking* that guarantees a lossless data exchange inside the IFB.

Due to the modular structure, we could easily extend the basic IFB and integrate further functionality into the interacting IFB parts. The final structure of the IFB Macro-Structure is illustrated in Figure 4.7. This model considers the interconnection of multiple communication partners including duplex traffic. Therefore, the PH_{IN} and the PH_{OUT} have been merged into only one PH, which comprises PH-Modes that can receive and send data.

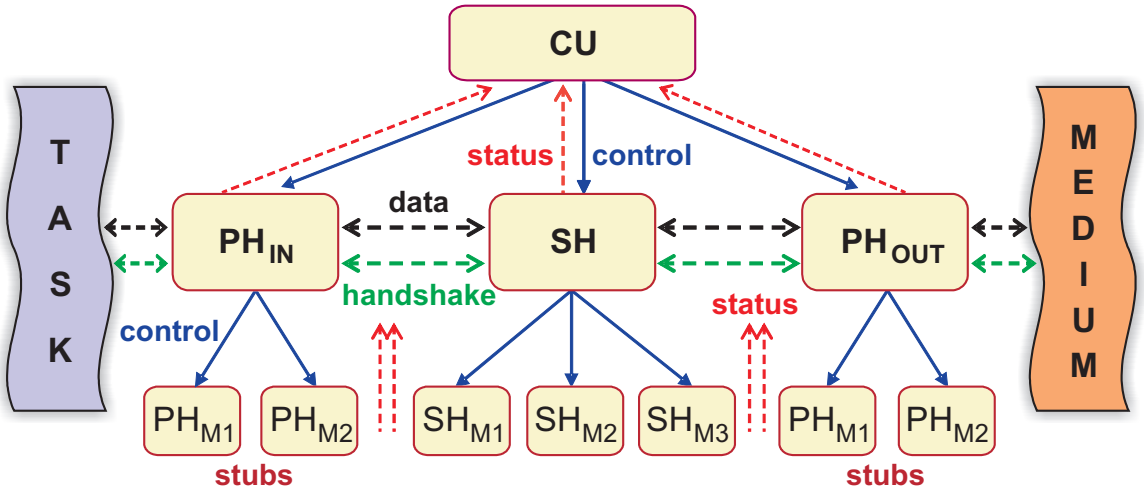


Figure 4.6: The basic IFB Macro-Structure as hierarchical tree representation

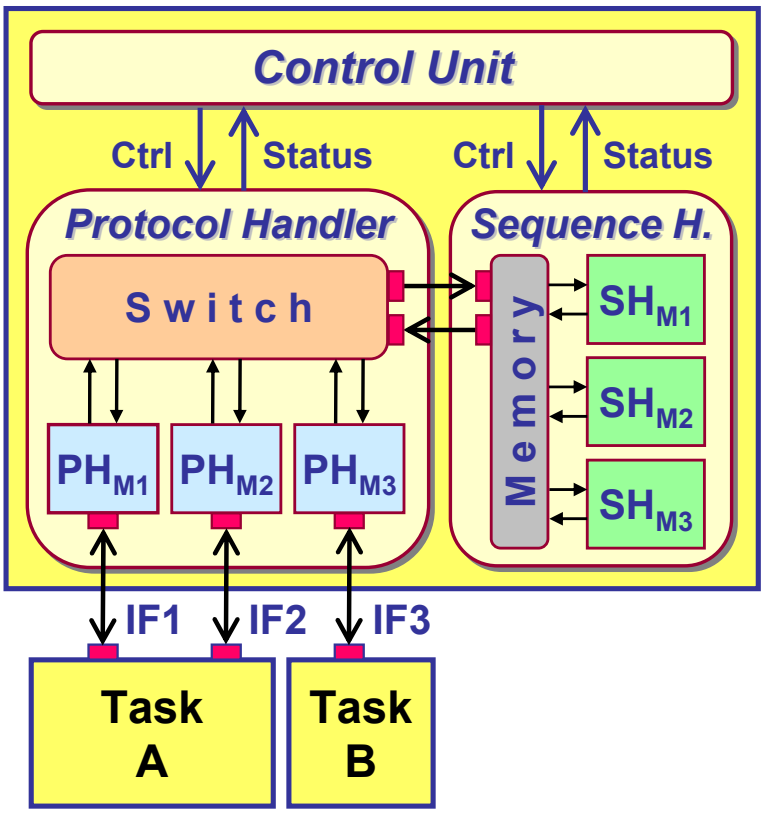


Figure 4.7: The IFB Macro-Structure

Protocol Handler

To establish the external communication with the tasks, the IFB provides a *stub* for each connected (component) interface. These stubs are implemented by the so-called *PH-Modes* inside the Protocol Handler (PH). To perform a transparent protocol adaptation, each PH-Mode implements the *complementary protocol* of the related interface in the form of a finite state machine. Once started, a PH-Mode executes this state machine continuously, which is synthesized based on the associated IFD and extended with additional states to interact with the control unit and the sequence handler. To create the state machine of a stub, the waveform state machine given in the IFD is transformed into a *protocol state machine*. The detailed *protocol synthesis* is handled in Section 5.2.5.

Usually the interconnected interfaces are independent of each other; therefore, the PH-Modes have to work in parallel to serve all interfaces continuously. It is easy to implement this aspect in hardware, as parallelism is a natural paradigm of hardware. One solution for software could be the realization as "semi-parallel" threads.

A PH-Mode *extracts* the *relevant information* from the incoming data, which is mapped in the IFD-Mapping, and provides it to the SH. Therefore, the PH-Mode stores the extracted data bits in an IFB internal memory inside the SH. Similar to the extraction of incoming information, a PH-Mode *merges* the *processed values* into the outgoing data, which is transmitted to the task. To sequentialize the memory access of the parallel PH-Modes, we added the *switch* inside the PH, which acts as memory interface to the internal memory.

Sequence Handler

The Sequence Handler (SH) does not have to communicate with external interfaces. It *stores* the IFB *internal data* and performs the *data transformation* between the PH-Modes according to the mapping functions which are specified in the IFD-Mapping.

There exists several concepts to implement the internal memory. We decided on a dedicated memory that provides the space for exactly one instance of each incoming and outgoing data word, instead of utilizing data queues. This decision is based on the IFD-Mapping and the resulting protocol translation that considers only the current instances of incoming and outgoing data packets. In hardware, the internal memory can be provided by a default memory, like a Block RAM, or created as dedicated memory within the synthesis process.

The IFD-Mapping specifies the scenario-based *protocol translation* in the form of *mapping functions*. Each mapping function is implemented by an SH-Mode that performs the data processing based on the four mentioned data mapping operations (see Section 4.1.2, Page 64) as there are: assign constant values and shuffle incoming data in combination with the guarded assignment of data and the complex data processing based on FSMs. In this way, the SH-Modes comprehend a multifaceted functionality, starting from a simple forwarding of data up to complex modification algorithms. Furthermore, the FSMs allow us to store passed data intermediately as this *history information* can be coded into the available states. Nevertheless, this way to store the passed data is very resource-inefficient.

To activate an SH-Mode, the input data given in the *mapping function* has to be completely received and stored inside the local memory. When the data processing has finished, the SH-Mode stores the computed output value in the internal memory and informs the CU to grant permission for this data packet to be sent by the dedicated PH-Mode.

The four data processing operations, in combination with the ability to store data, span a wide area of possible applications. To overcome the limitation of FSMs we afford to define *variables*. This extends our FSMs to *linear bounded automata*, which allow us, for example, to model a complete processor inside an SH-Mode. Admittedly, this is not the actual intention of an SH-Mode – thus, we restrict ourselves to process incoming into outgoing data.

In the following example, we demonstrate a reasonable application for an SH-Mode. Furthermore, we discuss the according code of the mapping functions. The example descends from the cryptographic domain and handles the *block cipher en-* and *decryption* inside the IFB. Therefore we expect a sender which offers its data as plaintext, and a receiver that requires this data to be coded as ciphertext, or vise versa.

Example: Block ciphers map blocks of a fixed size to other blocks of the same size. Both, plaintext and ciphertext are element of the set Σ^n of words with the block-size $n \in \mathbb{N}$ over the alphabet Σ. The applied plaintext-space (P) and ciphertext-space (C) is $P = C = \Sigma^n$. The key-space is the set $S(\Sigma^n)$ of all permutations of Σ^n, which contains $(|\Sigma|^n)!$ elements. For one particular key $\pi \in S(\Sigma^n)$ we define the encoding-function $E_\pi : \Sigma^n \to \Sigma^n, v \mapsto \pi(v)$ and the according decoding-function $D_\pi : \Sigma^n \to \Sigma^n, v \mapsto \pi^{-1}(v)$.

An interesting example of the block ciphers is the *permutation code*. It applies keys that allow only such permutations that result from exchanging the position of the characters. If $\Sigma = \{0, 1\}$ we call this a *bit-permutation*. In this case, the key-space belongs to the group of permutations S_n. For $\pi \in S_n$ we define: $E_\pi : \Sigma^n \to \Sigma^n, (v_1, ..., v_n) \mapsto \pi(v_{\pi(1)}, ..., v_{\pi(n)})$ and $D_\pi : \Sigma^n \to \Sigma^n, (v_1, ..., v_n) \mapsto (v_{\pi^{-1}(1)}, ..., v_{\pi^{-1}(n)})$. A simple cipher that utilizes the bit-permutation is the *electronic codebook mode*.

Electronic Codebook Mode (ECB-Mode) The electronic codebook (ECB) mode splits the messages into blocks of a given block-size and encrypts each of them separately as depicted in Figure 4.8. The disadvantage of this method is that identical plaintext blocks are encrypted to identical ciphertext blocks; therefore, it does not hide data patterns. The term "Mode" in "ECB-Mode" should not be mistaken with that one in "SH-Mode" or "PH-Mode".

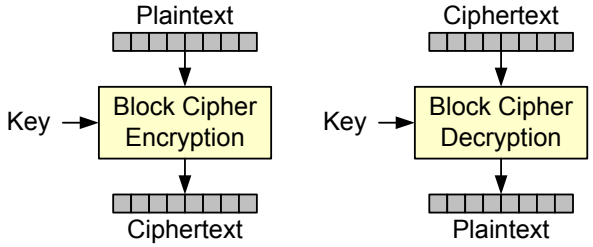


Figure 4.8: Electronic codebook (ECB) mode encryption and decryption

The key for a simple right-shift would look like: $\pi = \begin{pmatrix} 1 & 2 & 3 & ... & n \\ n & 1 & 2 & ... & n-1 \end{pmatrix}$.

An SH-Mode can easily perform the encoding and decoding conform to the ECB-Mode. The related IFD-Mapping is quite simple, since it comprises only one operation for the relocation of data bits: $P_{out} <= P_{in}[n] \ + \ P_{in}[1 : n-1]$; The precise syntax and semantics of the IFD-Mapping language are presented in Section 5.2.6.

Cipher-Block Chaining Mode (CBC-Mode) In the cipher-block chaining (CBC) mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way each ciphertext block is dependent on all plaintext blocks up to that point.

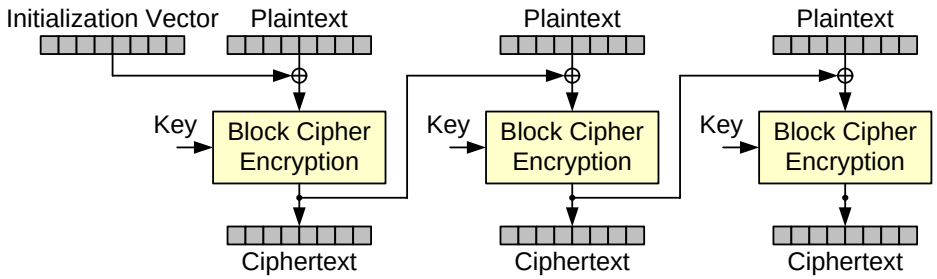


Figure 4.9: Chipher block chaining (CBC) mode encryption

An SH-Mode can also perform as CBC-Mode. Therefore, we could model an FSM that covers all possible words (in the case of n-bit words, it would be 2^n states) to remember the last computed value for the XOR-operation with the current value. IT obviously makes no sense to realize the CBC-Mode for a greater bit-width in this way. Therefore, the IFD-Mapping language provides *variables*. In the case of a software IFB, they become "variables" in the classical meaning; in hardware we implement them as registers, which belong to the related SH-Mode. These variables represent the memory of the linear bounded automata and allow us to also implement complex behaviors resource-efficiently.

The IFD-Mapping for the CBC-Mode looks like:

1. $bit\,[n]\ prevVal$; // Declaration of the variable
2. $P_{out} <= P_{in}[n]\ xor\ prevVal[n]\ +\ P_{in}[1:n-1]\ xor\ prevVal[1:n-1]$;
3. $prevVal := P_{in}$; // Store value of P_{in} in variable after computation of P_{out}

In general, the SH-Modes can process data in parallel similar to the PH-Modes. However, three conditions have to be fulfilled to perform a parallel protocol transformation. First of all, the platform has to support the parallel execution of multiple SH-Modes. This aspect is similar to the PH-Modes and has already been discussed before.

The second aspect is also of technical nature. If multiple SH-Modes shall work in parallel, they have to be able to read from and write into the IFB internal memory in parallel. From our point of view, this requirement can only be met in hardware with the help of a synthesized register file with a direct connection to the SH-Modes.

Thirdly, in contrast to the PH-Modes, the SH-Modes are not completely independent of each other. When different mapping functions demand identical pieces of input data, we obtain SH-Modes that consume the same bits from the internal memory. Depending on the complexity of the mapping functions, the computation time of these SH-Modes can vary. To make sure that each SH-Mode reads out each piece of data exactly once, the interdependent SH-Modes have to be synchronized. As all PH- and SH-Modes have to be modular and self-containing to be reconfigurable, the synchronization of the SH-Modes has become a task of the CU. Thus, the SH-Modes themselves are not aware of existing dependencies. All three aspects have to be satisfied truly to process data in parallel.

Control Unit

The synchronization of the SH-Modes is only one aspect of the CU functionality. Altogether, the CU provides the *IFB internal control* and is responsible for the *IFB reconfiguration*.

If there is more than one receiving or sending interface connected to an IFB (*Multi-Task IFB*) or if complex mapping functions have to be executed, an active control of the PH- and the SH-Modes is required to guarantee the *causality* of the *I - P - O* scheme. Therefore, the CU is responsible for the overall coordination of the *data flow* inside the IFB. It aims to optimize this data flow in order to minimize the latency produced by the IFB.

The *latency* is a critical aspect for the reaction time of communicating nodes. It represents the *delay* of the utilized *communication link*. As a transparent part of this link, the latency produced by an interface adapter adds on the latency of the communication link. Therefore, it is a major goal to minimize the latency of the IFB in order to optimize the latency of the communication link, and thereby, to maximize the possible data transfer rate [5].

For this reason, the PH-Modes and SH-Modes work in an *interleaved* manner as a *pipeline* [26]. This is possible since all modes are self-contained and decoupled by the internal memory located inside the SH. As there exists exactly one dedicated memory cell for each incoming and outgoing data word in this memory, the IFB pipeline structure (depicted in Figure 4.10) is comparable to a classical three-staged execution pipeline. The pipelined data processing is another crucial reason for the decision to utilize a dedicated memory instead of queues.

To induce a *maximum workload* for an IFB, we assume tasks that *continuously* try to *send* and *receive* data wherever possible. The *I - P - O pipeline* executes this *sequence of actions*:

1. If the *condition to receive data* is met, the CU admits the dedicated PH-Mode to receive data. Therefore, the CU grants the write-access to the memory via the PH-Switch.
2. The PH-Mode reads the incoming data and transfers it into the incoming memory.
3. If the *condition for processing data* is met, the dedicated SH-Mode is activated.
4. The PH-Mode processes the stored data according to its mapping function and writes the result into the outgoing memory.
5. If the *condition to send data* is met, the CU admits the dedicated PH-Mode to transmit data. Therefore, the CU grants the read-access to the memory via the PH-Switch.
6. The PH-Mode reads the processed data from the outgoing memory and merges it into the outgoing data.

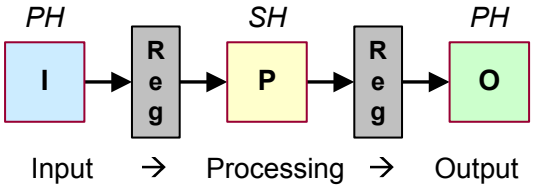


Figure 4.10: Pipelined *Input - Processing - Output* execution

Causality condition to *receive data*

- The PH-Mode reaches a state in its protocol where it has to read mapped data.
- The previous instance of the current data word has been processed by all relevant SH-Modes.

Causality condition to *process data*

- The input data of an SH-Mode has been completely received and stored inside the incoming memory.
- The previously processed data word has been sent by the dedicated PH-Mode.

Causality condition to *send data*

- The PH-Mode reaches a state in its protocol where it has to write mapped data.
- The required data word has been completely processed by the dedicated SH-Mode.

To check the presented *causality conditions*, grant access to the PH-Switch, and start the SH-Modes, the CU performs an active *resource management* including the

- *arbitration* of *memory bus* inside the PH-Switch and the
- *memory management* of the *internal memory*.

As we will see in Chapter 6, we ensure the causality of the I - P - O pipeline in hardware with the help of a *scoreboard*, which is responsible for the *memory management*. The scoreboard notifies incoming and leaving data of both memories and utilizes this information to start the data processing (SH-Modes) and evaluate the three mentioned conditions.

In case of a *Multi-Task IFB*, we also have to *schedule* the memory-access of the PH-Modes via the PH-Switch. This is essential, as the memory interface allows a maximum number of one writing and one reading PH-Mode at the same time. In conformance with the internal memory, the PH-Switch is separated into two parts: the incoming and the outgoing memory-bus interface. The CU provides a *controller* for each memory interface which manages the memory access of the PH-Modes. Thus, the *scheduling* of the *Input* and the *Output* of the I - P - O scheme is accomplished as *arbitration* of the memory-bus interface.

Depending on the system requirements, we can implement different *scheduling strategies* for the PH-Modes; for example, *fair* schedulers, which treat all interfaces equally or *priority-based* schedulers that prefer special interfaces. In order to make an estimation about the *schedulability* of given scenarios, we have to apply the concepts of *scheduling theory*. However, the *schedulability analysis* requires a more detailed model of the IFB than we have presented up until now. In Section 6.3 we demonstrate our solution for the hardware IFB.

Up until now, we explained the functionality of the "static" IFB, which can be employed for the development of non-changing environments, e. g. in the ASIC design. Every time we cope with dynamic environments, which allow the interchanging of tasks at runtime, we create a *runtime reconfigurable IFB* (*RTR-IFB*). Within an RTR-IFB, the second task of the CU is to handle the IFB reconfiguration. This requires a *Reconfiguration Controller*, which interacts with an external *Reconfiguration Control Unit* (*RCU*). Our interface synthesis process supports the creation of both IFB variants.

4.4 IFB Reconfiguration

The IFB reconfiguration that we present in this work offers the *basis technology* to reconfigure communicating tasks at runtime without a communication gap [9, 8]. Thus, it has been our job to model the IFB internal reconfiguration and develop techniques for its technical implementation. The more abstract part of modeling the reconfiguration process on the system-level, including innovative reconfiguration strategies, is developed in a cooperation by Meisel and Hardt [242, 137]. We address this work only briefly with respect to the RCU.

As depicted in Figure 4.11 we classify the diverse *reconfiguration* modes into the categories *static reconfiguration* and *dynamic reconfiguration*. In the literature, the static reconfiguration is also known as *offline reconfiguration*, while the dynamic reconfiguration is often referred to as *online reconfiguration* or *runtime reconfiguration*.

Static (Re-) Configuration

We apply static (re-) configuration to devices that we can configure just once or that we initialize offline before the complete system is started. Due to this limitation, we can handle only a fixed set of a-priori implemented tasks, which are completely available and attached to the considered system.

Of course, statically reconfigurable systems can pretend reconfiguration by the means of "context switching" which we introduced in Chapter 3. This process of reconfiguration can be understood as an activation of subsets of the implemented functionality. In this way, a restricted set of behaviors can be executed due to the combination of differently activated tasks. We use static reconfiguration in systems, where we own enough resources to place all functionalities in parallel. In this way, we save the complicated reconfiguration controller and do not waste time for the replacement and the establishment (transfer of the task's state) of tasks. The switching of tasks produces only a minimum of inevitable overhead.

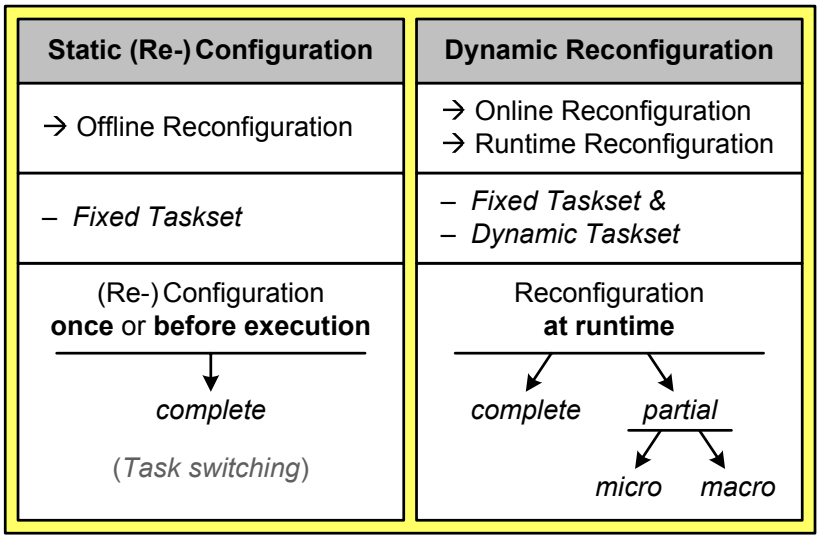


Figure 4.11: Classification of reconfiguration modes

Dynamic Reconfiguration

Due to the ability to reconfigure the devices of this class at runtime, we can handle dynamic and fixed task sets.

Fixed task set: Here, we consider predefined systems with a constant set of reconfigurable modules. This is comparable to the static reconfiguration, with the difference that not all tasks have to be present in the target platform at all times.

Dynamic task set: In this class, we cope with dynamic sets of tasks, which can change at runtime. An acceptance test has to be satisfied before a new task may join the current task set. E. g., the task has to fit into the slots and match the fixed points of a slotted architecture. In real-time systems, an additional scheduling analysis has to be passed.

To integrate a new task into the current scenario we have to pass through the IFS-Flow once again to carry the changes forward to the IFB. Actually, the IFS-Flow is uncoupled from the reconfiguration process itself. Whenever necessary, we create a new IFB, or parts of it, for the partial reconfiguration process, with the help of the Interface Synthesis Design Flow. Therefore, we rerun our design flow as stated in Figure 4.1 by the dotted arrow.

Dependent on our platform we are able to perform a complete or a partial reconfiguration (see Chapter 3). We distinguish between two models of partial reconfiguration:

Macro reconfiguration: The macro reconfiguration allows us to exchange complete tasks. Often these tasks are precompiled and available for the reconfiguration from some kind of memory, like a RAM, a ROM, or an EPROM. The RCU loads complete tasks and configures the complete or a part of the execution platform with it.

Micro reconfiguration: The micro reconfiguration aims at the reconfigured execution of the tasks themselves. This is reasonable for tasks whose functionality is composed of a large number of atomic operations. With the help of the micro reconfiguration, it is possible to reuse the same resources of one execution platform for sequential operations inside one task. *Spatial reconfiguration* techniques have been developed to manage these kinds of shared resources over time [59, 58, 113]. The micro reconfiguration is a kind of recursive application of the macro reconfiguration, when we assume that the atomic operations inside one task themselves are representable by a task graph.

The decision of where to apply which kind of reconfiguration is strongly affected by the field of application and the reconfigurable architecture. Only a few devices, e. g., the Xilinx Virtex and Xilinx Spartan family support the whole palette of reconfiguration techniques. Figure 4.12 illustrates an overview on where to deploy complete and partial, as well as micro and macro reconfiguration concerning a Xilinx FPGA in dependency of the reconfiguration time t_R and the computation time t_C. In our example $t_R = 1-5$ milliseconds (Virtex).

The *complete reconfiguration* makes sense for systems with $t_C/t_R \gg 1$. Examples for this category could be the Hubble telescope or a traffic light that work for hours or perhaps even for years with the same functionality until they are reconfigured. In these systems, the reconfiguration overhead is nonrelevant compared with the runtime. Therefore, we can afford a costly procedure that carries out the complete reconfiguration. Nonetheless, we have to take care of the "short" time in which the reconfiguration disables the device.

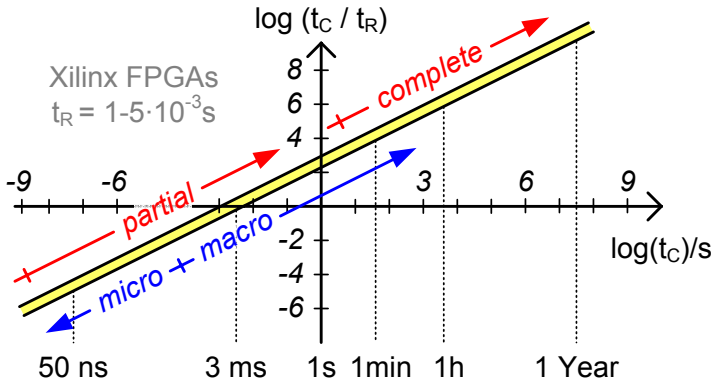


Figure 4.12: Micro vs. macro reconfiguration

If we cannot afford to switch off the complete device we can deploy *partial reconfiguration*. The micro reconfiguration model for the reconfigured execution of tasks works at high clock frequencies even for $t_C/t_R \ll 1$. A precise scheduling of the reconfigured resources is required to create the necessary data and control path in time. The micro reconfiguration model is mainly deployed to coarse-grained architectures where the complexity and the associated reconfiguration overhead are still manageable.

Fine-grained architectures are ideal for the macro reconfiguration. Due to their high scalability, we can easily map the "macroscopic" tasks onto the reconfigurable architecture; e. g., into the slots of a slotted FPGA architecture. Based on the presented techniques to hide the reconfiguration time, we can gain an increase in the overall performance even for $t_C/t_R < 1$.

The IFB was designed to cope with complex communication protocols as they are utilized on the macro level and above. On the micro level, an IFB would add quite a lot of avoidable overhead. Here, simple registers would fit better in most cases. For the static reconfiguration we create static IFBs, whereas an RTR-IFB is required for the runtime reconfiguration of tasks at the macro level. In Section 6.4.2 we present an approach to minimize the required IFB resources with the help of micro reconfiguration (reconfigured IFB execution).

4.4.1 The Runtime Reconfigurable IFB (RTR-IFB)

Every time we exchange tasks at runtime, the affected parts of the interconnected interface adapters have to be reconfigured as well. In the case of an IFB, the affected parts are those PH-Modes and SH-Modes that are dedicated to the exchanged tasks. To support the runtime reconfiguration of modes, the IFB has been extended with a *Reconfiguration Controller* inside the CU, and a tri-state bank (bus macros) for the hardware realization inside the PH and the SH for enabling the physical disconnection of the modes. The Reconfiguration Controller interacts with an external *Reconfiguration Control Unit* (*RCU*) that determines which tasks are exchanged and the point in time to perform the reconfiguration. Afterwards, we discuss the following challenges concerning the RTR-IFB:

1. When to reconfigure the RTR-IFB
2. The RTR-IFB reconfiguration flow
3. Modeling reconfiguration scenarios
4. Mapping the RTR-IFB to a reconfigurable architecture

Valid Points in Time to Reconfigure the RTR-IFB

It is an important feature for an interface not to loose data. Therefore, an IFB cannot be reconfigured at any point in time. For example, it is not allowed to interrupt the reception or the transmission of data (PH-Modes). As a transparent adapter module, the reconfiguration of the IFB has to be transparent for the tasks as well. Thus, a task must not recognize when the IFB is reconfigured. This implies, of course, ongoing transmissions. Furthermore, the processing of data (SH-Modes) may not be interrupted, as this could invalidate the state of the internal memory management.

Nevertheless, to reconfigure an IFB we have to exchange affected PH-Modes and SH-Modes. Thus, we have to identify valid *points in time* where we are allowed to *halt* these *modes* for reconfiguration. We demonstrate the main idea on where to find such moments with the help of the *I - P - O graph*.

I - P - O Graph & Communication Graph

The *communication graph* [11] is the first step on our way to the *I - P - O graph*. As presented in Figure 4.13 the communication graph is a combination of the IFS System Architecture and the *specification graph model* [232]. Thereby, the architecture graph derives from the *Architecture-Components* of the System Architecture. Task nodes $(T_1, ..., T_5)$ represent the available tasks which are connected by communication nodes $(C_1, ..., C_4)$. A mapping binds the task nodes to the available processing units (PU) of the architecture graph. One possible processing unit could be an FPGA.

At least one IFB has to be allocated to interconnect multiple incompatible tasks. In contrast to the specification graph, the IFB, which can be seen as the realization of one or multiple *communication nodes*, is bound to an implementation platform as well. In this way, all *channel nodes* of the architecture graph reduce to simple interconnections without any own behavior. Thus, the channel nodes were omitted in the communication graph.

Multi-Task IFBs can adapt several interfaces. Therefore, multiple communication nodes of the communication graph can be mapped to an IFB node as depicted in Figure 4.14. We call this the *interface centered design* style.

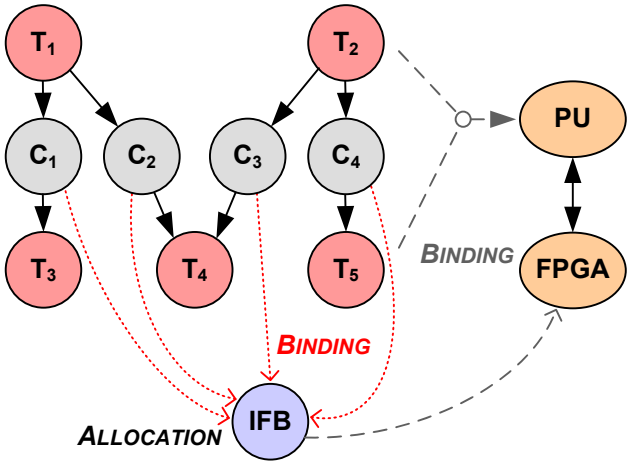


Figure 4.13: Communication graph

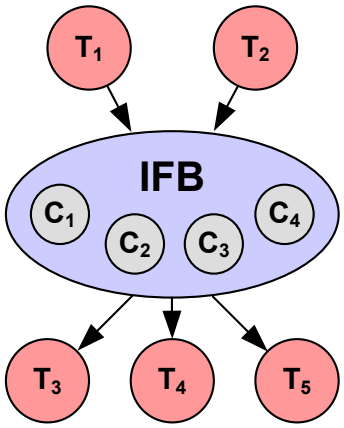


Figure 4.14: Interface centered design

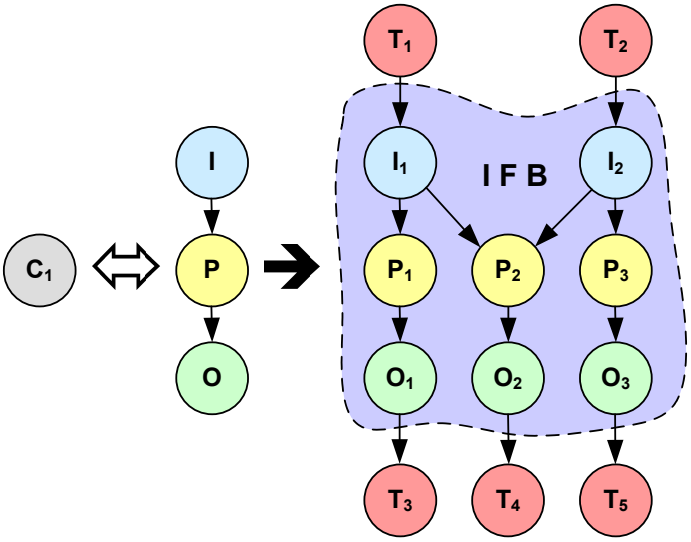


Figure 4.15: Combination of I - P - O and communication graph

In the next step on our way to I - P - O graph, we consider the pipelined execution of the IFB. Figure 4.15 illustrates the combination of the communication graph and the I - P - O based data processing of the IFB. Inside the IFB each communication node is implemented as an I - P - O sequence. The *PH-Modes* implement the *I* and *O nodes* while the *SH-Modes* realize the *P nodes*. In the case of several tasks (in our example $T_1 \dots T_5$) the resulting graph is constructed as follows (under the assumption that each task comprises exactly one interface which reads the output of its previous tasks and offers the input for the succeeding tasks):

1. Add one *input node* (→ PH-Mode) for each task that delivers data.
2. Create one *output node* (→ PH-Mode) for each consuming task.
3. Insert *processing nodes*: As each processing node represents a mapping function, which allows multiple inputs, we require exactly one processing node per output node.

The precise construction of the communication cycle is presented in Figure A.1. As usual for communicating systems, we assume the task-graph to be executed repeatedly. From the viewpoint of an IFB, only the input, processing, and output nodes that are mapped to this IFB are relevant. One iteration through the I - P - O nodes that belong to one IFB is called *communication cycle*. To execute multiple communication cycles, we insert an edge that leads from the output nodes back to the input nodes as depicted in Figure 4.16.

The dotted line presents a notional *progress* of the *I - P - O pipeline*. When the *progress line* runs through a *node*, the related stage of the I - P - O pipeline is currently executed. After a node finishes processing, the progress line advances to the outgoing *edges*. In our example, the stages related to I_1 and P_1 already finished, while I_2 is currently receiving data. Thus, O_1 is the only stage that is ready for execution. P_2 and P_3 cannot start until I_2 finishes.

We may halt affected PH-Modes and SH-Modes in any position where the I - P - O progress line does not intersect the node of an affected mode. Otherwise, the related mode would be currently active. If this claim is fulfilled, our condition for reconfiguring, which is that all affected modes are idle, is satisfied and there is no data lost in the IFB. Inside the communication cycle, there exists three kinds of edges for the progress line to intersect:

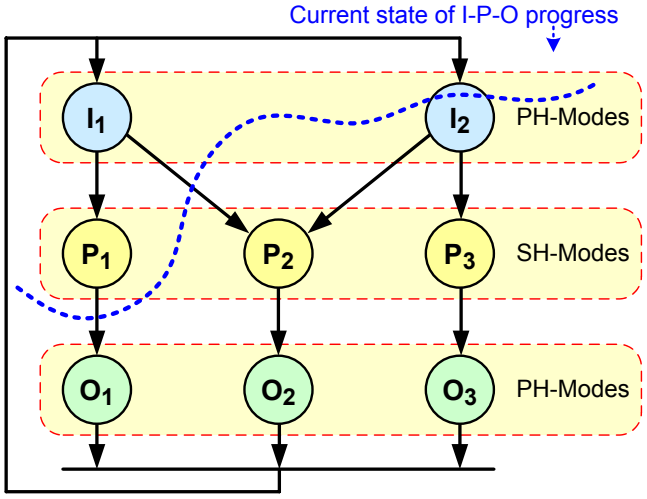


Figure 4.16: *I - P - O graph* including the current state of I - P - O progress

1) An edge between an *output* - and an *input node*:
   ⇒ In the beginning of a communication cycle the IFB holds no data to loose.
2) An edge between an *input* - and a *processing node*:
   ⇒ The received data is processed by the related SH-Recon-Mode.
3) An edge between a *processing* - and an *output node*:
   ⇒ If the receiving tasks is reconfigured the computed data is obsolete anyway and will be overwritten in the next communication cycle. Otherwise, the PH-Mode, which is related to the output node, transfers the processed data to the receiving task.

To guarantee that the actual state of the I - P - O progress is not influenced by the reconfiguration process, it is handled inside the CU. The Reconfiguration Controller in the CU also deals with the correct accomplishment of the IFB internal reconfiguration process.

The RTR-IFB Reconfiguration Flow

The reconfiguration flow of an IFB is illustrated in detail by Figure 4.17 [8]. When for some reason the reconfiguration of a task has to take place, the RCU informs the Reconfiguration Controller to activate the IFB internal reconfiguration procedure. Therefore, the RCU has to declare the IDs of the exchanged tasks. Based on these IDs the IFB determines which modes (*PH-Modes* and *SH-Modes*) are affected by the *current reconfiguration process*.

As a first step of the IFB reconfiguration, the control unit *halts* the reconfigured PH-Modes and SH-Modes. For this reason, we extended the scoreboard with the functionality to await all affected modes to become idle and keep them from further processing. Remember, a PH-Mode is idle if it does not to read or write data, an SH-Mode if it does not process data.

After halting the affected modes, the CU *switches* between the *reconfigured SH-Modes* and the related *SH-Recon-Modes*. Therefore, each SH-Mode possesses an SH-Recon-Mode that represents a user-defined behavior, specified at design time, which is executed during the reconfiguration. Thus, we guarantee a deterministic behavior for real-time environments. As illustrated in Figure 4.18 an SH-Recon-Mode comprises an FSM that offers the reconfiguration behavior for different scenarios in the form of modification state sequences.

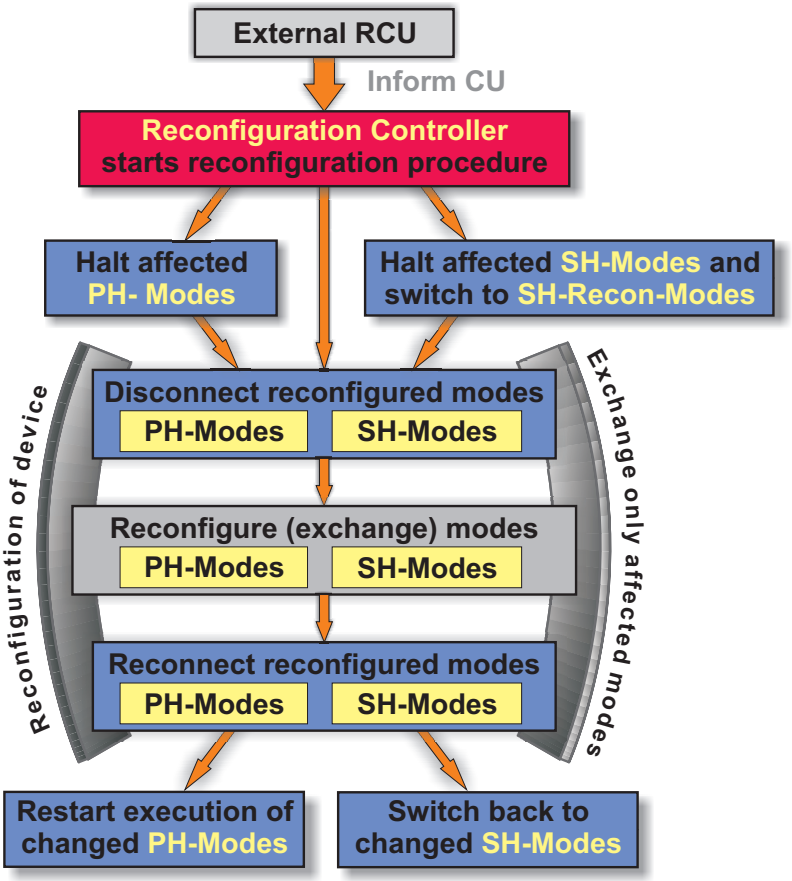


Figure 4.17: RTR-IFB reconfiguration flow

Afterwards, the affected modes are disconnected by setting the bus macros to high impedance. Now, the reconfiguration of the tasks in combination with the previously generated PH-Modes takes place. The affected SH-Modes are reconfigured as well: for additional tasks new SH-Modes are inserted, for replaced tasks the obsolete SH-Modes are replaced. After this, the exchanged modes are reconnected by the bus macros and the PH-Modes are activated. Synchronously, the SH-Recon-Modes are switched back to the SH-Modes. A special partitioning of the RTR-IFB model assures the reconfigurability of the modes inside the IFB Macro-Structure. The placement of this partition is described afterwards.

Modeling and Implementation of Reconfiguration Scenarios

An SH-Recon-Mode is able to provide the reconfiguration behavior for each possible reconfiguration scenario, which means for all combinations of exchanged tasks. As we can see in Figure 4.18 an SH-Recon-Mode is implemented as FSM. In its initial state, the FSM evaluates the IDs of the reconfigured tasks, which are provided by the RCU.

If only one out of n Tasks can be reconfigured simultaneously, we have to distinguish between n different reconfiguration scenarios. If an arbitrary combination is allowed, we have to cope with a maximum of $2^n - 1$ different scenarios. The scenario in which no task is reconfigured does not have to be considered here. More or less, the SH-Mode covers this scenario.

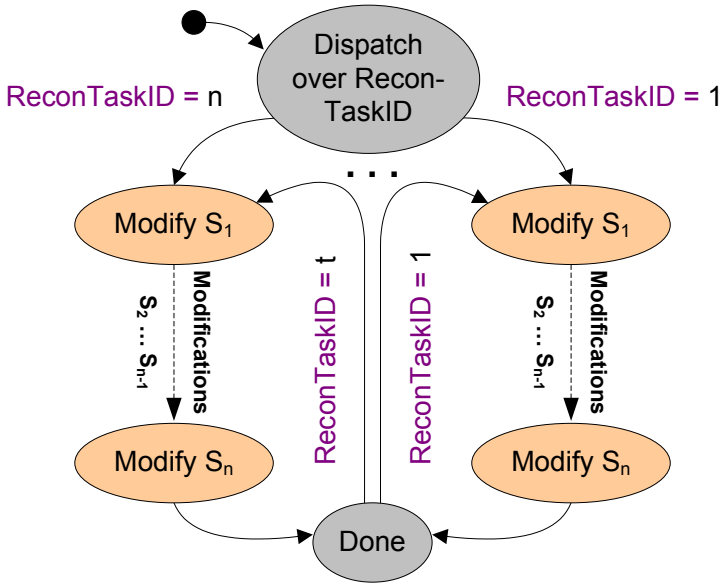


Figure 4.18: Execution of the reconfiguration behavior as FSM

Nevertheless, the functionality of an SH-Mode must not be joined with its SH-Recon-Mode, as the two modes have to be separated for the reconfiguration; one mode is always active, while the other one can be reconfigured. This leads to the fact that not only SH-Modes can be reconfigured, but also SH-Recon-Modes. In this way, we can adjust the behavior for particular reconfiguration events. In systems that reconfigure only at well-known points of time, we could even free the resources that implement the SH-Recon-Modes and use them ulteriorly. In this case, the SH-Recon-Modes would have to be reestablished just in time for the scheduled reconfiguration.

Similar to the SH-Modes, we employ linear bounded automata in the SH-Recon-Modes to model the reconfiguration behavior; for example, a static sequence of modification states or a simple work around based on the last transmitted value. Static state sequences can be useful for systems that provide a kind of "default behavior" or "fail-safe behavior". During reconfiguration, we could force the system to execute this behavior. We can also simulate a tri-state behavior by doing nothing. For controllers it could be a clever idea to repeat the last received data or to apply the results of a "mini-controller". In general, the possible behavior depends on the reconfiguration scenario that determines which tasks are reconfigured and which ones have to be served furthermore.

Let us illustrate a representative *reconfiguration scenario* through an example. Therefore, we assume five tasks, which offer and / or request data packets. As we will see later, packages are the central objects for the protocol translation. We identify the packages within the synthesis flow from the given protocol descriptions inside the IFDs. T_1, for example, provides the incoming package $P_{in}1$ and consumes the two outgoing packages $P_{out}1$ and $P_{out}2$.

Five interacting tasks T_1, ..., T_5:

$T_1 \rightarrow \{P_{in}1, P_{out}1, P_{out}2\}$, $T_4 \rightarrow \{P_{in}2, P_{out}3\}$,
$T_2 \rightarrow \{P_{in}3\}$, $T_5 \rightarrow \{P_{in}6, P_{out}4\}$
$T_3 \rightarrow \{P_{in}4, P_{in}5\}$,

Mapping functions of this scenario:

1) $P_{out}1 <= f_{map}(P_{in}2, P_{in}5)$,
2) $P_{out}3 <= f_{map}(P_{in}3)$,
3) $P_{out}4 <= f_{map}(P_{in}4, P_{in}6)$

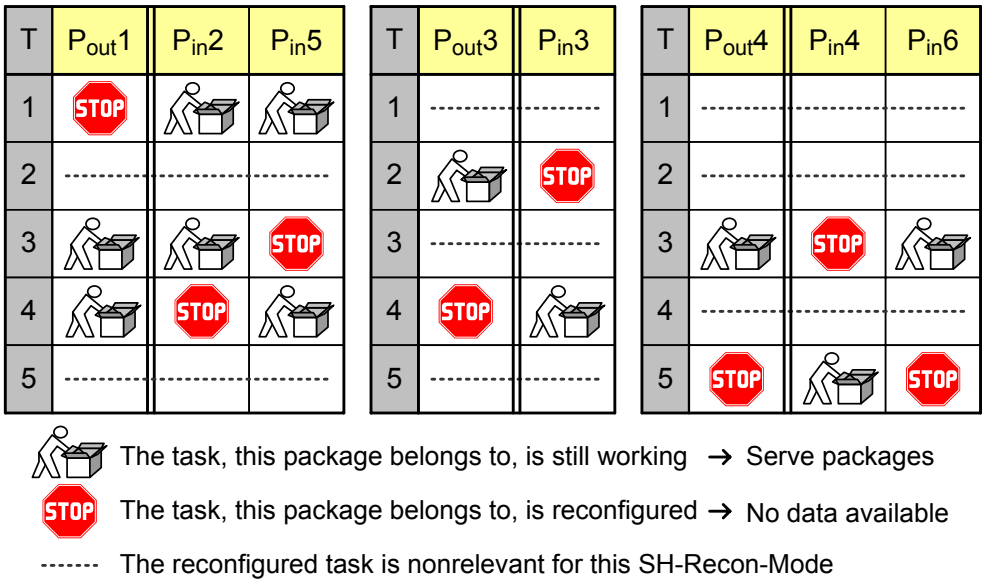


Figure 4.19: How to determine the correct reconfiguration behavior

Three mapping functions define the data mapping for this communication scenario. The first mapping function, e. g., requires the data packages $P_{in}2$ and $P_{in}5$ to process $P_{out}1$. Here, the precise *mapping operation* that is specified by the mapping functions is nonrelevant. Based on the tasks and the mapping functions, we created the three tables depicted in Figure 4.19. Each row stands for one of the five tasks ($\rightarrow$ reconfiguration scenario), while the columns represent the parameters of the associated mapping functions. There exists always one column for the dedicated package P_{out} and $0\,..\,n$ columns for the incoming packages P_{in}.

If a mapping function is completely independent of a task, which means it does not map any data packages related to one of its interfaces, the row of this task is marked by a dashed line in the related table. The left table, for example, which is dedicated to the first mapping function, is independent of task T_2 and T_5. The SH-Recon-Mode which implements this mapping function is unaffected by a reconfiguration of these tasks. Thus, their IDs does not have to be evaluated in the initial state of the FSM.

In case of a mapping function being affected by a task, stop signs depict which packages are not available, whereas worker symbols show which packages have to be further served during the reconfiguration of this task. The first mapping function (left tabular), for example, depends on the tasks T_1, T_3 and T_4. In case T_4 is reconfigured, the incoming data package $P_{in}2$ is unavailable during the reconfiguration process. However, the data package $P_{in}5$ is still available. The designer can now specify a reconfiguration behavior that transforms the available packages $P_{in}5$ into the outgoing package $P_{out}1$. The last transmitted value of $P_{in}2$ is stored in the incoming memory and can be employed in order to create $P_{out}1$.

If several tasks are reconfigured simultaneously, we have to combine the affected lines. Thereby, a worker symbol overwrites a dotted line; a stop sign overwrites all others. In this way, we obtain a maximum of $2^n - 1$ rows (reconfiguration scenarios) for each table.

The presented tables define precisely which data is unavailable and which packages have to be further served in a particular reconfiguration scenario. In this way, these tables simplify specification of the reconfiguration behavior for particular scenarios.

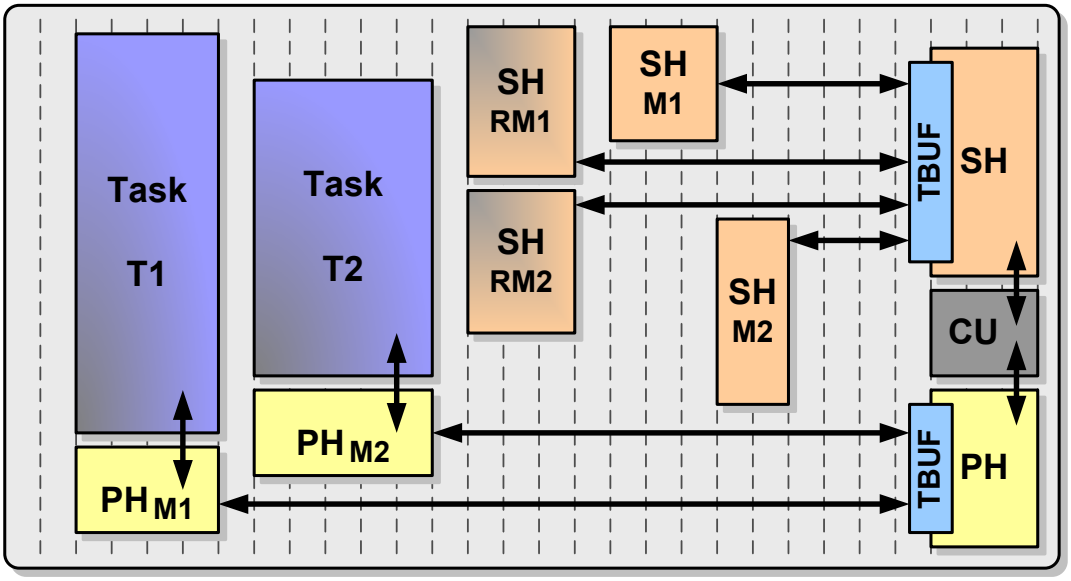


Figure 4.20: Placement of an RTR-IFB on an FPGA

The FPGA-Placement of an RTR-IFB

There exists different possibilities to place an RTR-IFB onto an FPGA [9, 8]. Figure 4.20 presents one possible mapping of an RTR-IFB to a slotted FPGA architecture, which offers the necessary degree of freedom for the reconfiguration of particular modes. Only the skeleton of the IFB Macro-Structure is placed into a fixed slot, here depicted on the right side. The skeleton of an IFB consists of the CU, the PH and the SH, and is never changed during runtime. As we will see in the realization chapter, it is important to construct the static IFB skeleton in such a way that it can handle all those interfaces, including the derived PH-Modes and SH-Modes, which it has to adapt during its life cycle.

The PH-Modes, SH-Modes, and SH-Recon-Modes are connected to the skeleton by bus macros and implemented in separated slots that can be reconfigured independently, managed by the CU in cooperation with the RCU. This facilitates the highest degree of freedom for the reconfiguration of particular modes. According to our introduction of reconfigurable architectures in Chapter 3, each mode provides the necessary fixed points to be connected to the concerning PH or SH inside the RTR-IFB skeleton.

If possible, we place a PH-Mode into one slot together with the implementation of its related task. On one hand, this is more resource-efficient than to locate them into separate slots; on the other hand, it means less overhead for the RCU as the PH-Modes always have to be reconfigured together with the task. The combination of a task and its stub can be easily created in the low-level synthesis after the modules of the IFB have been created in the Interface Synthesis Design Flow.

During reconfiguration, only the affected SH-Modes are switched to the SH-Recon-Modes, which then operate in parallel to the unaffected SH-Modes. To facilitate arbitrary reconfiguration scenarios, we place each SH-Mode in a separate slot. In case the IFB is actually not reconfigured, all SH-Recon-Modes are idle. Therefore, we implement them altogether in one slot, which can be reconfigured at any time of the “normal” operation.

4.4.2 Formalization of the FPGA-Placement

As mentioned earlier, an FPGA consists of atomic units called *slices*. It is reconfigured in *columns* that comprise two slices each. Several columns can be grouped into a reconfiguration block (RB). Notice, the slots of a slotted architecture are special RBs which are all of the same size. For technology reasons, two reconfiguration blocks are separated by bus macros (BM). In Figure 3.10, reconfigurable and fixed blocks are distinguished, but a fixed block can be understood as a special case of an RB. This leads to the following definitions [9, 15]:

In accordance with the slotted architecture, we define reconfigurable *implementation units* (U_{impl}) as paired reconfiguration block and a bus macro:

$$U_{impl} = (RB \times BM)$$

The slots of an FPGA, which hold the active configuration, can be seen as a sequence of implementation units. The length of this sequence depends on the FPGA type which is used. The size of a single BM or RB may vary, but the pairwise sequence has to be ensured.

$$FPGA = \{ u_1, u_2, ..., u_{max} \}, \quad \text{with } u_i \in U_{impl},\ i = 1 \,...\, max$$

To create the communication graph of our system S we apply the *problem graph* [232]. Therefore, we have to define a task set T and a set of communication nodes C:

$$\begin{aligned} \text{Task set}: &\quad T = \{ t_1, t_2, ..., t_n \} \\ \text{Set of communication nodes}: &\quad C = \{ c_1, c_2, ..., c_m \} \end{aligned}$$

The IFB is defined as a tuple of protocol handler, sequence handler, and control unit:

$$IFB = (PH, SH, CU)$$

The PH-Modes and SH-Modes are collected in sets:

$$\begin{aligned} PH &= \{ phm_1, phm_2, ..., phm_i \} \\ SH &= \{ shm_1, shm_2, ..., shm_k \} \end{aligned}$$

Now, each communication node c is mapped to exactly one IFB:

$$\forall\, c \in C:\ c \mapsto ifb_a \wedge\ c \mapsto ifb_b \Rightarrow ifb_a \equiv ifb_b \neq \varnothing \ \mid ifb_a, ifb_b \in IFB$$

To model the different application scenarios, we define *system configurations* $\mathcal{C}_S$ as $\mathcal{C}_S(t_1, ..., t_c)$. In dependency of the executed tasks, each configuration represents the implementation units required to implement a particular scenario (task set) including the IFB components.

$$\mathcal{C}_S(t_1, ..., t_c) = \{ u_1, u_2, ..., u_z \} \subseteq FPGA \mid (t_1, ..., t_c) \subset T$$

The mapping function $\mathcal{M}$ binds elements (here: tasks) to available implementation units:

$$\forall\, t \in T : \exists\, u_t \in \mathcal{C}_S\ :\ t \overset{\mathcal{M}}{\longmapsto} u_t$$

In the same way we map the PH, the SH, the CU as well as the PH-Modes (phm), the SH-Modes (shm), and the SH-Recon-Modes ($shrm$):

- $\exists\ PH,\ SH,\ CU : \exists\ u_{ph}, u_{sh}, u_{cu} \in \mathcal{C}_S :$
  $PH \overset{\mathcal{M}}{\longmapsto} u_{ph}\ \wedge\ SH \overset{\mathcal{M}}{\longmapsto} u_{sh}\ \wedge\ CU \overset{\mathcal{M}}{\longmapsto} u_{cu} \Rightarrow\ u_{ph} \equiv u_{sh} \equiv u_{cu}$
- $\exists\ phm \in PH,\ t \in T : \exists\ u_{ph}, u_t \in \mathcal{C}_S :$
  $phm \overset{\mathcal{M}}{\longmapsto} u_{ph}\ \wedge\ t \overset{\mathcal{M}}{\longmapsto} u_t \Rightarrow u_{ph} \equiv u_t$
- $\exists\ shm_a,\ shm_b \in SH : \exists\ u_a, u_b \in \mathcal{C}_S :$
  $shm_a \overset{\mathcal{M}}{\longmapsto} u_a\ \wedge\ shm_b \overset{\mathcal{M}}{\longmapsto} u_b : shm_a \neq shm_b \Leftrightarrow u_a \neq u_b$
- $\forall\ shrm_a,\ shrm_b \in SH : \exists\ u_a, u_b \in \mathcal{C}_S :$
  $shrm_a \overset{\mathcal{M}}{\longmapsto} u_a\ \wedge\ shrm_b \overset{\mathcal{M}}{\longmapsto} u_b : shrm_a \neq shrm_b \Leftrightarrow u_a \equiv u_b$

The equations formally express the presented conditions for an IFB. Each PH-Mode is mapped to the same implementation unit as its related task. SH-Modes never share an implementation unit; whereas all SH-Recon-Modes, as well as PH, SH, and CU, are packet into one implementation unit. Obviously, the implementation units of a configuration $\mathcal{C}_S$ are pairwise disjoint.

4.4.3 Runtime Self-Reconfiguration Using the RCU

The RCU affords the *automated reconfiguration* of communication nodes in combination with the IFB, which is executed whenever a task (-node) is reconfigured. Therefore, we introduce $\mathcal{R}(t)$ that indicates the *implementation units* that are related to the task $t \in T$.

$$\mathcal{R}(\,t\,) = \{\, t \overset{\mathcal{M}}{\longmapsto} u_t,\ t \mapsto (phm \overset{\mathcal{M}}{\longmapsto} u_{ph}),\ t \mapsto (shm \overset{\mathcal{M}}{\longmapsto} u_{sh})\,\}$$

To reconfigure a task t by t_R, we have to update the implementation units determined by $\mathcal{R}(\,t_R\,)$. If we cope with complete configurations instead of single tasks, the related units are determined by $\mathcal{C}_S = \bigcup_{i=1..c} \mathcal{R}(t_i)$. We can minimize the set of reconfigured units when tasks belong to several configurations. $\Delta\mathcal{R}(\,\mathcal{C}_{S_R}, \mathcal{C}_S\,) = \mathcal{C}_{S_R} \backslash \mathcal{C}_S$ lists only those implementation units of the new context $\mathcal{C}_{S_R}$ that differ from the current configuration $\mathcal{C}_S$. The key function $\mathcal{K}(t_R) \rightarrow\ Addr(\mathcal{R}(t_R))$ delivers the addresses of the units related to t_R. In hardware, we implement $\mathcal{K}$ as a lookup table that delivers the memory locations of the reconfigured modes.

The algorithm for the *self-reconfiguration* based on *configurations* consists of five steps:

0) The configuration $\mathcal{C}_S$ shall be reconfigured by $\mathcal{C}_{S_R}$
1) The RCU informs the affected IFBs to prepare for the reconfiguration of $t \in \mathcal{C}_S$
2) The IFBs disconnect the related modes and execute the dedicated reconfig. behavior
3) The RCU reconfigures the implementation units resulting from $\Delta\mathcal{R}(\,\mathcal{C}_{S_R}, \mathcal{C}_S\,)$
4) The IFBs reactivate the reconfigured modes
5) The RCU finishes the current reconfiguration process

The Reconfiguration Control Unit

As we have seen, one *Reconfiguration Control Unit* (*RCU*) can serve multiple tasks and IFBs. Thus, we require only one RCU instance in a reconfigurable device. We implement the RCU as FSM that we map to an additional implementation unit. If there were enough space in the fixed slot of one IFB skeleton, it could be also implemented therein.

The RCU implements a plug-and-play mechanism for tasks on FPGAs. Whenever we want to exchange a task, the RCU triggers the automatic reconfiguration of the affected IFBs. Therefore, we store the previously synthesized bit-streams of the tasks and the modes in a memory, e. g., the BRAM of the FPGA. To reconfigure an implementation unit, the RCU activates the download of the dedicated bit-stream from the memory into the FPGA via the reconfiguration port. To address the right bit-stream the RCU makes use of the key function $\mathcal{K}(t_R) \rightarrow \ Addr(\mathcal{R}(t_R)) = \{ MemLoc_1, MemLoc_2, ..., MemLoc_n \}$ that holds the memory locations ($MemLoc$) of all modes. The addresses of the tasks do not have to be stored, as this information is implied by the location of the related PH-Modes.

4.4.4 Example: A Multi-Controller Design

Before we go on with the fail-safe behavior of an IFB, we deliver an example where we applied an RTR-IFB to a multi controller design [8]. The design comprehends multiple controller variants for the control algorithm, which can be exchanged by reconfiguration. The controller has been implemented and evaluated in our working group [87, 86, 55] using the Xilinx design flow for partial reconfiguration.

In the original design, a high-level control exchanges the alternative instances of the control algorithm via reconfiguration at runtime. A multiplexer, which also provides the necessary fixed points for the inter-module communication, implements the context switching. To keep the example simple, we assume an identical sampling rate for all controller variants.

If the sum of reconfiguration and computation time remains within the sampling period, we can seamlessly carry out the reconfiguration next to the computation in a one-slot solution. Otherwise, the active controller has to operate during the reconfiguration process until the circuitry of the new controller is fully established and the multiplexer is able to switch between them. This kind of reconfiguration requires a two-slot solution where it is not possible for the controllers to share the same resources.

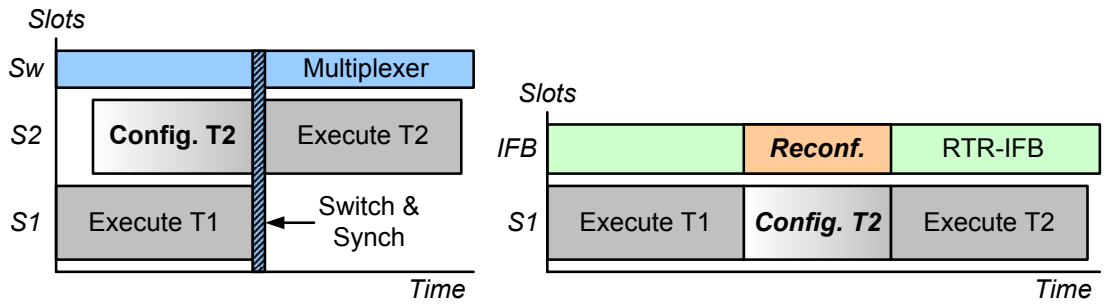


Figure 4.21: Multi controller scheduling

The left scheduling diagram in Figure 4.21 illustrates the reconfiguration procedure for the two-slot solution. Control variant T1 is computing in slot *S1*, while T2 is configured into slot *S2*. The slot named *Sw* holds the multiplexer. The hatched area represents the switching process of T1 and T2, which happens immediately after T2 is established. However, to avoid reconfiguration-based communication gaps, we have to allocate two slots for T1 and T2.

Timing Estimations

We assume a worst case computation-time t_C of 500 ns for our controller variants. The time for the complete reconfiguration of a small FPGA is about 1 – 5 ms and ranges up to 500 ms for large ones, like the Virtex-II Pro. When we divide our slotted FPGA architecture into ten logical slots, we have to cope with configuration times t_R from 100 μs to 50 ms per slot. As we can see, the reconfiguration lasts 200 to 100,000 times longer than the computation. This is an example of a macro reconfiguration with a very frequent reconfiguration.

To compensate the configuration time t_R in a two-slot solution, the *active time* t_A of a controller, which specifies the time interval between initialization and replacement by means of reconfiguration, has to be $t_A > t_R + t_C$. Depending on the sample-rate f_S (our controllers operate at a sample-rate of $f_S = 10\,\text{kHz} \rightarrow t_S = 1/f_S = 100\,\mu\text{s}$), we have to cope with $t_R \,/\, t_S = 1 - 500$ samples which are affected by a single reconfiguration process. This makes clear that the task-switching time cannot be discounted as a negligible side-effect.

The Embedded RTR-IFB

The presented timing characteristics indicate that it makes sense to introduce an RTR-IFB into this design. This means to replace the high-level control by an RCU and to substitute the multiplexer with an RTR-IFB. The RCU is comparable to the high-level control in functionality and size. Admittedly, an RTR-IFB is much larger than a single multiplexer; however, it saves to allocate the second slot as presented in the scheduling diagram on the right side of Figure 4.21. Depending on the size of the controller algorithm, it is actually possible to reduce the total size of a design using an IFB.

With the help of the SH-Recon-Modes, we are able to *transparently* perform a deterministic behavior during the reconfiguration process for the *unaffected tasks*. For example, the SH-Recon-Mode could repeat the last set-value of the exchanged controller, or compute new values based on a mini-controller, if the controller's sensors are connected to the IFB as well. As this process happens transparently for the sensors and actuators, we can remove the current controller immediately and configure the new one into the same slot (see Figure 4.21).

Results: The RTR-IFB Implementation

We used the design tool ISE from Xilinx Inc. to generate the target code for our hardware platform, a board named Digilab 2E from Digilent Inc. carrying a Xilinx Spartan2E FPGA. Figure 4.22 and 4.23 illustrate our results of the implemented IFB.

In Figure 4.22 a screen shot of the *floorplanner* can be seen. The blue parts on the right and left represent the protocol handler. The yellow area implements the reconfigurable parts including the fixed TBUF fixpoints. The control unit and the RCU are located on the very left. Figure 4.23 illustrates the implemented IFB using the Xilinx *FPGA Editor*. This illustration depicts the final circuit that is downloaded into the FPGA.

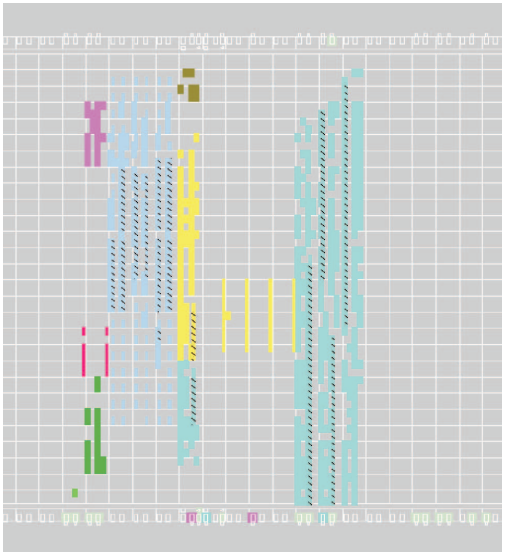


Figure 4.22: Placement and routing of the synthesized IFB

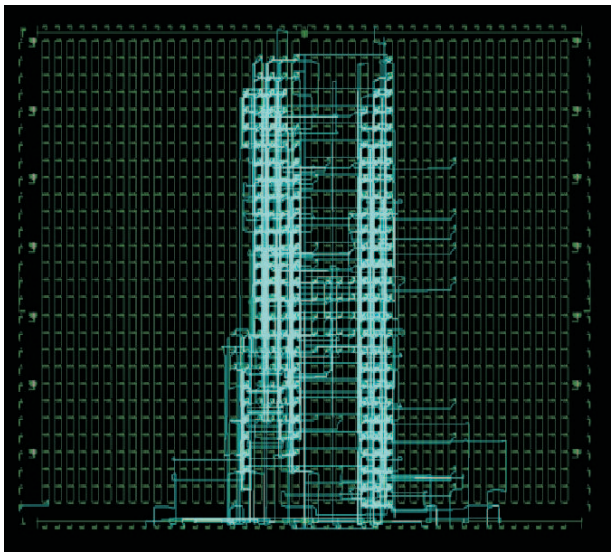


Figure 4.23: The synthesized IFB visualized by the Xilinx FPGA Editor

What Did We Learn From the Multi Controller Example?

The first question is: Which generality does the presented example have? Embedded systems are mainly separated into the data-processing and the control domains. As our controllers apply typical control values, the presented example is significant for many applications. The positive effects that we can observe in the multi controller example are:

- The Interface Synthesis Design Flow speeds up the design process due to a comfortable integration of tasks, especially those with heterogenous interfaces.
- The IFB handles the synchronization of the complete reconfiguration steps.
- The designs are no longer restricted to task-specific demands like sampling rates in relation to the reconfiguration times.
- We can easily specify the designated reconfiguration behavior as an SH-Recon-Mode. This prevents reconfiguration based communication gaps and guarantees a predictable behavior during the reconfiguration.
- Decreasing the number of slots increases the utilization of the reconfigurable device.

There are also two possible drawbacks resulting from the integration of the RTR-IFB:

- The reconfiguration time increases because of the additional RTR-IFB components.
- The design space exceeds the “hand-made” solution.

In cases where we save a sufficient number of slots, we can even reduce the total design size and thus shorten the configuration time. In the motivation we mentioned the Design Gap, which concludes that there are so many implementation resources available that we cannot manage them efficiently anymore. Therefore, we are convinced that the mentioned drawbacks are acceptable or even negligible compared to the advantages.

4.5 Fail-Safe Behavior

Next to the real-time capability, it is an important feature of the IFB to provide *fail-safe functionality* for the application in *fail critical embedded systems*. In [6, 32] we present a methodology to extend the IFB with fail-safe behavior.

Some helpful applications in the embedded systems domain have been innovated just to improve our living standard. Restricted to these systems, a reduced performance is uncritical even if it is not satisfactory for the user. Such systems are categorized as *fail-soft*. On the other hand, applications that control safety critical systems with real-time constraints have to consist of a deterministic behavior, even in case of an error.

As shown in Table 4.1 we distinguish between six states of system behavior. The most important execution modes for safety critical systems, next to the default state *go*, are the states *fail-operational* and *fail-safe* or *fail-stop-safe*. As long as we can overcome all occurring errors with the help of fault tolerance mechanisms, *fail-operational* is to be preferred to *fail-safe*. If an error correction cannot be guaranteed, a *fail-safe* behavior must be implemented.

System state	**System behavior**
go	System runs *safe* and *accurate*
fail-operational	System runs *fault tolerant*, without a loss of performance
fail-soft	System *execution* is *safe*, but the performance is restricted
fail-safe, fail-stop-safe	Just *system safety* is guaranteed, maybe with no performance
fail-unsafe	*Unpredictable* system behavior

Table 4.1: System failure states [126]

Fail-safe means to bring a running system into a safe state [230, 126] where only the system safety is guaranteed, maybe without any remaining performance. Thereby, unpredictable behavior is prevented. A system has to be forced into the fail-safe state when a malfunction would result in the damage of people, materials, or the system itself. Such a malfunction could arise from the breakdown of partial system functionality or result from a communication error in the form of corrupted or lost signals.

Our approach copes with the error processing for a large bandwidth of communication errors. The process of *communication error processing* has been divided into an *error detection* and a *fail-safe behavior* phase. We developed a methodology to transparently relocate both phases from the interacting nodes into the Interface Block.

The advantage of this methodology is that IFBs already deal with communication, and especially the protocols. From the implementation view, it is much easier to extend an IFB by a communication error processing than to implement the functionality in the task itself. To perform the error detection, an adequate *protocol guard* has to be integrated in the dedicated PH-Mode. A definition of the expected fail-safe behavior allows us to extend the protocol transformation. In this way, communication failures are hidden from the tasks already due to the interface. Fail-operational, as well as fail-safe behavior, is transparently executed based on the interface adapters of the interacting nodes.

4.5.1 Basic Concepts of Error Processing

Safety critical systems demand a predictable system behavior in case a failure occurs. In order to guarantee predictability, fail-safe mechanisms are used next to the important concept of redundancy. Some strategies for error detection are explained in the following section and an overview of fail-safe strategies used in well-known protocols is discussed afterwards.

Concepts of Error Detection

To integrate communication error processing into the IFB, it is necessary to be aware of the failures that may occur during the communication and how to discern them. A set of communication errors is presented in Table 4.2 next to a collection of well-known error detection techniques. Transmission errors and insertion change the bit pattern of a message. The disappearance of messages on the receiving side is called loss, in contrast to repetition. A false sequence of packages is critical, especially for control-dominated systems. The disguise of messages changes the target address and leads to wrong data receivers. If a message is falsified before it is provided with any safety mechanism, it is called falsification. A wrong appearance of signals in time is called timing error.

As shown in Table 4.2, a parity-bit is a simple technique to detect transmission errors. It is easy to implement, but the number of recognizable errors is limited to an odd number of bit faults. Rectangle codes extend the parity check to a two-dimensional check over several messages. Hamming codes add redundancy for detecting and correcting the wrong bit pattern. Techniques like cyclic redundancy check (CRC), PID-check, or frame-check detect most transmission errors. Using more complex data structures, we are able to discern repetition, loss, false sequences, and timing errors. Representatives of this class are acknowledge-fault detection and the message descriptor list (MEDL) [168].

In order to reach a higher degree of error detection, different methods can be combined. Due to this combination, a reliable error detection for safety critical systems can be obtained.

mechanism / **fault**	transm. error	repetition	loss	insertion	false sequence	falsification	disguise	timing error
Parity	X							
Rectangle code	X			X				
Hamming code	X			X				
CRC	X			X				
Frame-Check	X			X				
PID-Check	X			X				
ACK-Fault Det.		X	X		X			
MEDL		X	X		X			X

Table 4.2: Communication failures and detection mechanisms [126]

Error Checks of Popular Protocols

Compared to the ISO/OSI reference model, communication failure checks can be found in each layer. They vary in functionality and complexity. As we will see in the following section, complete sections of the reference model have to be implemented by the designer if there is no support given on the existing implementation platform. An IFB can be used to overcome gaps, which result from missing or incompatible layers inside the ISO/OSI model. The communication failure checks, which originally would be realized in the bypassed layers, are implemented in the IFB instead.

Many popular protocols contain methods for error detection and correction. For instance, RS-232 adds a parity bit; USB uses a CRC of 5- or 16-bits depending on the transmitted package. Protocols, which have been designed for safety critical systems, often combine several failure detection methods as time-triggered protocols (TTP) do. TTP/A employs a message descriptor list (MEDL) in cooperation with a parity bit. TTP/C combines a MEDL with a CRC, which is more powerful in error detection [168].

Other protocols like LVDS and RS-458 define only electrical parameters of the physical layer. The developer, himself, has to implement the functionality between the application and the physical layer. This challenge can be rather complex, faulty, and costly in terms of time and money.

4.5.2 Integrating Error Processing into an IFB

The integration of communication error processing into an IFB has to cope with two phases: the *error detection* and the execution of *fail-safe behavior*.

Executing Error Processing Inside an IFB

From our point of view, communication errors may result only from the adapted protocols, never from the IFB itself. Thus, the functionality for discerning communication errors has to be located inside the PH-Modes. We implement the error detection as a so-called *protocol guard* which performs a *monitoring* of the executed protocol. An additional FSM that is closely coupled to the FSM, which implements the communication protocol, provides the dedicated monitoring functionality.

As one advantage of this approach, we can perform all error checks presented in Table 4.2, except semantic tests (falsification, disguise), in parallel to the data processing. To inform the CU about a communication error, we make use of failure IDs. We specify as many failure IDs as required for each PH-Mode to activate the various fail-safe behaviors by the CU.

To integrate fail-safe behavior into the IFB, we reminisce about the method to implement reconfiguration behavior (see Section 4.4.1). The idea is to provide additional user-defined SH-Modes that can be activated during runtime. Within these modes, the designer specifies the behavior that is executed while the fail-safe mode is activated. Deterministic behavior can also be assured for real-time environments in this way.

If we can assure that no communication error occurs during the reconfiguration of the PH-Recon-Modes, we can even integrate the fail-safe functionality into the PH-Recon-Modes as a further part of the user-defined FSM.

The way we handle communication errors is quite similar to the reconfiguration flow depicted in Figure 4.17. Our method of handling communication errors excludes undefined states and unpredictable system behavior. The detailed *procedure* for coping with *communication errors* consists of six steps:

1) The communication error is detected by a *protocol guard*, e. g. a timing violation in a timed transition.
2) The protocol guard informs the CU that a failure occurred by updating the failure ID. The ID specifies which fail-safe behavior has to be executed.
3) The CU determines the affected SH-Modes and switches between active and fail-safe SH-Mode (or SH-Recon-Mode where applicable), which has been previously specified by the system designer.
4) In parallel to 3), the task dependent PH-Mode is reset and reactivated in the initial state of the protocol. The PH-Mode tries to synchronize itself with the protocol to reestablish the connection. Meanwhile, the protocol guard in the PH-Mode listens for a reactivation of the communication.
5) In the case of a successful reactivation, the protocol-guard informs the CU.
6) The CU switches immediately back to the default SH-Mode.

Integration into the Communication Synthesis Flow

To integrate communication error processing into the Interface Synthesis Design Flow, the failure detection and the description of fail-safe behavior have to be included in the abstract description level as well. The detection of communication failures belongs to the external protocols and therefore has to be part of the IFDs. Only the designer of an IFD is aware of the functionality of the protocol guard. Thus, the specification of the protocol guard is composed a-priori and gets part of the protocol description inside the IFD. Of course, the protocol synthesis has to treat this part of the IFD differently from the original protocol description. To speed up the design process, a set of predetermined error-detection techniques, as presented in Table 4.2, can be provided to the designer. The designer has to specify the fail-safe behavior for dedicated communication errors in the form of an SH-Mode or a state sequence in an SH-Recon-Mode.

Actually, the fail-safe behavior has not been integrated into the Interface Synthesis Design Flow. Nevertheless, the presented theory copes with all challenges of this problem. Therefore, there remains the pure implementation aspect which is part of future work.

4.5.3 Case-Study: Robot Scenario

An example that has been implemented by hand shall demonstrate the functionality and limitations of our approach. The scenario given in figure 4.24 consists of a task which sends a one-byte set-value to a robot controller with four degrees of freedom. We assume a high-level controller or a user to be represented by this task. The communication between the task and the controller is established by an IFB. We use the serial RS232 protocol that implies 8 bit of

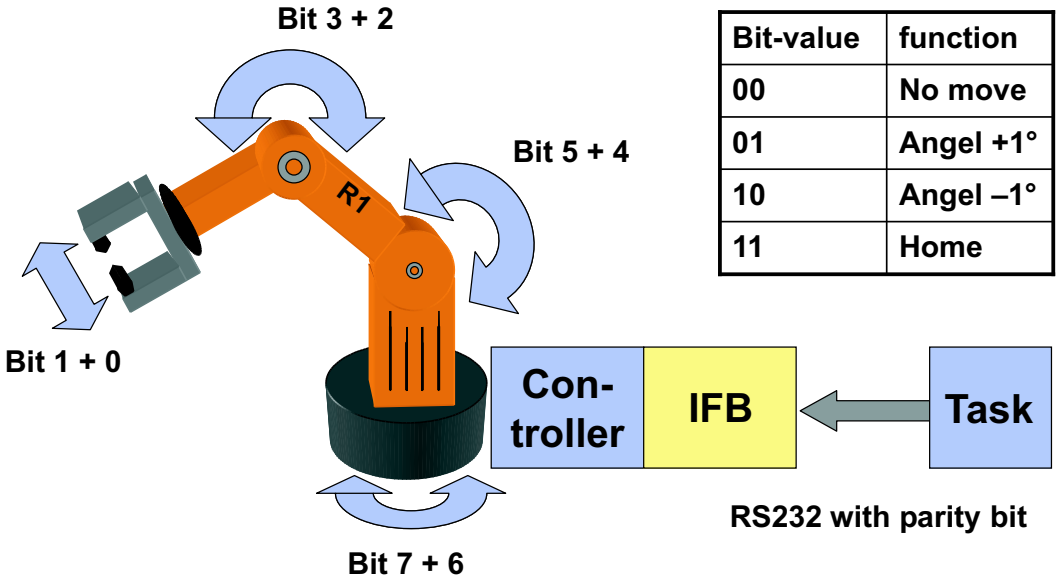


Figure 4.24: Error handling in a robot scenario

data and one parity bit. The parity bit is evaluated to avoid transmission errors. Therefore, we use a *parity checker* as one protocol guard. Additionally, the controller expects a message from the task every millisecond. This time condition is guarded by a *watchdog*.

If one protocol guard observes its error condition to be fulfilled, the dedicated fail-safe mode in the SH is activated and generates the fail-safe control data for the robot controller. The fail-safe data for a wrong parity is defined as ”00000000”. By transmitting this data to the controller, it is guaranteed that no further movement takes place by stopping the complete robot arm. A timing error causes the robot to go back to its default position by sending ”11111111”. A properly received message would reestablish the communication here. These actions guarantee a predictable behavior in each situation.

The descriptions for the parity checker and the watchdog have been modeled as FSMs and were linked into the existing IFD. Due to this linkage, the protocol guards are able to exchange data with the protocol FSM and the control unit. Therefore, the signals of the protocol guard were hooked into the vertical and horizontal communication structure of the IFB. The same is true for the fail-safe behavior. As a further SH-Mode, the fail-safe behavior is treated as equivalent to the synthesized SH-Modes.

A large bandwidth of communication errors can be treated in the same way as given in our robot example. By definition, our approach is actually limited to syntactical checks and thus does not perform any semantic error detections. This is also the reason why we cannot detect falsification or disguise in general. This limitation has been committed to keep semantic understanding out of the connecting interface. In fact, the technical limitation of the communication error processing algorithms is restricted to the expressive power of linear bounded automata.

4.6 Relation to the ISO/OSI Model

Every communication between component interfaces is based on communication protocols. To compare different forms of communication we often use the ISO/OSI reference model. This model allows us to classify a communicating component related to one of the seven ISO/OSI layers (see Figure 4.25). Those layers, which are implemented by a particular component interface, are called *protocol stack*. Beginning with the lowest layer, the protocol stack grows up to that layer which is required for a particular application. Therefore, applications with a simple I/O protocol, like a computer keyboard, implement only a small protocol stack; whereas complicating applications, like a program for home banking, grow up to the application layer.

Until now, we introduced the *reconfiguration based communication gap*, which copes with I/O problems during runtime reconfiguration. The second kind of gap is the *ISO/OSI layer based communication gap* [4]. It describes the challenge of adapting interfaces of different ISO/OSI layers. We can deploy an IFB *above* the *physical layer* to overcome these gaps, which result from *missing* or *incompatible layers* inside the ISO/OSI model. Therefore, the functionality that originally would be realized in the bypassed layers, for example, the handling of communication failures, has to be implemented in the IFB instead.

The ISO/OSI model describes scenarios where tasks are interconnected by communication media. Media are located below the physical layer while tasks access the protocol stack from above. Therefore, it is our aim to adapt high-level task interfaces with low-level media interfaces, which are separated by an ISO/OSI layer based communication gap as illustrated in Figure 4.25. The figure illustrates four media that implement different protocol stacks: RS-232 and RS-485 define only the physical layer, CAN implies also the link layer, while Firewire (IEEE 1394) goes up to the network layer. The upper two layers of the ISO/OSI model are dedicated to the tasks. It is not an intended aim of the IFB to adapt the abstract protocols of this high-level communication. The same is true for the layers four and five, that we denoted as *Abstract Communication Layer* (*ACL*). The ACL is part of our real-time environment for the rapid prototyping of communication.

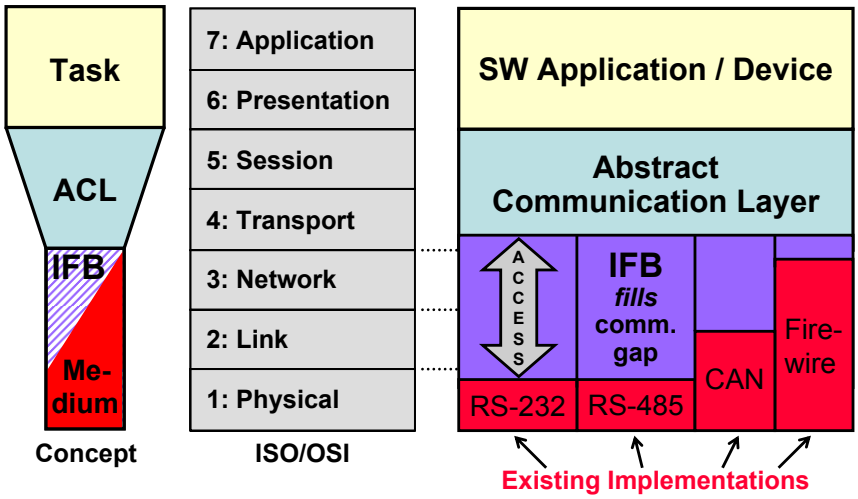


Figure 4.25: ISO/OSI layer based communication gap

4.7 Prototyping of Real-Time Communication

To facilitate the *rapid prototyping* of *real-time communication* scenarios, we developed a specific approach based on the ISO/OSI model. Figure 4.26 illustrates a common situation: A sending task generates data that has to be transmitted to the receiving task via a medium. It is the duty of the IFB to accept, modify and generate protocol conform bit-streams. The *abstract communication layer* (ACL) provides *real-time capable communication channels*.

In conformance to the ISO/OSI standard, each layer provides a number of services to the upper layer and thus hides all technical implementation details of the lower layers. For that purpose, the ACL provides various services to the tasks like *transmit* - and *receive messages* on *logical channels*. Both messages and channels support *priorities* as well as *transmission deadlines* [4].

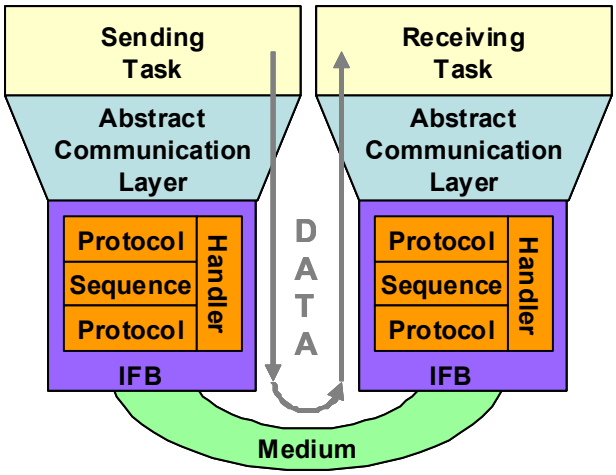


Figure 4.26: Rapid prototyping of communication

Abstract Communication Layer (ACL)

We created the abstract communication layer to add a higher class of service and safety to the programming of embedded systems. The ACL encapsulates the lower level interfaces of real-time capable media and thus hides them to tasks, as shown in Figure 4.27. Without the ACL, the computational nodes would have to also consider the details of the real-time communication on the session and transport layers.

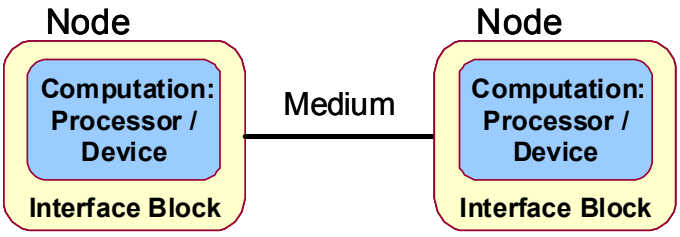


Figure 4.27: Abstract view of communication

By including the ACL in the design process, the programmer may concentrate his efforts in developing the tasks and just call the API functions of the ACL. The most important functions are the *reservation* of a *channel*, and specifying the media that one wants to use, including priorities and time restrictions (see Figure 4.28). Further, there exist functions to *schedule* (*send*) or *read* messages from a specific channel.

To optimize the utilization of the available real-time connections, multiple logical channels are mapped onto one physical medium. The reservation of a channel has to pass an acceptance test based on a schedulability analysis, which guarantees that the maximum utilization of a particular medium (100 %) is never exceeded.

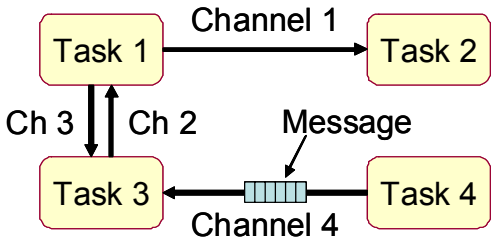


Figure 4.28: ACL in a multi-task communication system

Each logical channel behaves as a single entity. It provides a one way communication between two tasks with a unique system-wide priority level. The messages themselves consist of priorities as well. We determine the order in a transmit process firstly by the unique channel priority and secondly by the message priority. If there are two messages in a channel with the same priority, the message that was scheduled first will be sent first.

An IFB provides the technology to support a wide range of protocol stacks. In this way, the IFB is able to provide a uniform API for the ACL for different kinds of media. ACL and IFB communicate via memory mapped I/O. According to the different protocol stacks the IFBs have to adapt to the physical layer (1), the link layer (2), or the network layer (3) of the ISO/OSI model (see Figure 4.26). Because of the transparency of an IFB, which is in accordance with the ISO/OSI concept to hide lower level details, the ACL is not able to determine between different communication media.

To make use of the IFB, we have to describe the IFD of each protocol that we like to support. RS-232, RS-485, CAN, asynchronous- and isochronous FireWire are our favorite real-time media. However, RS-232, RS-485, CAN and FireWire (in the asynchronous mode) do not guarantee a real-time communication by nature. We can solve this problem by scheduling the medium access, for example, using time slots similar to TTP.

Until now, we modeled and tested the IFB for the two protocols RS-232 and RS-485 in this scenario. Additionally, we implemented an ACL solution that deploys an isochronous FireWire communication based on shared memory [4]. Apart from this real-time example, we created an IFD also for the I^2C bus.

4.8 Summary

In this chapter, we presented our Interface Synthesis methodology. We introduced the Interface Synthesis Design Flow and its partitioning into design phases and design steps. Structuring the design flow in this way helps us to cope with the complexity of automatically generating reconfigurable IFBs. Furthermore, we introduced our modeling concept which accomplishes the three domains XML, Java, and UML.

Afterwards, we explained our interface adapter module, the Interface Block. On one hand, we refer to the IFB Macro-Structure, which defines the conceptual design of the IFB; on the other hand, we explained the features and the functionality of the runtime reconfigurable IFB. In the end, we presented how to employ an IFB in fail-critical application scenarios and described its relationship to the well known ISO/OSI model.

Interface Synthesis Design Flow

The first design phase of the Interface Synthesis Design Flow encapsulates the modeling aspect. Within this phase, we specify the System Architecture that provides the IFDs and TPDs as input for the automated IFB synthesis. With the help of the System Architecture we are able to model the communication infrastructure of complex communication scenarios very elegantly, which is interesting for the rapid design of distributed embedded systems.

As our approach proposes a scenario-based interface synthesis, we have to clearly distinguish between the syntax and semantics of protocols, which are modeled inside the IFDs. Our IFDs are limited to the protocol syntax that defines the use-case independent structure of protocols. The semantics of particular bits does not have to be specified in our approach, which is especially interesting for the adaptation of generic IPs.

The information on how to process the data inside the IFB for a dedicated scenario is added in the interface synthesis phase. The strict separation of static structure and flexible behavior is especially helpful for the rapid prototyping of communication systems, as we are able to evaluate different IFBs without any modifications in the System Architecture model.

The subsequent synthesis phase divides into the creation of a target language independent IFB model and a code generation design step based on standard compiler techniques. As the next chapter is dedicated to the synthesis phase, we gave only a rough overview of the two design steps and the subsequent integration phase.

Modeling Concept

Afterwards, we presented our *IFS Modeling Concept* with respect to XML, Java, and UML. We highlighted the relevance and the interaction of the three modeling languages related to our approach. In the form of an XML scheme, the IFS-Format defines the formal model for the System Architecture and the exchange format of the IFS-Editor. We implemented the IFS-Format as IFS-Data-Structure, which defines the data model in the M–V–C architecture of the IFS-Editor.

To provide an intuitive and abstract modeling of the System Architecture, we developed a UML 2.0 profile. It is an outstanding feature of our approach to provide an integrated design flow, which is completely implemented by the IFS-Editor, reaching from an abstract UML 2.0 model down to executable hardware. Therefore, we explain our UML 2.0 profile and its model transformation in the following chapter in greater detail.

The Interface Block

In the second part of this chapter we introduced the Interface Block. We presented the IFB Macro-Structure that is composed of the protocol handler, the sequence handler, and the control unit, and discussed the functions and interaction of these components. The PH provides the PH-Modes, which act as stubs that pack and unpack the information that is exchanged with the adapted interfaces. The SH-Modes process this information according to the mapping functions specified in the IFD-Mapping. With the help of the mapping functions, we can assign constant values, shuffle incoming data in combination with the guarded assignment of data, and perform complex data processing based on FSMs.

Based on the IFB Macro-Structure, we explained the pipelined data processing inside the IFB, which is an important aspect for the reconfiguration of an IFB. To motivate runtime reconfigurable IFBs, we classified the reconfiguration in static- vs. dynamic-, complete- vs. partial- as well as micro- vs. macro reconfiguration. While a simple IFB fulfills all needs of the static reconfiguration, we require a runtime reconfigurable IFB for systems using macro reconfiguration. We answered the questions of when to reconfigure an RTR-IFB and explained the detailed reconfiguration flow. Afterwards, we presented the modeling of reconfiguration scenarios and discussed a flexible solution to map RTR-IFBs to a slotted FPGA architecture. The RTR-IFBs prevent reconfiguration-based communication gaps by executing deterministic behavior during runtime.

An important feature for the embedded domain is the application of the IFB as a fail-safe interface adapter in fail critical environments. We discussed how far the IFB is able to detect and handle communication errors and presented a realistic robot example.

Finally, we explained the relationship of the IFB to the ISO/OSI model. To handle ISO/OSI based communication gaps, the IFB emulates the bridged functionality within its SH-Modes. In this way, we can adapt even incompatible or separated ISO/OSI layers. The ACL is an environment for real-time communication that provides logical channels for real-time communication. In combination with an IFB, the ACL is able to employ various real-time capable media like RS232, RS485, and FireWire.

CHAPTER 5

The Detailed Interface Synthesis Design Flow

After introducing the composition and the coherences of our design flow in the last chapter, we now highlight important aspects of the Interface Synthesis Design Flow. The first point of interest is the developed UML 2.0 profile, related to the modeling phase. Afterwards, we discuss the synthesis design phase with a main focus on the protocol synthesis and the assembly of the intermediate representation of the IFB model. The chapter concludes with an insight into the external integration phase. We present the relation to Part-Y, an EDA tool for the partial runtime reconfiguration that we have developed in our working group.

5.1 Modeling-Phase

Figure 5.1 highlights the important aspects of the *modeling phase*. The main goal is the modeling of the communication infrastructure in the form of the System Architecture as input for the interface synthesis. To maximize the reuse of our models, we make them available in an IFS IP data-base. In Figure 5.1 we distinguish between two data-bases, one for communication components and the other for architecture components. The stored IPs are IFS-Instances in the XML format presented in our modeling concept (see Section 4.2).

The Interface Synthesis Design Flow supports three ways to compose new components of the System Architecture or to add externally created models:

- XML instances (including IP meta-data)
- IFS-Editor (graphical component view + interactive masks)
- UML 2.0 (Fujaba)

XML instances are suitable to *store* and *exchange* information. Therefore, we can add components to the System Architecture by loading components from the IFS IP database or by importing the meta-data of an IP. However, it is error prone and inefficient to model XML instances by hand. Therefore, we support the design entry within the IFS-Editor.

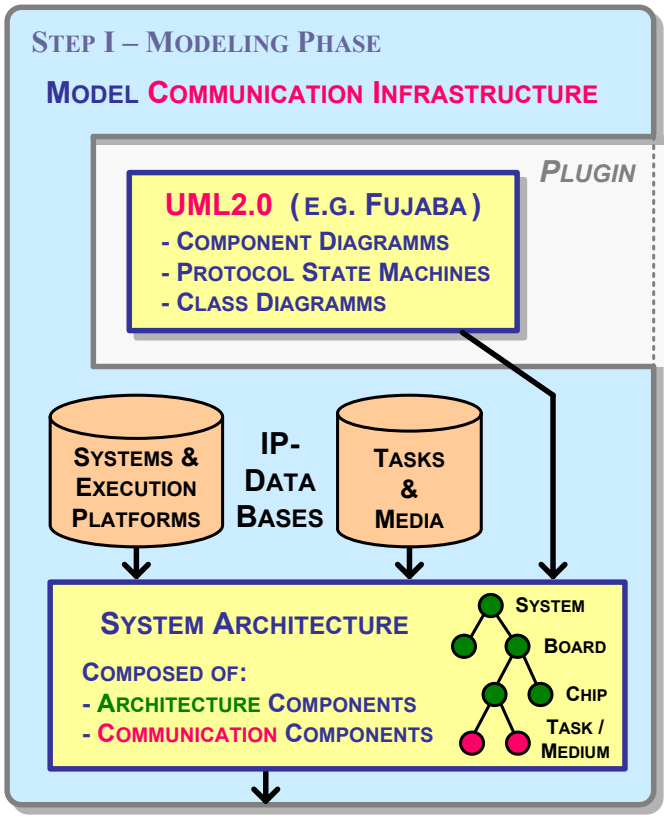


Figure 5.1: Interface Synthesis Design Flow: the modeling phase

The IFS-Editor provides a graphical component view in combination with interactive masks for entering the detailed parameters. Furthermore, we implemented a visualization of the waveform state machines as protocol traces in the form of timing diagrams.

Since UML 2.0 and the related CASE tools are familiar to many designers, we extended our modeling concept. Therefore, we developed a UML 2.0 profile that we transferred to the CASE tool Fujaba, which offers a comfortable way for modeling the System Architecture. Thus, we dispose of an integrated design flow that copes all the way from UML 2.0 models all the way down to executable hardware. Afterwards, we give a short introduction into the defined profile.

The main intention of the UML 2.0 profile is to deliver the input for the IFB synthesis similarly to the System Architecture including architecture- and communication components. The IFD-Mapping remains part of the IFS-Editor because of the IFS specific mapping language, which is a central part of the synthesis process. For this reason, we translate the composed IFS-Models into the IFS-Data-Structure in order to generate the required IFBs.

5.1.1 Modeling the UML 2.0 Profile

We analyzed the IFS-Format and UML2.0 to define an intuitive *UML 2.0 profile*. For that purpose we selected a set of *UML diagrams* and defined *precise semantics* related to the IFS-Format. The developed UML 2.0 profile comprises *component diagrams*, *class diagrams*, and *protocol state machines*. In the following section, we present important aspects of our UML 2.0 profile and the model transformation. A detailed description is given in [1, 36].

Modeling the System Architecture

We employ *component* and *class diagrams* to describe the *structural* aspects of the IFS System Architecture. With the help of these diagrams, we model the *component hierarchy* according to the component tree in Figure 2.7, including the electrical properties of the interfaces. UML *ports* provide the *connections* between the components. *Protocol state machines*, which are related to the UML ports, define the communication *behavior*. As usual for extensions of the UML meta-model, we define the added classes as *stereotypes*.

Structural Description

Figure 5.2 presents the IFSComponent diagram, which models the hierarchical structure of the System Architecture. A central class of the diagram is the stereotype *IFSComponent* that extends the meta-model class *Component*. According to the IFS-Format, this class provides a unique identification and a version stereotype. The stereotypes CommunicationComponent and ArchitectureComponent derived from IFSComponent.

On one hand, the stereotypes System, Board, Chip (right circle) represent the available architecture components; on the other hand, Medium and Task (left circle) define the interacting communication components. While we are allowed to assign media to each architecture component, tasks are bound to the chip level.

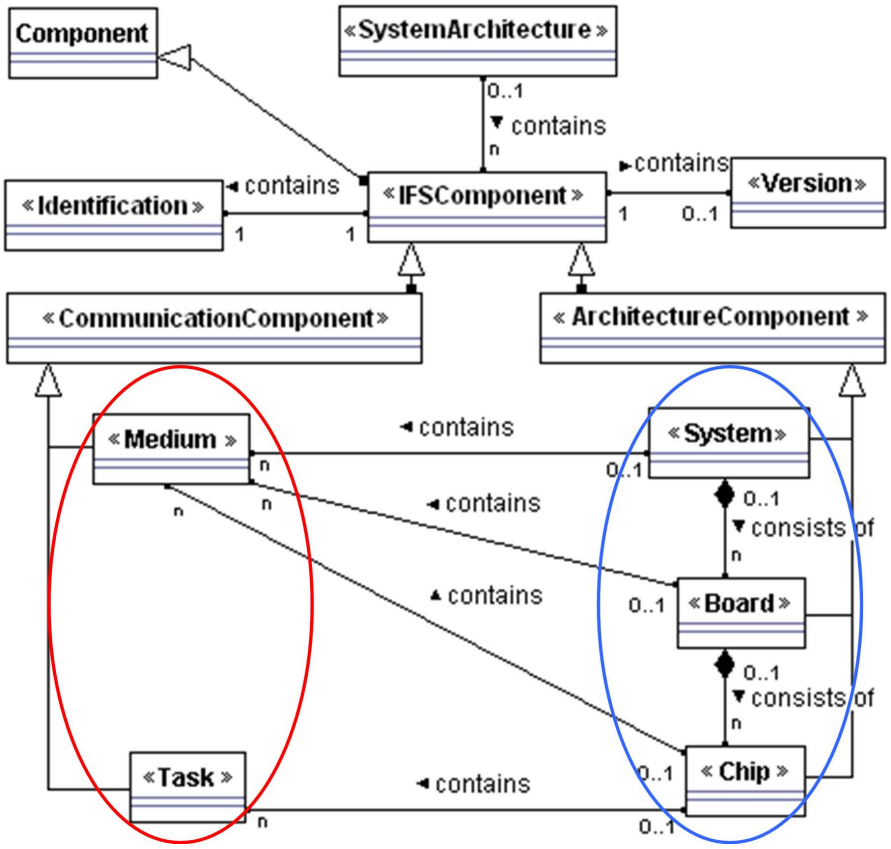


Figure 5.2: IFSComponent diagram

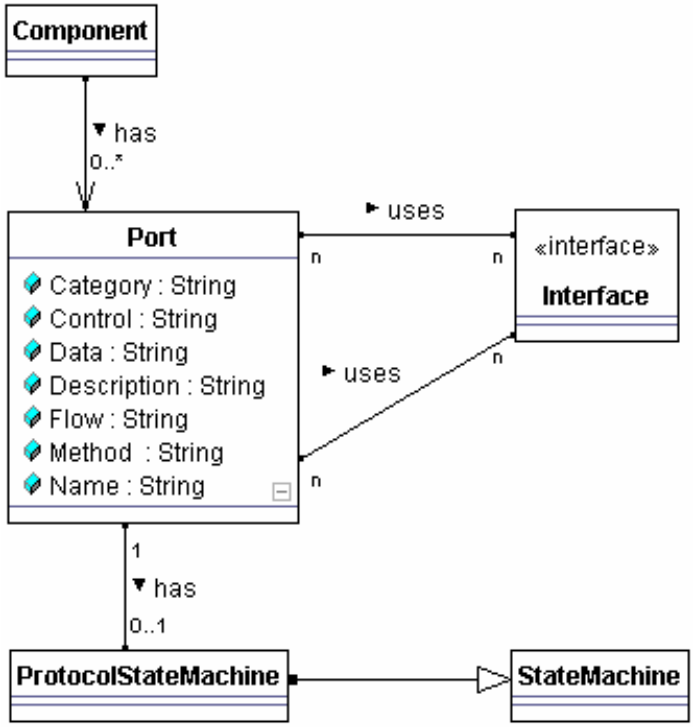


Figure 5.3: Interrelation of *Component*, *Port*, *Interface*, and *Protocol State Machine*

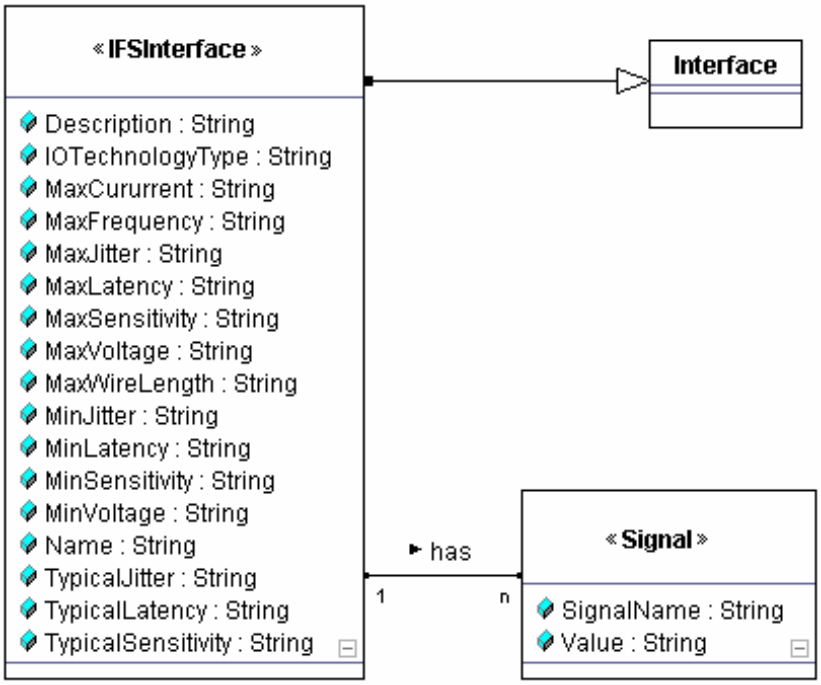


Figure 5.4: Stereotype *IFSInterface* and *Signal*

As we can see in Figure 5.3, which is an excerpt of the UML 2.0 meta-model, the class *Port* acts as connector between Component, Interface, and Protocol State Machine. In our profile, we define the stereotype *IFSPort* as a refinement of the meta-model class *Port*. With the help of the IFSPorts, we are able to combine the structural, electrical, and behavioral aspects of an IFSComponent interface similar to an IFD.

The *electrical properties* of the physical interfaces are incorporated as attributes in the stereotype *IFSInterface*, which is illustrated in Figure 5.4. Furthermore, this class defines the *interface topology* implemented as an association with the stereotype *Signal*. Each signal instance consists of a *name* and a *value* that specifies the initial value and the pin width. Additional to the physical interface, these pins also represent the ProtocolPins, which are required for the *waveform state machine*. In this situation, we fundamentally *redefine* the *semantics of UML*, as we do not think of interfaces in the meaning of provided - and requested interfaces using function calls. The waveform state machines based on ProtocolPins are dedicated to specify timing diagrams for a couple of interconnected components.

An example for a signal could be {`SignalName = sig, Value = "0X1Z"`}. Based on this input, our model transformation would create a port named "sig" consisting of four pins with the default values {`0, X, 1, Z`}. Until now, we support only the basic data types "bit" and "std_logic". In this example, the resulting data type would be "std_logic" as the values "high impedance" and "unknown" are included.

Figure 5.5 illustrates an exemplary System Architecture including two connected tasks and media. The ports of the architecture components inherit their behavior from the connected communication components. In the example we depicted only compatible tasks and media, which can be directly connected via the available ports of the architecture components.

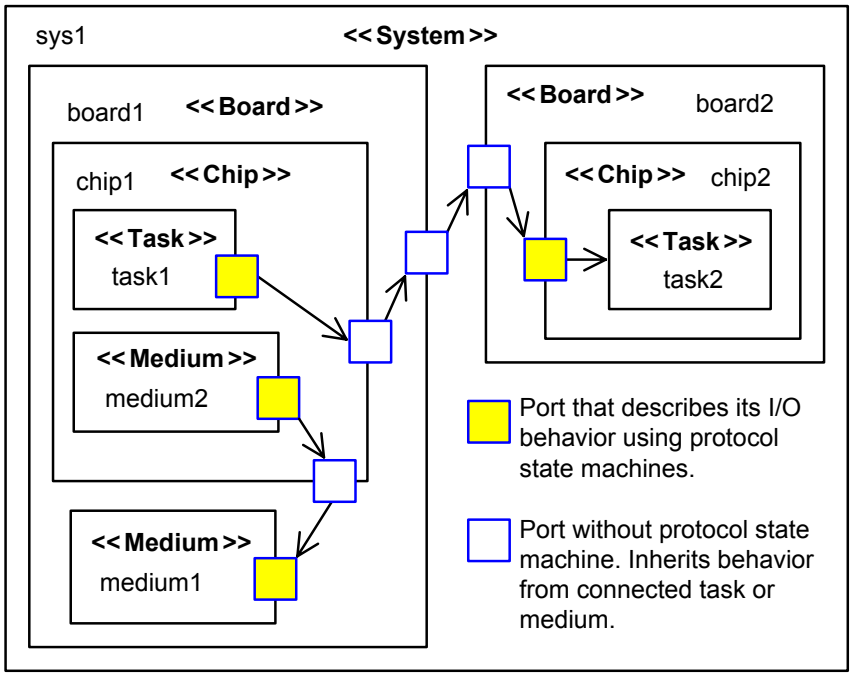


Figure 5.5: IFSComponent diagram presenting an exemplary System Architecture.

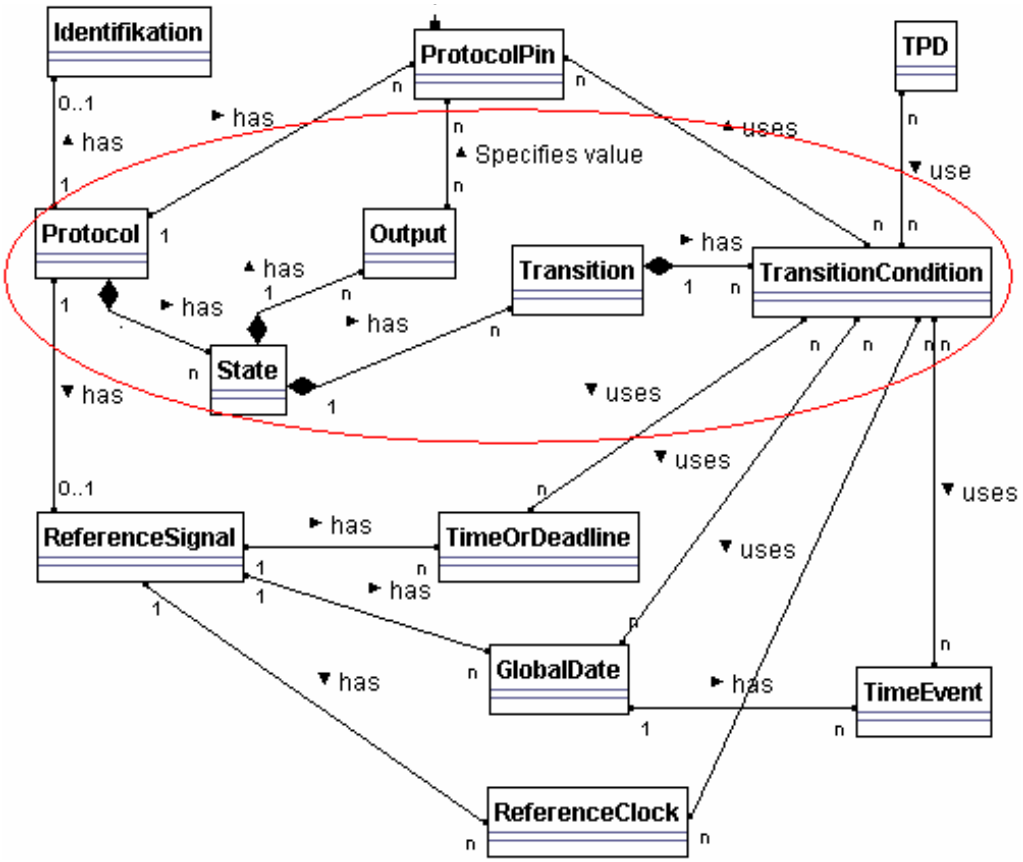


Figure 5.6: Waveform State Machine as UML class diagram.

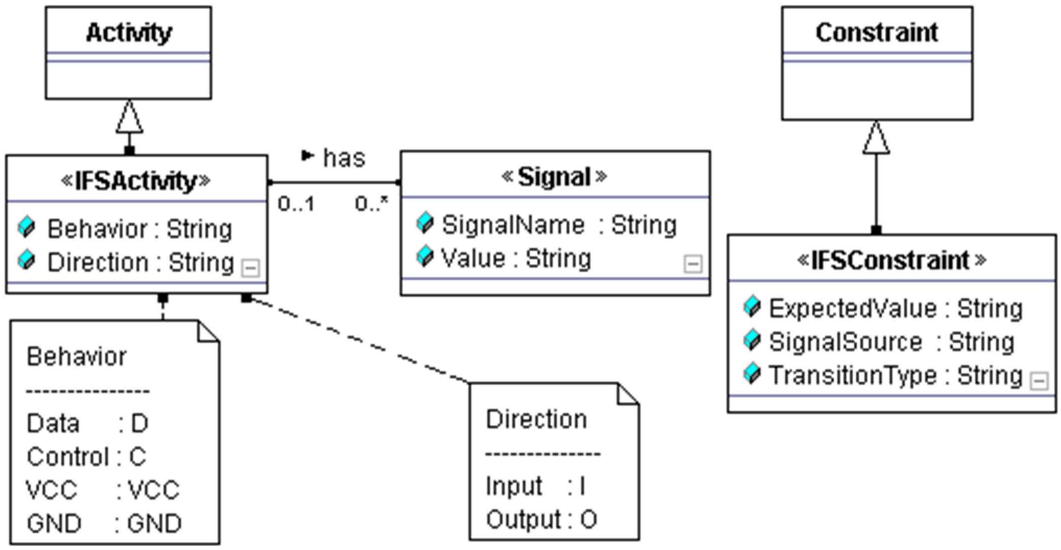


Figure 5.7: Extension of the *protocol state machine* in the UML 2.0 meta-model.

Behavioral Description

To keep the inner functionality of communication components as a black box, we restrict our profile to the behavior descriptions of I/O protocols. For our UML 2.0 profile this means that we exclusively support behavior descriptions related to IFSPorts. As defined in Section 2.1.4, we model protocols as timing diagrams in the form of waveform state machines.

Figure 5.6 illustrates the IFS-Model of the waveform state machine as a state chart. The classes within the circle define the state machine, consisting of states, outputs, transitions, and transition conditions. The class *ProtocolPin* in this diagram corresponds to the stereotype *Signal* in Figure 5.4. The output of the state machine defines the values of the related signals from the IFSInterface for each state. We implement the static attributes of the waveform state machine as attributes of the stereotype IFSPort. The dynamic aspects are implemented as protocol state machine related to the IFSPort.

To deal with waveform state machines, we adjusted the meta-model of the protocol state machine class diagram (as depicted in Figure 5.7). We extended the classes *Activity* and *Constraint* to the *IFSActivity* and *IFSConstraint* stereotypes. IFSActivity introduces the *Behavior* and *Direction* attributes to the output, while IFSConstraint extends the transitions with the *ExpectedValue*, *SignalSource*, and *TransitionType* attributes according to Figure 5.8.

The example depicts a protocol state machine consisting of three states. In `state0` we specify the signal values for the two ProtocolPins `X_Motor` and `Y_Motor`. The automata output `C.O:Y_Motor = '10'`, for example, appoints the values `'1'` and `'0'` to the two pins of the port `Y_Motor` acting as `outgoing control` in this state. Remember, all these attributes describe only the protocol syntax; we do not specify any meaning by this.

The transition condition `(S.PP:Y_Motor = '10')` awaits the two ProtocolPins of `Y_Motor` to become `'10'`. This condition is ORed with `(S.GD:TimeEvent_1 = 'HV')` that describes a transition condition triggered by a high value on the synchronous global date `TimeEvent_1`. A global date is one of the available *reference signals* that are available to model *time*.

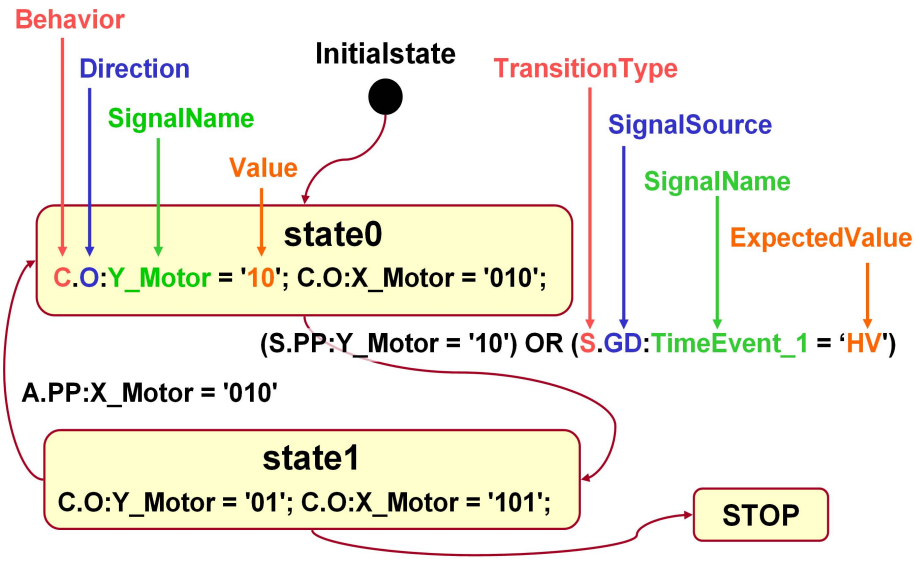


Figure 5.8: Example of the IFS protocol description syntax.

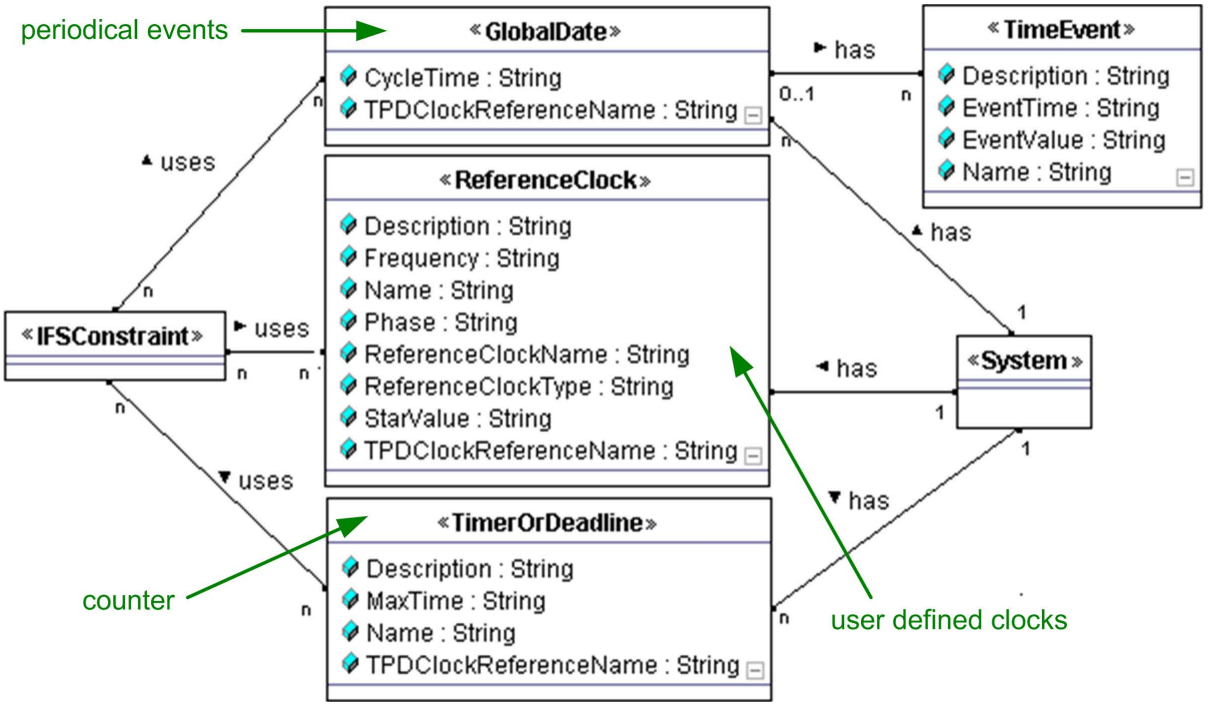


Figure 5.9: Stereotypes for reference signals

As depicted in Figure 5.6, a transition condition can depend on five different kinds of *triggers*, which divide into *signal events* (ProtocolPin) and *time events* (all others):

- ProtocolPin
- TPD clock
- Reference clock: Clock
- Reference clock: TimerOrDeadline
- Reference clock: GlobalDate

ProtocolPins provide triggers for the (incoming control) signal events *high* and *low value* as well as *rising* and *falling edge*. With the help of *TPD clocks* and the *reference signals*, we are able to react on *time events* and thus to model timed protocols for real-time communication. TPD clocks are periodic signals provided by the target platform. As illustrated in Figure 5.9, there are three kinds of reference signals available: *TimerOrDeadlines* (specify time intervals), *ReferenceClocks* (allow user defined clocks), and *GlobalDates* (can be used to model time events in a cyclic and static time schedule).

Reference signals are artificial times that we derive from the available target platform clocks, modeled in the target platform description of the stereotype chip. To be able to reconfigure PH-Modes that refer to the specific reference signals, we implement the reference signals directly within the PH-Modes. The drawback of this method is that we create one instance for each reference signal, which can lead to redundant reference clocks and global dates. The extended UML 2.0 meta-model constitutes the basis for the model transformation into the IFS-Data-Structure.

5.1.2 Tool Coupling of the IFS-Editor with the CASE tool Fujaba

The UML 2.0 profile allows the composing of IFS-Model-Instances; and in our case, with the help of the CASE tool Fujaba. To close the gap between UML 2.0 and Java/XML, we developed an automatic transformation of the IFS-Model into the IFS-Data-Structure. The model transformation is divided into the coupling of the IFS-Editor with Fujaba and the translation of the IFS-Model-Instance. Figure 5.10 presents the class diagram of the implemented model transformation.

The IFS-Editor and Fujaba exist as packages. The class FujabaHandler establishes the tool coupling in such a way that the editor can start, close, and import the current component diagram from Fujaba. The class SystemArchitecture represents those Java classes which implement the IFS-Data-Structure. The classes on the right describe the translator with the central class `Synch2Fujaba()`, which implements the Java interface Synch2IFS. The interface was defined to offer basic classes for the transformation of the Fujaba model into the IFS-Format. Therefore Synch2X4J, which creates the IFS-Object structure from the XMLElement tree when loading an IFS-Instance (XML file), also implements the interface.

FujabaTools offers features to parse and evaluate expressions in the IFS-Model-Instance like transition conditions and automata outputs. The InterfaceProtocolFactory is part of the data structure. It handles the creation of IFS interfaces from IFS protocols and vise versa.

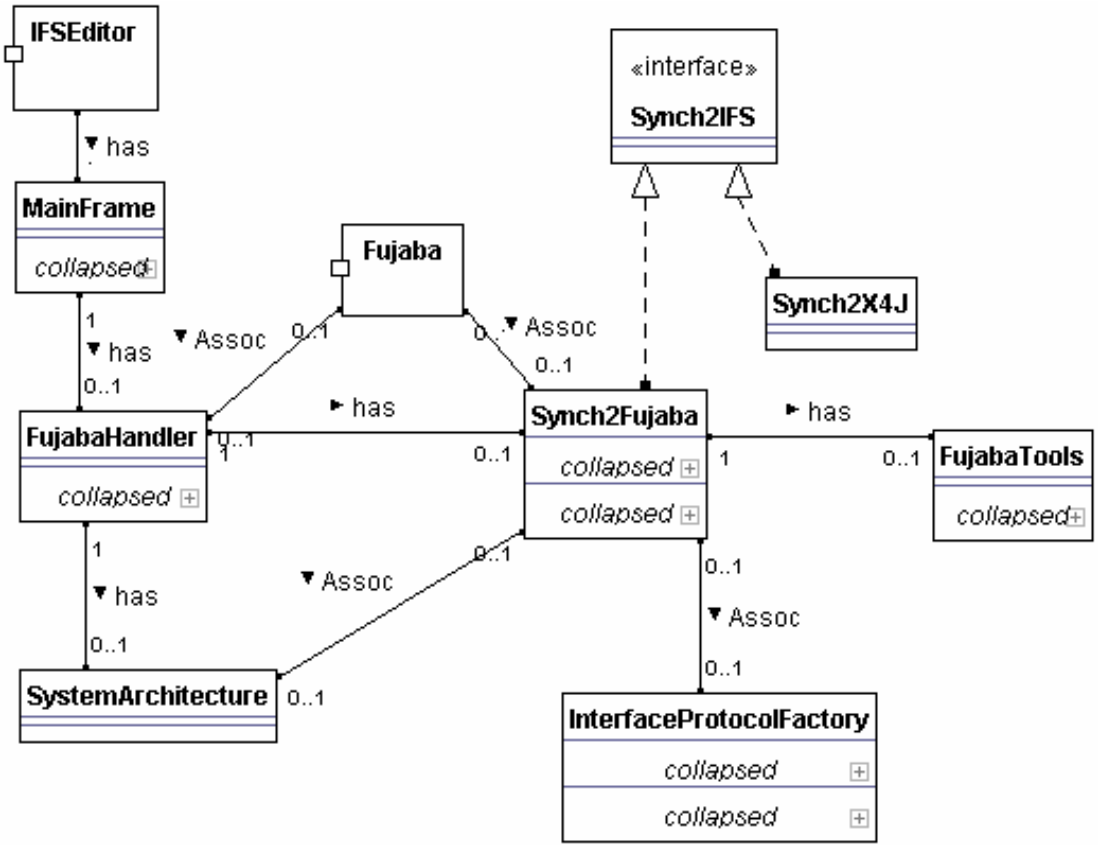


Figure 5.10: Class diagram: Tool coupling and UML model transformation

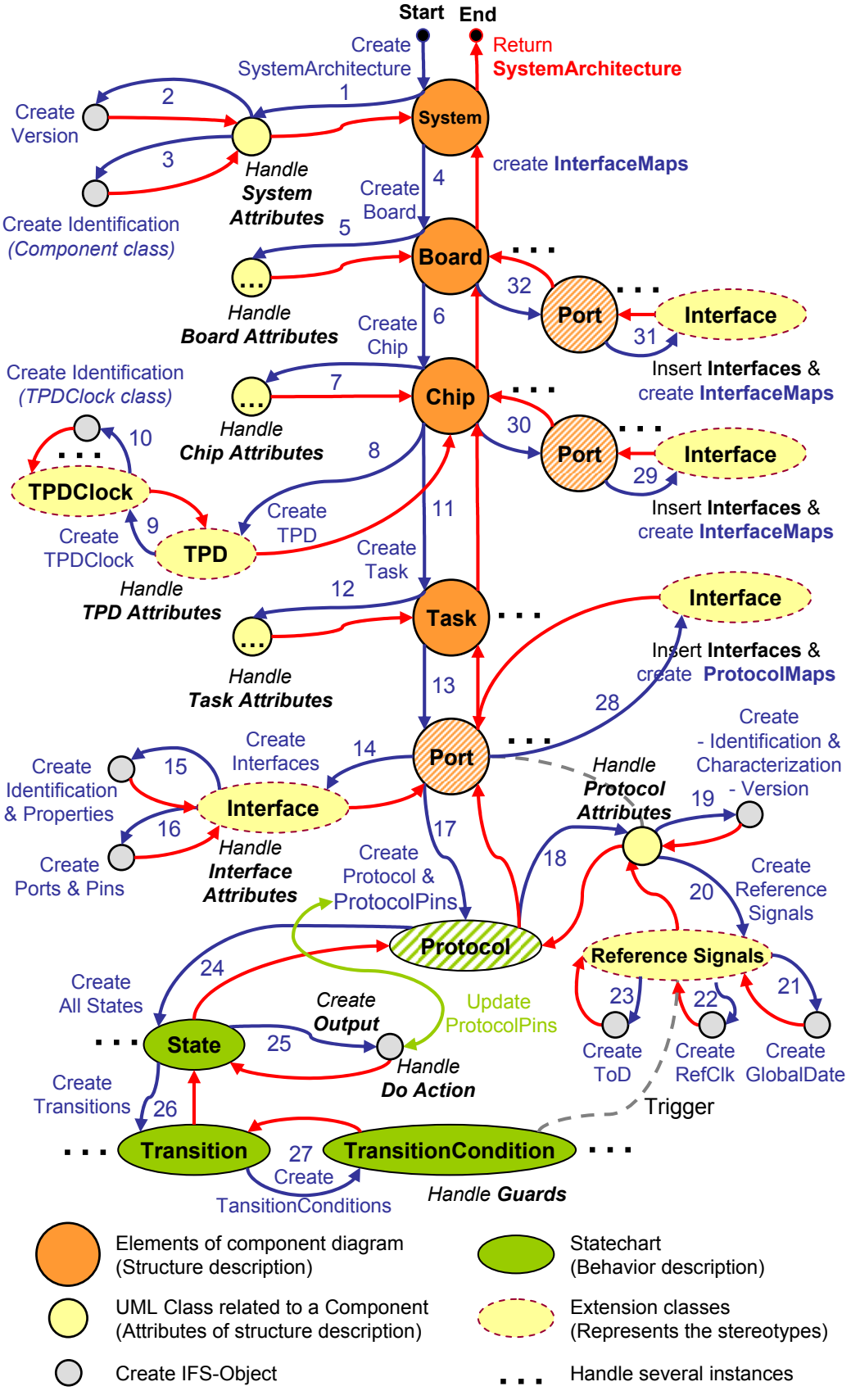


Figure 5.11: Recursive descend of UML model transformation

5.1.3 Model Transformation from UML 2.0 to Java

The developed *IFS-Model translator* converts *IFS-Model-Instances* into *IFS-Objects*. For the implementation, we resorted to well-known *compiler techniques* [44]. A typical compiler front-end consists of a lexical, a syntax, and a semantic analysis. In our approach, the lexical analysis is integrated in the `set()`-Methods of the IFS-Objects (see Section 4.2.1). This works *fault tolerantly* and ignores faulty inputs. The construction of the syntax tree is redundant, as the IFS-Model-Instances are available as object trees. A syntax - and semantic check of this tree is not necessary since the data structure is generated by the CASE tool.

The compiler back-end performs a *pattern-based model translation*. Each pattern defines a set of translation rules for specific object constellations within the IFS-Model. To find these patterns, a *parser* traverses the UML 2.0 data structure of the IFS-Model-Instance by a *recursive descent* (as illustrated in Figure 5.11). If the parser detects a known object constellation, it creates the instances of the according IFS-Objects and initializes them with the attributes it came upon. Some objects cannot be created in a single step. These objects are intermediatly stored and finished when the parser detects the missing information.

Figure 5.11 illustrates the recursive descend that we perform when `synch2Fujaba()` is called. The parser obtains the handle of the current UML-Model-Instance as a parameter. Primarily, `synch2System()` creates the IFS-Object *SystemArchitecture* if it detects the IFSComponent *System* in the UML 2.0 data structure. If this was successful, `synch2System()` recursively tries to create its subcomponents. When `synch2Fujaba()` finishes the recursive descent, it returns a reference on the resulting *IFS-Object tree* to the FujabaHandler.

To *complete* the *modeling phase*, we interlink unconnected interfaces inside the IFS-Editor, as presented in Figure 5.12. According to Section 2.1.5, we can directly connect interfaces, that are *compliant*, *connective*, and *compatible*, via interface maps. Otherwise, we have to generate an IFB. Therefore, the interfaces have to be at least (electrically) compatible. After creating the IFB, it is interlinked to the adapted interfaces via interface-maps. In the following section, we describe the IFB synthesis flow including the IFD-Mapping.

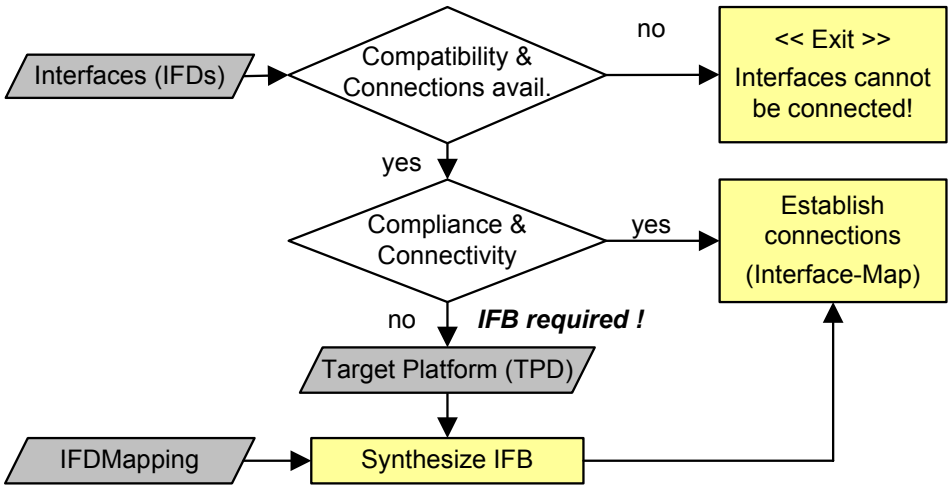


Figure 5.12: Connecting interfaces in the System Architecture

5.2 Synthesis Phase – Design Step 1: IFB Model Synthesis

Figure 5.13 delivers an overview of the *first design step* of the *synthesis phase*. The aim of this design step is to create a *target language independent model* of the IFB. This *intermediate representation* of the IFB is added to the System Architecture when the synthesis finishes successfully. Similar to other components, an IFB can be exported and imported as *IFB-IP*.

It is a specialty of our approach that we do not store the final result of the interface synthesis (the IFB intermediate representation), but the *input* of the synthesis algorithm, namely the selected IDFs, the TPD, and the IFD-Mapping. This is done for two reasons. Firstly, the amount of stored data is largely reduced. However, we have to perform the intrinsic part of the IFB synthesis each time we load an IFB (in Figure 5.13 denoted as "Create IFB"). Since the import process of XML files is also a time-consuming process (remember the API-stack presented in Figure 4.5), both effects cancel each other out more or less.

Secondly, storing an IFB as input of the synthesis facilitates the *further development* of the *synthesis algorithms*, as we can still import previously generated IFBs. The result is an updated IFB model, which is consistent with the modifications of the synthesis algorithm without any user assistance.

As depicted in Figure 5.13, the synthesis flow is composed of several consecutive processing and synthesis steps, which are grouped into *modules* (yellow rectangles). The IFS-Editor encapsulates these modules as *plugins* within the so called *Synthesis Wizard*, which guides the designer through the particular stages of this design step. Additionally, the Synthesis Wizard manages the input and the output of the particular modules (green blocks). The IFB model synthesis is divided into the following user I/O - and synthesis steps:

Interactive User I/O Steps: Manage Input and Output

Select Input: In the first step, we collect the IFDs of the adapted interfaces and the TPD of the platform that executes the IFB from the System Architecture.

Map Data: Before we create the IFD-Mapping, we have to adapt the selected input, which is part of the synthesis steps. In a first step, we flatten the IFDs and complement the specified directions. Afterwards, we assure the descriptions to satisfy our design rules. In the next step, which represents the first part of the protocol synthesis, we transform the waveform state machines into protocol state machines.

Now, we are able to specify the IFD-Mapping with the help of the IFD-Mapping Editor. Therefore, the designer interactively defines the scenario-based data processing or loads a previously created IFD-Mapping.

Define Reconfigurability: After finishing the IFD-Mapping, we specify whether to create the static or runtime reconfigurable version of the IFB. In the case of a RTR-IFB, we have to provide the appropriate Reconfiguration Control Unit.

Connect IFB : When the IFB model has been successfully generated, we interlink the IFB interfaces to the adapted interfaces inside the System Architecture by creating the necessary interface-maps. Furthermore, we define and interlink the clock and the reset signal with the IFB.

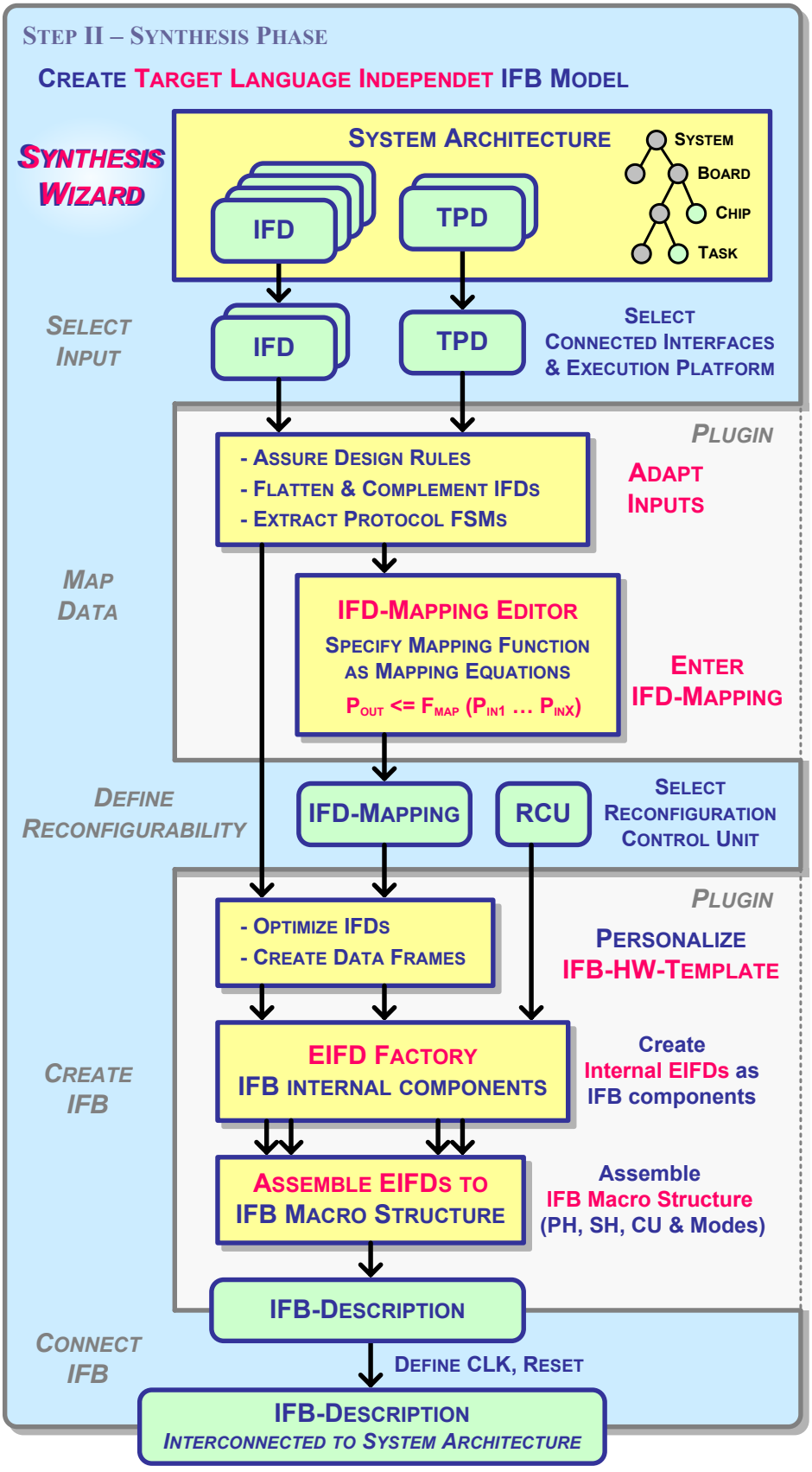


Figure 5.13: Interface Synthesis Design Flow: the IFB model synthesis phase

Automated Synthesis Steps: Construct IFB Model

Adapt Inputs: As mentioned earlier, we have to preprocess the inputs and transform the waveform state machines into the protocol state machines (protocol synthesis).

Create IFB : The aim of this step is the assembly of a target language independent IFB model, which is consistent with the IFB Macro-Structure. The model is composed of a hierarchy of *extended IFDs* (*EIFD*) which have been created by the *EIFD Factory*.

To create the EIFDs, we optimize the adapted IFDs based on the IFD-Mapping and identify the resulting data frames.

5.2.1 Prepare Synthesis Input

To improve the modeling process, we support complex expressions for the *Repetition* and *AutomataOutput* protocol attributes. We defined these expressions in a Backus-Naur form as presented afterwards. Our synthesis algorithms cannot handle these complex expressions. Therefore, we flatten them right in the beginning.

Flatten IFDs

Definition 5.1 Complex expression for the transition attribute *Repetition*

```
Repetition := [1-9][0-9]*␣*(␣*:␣*[1-9][0-9]*)?
```

The expression, which defines the *repetition* attribute of a *transition*, consists of an integer value that determines the number of repetitions optionally followed by an ID. The repetition value specifies how often the current state has to be executed before the transition to the next state is taken. Thereby, the given transition condition `TC` is applied for the *n-1* self-transitions as well as for the final transition to the succeeding state. Figure 5.14 depicts the simple case without ID : `Repetition := 2`. To flatten repetitions we insert *n-1* clones of the repeated state to the protocol in our example, $S_{1'}$.

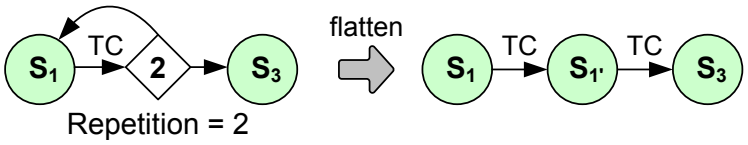


Figure 5.14: Flattening the transition attribute *repetition*

Figure 5.15 illustrates a repetition value including an ID : `Repetition := 2:1`. The ID specifies the first state of a complete sequence of repeated states. A loop, defined in such a way, must not overlap with another loop. In this case the flattening process for repetitions results in a simple *loop unrolling*, which also supports *branches* and *nested loops* inside the loop.

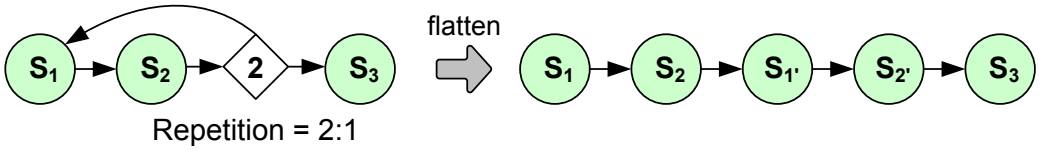


Figure 5.15: Loop unrolling for repetitions

Definition 5.2 Complex expression for the attribute *RefProtocolPin*

```
Interval := [1-9][0-9]*␣*(␣*:␣*[1-9][0-9]*)?

AutomataOutput := Interval(␣*,␣*Interval)*
```

With the help of the complex attribute "*RefProtocolPin*" (inside an AutomataOutput) that refers to those ProtocolPins which are defined by this AutomataOutput, we can specify a particular value for a complete set of ProtocolPins by listing up the referenced ProtocolPin IDs. Some examples for valid expressions are: (`2`), (`1:3`), (`5:  2, 5, 6:8`). The complex terms are comma separated enumerations of ranges of ProtocolPin IDs. To flatten this attribute, the algorithm creates one automata output for each referenced ProtocolPin.

Assure Design Rules

After flattening the IFDs, we assure important *design rules* to enable a successful synthesis. Therefore, we analyze the IFDs and the TPD. Afterwards, one can find a list containing the most relevant actions, performed within this process:

- Remove unreferenced or incompletely specified Pins & ProtocolPins
- Remove dead states
- Remove invalid AutomataOutputs & Transitions
- Remove unreferenced TPD clocks

Our motivation for shrinking the descriptions is for avoiding synthesis errors and for minimizing the number of resources that we allocate for the IFB. For example, it would be a waste of resources to create signals for unreferenced pins that are never written or read.

An additional feature updates the object identifiers to a VHDL compliant notation. For example, the signal name "`My Ack`" would be changed into `My_Ack`. Otherwise, the generated VHDL code would imply syntactical errors from the beginning.

Complement IFDs

The IFB operates as a communication partner of the tasks. Since the designer models the waveform state machine inside the IFD from the viewpoint of the task, we have to *invert all directions* to create the protocol state machines inside the IFB:

- Input $\Longrightarrow$ Output
- Output $\Longrightarrow$ Input
- Bidirectional $\Longrightarrow$ Bidirectional

In our protocol synthesis algorithms we apply *specific data structures* that abstract from the IFDs. A fundamental understanding of these data structures is essential for comprehending the developed synthesis algorithms. First of all, we can subdivide each protocol, defined as waveform state machine, in a set of non-branching state sequences, the *basic blocks*. The *protocol matrix* represents each basic block in the form of a *two dimensional matrix*. This internal protocol representation is input for the protocol synthesis and allows us to detect *packages* and *frames*, which are essential to create and evaluate the IFD-Mapping.

5.2.2 Basic Blocks

The IFDs utilize waveform state machines to describe behavior. We can visualize the implied protocols as *directed graphs*. In these graphs, the vertices represent the states and the directed edges stand for the state transitions. In this thesis, we define basic blocks similarly to the well-known pendant in software.

Definition 5.3 *Basic Block (BB)*

A *basic block* is an ordered set of states that are on a path of maximum length in a protocol graph. On this path, only the first and the last vertex can have a degree higher than one. The graph representation of a waveform state machine in the form of BBs results in a *Control Flow Graph*.

Definition 5.4 *Control Flow Graph (CFG)*

$CFG = \{V; E\}$
V = Basic Blocks
$E \subseteq V \times V$, a set of edges (= *transitions*) between BBs.

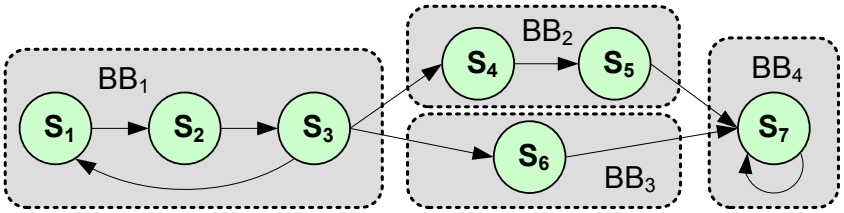


Figure 5.16: The basic blocks of a protocol visualized as CFG.

Figure 5.16 represents a CFG that consists of the four basic blocks BB_1 ... BB_4. The CFG was created from a protocol with the states S_1 to S_7. A detailed example for the creation of BBs is given in Figure A.2. In our design tool, we implemented a *protocol parser* that automatically determines the basic blocks of a given protocol and returns the constructed CFG. A *protocol viewer* allows the visualization of protocols, in the form of waveform diagrams, based on valid traces of basic blocks in the CFG. Furthermore, the basic blocks provide the foundation for computing packages and frames.

5.2.3 Protocol Matrix and Protocol Packages

Protocol packages play a central role in the Interface Synthesis Design Flow, especially for the IFD-Mapping. The *mapping functions* are defined on *data packages*, which exist next to the *control packages*. Packages abstract from the structure of the protocol and the basic blocks in order to facilitate an intuitive specification of the data mapping.

Afterwards we show that basic blocks are adequate for detecting protocol packages, since packages do not proceed the borders of a basic block. This has to do with the fact that data words do not begin in one basic block and continue in a succeeding one [37]. A consequence of this property is that we use basic blocks as the basic data structure for the IFB synthesis. Nevertheless, basic blocks are not an efficient data structure for the detection of packages.

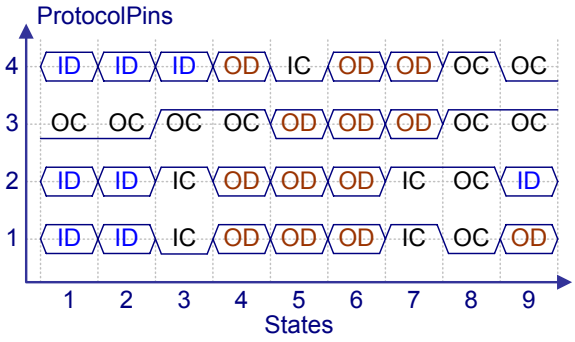


Figure 5.17: Annotated waveform diagram

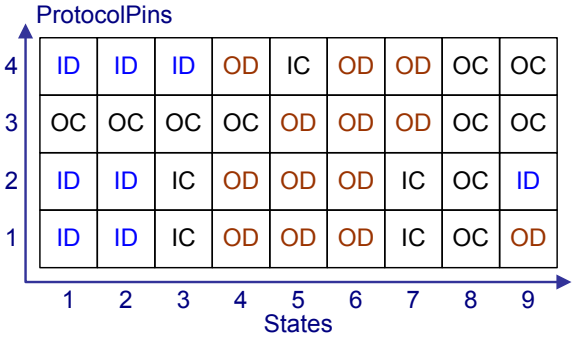


Figure 5.18: Derived protocol matrix

Therefore, we derive a *protocol matrix* from each basic block. Similar to waveform diagrams, each column of the matrix represents a state of a basic block and each row stands for a particular ProtocolPin. Figure 5.17 shows a fictive waveform diagram including the annotations for the attributes *Direction* and *UseCase*: $\{\, I\text{ncoming}, O\text{utgoing} \,\} \times \{\, D\text{ata}, C\text{ontrol} \,\}$.

The deduced protocol matrix is depicted in Figure 5.18. Each *protocol matrix node* (PMN) comprises the complete information that is defined by the AutomataOutput for the current state and the related ProtocolPin: {Direction, UseCase, Value}. In the depicted version of the protocol matrix, we display the attributes *Direction* × *UseCase*. We call this the *Direction-UseCase Matrix*. Next to this, we require the *Value Matrix* for the protocol synthesis that illustrates the attribute *Value*. Remember, both derivations are one and the same protocol matrix where we depict different parameters, which is helpful for the graphical explanation of our algorithms.

In the IFS terminology, *protocol packages* (or simply *packages*) consist of homogenous bits which are adjacent PMNs in the protocol matrix. Thereby, packages can be two dimensional. One dimension refers to the number of ProtocolPins, which specifies the amount of bits that are transmitted in parallel. The other dimension is the number of succeeding states that indicates the length of a package. However, the length of a packet is not equal to the quantity of sequentially transmitted bits, since one bit may be pending for several states. To find out the real number of data bits is a challenge of the protocol synthesis algorithm.

In the Direction-UseCase Matrix, each package is a *rectangular field* of the *maximum size* consisting of *homogenous protocol matrix nodes*. Figure 5.19 illustrates the data packages of our exemplary protocol matrix. As we can see, a package can also be a single PMN.

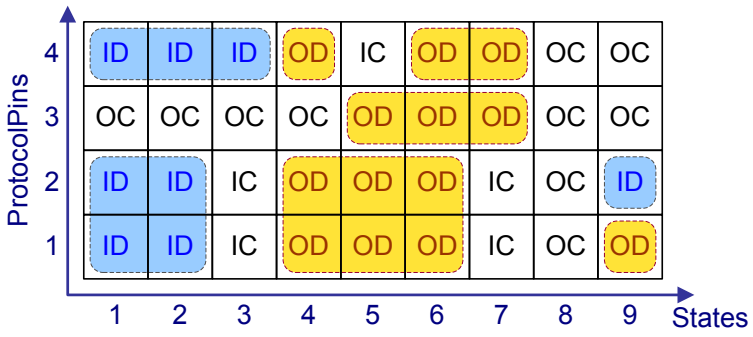


Figure 5.19: Protocol packages of protocol matrix

The algorithm for detecting protocol packages is quite simple. In the first step, it analyzes the Direction-UseCase Matrix line-by-line for *horizontally adjacent* and *homogenous PMNs*. It creates packages of a height one and the respective maximum width. In the second step, the algorithm merges the one-dimensional packages into two-dimensional packages by affiliating homogenous packages of the same width and horizontal position (states). For more details on basic blocks, protocol packages, and the underlying protocol parser see [34, 1, 37].

As presented in Figure 5.19, data packages can overlap themselves by their covered states. To handle these *parallel packages*, we introduce another data structure, the *protocol frame*.

5.2.4 Protocol Frames

The IFB internal communication between PH and SH provides exactly one memory bus for incoming and one for outgoing data. Each bus can serve at most one sender/receiver at each point of time. Therefore, we encapsulate *parallel packages* that overlap in the state space in *protocol frames* (simply called *frames*). The *memory bus* can be requested exclusively for reading or writing data in the form of *complete frames*. For this reason, each frame comprises a unique ID that we utilize for assigning the encapsulated data packages to a dedicated memory location inside the internal memory. In this way, we ensure the continuous transmission of parallel packages between PH and SH.

Each frame encapsulates a *maximum set* of *overlapping data packages*. In contrast to the packages, frames are one-dimensional and must not overlap with other frames. By merging overlapping packages into frames, each frame defines an interval of states that does not overlap with the interval of another frame. A state that is not covered by a data package does not become part of a frame, either. As consequence of the packages, frames are confined to exist inside of a basic block.

Figure 5.20 illustrates the frames that result from the packages of our exemplary protocol matrix: *Frame 1* for incoming data, *Frame 2* for outgoing data, and *Frame 3* being a mixed frame. The states 1 – 3 hold two overlapping incoming data packages which are merged to *Frame 1*. As no other packages cover these states, there are no more packages in this frame. The second frame, *Frame 2*, comprises four outgoing data packages. As we can see in this example, it is not necessary for all packages inside of one frame to share one collective state; it is sufficient if there exists pairs of at least two data packages, each with a shared state.

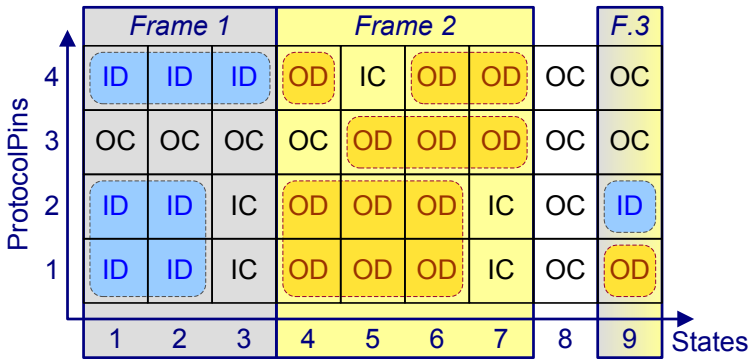


Figure 5.20: Protocol packages merged to protocol frames

Pure vs. Mixed Frames

As presented in Figure 5.20, we distinguish between two kinds of frames: the *pure frames*, which comprise either incoming or outgoing data packages (*Frame 1*, *Frame 2*) and *mixed frames*, which comprehend both kinds of packages (*Frame 3*). Pure frames are sufficient for *simplex communication*, while we have to permit mixed frames for *duplex communication*.

Our methodology is able to cope with both variants. However, in the current IFB hardware implementation, we restricted ourselves to pure frames in order to simplify the synthesis process and the resulting IFB design. Currently, each PH-Mode consists of a set of states and signals for communication with the PH-Switch and the memory bus arbiter inside the CU. These IFB internal communication resources, including the control overhead within the memory bus arbiter, would have to be doubled to permit duplex traffic.

Super-Frames

With the help of the IFD-Mapping, we map incoming data to outgoing data. To be more precise, we map incoming data packages to outgoing packages. To cope with packages of a *different capability* (= *height* · *width* of a data package) we support *package instances*. For example, it is possible to receive a parallel 32-bit word that has to be transmitted in four serial 8-bit packages. This necessitates a repeated transmission of the outgoing frame that comprises the 8-bit package, which again implies the repeated execution of the basic block that contains the outgoing frame. The same problem exists when several instances of small incoming data packages are assembled into a large outgoing data package.

As depicted in Figure 5.21, the *repeated execution* of input or output frames *disagrees* with the *pipelined execution* of the communication cycle (*I - P - O graph*). To handle multiple input or output frame instances, we create a *super-frame* which encapsulates as many repetitions of the related basic block as defined by the IFD-Mapping. To construct a super-frame, we *flatten* the repeated basic blocks into a linear sequence of states. Therefore, the related basic blocks have to provide a *self transition*. As we can flatten only complete basic blocks into super-frames, multiple packages can be affected as result of the repetition of a single package. These packages have to be considered in the IFD-Mapping as well to not to loose any data. For more details about frames and super-frames see [1, 26, 34].

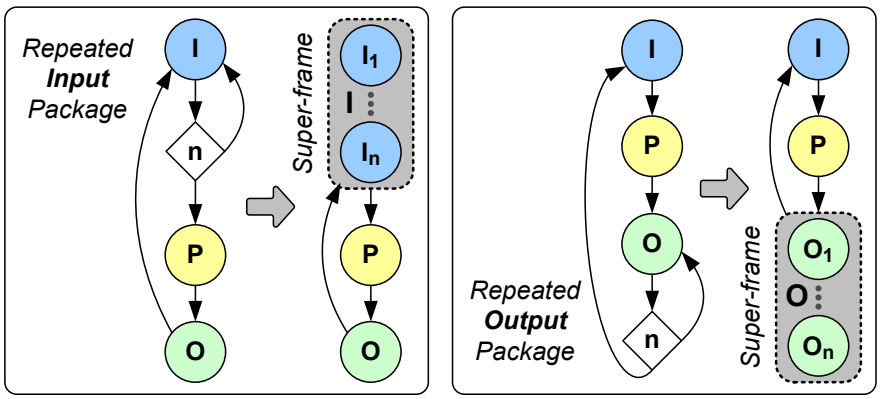


Figure 5.21: Generation of incoming and outgoing protocol super-frames

5.2.5 Protocol Synthesis – Generation of the Protocol State Machines

A PH-Mode acts as the *opposite communication partner* of a component interface. Therefore, the IFB behavior has to be *complemented* to the one of the interface. The *protocol synthesis* takes the flattened and complemented IFDs as input to generate the behavior model of the PH-Modes. This model is a finite state machine, the so called *protocol state machine* (*PSM*). To construct the PSM, we transform the *waveform state machine* (*WSM*), given in the IFD, in such a way that it implements the *complemented communication protocol*.

The WSM, depicted on the left side of Figure 5.22, specifies the communication protocol for two or more interacting entities as defined in Section 2.1.4. Thereby, the WSM represents the *complete waveform diagram* of the communication protocol from the *viewpoint of the component interface* (indicated by the attribute *direction*). To adopt the viewpoint of the IFB, we invert the directions as presented in the previous synthesis step.

To generate the PSM of a PH-Mode it is *not necessary* to know the *complete PSM* of the component interface. For example, it is not required for the IFB to consider the signal that starts a serial transmission; it is sufficient to recognize the *observable* result: a change on the transmit data signal. However, the *limitation* to *reactions on observable signal events*, which are modeled by the AutomataOutputs of the WSM, restricts the possible PSMs that we can synthesize. The timing, which is a non observable attribute, constitutes the only exception since it is specified within the transition conditions of the WSM.

Nevertheless, as there is no chance to access IP internal signals, the limitation to observable signals is crucial for an IP based interface synthesis approach. Therefore, the created PSM has to get by with the observable signals between the IFB and the component interface. For this reason, the WSM model comprehends exactly that subset of the interacting protocol state machines, which causes *observable effects* on the shared signals. To derive the PSM from the WSM we have to perform the following synthesis steps:

1) Identify the *states* that belong to the PSM
2) Create *signal based transition conditions*
3) Identify the particular *data bits*

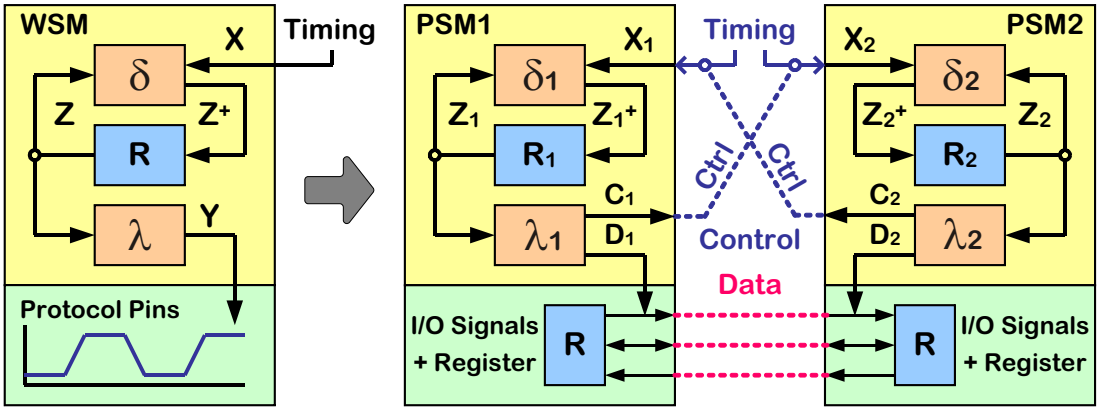


Figure 5.22: Transformation of the WSM into the PSM

Definition of Protocol State Machine

Before explaining the three synthesis steps, we have to define the *formal model* of the *protocol state machine*. For details of the terms used in the PSM definition (S, TC, PP, δ, and λ), we refer to the definition of the WSM (see Definition 2.5):

Definition 5.5 *Protocol State Machine (PSM)*

$Protocol := (S, TC, PP, \delta, \lambda, s_0)$

S	:	A finite number of states
TC	:	A set of *transition conditions*
PP	:	A set of *protocol pins*
δ	$: S \times TC \rightarrow S$	The *state transition function*
λ	$: S \rightarrow (Val \mapsto PP)$	The *(Moore) output function*
s_0	$\in S$	The *start state*

Obviously, the PSM model consists of the same elements as the WSM model. However, the meaning of the PSM model is different from the one of the WSM. As depicted on the right side of Figure 5.22, a PSM describes a *control path* (blue) and a *data path* (red). Since the WSM comprises states which are related to multiple communication partners, it is one challenge of the protocol synthesis algorithm to find out the *states* that belong to the *control path* of the PSM which implements the PH-Mode.

In the WSM model we utilize the AutomataOutputs to define the values of ProtocolPins related to specific states as representation of the waveform diagram. The supported values are (In, Out) × (Control, Data) × (0, 1, Z, X). To create the PSM, we interpret the given outputs in step 2) and 3) of the protocol synthesis algorithm according to the table presented in Figure 5.23. We apply the control signals to specify the *transition conditions* (TC) of the control path. As illustrated in Figure 5.22, the *outgoing control* signals (C) remain an *AutomataOutput* in λ, while *incoming control* signals are transformed into TCs in δ. It is reasonable to evaluate only constant values in a TC. Terefore, dynamic values are forbidden.

The data signals model the data path (D) in the form of AutomataOutputs in λ. Dynamic incoming data is written into the IFB internal memory in the SH; dynamic outgoing data is read from this memory. Since the transmission of incoming data constants is redundant, we ignore them. Outgoing data constants are directly integrated into the PH-Mode.

		Control	Data
In	Constant (0, 1, Z)	Transition Conditions	< *Redundant* > *(ignored)*
	Dynamic (X)	< *Forbidden* >	Automata Outputs (Dynamic values written to SH)
Out	Constant (0, 1, Z)	Automata Outputs (Fixed values in PH-Mode)	Automata Outputs (Fixed values in PH-Mode)
	Dynamic (X)	< *Forbidden* >	Automata Outputs (Dynamic values read from SH)

Figure 5.23: Interpretation of automata outputs in the PSM

Protocol Synthesis Step 1) : Identify States

As we transform the WSM into the PSM, the states of the PSM are a subset of the WSM: $S_{PSM} \subseteq S_{WSM}$. There exist three possible cases for a state belonging to the PSM:

I) A state belongs to the PSM when it comprises an *AutomataOutput* that *changes the value* of an *outgoing signal* (independent of control or data). This means, a state s of the WSM belongs to the PSM when there exists at least one pair of AutomataOutputs related to s and its predecessor state s_p that assigns a differing value to one and the same ProtocolPin, while the direction in state s is *output*.

II) The state s has to be present in the PSM if there exists a timed transition in s.

III) Furthermore, we have to consider a state in the PSM when all *AutomataOutputs* of the current state s are pairwise *equal* to the predecessor state s_p, and at least one *ProtocolPin* provides *incoming data*. We defined this case, which is not covered by the two previous conditions, to model *separate states* for *reading data*.

Based on these rules we can identify the states that belong to the PSM. In a succeeding synthesis step, where we create the final state machine of the PH-Mode, we insert additional states to the PSM which secure the IFB internal communication.

To explain the *state detection algorithm* we refer to the parallel data transfer protocol presented in Figure 2.4. In Figure 5.24 we depicted the representation of this protocol as Δ Direction-UseCase Matrix and Δ Value Matrix. Note that we split the fourth state into the two states "Read Data" and "Set DA". For a better overview, we sort the incoming signals to the top and the outgoing signals to the bottom of the matrices.

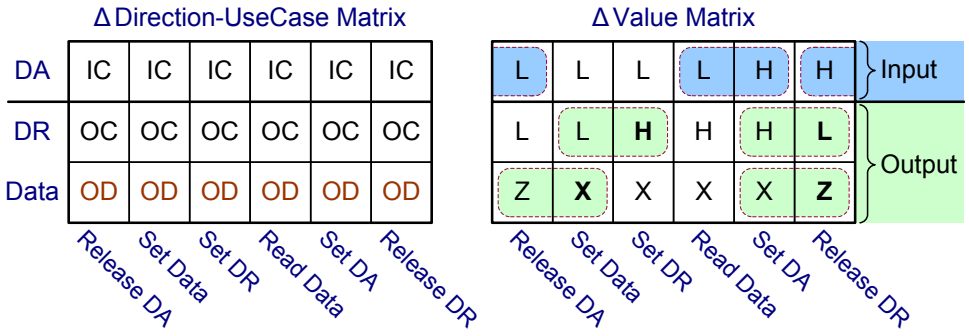


Figure 5.24: Δ Direction-UseCase Matrix and Δ Value Matrix of parallel sender

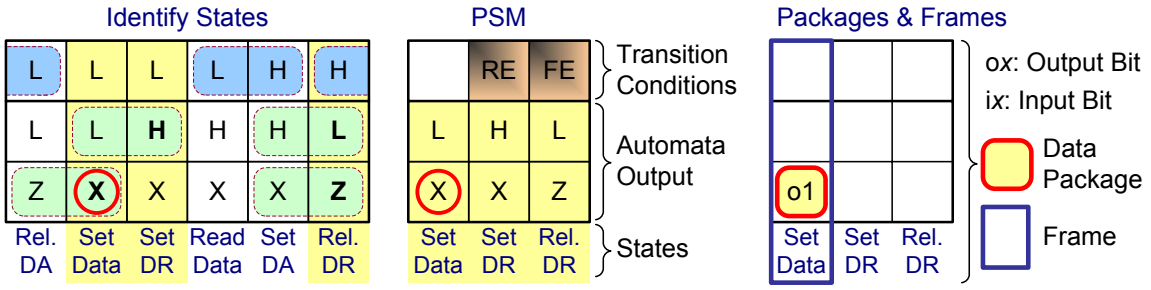


Figure 5.25: Detect states, packages, and frames of parallel sender

To identify the states that fulfill the first condition, we search for unequal horizontally adjacent nodes in the Value Matrix. In this way we obtain the Δ Value Matrix as depicted in Figure 5.24. The *green boxes* visualize the available pairs of unequal outgoing PMNs. Each state that belongs to the *right PMN* of such a green box (bold font) becomes a state of the PSM as highlighted on the left side of Figure 5.25.

In our example – in which the IFB acts as a parallel sender – we identify the states `Set Data`, `Set DR` and `Release DR`. To consider all states, the algorithm compares also the last state of a basic block with the first state of all succeeding basic blocks. In this case, a basic block provides a self-transition, this means comparing the last with the first state of the basic block. In this way we detected the Δ for the `DA` signal covering the first and the last state.

Based on the parallel protocol we also can explain the third condition to consider a state in the PSM. Therefore, the IFB has to act as parallel receiver as presented in Figure 5.26 and Figure 5.27. As we will see, reversing the direction results in the inverse set of states. With the help of the first condition we identify the states `Release DA` and `Set DA`. In addition, the state `Read Data` and its previous state `Set DR` also satisfy the third condition: Both columns comprise pairwise equal PMNs (dotted rectangle in the Δ Value Matrix) and own at least one ProtocolPin that is incoming data (dotted shape in the Δ Direction-UseCase Matrix). Therefore, we add also the state `Read Data` to the PSM. As we will see in the third protocol synthesis step, we read incoming data within this state.

To illustrate the second condition we switch to the serial *RS 232* protocol. In contrast to parallel protocols, we can also find $\Delta's$ in the Δ Direction-UseCase Matrix for serial protocols, as particular ProtocolPins act as control- as well as data signal.

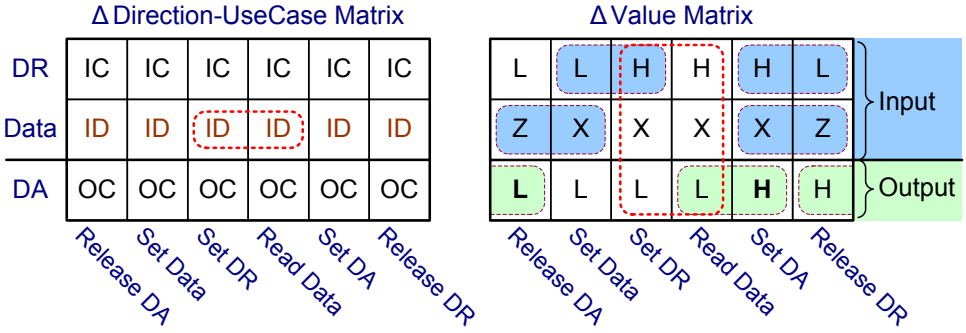


Figure 5.26: Δ Direction-UseCase Matrix and Δ Value Matrix of parallel receiver

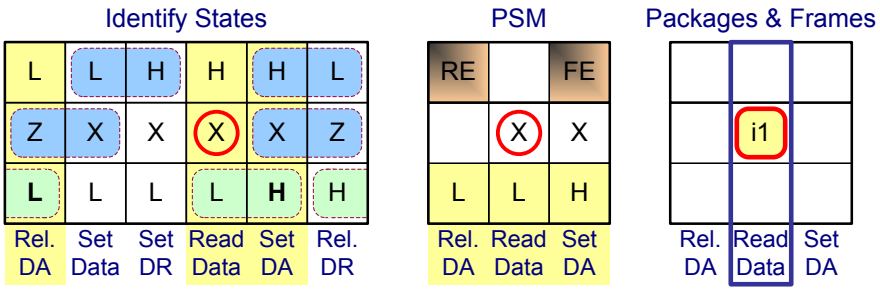


Figure 5.27: Detect states, packages, and frames of parallel receiver

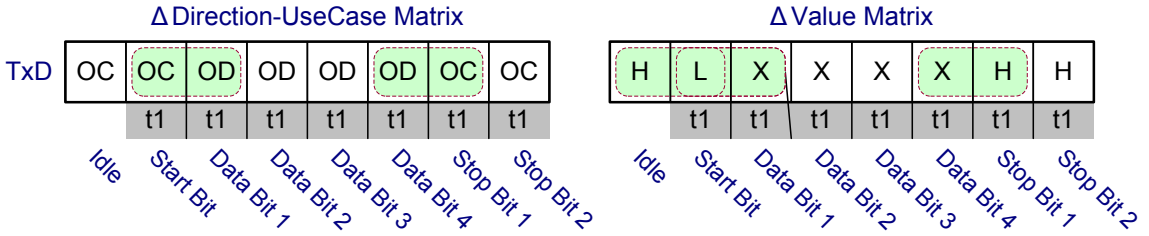


Figure 5.28: Δ Direction-UseCase Matrix and Δ Value Matrix of serial sender

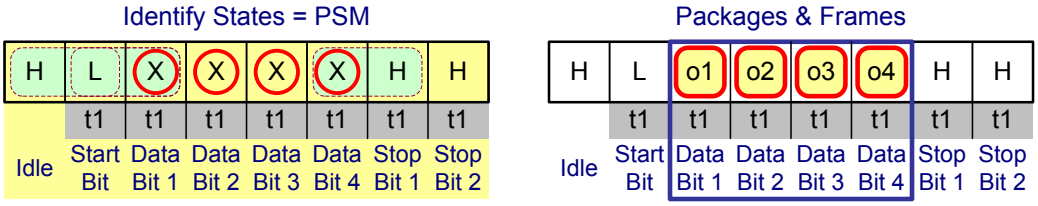


Figure 5.29: Detect states, packages, and frames of serial sender

The RS 232 protocol presented in Figure 5.28 and Figure 5.29 comprises 1 start bit, 4 data bits, and 2 stop bits next to an initial idle bit. Each state except of the *idle* state consists of a timed transition (timer $t1$), which has been derived from the WSM (see Section 2.1.4). Therefore, all of these states belong to the PSM in order to reproduce the given timing.

Furthermore, we include the first state of the WSM as an exception of the first condition. If the *start state* of the WSM assigns outgoing signal values to at least one ProtocolPin, it has to be added to the PSM to guarantee a correct initialization of the protocol. In the case of the RS 232 protocol, the first state would be deleted otherwise, as it is not required for the repeated execution of the protocol. However, for the proper start of the PH-Mode it is necessary to include this state.

Protocol Synthesis Step 2) : Create Signal Based Transition Conditions

As presented in our UML 2.0 profile, we know *five* different kinds of *transition conditions*. Four of them (TPD clock + 3 types of reference signals: TimerOrDeadline, ReferenceClock, and GlobalDate) are explicitly modeled in the WSM. Therefore, the second step of the protocol synthesis algorithm is to cope with the *signal based transition conditions*. These transition conditions represent those *signal dependencies* that we can derive automatically from *observable events* on the *ProtocolPins*. In timing diagrams, we usually model such dependencies in the form of arrows as visualized in Figure 5.30. To recognize the *observable signal dependencies*, we analyze the *existing AutomataOutputs* of the related *WSM*.

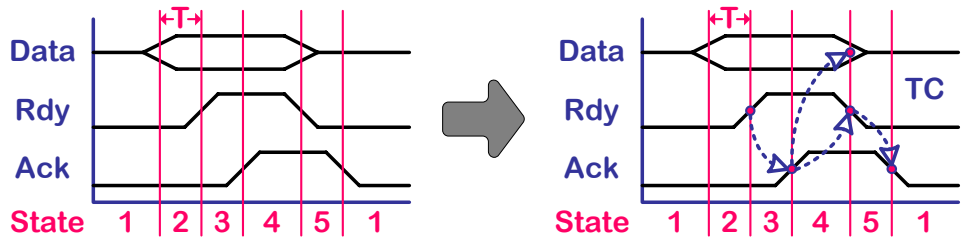


Figure 5.30: Signal dependencies of the waveform diagram

To create the signal based transition conditions (TC) for the current state $S_{curr} \in PSM$, we interpret the *changes* on *incoming control signals* $\in WSM$, which occur between S_{curr} and the succeeding state $S_{succ} \in PSM$, as the *observable reasons* for the *signal based TCs*. A drawback using this approach is that we cannot exactly detect the *proper reasons* for a state change. However, by converting *all changes* on *all incoming control signals* within the above mentioned interval into TCs, the actual (observable) causes are *always included*. Therefore, it is satisfying to apply all created TCs (without selecting specific ones) as we derive all of them directly from the WSM, which specifies the executed protocol.

To create signal based TCs, the protocol synthesis analyzes the Δ Value Matrix for $\Delta's$ on incoming control (`IC`) signals (in comparison to the state recognition procedure), which are then transformed into TCs and added to the transition list of the related state. The change from low to high on the `DR` signal in the 4th – 5th state in Figure 5.25, for example, is transformed into a TC that triggers on a rising edge (RE) in the state `Set DR`.

If we identify a *sequence* of *incoming control* $\Delta's$ related to one and the same ProtocolPin, which indicates multiple sequential transition conditions, we have two possibilities: either we opt for the *last event* or we assume the complete sequence to be the cause for the state change. In the second case, we have to add additional states to the PSM to reconstruct the complete sequence of transition conditions. An example for a sequence of incoming control signals is the port knocking technique used in the firewall domain.

Protocol Synthesis Step 3): Identify Data Bits

The data bits included in the protocol matrix carry the value X. Depending on the protocol, one data bit can be pending for several states. Therefore, the third part of the protocol synthesis has to identify the $X's$ which represent discrete data bits.

In general, *a continuous sequence of* $X's$ is interpreted as *one and the same data bit*. Two data bits are either separated by a state that assigns a *value different from* X to the related ProtocolPin, or the state that holds a data bit provides a *timed transition*. Figure 5.25 shows an example for a data bit that covers four states in the WSM. As the sequence of the four $X's$ is neither separated by a differing value nor covers a state that comprises a timed transition, we assume these four PMNs to represent only one data bit.

In the case of outgoing data, the *first state of a sequence of* $X's$ represents the data bit, since outgoing data must be initialized in the first state. As we will always detect an outgoing Δ in the first state of a sequence of $X's$, this state becomes definitely part of the PSM (guaranteed by the first condition). Figure 5.25 illustrates the detected data bit in the state `Set Data` for the parallel sender, including the resulting protocol package and protocol frame. The outgoing data bits in Figure 5.29 model a large protocol package that consists of four bits, as each of the states provides a timed transition.

As mentioned before, we can specify a separate state that is recognized to read incoming data (see incoming data package in the state `Read Data` in Figure 5.27). If such a state is not found for an incoming data sequence we act similarly to outgoing data and decide for the first state that belongs to the PSM. To mark the correct data bit in the PSM, we invalidate all other data bits of the sequence in such a way that they are ignored in the following IFB synthesis process. For this reason, we have to write or read data in each state of the PSM that comprises a (remaining) AutomataOutput for outgoing- or incoming data.

5.2.6 IFD-Mapping

Based on the generated PSM we are now able to explain the *IFD-Mapping* that defines the *scenario based data processing* inside the IFB. Therefore, we extract the data packages from the PSM, which works similar to the identification of the packages in the WSM model. This is possible as the WSM- and the PSM model are equivalent in respect to this aspect.

The IFD-Mapping does not define "classical" *protocol semantics* in the form of a grammar based language (that specifies terminal symbols and so on), which are applied to map items between the adapted protocols. As already mentioned in Section 4.1.2, it provides a set of *data processing operations*, which are formulated as *mapping functions*. In general, a mapping function affords the complex assignment of incoming to outgoing data:

$$\textit{Mapping Function}: \; dataOut <= f_{Map}\,(\,dataIn_1, \ldots, dataIn_k\,)$$

As shown before, the processed data is always encapsulated by *data packages*. Thereby, the identified packages of the adapted interfaces result in two sets: the outgoing- and the incoming packages. With respect to these two groups we refine our mapping functions:

$$\textit{Mapping Function}: \; P_{Out} <= f_{Map}\,(\,P_{in}1, \ldots, P_{in}k\,)$$

On the left side of Figure 5.31 we provide a graphical representation that visualizes this definition of the mapping function. Afterwards, we customize this graphical template to illustrate the different processing operations. As presented in Section 4.1.2 the mapping function f_{Map} allows us to model four basic *data processing operations*:

1) Assign constant values
2) Shuffle incoming data
3) Guarded data assignment & Boolean equations
4) Complex data processing including a data processing library → FSM

The first processing operation assigns constant values to an outgoing package. The second one supports the arbitrary mapping of incoming data bits to outgoing data bits (shuffle data). Furthermore, our mapping allows guarded assignments and the application of boolean equations on incoming data bits. To also handle complex behavior, the designer can model linear bounded automata to process incoming bits.

With respect to the processing operations 1) – 3), f_{Map} represents a "classical" mathematical function, for example, a boolean function. Because of the application of state machines in 4), f_{Map} can also consist of automata functions.

To express the IFD-Mapping, we developed the *IFD-Mapping Language*. This language is based on a *context-free grammar* [146] (Type-2 in the Chomsky hierarchy). In Section A.3 we depict the complete grammar in an *EBNF notation*. The IFD-Mapping Language defines the syntax of valid IFD-Mapping instances that the system architect can specify.

Afterwards, we explain the four presented mapping operations by representative examples of the IFD-Mapping Language. A complete documentation and explanation of the grammar and the IFD-Mapping Language is presented in [34].

IFD-Mapping: Data- and Constant Assignment, Boolean Equations

In Chapter 7 we depict a real IFD-Mapping created by the IFS-EDITOR. However, in the following examples we refer to a simple scenario including two task interfaces: we map one incoming- onto one outgoing data package. Therefore, Figure 5.31 depicts an incoming data package resulting from a parallel sender, which delivers a burst of 4×4 bits, and an outgoing package related to a serial receiver with a capability of 8 sequential data bits.

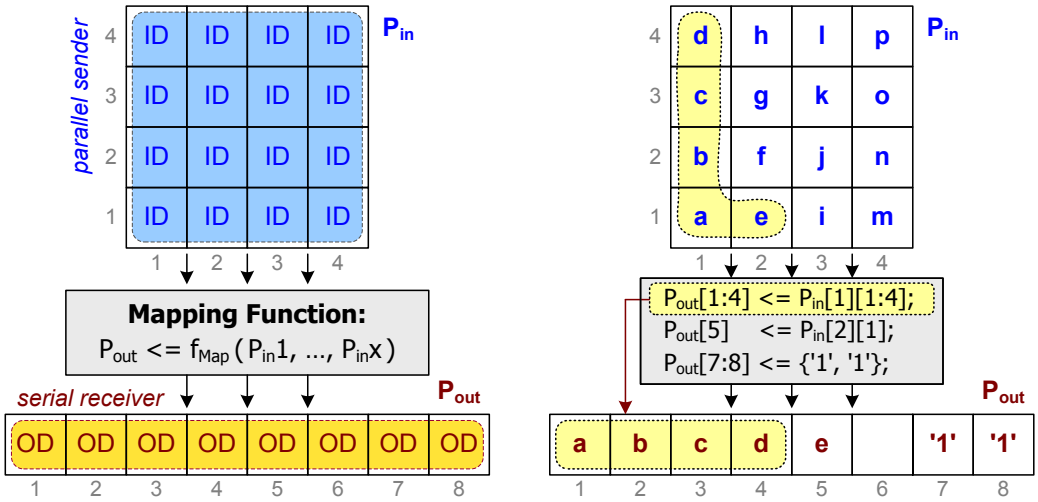


Figure 5.31: IFD-Mapping: Assignment of data bits

An exemplary mapping function for the *assignment* of incoming to outgoing bits is depicted on the right side of Figure 5.31. For a better readability we use an abbreviated form of the package identifiers: we write "P_{out}" instead of "OP_1" and P_{in} instead of "IP_1":

1) $P_{out}[5] <= P_{in}[2][1];$

2) $P_{out}[1{:}4] <= P_{in}[1][1{:}4];$

The package identifiers denote the incoming and outgoing data packages. As the packages are nothing more than two-dimensional matrices (remember the internal representation as *protocol matrix*), the notation of the IFD-Mapping Language was motivated by MatLab[1]. For this reason, we specify a subset of the protocol matrix in the form of squared brackets. The first bracket refers to the column (state) of a package. Thereby, we can address a single column ($P_{out}[5]$) or an interval of columns ($P_{out}[1{:}4]$). When a package covers multiple signals (ProtocolPins), the row can be selected similar to the column by a second squared bracket. The intersection of the specified row- and column interval defines the selected bits (protocol matrix nodes) of the data package.

Mapping function 1) maps a single bit of P_{in} (column 2, row 1) to P_{out} (column 5, row 1). If there is no row specified explicitly, as given in $P_{out}[5]$, the vertical range is automatically set to "all rows". Therefore, in our example $P_{out}[5]$ is equal to $P_{out}[5][1]$. Mapping function 2) assigns four bits. A more elegant notation to assign the five bits of 1) and 2) is based on the *concatenation* ("+") of bits: $P_{out}[1{:}5] <= P_{in}[1][1{:}4] + P_{in}[2][1]$.

[1] MatLab® is a programming language for the scientific computation, specialized on linear algebra.

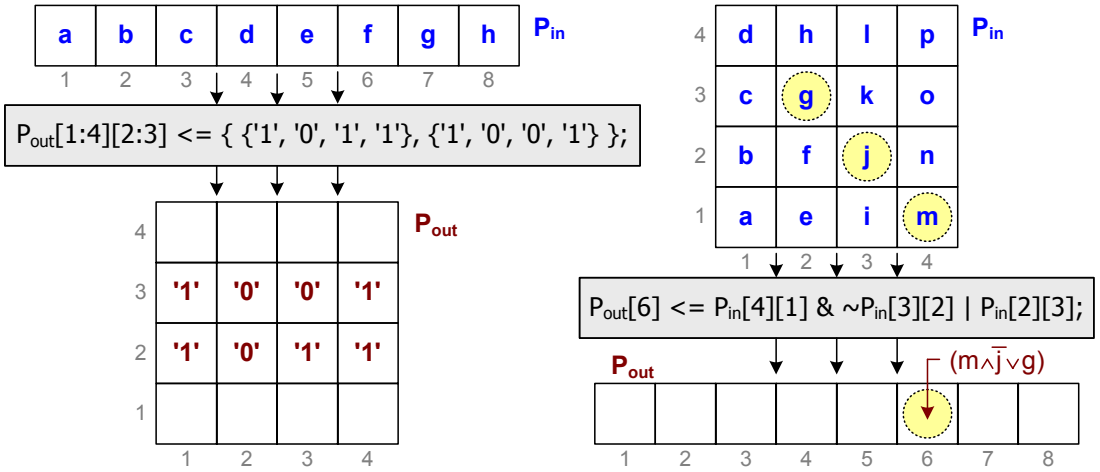


Figure 5.32: IFD-Mapping: Boolean equations and constants

In mapping function 3) we assign two constant values to the last two bits of the outgoing data package P_{out}:

3) P_{out}[7:8] <= {'1', '1'}; //≡ (P_{out}[7] <= '1'; P_{out}[8] ⇐ '1');

Instead of assigning individual constants, we can specify two-dimensional constant arrays, which comprise comma separated constants encapsulated by braces, as visualized on the left side of Figure 5.32. Mapping function 4), which derives from this figure, assigns constant values to the columns 1 – 4 of the rows 2 – 3:

4) P_{out}[1:4][2:3] ⇐ { {'1','0','1','1'}, {'1','0','0','1'} };

5) P_{out}[6][1] <= P_{in}[4][1] & ~P_{in}[3][2] | P_{in}[2][3];

An example for the application of boolean equations is given in mapping function 5), depicted on the right side of Figure 5.32. The sixth bit of P_{out} results from the boolean equation $P_{out}[6] = (p \wedge \overline{k} \vee f)$. We define the boolean operators as follows: & ≡ and, ~≡ not, | ≡ or.

IFD-Mapping: Guarded Assignments and State Machines

To model complex behavior we support the definition of *guarded assignments* (conditional assignments) and *state machines*. To express conditions, we provide *if-then-else* statements comparable to common programming languages. Guarded assignments allow us to dynamically process incoming data in dependency of (other) incoming data:

```
6) if ( Pin[1:2] == {'0','1'} ) {
       Pout[1:4] <= Pin[1:4];
   } else {
       Pout[1:4] <= Pin[4:1];
   }
```

In mapping function 6) we assign the values of P_{in}[1:4] to P_{out}[1:4] if the first two bits of P_{in} are equal to {'0','1'}, if not, we reverse the order. As we can deduce from the term P_{in}[4:1], we are allowed to specify intervals also in the inverse direction.

By a combination of guarded assignments and *variables* that are used as state variables, we can define state machines. An exemplary state machine that extends the guarded assignment, given in mapping function 6), is defined by the mapping functions 7) and 8):

7) bit stateVariable = '1';

8) if ((P_{in}[1:2] == {'0','1'}) && (stateVariable != '1')) {
   P_{out}[1:4] <= P_{in}[1:4];
   stateVariable = '1';
   } else {
   P_{out}[1:4] <= P_{in}[4:1];
   stateVariable = '0';
   }

The presented state machine applies a bit variable to distinguish between two states and comprises a set of Moore outputs. As we use *linear bounded automata*, we can *buffer values*, which means to store packages in predefined *variables* [146]. In the succeeding IFB synthesis steps, we transform the mapping functions into PSMs, which represent the SH-Modes.

IFD-Mapping: Package Instances

To handle packages of a differing capacity, we support *package instances*. Figure 5.33 depicts a scenario where a large incoming package is mapped to two outgoing package instances. We already discussed package instances in Section 5.2.4, when we introduced the super-frames. As presented in mapping function 9), a package instance is characterized by an integer value, encapsulated in parentheses. The first instance of P_{out}, for example, is written as $P_{out}(\mathbf{1})$. When only one instance is needed, we can leave out the package declaration.

9) $P_{out}[1:8](\mathbf{1})$ <= $P_{in}[1][1:4]$ + $P_{in}[2][1:4]$; $P_{out}[1:8](\mathbf{2})$ <= $P_{in}[3][1:4]$ + $P_{in}[4][1:4]$;

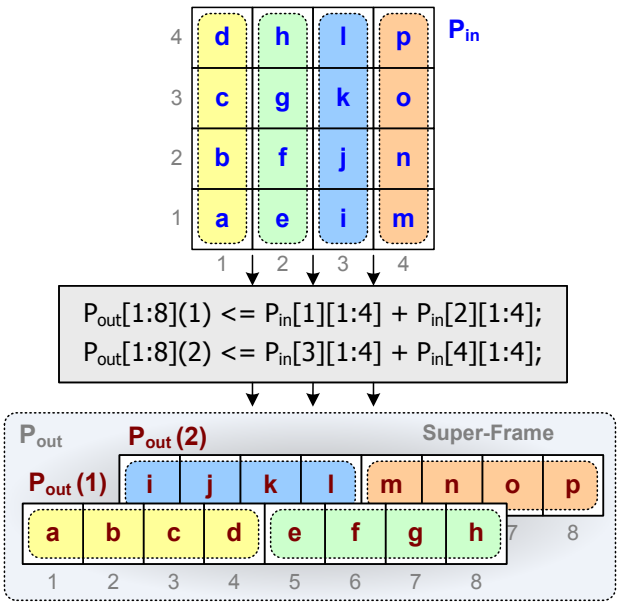


Figure 5.33: IFD-Mapping: Mapping multiple package instances of a super-frame

Syntax- and Semantic Analysis of the IFD-Mapping

To check the *correctness* of the defined *mapping functions*, the IFD-Mapping provides both a *syntax*- and a *semantic analysis*. Therefore, the IFS-EDITOR disposes of a graphical user interface for the entry and the analysis of the mapping functions. We developed a compiler frontend that implements the analysis as part of the IFB model synthesis. The related compiler backend is part of the code generation in the second IFB synthesis phase.

We specified the grammar of the IFD-Mapping Language in such a way that it can be processed by JavaCC (Java Compiler Compiler) [153]. Based on a predefined grammar, this tool creates a lexical analyzer and a parser to verify the lexical correctness and the syntax of the IFD-Mapping, respectively. To construct a derivation tree for the semantic analysis and the usage in the code generation, we utilize the tool JJTree [153], in addition to JavaCC.

When the IFD-Mapping has passed the lexical and the syntax check successfully, we ensure the correctness of the following aspects within the *semantic analysis*:

- The use of correct *assignment operators* (<= for packages, := for variables)
- The *data types* of assigned expressions have to match
- *Variables* have to be *defined* before they are used
- The *dimensions* of assigned packages and constant arrays have to match
- The *direction* of the assigned packages has to be input to output
- *Boolean operators* may be applied only to bit-values

Another condition for a valid IFD-Mapping is a *conflict-free mapping* of the data packages. Conflict-free means that there exists no mapping function which either violates the *causality* of the I-P-O execution cycle or leads to a *deadlock* in the data processing.

Causality of Data Packages

The CU assures the proper execution of the communication cycle according to Figure 4.16. In order to create an executable CU, it is essential that all mapping functions respect the *causality* of the *mapped packages*, which means that all incoming packages are read before the outgoing package is created. To evaluate the causality, we utilize the *package graph*.

We construct the package graph from the given PSMs in combination with the IFD-Mapping as presented in Figure 5.34. Sequences of bits, which are mapped from an incoming to an outgoing data package, are abstracted as *packages* (dotted boxes on the left). These packages present the nodes of the package graph (on the right side). The edges of the package graph result from the given transitions of the state graph in combination with edges that arise from the mapping functions between the created packages.

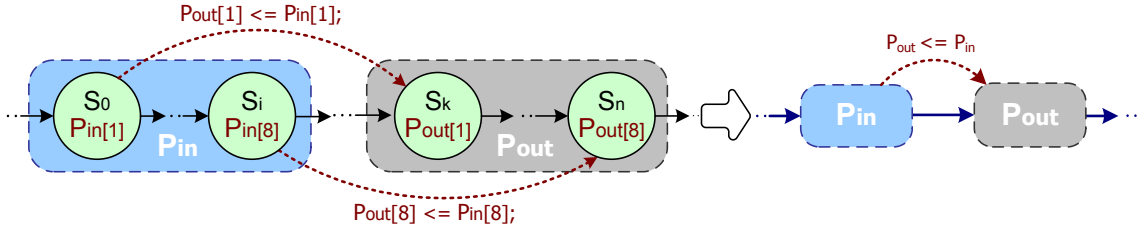


Figure 5.34: Construction of the package graph

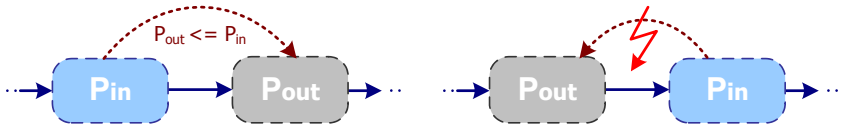


Figure 5.35: Causality in the package praph

The mapping functions respect the causality if the related package graph is *cycle-free*. On the left side of Figure 5.35 we can see a valid mapping function, whereas we can observe a violation of the causality on the right. In contrast to a valid mapping function, an *invalid mapping function* adds a *backward edge* to the graph that results in a cycle.

We can utilize the *depth-first-search* (DFS) algorithm to analyze the package graph for cycles [82]. This is an efficient way with a complexity of $\Theta(V+E)$, where V represents the number of nodes and E stands for the number of edges in the package graph.

Deadlocks

In the case we process the data packages of multiple component interfaces inside the IFB, invalid mapping functions lead to *deadlocks* during the execution of the protocol translation. With the help of the package graph that we create from the affected PSMs and the IFD-Mapping, we can identify deadlocks as a special case of a causality violation. Therefore, the IFD-Mapping is deadlock-free if the package graph is cycle-free.

Figure 5.36 illustrates an exemplary package graph that was created from two interfaces and two mapping functions. The upper two packages origin from the first PSM, the lower two packages from the second one. We map $P_{in}1$ to $P_{out}2$ and $P_{in}2$ to $P_{out}1$. As the package graph is free of cycles, the related IFD-Mapping is deadlock-free. If we switch the sequence of $P_{in}1$ and $P_{out}1$ in the first PSM, the resulting package graph comprises a cycle (see Figure 5.37). Independent of the structuring of the package graph, there exists a backwards edge which indicates the deadlock.

After the designer entered a valid IFD-Mapping (the syntax- and sematic analysis passed successfully), we employ the IFD-Mapping and the created PSMs to construct the IFB model.

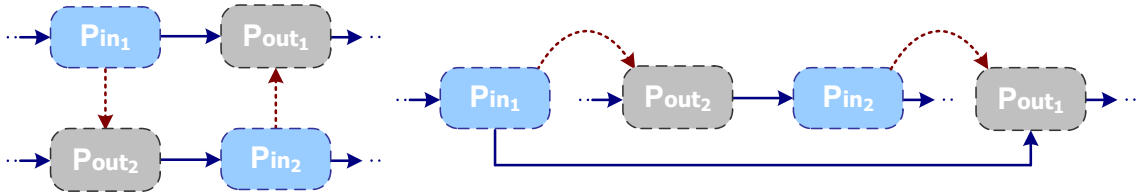


Figure 5.36: Example for a valid IFD-Mapping

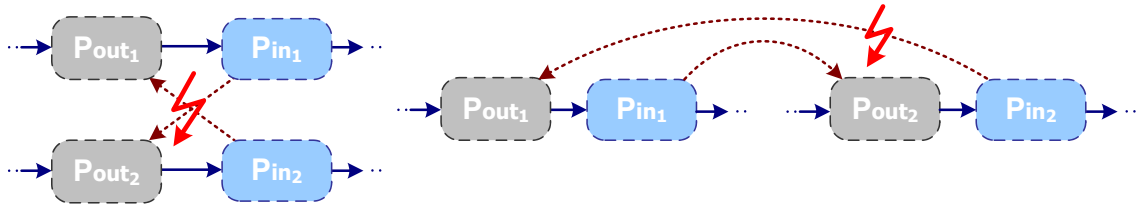


Figure 5.37: Example for an invalid IFD-Mapping that leads to a deadlock

5.2.7 IFD Optimization and Creation of the Protocol Frames

Before we create the IFB model, we adopt the information provided by the IFD-Mapping to optimize the PSMs related to the PH-Modes. The aim of the optimization is to *minimize* the *amount of data* that we have to transmit between PH and SH. This can be done by *reducing* the available data packages with respect to the mapped data bits.

For this reason, the synthesis algorithm marks the *protocol matrix nodes* which are related to the *mapped data bits* as "*required*" (unmapped PMNs are denoted as "*don't care*") and reruns the package detection to *shrink* the *existing packages*. Therefore, we extended the *package detection algorithm* to ignore data bits which are marked as "don't care". As presented in Figure 5.38, an existing package can thereby resolve into multiple *sub-packages*.

The optimization of the data packages inside the PSMs provides the following advantages for the IFB synthesis:

Optimize traffic: The optimized data packages guarantee a *minimum traffic* inside the IFB, as we do not have to transfer redundant values (via the PH-Switch into the memory). This is especially important as the IFB internal *memory bus*, which is realized inside the PH-Switch, represents the *bottleneck* of the IFB architecture.

Bus allocation: The fragmentation of a package into sub-packages can involve the *division* of the *belonging frame* into several frames which cover less states, each. This affects the memory bus arbiter in the CU, which schedules the memory access (memory bus allocation) of the frames. In *priority based systems*, a greater number of small frames *shortens* the *average waiting time* (= starting time - arrival time) for the "execution" of a frame, as we may *not preempt* the ongoing transmission of a frame.

Memory consumption: We synthesize the IFB internal memory as a dedicated memory for each processed data word. Therefore, optimized packages guarantee a minimized memory consumption including a minimum memory interface.

When the optimization is finished, we replace the original WSMs with the optimized PSMs. Thereon, we apply the package- and the frame detection, which delivers the final packages and frames. We utilize this information, including the updated IFDs and the interpreted IFD-Mapping, as input for the IFB model synthesis. In case the designer wants to create an RTR-IFB, he additionally has to specify the related RCU in the Synthesis Wizard.

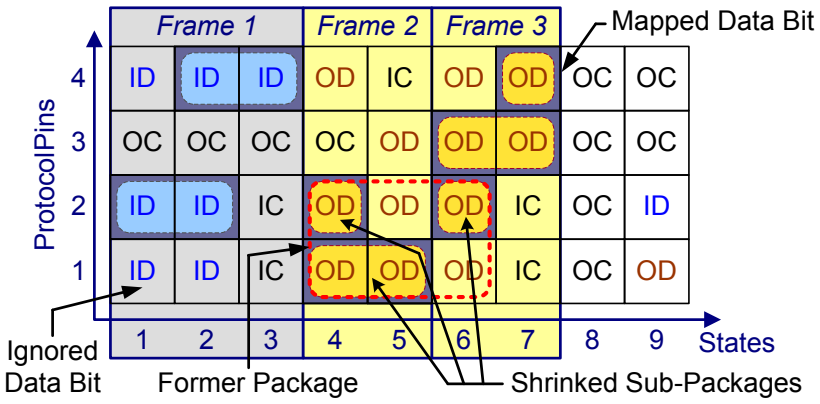


Figure 5.38: Optimized protocol matrix consisting of sub-packages

5.2.8 Assembly of the IFB Model (Intermediate Representation)

The class diagram in Figure 5.39 delivers a *structural overview* of the data model that we assembled in the preceding synthesis steps. It points out the major classes and their *correlations* as we can find them in the *data structure* of the implementation.

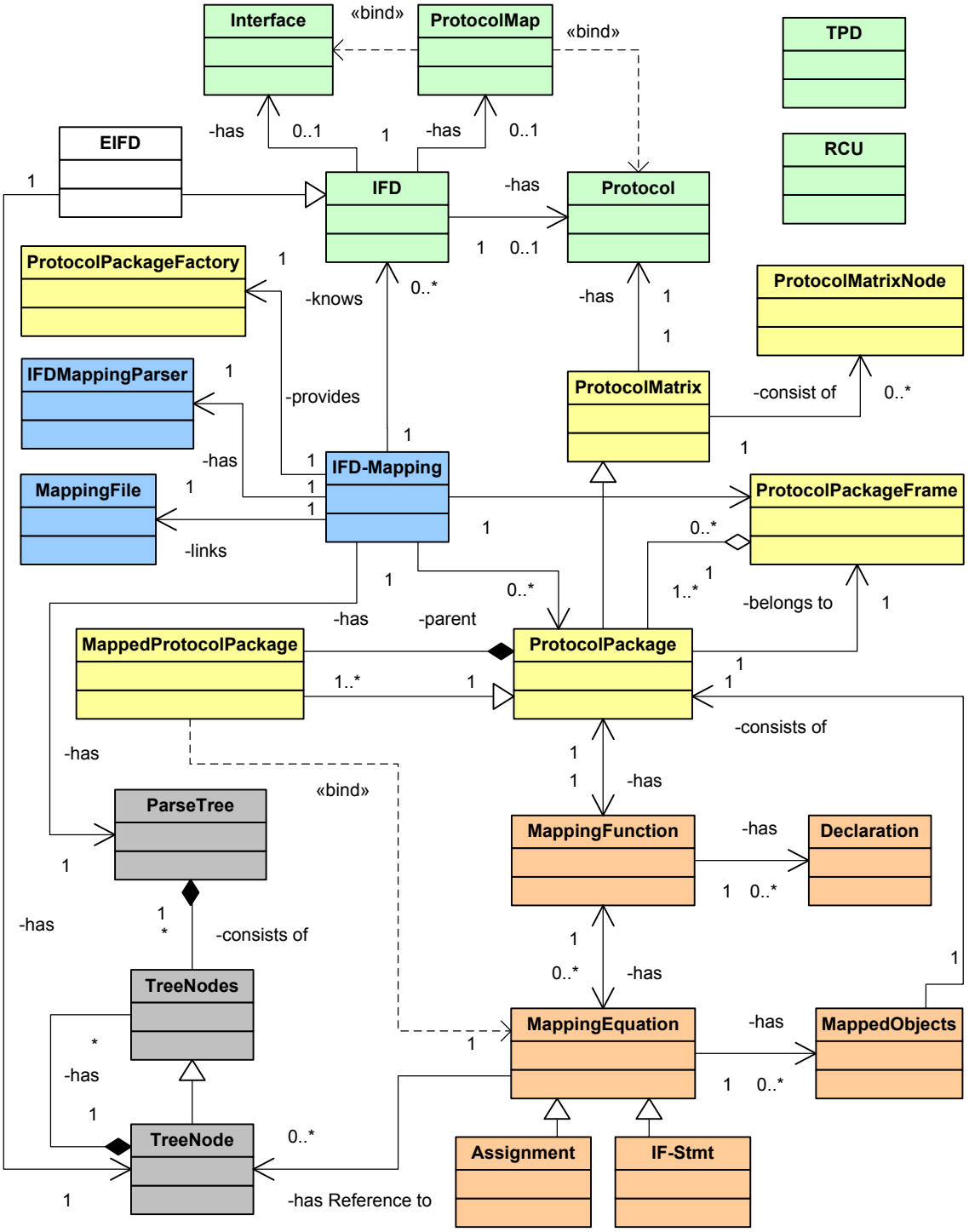


Figure 5.39: Input of the IFB model synthesis as class diagram

We divide the class diagram into five domains, which are represented by colors:

green : Synthesis input, selected from the System Architecture
blue : IFD-Mapping (with a focus on the compiler frontend)
yellow : Protocol matrix, protocol packages and frames
orange : Interpreted mapping functions (as a special part of the IFD-Mapping)
grey : Derivation tree, constructed from the IFD-Mapping

The green classes (IFDs, TPD, RCU) constitute the synthesis input that has been selected from the System Architecture. The class *IFD* represents those interface descriptions which belong to the selected component interfaces. It references the classes *Interface*, *Protocol*, and *ProtocolMap*. Remember that we replaced the original protocol descriptions (WSM) in the IFDs by the optimized PSMs in the preceding optimization step.

IFD-Mapping is a central class that provides a link to the mapping file (*MappingFile*). This is necessary to export the IFB model, which is stored in the form of its inputs. In contrast to the other IFS components that are stored in the IFS-Format (XML), the IFD-Mapping remains an ASCII file, which can be easily modified and reused by a designer. The class *IFDMappingParser* represents the mentioned compiler frontend, developed to analyze and evaluate the IFD-Mapping.

The classes in the lower right corner model the interpreted mapping functions. Technically, each *MappingFunction* is constructed by a number of *MappingEquations*, which can either be refined to an *Assignment* or an *IF-Statement*. Each *MappingFunction* is associated to that outgoing *ProtocolPackage* which is defined by this function. In order to optimize the data packages (preceding optimization step), each *MappingEquation* references its *MappedObjects*, which are exactly one outgoing and a set of incoming *ProtocolPackages*.

The packages that result from the adapted IFDs are instances of the class *ProtocolPackage*, which refines the class *ProtocolMatrix* that consists of *ProtocolMatrixNodes* and belongs to one of the given protocols. At this point, we leave out the basic blocks to simplify the class diagram. To create packages and frames, the IFD-Mapping provides algorithms that process the protocol matrices, here represented by the class *ProtocolPackageFactory*. Next to the packages, the IFD-Mapping knows all frames. One *ProtocolPackageFrame* consists of *ProtocolPackages*, whereas each *ProtocolPackage* belongs to exactly one frame.

The reduced packages are modeled by the class *MappedProtocolPackages*. Each sub-package references its original package and the related *MappingEquations*. The *MappingEquations*, in turn, are associated to dedicated tree nodes of the derivation tree, which results from the interpretation of the mapping functions. The *derivation tree* is defined by the *composite design pattern* [118] that allows us to model a tree structure consisting of TreeNodes. In order to simplify the diagram, we left out the different roles that a particular TreeNode can adopt. The possible roles are defined by the grammar of the IFD-Mapping Language, for example, an non-terminal symbol like an assignment or a terminal symbol like a package identifier. The root node of the derivation tree (*ParseTree*) is assigned to the IFD-Mapping.

For the succeeding synthesis steps we require an extended form of the IFD, the so called *EIFD* (Extended IFD). An EIFD can additionally refer to a *subtree* of the *derivation tree* in order to attach the data processing information specified in the IFD-Mapping to the AutomataOutputs of a PSM. Based on the additional information, we construct the specific data processing behavior of the SH-Modes within the IFB code generation process.

The EIFDs as Building Blocks of the IFB Model

Based on the presented data structure, we create the *building blocks* of the *IFB model*, called EIFDs. An EIFD is an adequate *basic unit* for the succeeding code generation (2nd synthesis step), as it allows the modeling of structure and behavior (see IFD definition in Section 2.1.4). With the help of the EIFDs we are able to model parallel and sequential behavior that can be translated into software (program code) or hardware (hardware design). In this way, we can create either a software or a hardware implementation of the IFB based on the intermediate representation.

An EIFD points to subtrees of the derivation tree inside of AutomataOutputs. As we cannot export the referred subtree based on the techniques available in the IFS implementation, an EIFD provides no (complete) representation in XML. This was another reason to store an IFB in the form of its inputs instead of the resulting intermediate representation.

We utilize three different *EIFD codings* [35] to increase the expressiveness of the EIFDs. As depicted in Figure 5.40, there exist the EIFD versions I-P-M, I-x-x, and x-P-x. An "x" stands for a not available (null) element in this EIFD.

I-P-M: The EIFDs of this category provide the full information (topology & behavior). The updated IFDs that we selected from the System Architecture are a representative of this class. Based on the given information, we can create an interface and a behavior that is linked to this interface. In Figure 5.41 we underlined all components that are modeled by such an EIFD with a red line.

In the case of VHDL, which is the target language we create in the succeeding code generation, we transform the interfaces I into *entity descriptions* and *component declarations*. The protocol P (e. g., the optimized PSM) is translated into a state machine inside the *architecture description*. Based on the protocol map M we map specific signals of the state machine to the related ones of the entity.

I-x-x: An EIFD that comprises only the interface information I can be used to create structural descriptions. In hardware, we can synthesize the component interfaces in hierarchical designs. The pendant in software is to create a method signature.

x-P-x: This class of the EIFDs provides only ProtocolPins and is required to specify internal signals. In this way, we can model local variables/signals that we use, for example, to interconnect hardware components on the same hierarchy level.

We introduce the different EIFDs to assemble the intermediate IFB representation. Depending on the coding and the position inside the IFB model, the code generator knows how to process (generate the target code of) an EIFD.

Figure 5.40: The three available EIFDs coding styles

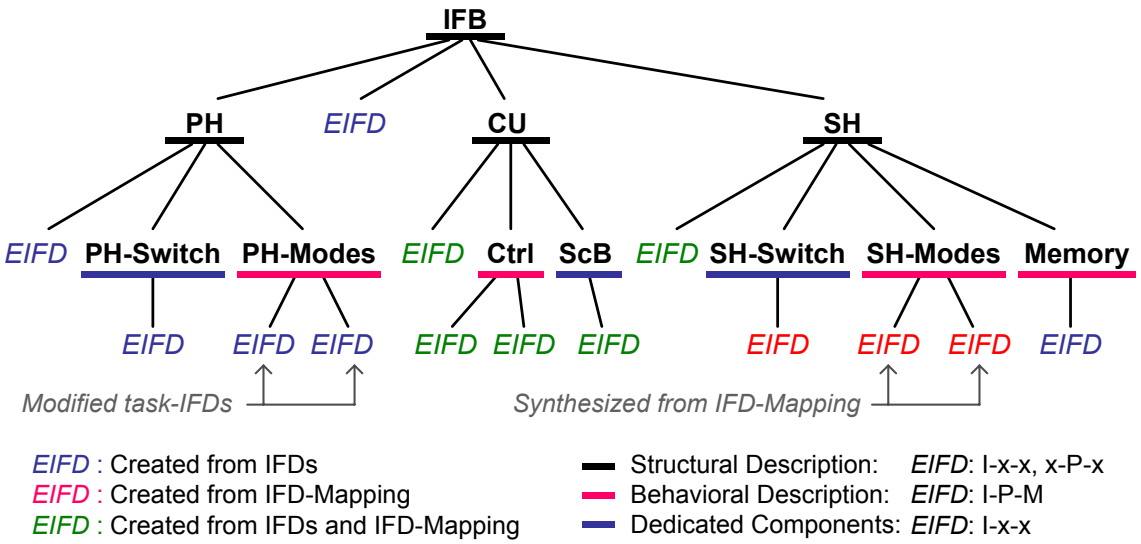


Figure 5.41: The IFB intermediate structure as EIFD-tree

Assembling the EIFDs to the IFB Macro-Structure

Before we explain the creation of the EIFDs in the *EIFD Factory*, we present an overview which EIFDs are required and how they are assembled to the *intermediate representation*. We construct the *IFB model* according to the IFB Macro-Structure by composing EIFDs to the tree structure given in Figure 5.41, which depicts the non-reconfigurable IFB model.

The tree nodes *IFB*, *PH*, *SH*, and *CU* are structural elements that define the *component hierarchy* of the IFB. For this reason we utilize $I\text{-}x\text{-}x$ and $x\text{-}P\text{-}x$ coded EIFDs to model these *components*, including their interfaces and interconnections. The *PH-* and *SH-Modes*, the *Memory* (*-interface*), and the *Ctrl* (arbitration of the memory bus, CU) are modeled by $I\text{-}P\text{-}M$ coded EIFDs, which support the creation of behavior in the form of a state machine in addition to the interfaces. To model the components PH-Switch, SH-Switch, and Scoreboard (ScB), we utilize $I\text{-}x\text{-}x$ coded EIFDs to define their interfaces. The behavior of these complex components, which cannot be expressed properly as FSMs, depends extremely on the target language and the available HW or SW paradigms. For this reason, the code generation in the succeeding synthesis step has to create the behavior of these components directly based on the IFDs and the IFD-Mapping (see class diagram in Figure 5.39).

To construct the EIFDs which represent the PH-Modes, we extend the modified task-IFDs. Furthermore, we personalize the EIFDs that model the PH, the PH-Switch, the Memory, and the IFB (blue EIFDs) with the help of the task-IFDs. To create the SH-Modes and the SH-Switch (red EIFDs), we refer to the IFD-Mapping. The EIFDs representing the SH and the complete CU (green EIFDs) depend on both kinds of information.

In the case of the RTR-IFB model, the EIFDs of the PH-Modes and the SH-Modes are placed on the top level of the tree next to the IFB node. Accordingly, the interfaces and interconnections of the PH, the PH-Switch, and the IFB have to be adapted, which is done by a modification of the related EIFDs in the EIFD Factory. The reason to place the reconfigurable modes on the top level results from the presented mapping of the IFB components to the hardware execution platform (see Figure 4.20).

The EIFD Factory

To assemble the IFB model, the EIFD Factory creates all necessary EIFDs. At this point we deliver only the most interesting aspects of the construction algorithm:

IFB, PH, SH, CU: To model the hierarchical component structure of the IFB, we require only $I\text{-}x\text{-}x$ and $x\text{-}P\text{-}x$ coded EIFDs. A created interface has to comprise all signals which are routed through the related component.

PH-Modes: The created PSMs, which model the external I/O, provide the basis for the PH-Mode EIFDs. We have to add the internal communication to the memory bus, which is specified by the protocol frames (see Section 5.2.4 and A.6). Therefore, we integrate the identified frames into the PSMs, as presented in Figure 5.42 for an incoming frame. Thereby, we have to add three states for each frame – two states to request the memory bus in the PH-Switch (`Init`, `IRQ`), and one state to release the bus ($\overline{\texttt{IRQ}}$).

Additionally, each state inside the frame is augmented by two more states (`Rdy`, `Ack`) for the fully interlocked handshake with the memory interface. Currently, we do not permit burst transmissions. To compensate for the additional time consumed by the memory I/O, we correct the affected timed transitions by subtracting this time (here: 5 clock cycles, see Section 6.2). Furthermore, we add one state that starts the related timer for each timed transition that comprises a TimerOrDeadline.

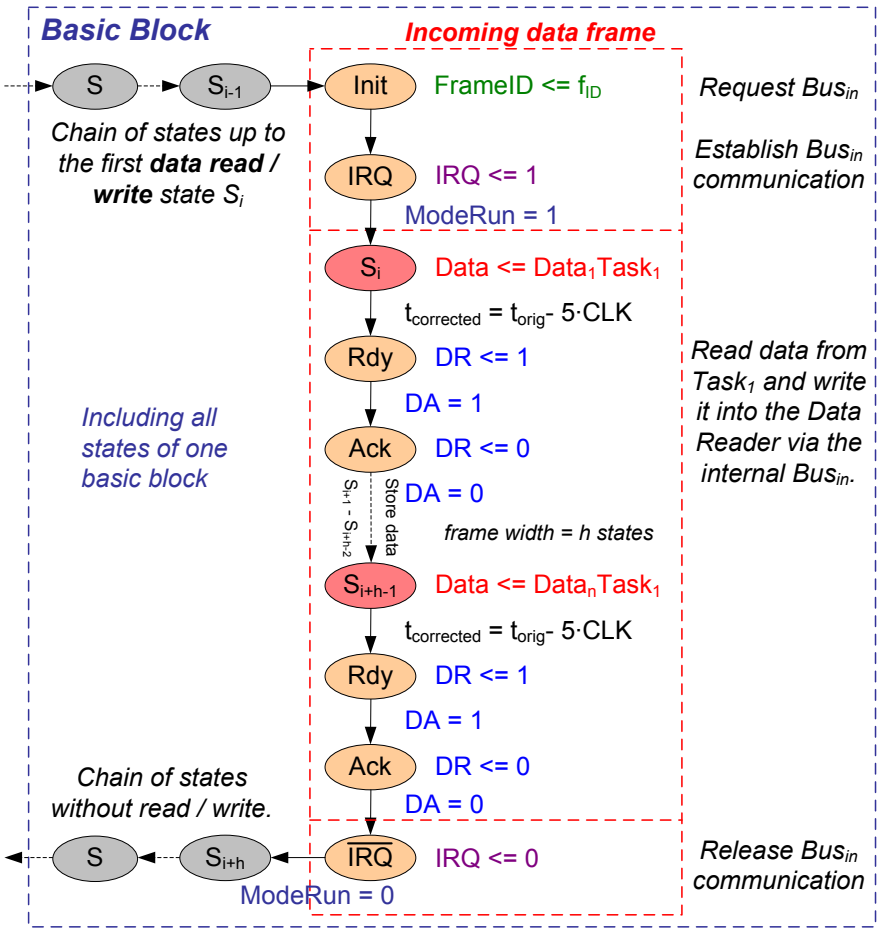


Figure 5.42: PH-Mode synthesis – Construction of a data frame (incoming data)

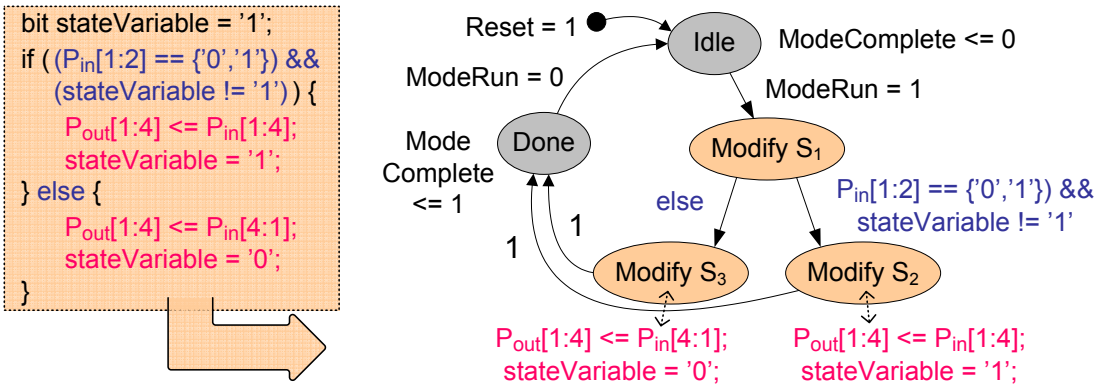


Figure 5.43: SH-Mode synthesis – Creation of the state machines for the data processing

SH-Modes: To create an SH-Mode we generate a PSM from the related mapping function. Thereby, we transform the available *assignments* into AutomataOutputs inside of the modify-states. Each output references a dedicated subtree within the derivation tree in order to specify the related assignment. In the case of *conditional mapping functions* or *state machines* we create additional states to implement the modeled branches.

Figure 5.43 illustrates the state machine that we would synthesize from the mapping function presented in Section 5.2.6. The states `Idle`, `Done`, and `Modify S1`, including the signals `ModeRun` and `ModeComplete` result from a *PSM template* that we use to construct an SH-Mode. The synthesis algorithm completes the *scenario specific parts* of the SH-Mode template: because of the binary condition in our example we create the two states `Modify S2` and `Modify S3`. The *transition condition* related to the *transition* to state `Modify S2` results from the *if-condition* in the mapping function, the other condition represents the *else* case. The outputs of the state machine are composed as Moore outputs, which reference the dedicated subtree of the derivation tree (red).

PH-Switch, SH-Switch, Scoreboard: These components comprise a complex behavior that cannot be expressed (efficiently) using IFDs. Furthermore, the implementation of these components varies quite a lot depending on the target language. Thus, we model only their interfaces here, using *I-x-x* coded EIFDs. The generation of the components' behavior has to be implemented directly within the code generator.

Memory: The internal memory of the IFB is synthesized as a *dedicated memory*. Therefore, it depends on the package information that we derive from the IFDs. Our synthesis algorithm generates one PSM for the incoming- and one for the outgoing memory. The two state machines implement the *memory interface* that communicates with the PH-Modes and stores (recalls) the identified data words in (from) dedicated registers. In our case (VHDL), the succeeding code generation creates an additional *register-file* that represents the data structure of the identified packages as a library.

Ctrl: Based on the identified frames we create the *memory bus arbiters* (Ctrl_{in} and Ctrl_{out}) inside the CU. The construction depends on the strategy that the bus arbiter has to perform to schedule the memory bus access. In our case, we implement a priority based scheduler, which is explained in the following chapter.

It is the task of the code generation to create an executable version of the IFB based on the presented intermediate representation. Therefore, the *code generation step* has to processes the generated EIFD-tree as the second step of the *synthesis phase*.

5.3 Synthesis Phase – Design Step 2: IFB Code Generation

The IFB model synthesis is followed by a completely automated code generation step that translates the intermediate representation into a dedicated target language using classical *compiler techniques*. An automation of the code generation is profitable for multiple reasons:

- When the target platform comprises specific resources, it makes sense to create the target code in dependency of the available components.
- When performed automatically, the coding is faster and less error prone. Furthermore, automation facilitates enforcing compliance with design rules and coding styles and therefore simplifies the reusability and maintainability of the created code.
- Furthermore, automated testing and verification allows us to increase the code quality and is essential during the development of safety critical systems.

To create the *IFB target code*, the code generator evaluates the generated intermediate representation in combination with the given IFDs and the IFD-Mapping as depicted in Figure 5.44. We developed a specific *compiler backend* [35, 1] to generate the *VHDL code* of the IFB as part of the Interface Synthesis Design Flow. It applies the well known code generation technique *frame processing* [178], based on the *IFB Hardware Template*, which defines a precise construction plan of the IFB device on the RT (circuit) and the algorithm (FSM) level. We explain the IFB Hardware Template in detail in the next chapter.

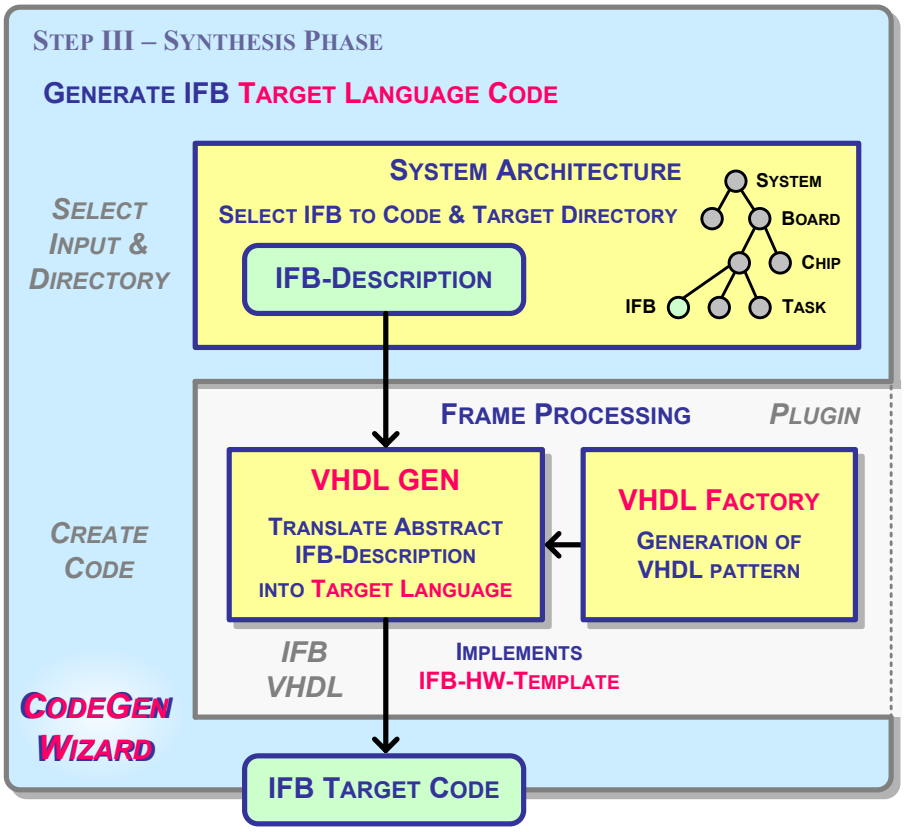


Figure 5.44: Interface Synthesis Design Flow: the IFB code generation phase

5.3.1 Frame Processing

Figure 5.45 illustrates a structural model of the *frame processing* technique, which comprises a *meta model* and a set of *code frames*. A code frame is composed of a set of *parameterizable slots*. In turn, each slot comprehends one or multiple *frame instances* and a number of *code fragments* (e. g., Type, Name, etc.).

Frame processing uses the meta model as a kind of "top-level frame" that instantiates and parameterizes the required frames. Furthermore, the meta model assembles the completed frames to a tree-like data structure. In this way, we create a frame hierarchy, which represents the dedicated target code. We generate the final code by exporting the created frames.

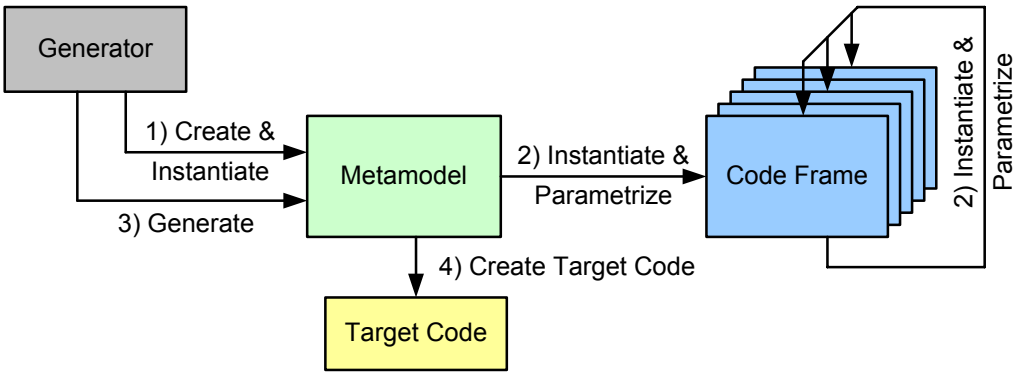


Figure 5.45: The code generation technique *frame processing*

VHDL is a component based language that facilitates the mapping of particular VHDL constructs to frames. Therefore, frame processing is an adequate code generation technique to create the IFB target code. We utilize the frames to create the particular elements of the IFB Hardware Template in the form of *VHDL pattern* as presented in the next chapter. Therefore, we adapted the frame processing model as presented afterwards.

5.3.2 Adapted Frame Processing Model

Figure 5.45 depicts the adapted model of the *frame processing*, which is integrated in the *CodeGen Wizard* as part of the IFS-Editor. Thereby, we implement the meta model by means of the module *VHDL Gen* and create the frames with the so called *VHDL Factory*.

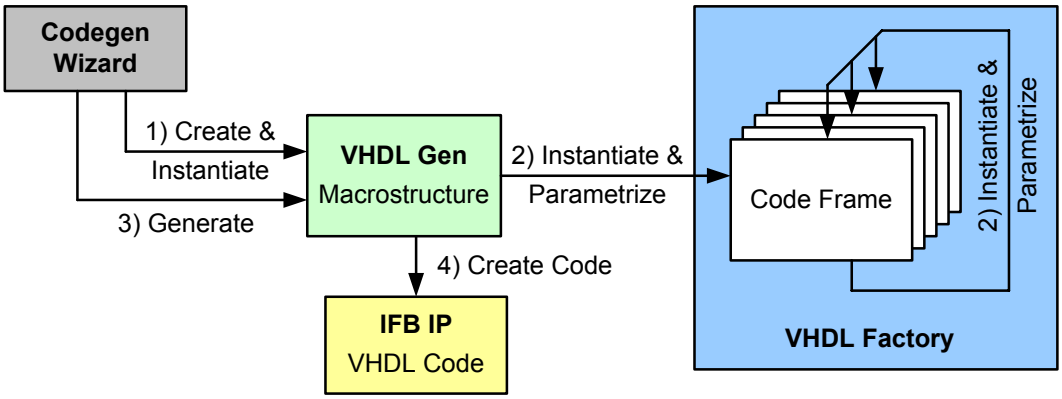


Figure 5.46: Adapted model of the *frame processing* for the IFB code generation

VHDL Gen

As *meta model* we apply the *IFB Macro-Structure* which already has been used to construct the IFB intermediate representation. To create the target code, VHDL Gen traverses the IFB model by a recursive descend and uses the VHDL Factory to translate the abstract EIFDs into VHDL. Starting point of the recursive descend is the top-level IFB node.

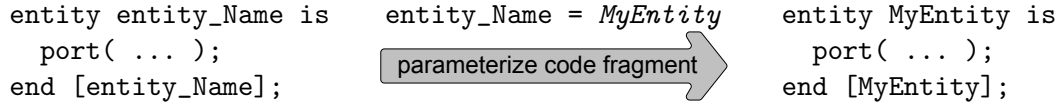
On one hand, our meta-model considers the *component hierarchy* as presented in the IFB Macro-Structure; on the other hand, it handles the internal structure of a VHDL file, for example, the composition of the respective entity - and architecture descriptions.

Furthermore, VHDL Gen *organizes* the generated *VHDL code fragments* with the help of a code-list. After the code generation is completed, we create the target code by exporting the code-list in the form of VHDL files.

VHDL Factory

The generation of the actual VHDL code is performed by *instantiating frames* inside the VHDL Factory. Each frame consists of slots, which comprehend further frames and code fragments. Thereby, each code fragment represents a *parameterized snippet* of *VHDL code*. During the initialization, the snippets receive their final specification.

A simple example that points out this procedure refers to the identifier of a VHDL entity. The related sippet comprises the identifier as a parameter (here: entity_Name), which is replaced during the frame initialization by the respective value of the IFB model.



In the following section we provide an overview of the top level *code pattern* (frames) that we generate in order to assemble the VHDL entities and architectures. Some VHDL examples are given in A.4. A detailed description of the code generation is presented in [1, 35].

5.3.3 Overview of the Generated VHDL Code Pattern

Entity : With the help of this frame we create the "*interfaces*" for all VHDL files

Architecture : This complex pattern creates the different forms of VHDL *behavior*:

- **Behavioral VHDL descriptions :** In this case we create *concurrent assignments* and *processes* including the definition of *local signals* and the synthesis of the modeled *reference signals* (TimerOrDeadlines, Clocks, and GlobalDates).
  1) State machines (PH-Modes, SH-Modes, Memory (-Interface), Ctrl)
  2) Dedicated RTL circuits (PH-Switch, SH-Switch, Scoreboard)
- **Structural VHDL descriptions :** Here, we create the behavior by composing existent components to a component hierarchy. Therefore, we have to declare, instantiate, and map the required components.
  1) Components (IFB, PH, SH, CU)

Library : To provide complex data types that represent the internal memory, we create a register file that consists of packages which are modeled as records of std_logic vectors.

5.3.4 The Three levels of IFS Code Generation

As presented in Figure 5.47, we can use the implemented code generator also to create the VHDL code related to a *chip* or even the *complete System Architecture*. This is possible as the complete system is composed of components which provide an IFD. Similar to the EIFDs that we utilize in the case of an IFB, our compiler is able to process IFDs, which simply lack the association to the derivation tree. This is not yet critical, as we require the associations only to create the SH-Modes. Thus, the VHDL Factory can construct the VHDL descriptions of architecture- and communication components without modifications.

In order to create the target code for these components (except of the IFB), we apply the System Architecture as meta model and provide the modeled communication system to the code generator. We distinguish between three levels of VHDL code generation:

I) **GenerateIFB()**

This version delivers the target code of an IFB in the form of an *IFB-IP* as presented in the previous section. The created IFB code is a set of hierarchical VHDL files.

II) **GenerateExecutables()**

GenerateExecutables() iterates through the available chips. Thereby, it creates the target code for these chips based on the contained tasks, media, and IFBs. In Chapter 3 (Related Work) we introduced this specific bunch of code as "*Configware*".

III) **GenerateSystem()**

GenerateSystem() iterates through the System Architecture by a recursive descend. It creates the hierarchical target code of the complete system, including all architecture- and communication components. The resulting code can be utilized for a simulation of the system. As the communication components are modeled as black box, we can only synthesize their interface behavior. For a complete simulation, the designer would have to add the internal component functionality by hand, as the automated generation of this functionality is out of the scope of the IFS methodology.

The code generator produces *synthesizable VHDL code* from the *synthesizable VHDL subset* [223, 175, 173] that is ready to be integrated into a VHDL based design flow [74, 50].

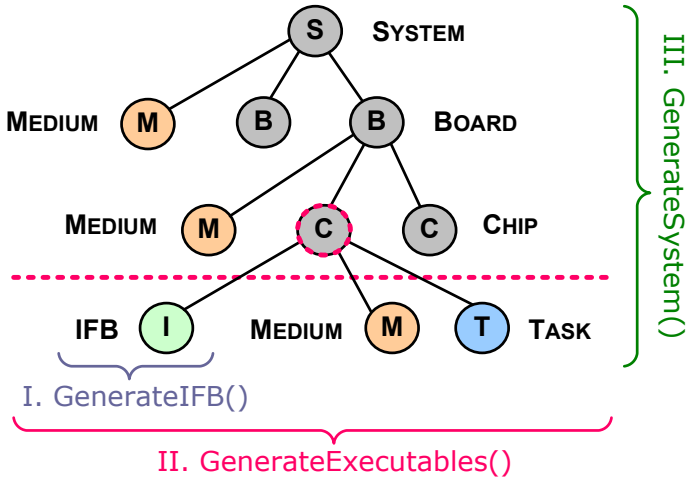


Figure 5.47: The three levels of the IFB code generation

5.4 Code Integration Phase

The generated IFB target code can be seen as an *IP*. In contrast to the IFB intermediate representation, which we regard as a flexible IP which can be stored in a design data base, it is the primary intention of the IFB target code to be integrated into an *existing design*. As we create an IFB for a specific scenario with respect to the applied language, the challenge of integrating an IFB depends on the utilized *tools* and *design-flows* (see Figure 5.48).

In hardware, we use *EDA tools* to create an *executable hardware design*, in the case of an FPGA, for example, a circuit or a configuration bitstream. This process depends mainly on the available target technology. In software, we employ *compilers* to generate the object code in the form of an executable binary file. This final processing of the design (e.g., the low level synthesis in the case of HW) is independent of the IFB synthesis. Therefore, the integration process, which can bee seen as a *concluding design step*, is located outside of the Interface Synthesis Design Flow.

An example for the integration of the hardware RTR-IFB is the *Part-Y* tool, developed in our research group [92, 91]. This tool is specialized in the partial design flow for Xilinx devices. With the help of the tool, we can combine the RTR-IFB with the VHDL files of the tasks and create the partial bitstreams (= *Configware*) to configure the FPGA. Thereby, Part-Y provides a single UML class diagram as backend that is used to visualize the characteristics of the reconfigurable system. The tool allows to navigate on the design under development and thus provides a sophisticated means to integrate the IFB VHDL files into the design.

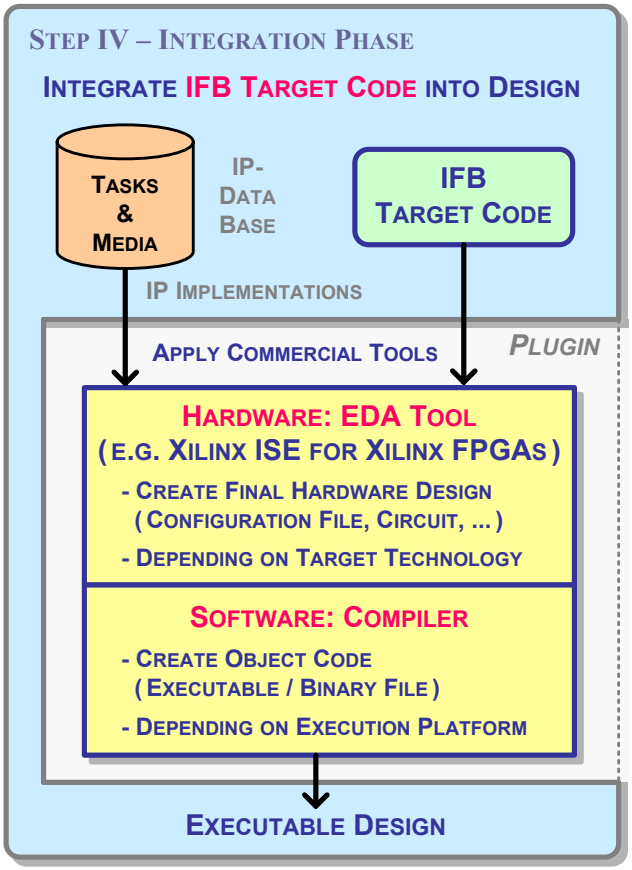


Figure 5.48: Interface Synthesis Design Flow: the IFB code integration phase

5.5 Extension of the Interface Synthesis Design Flow

5.5.1 Creation of a Globally Optimized Communication Infrastructure

In this section we introduce an extension of the presented Interface Synthesis Design Flow. The idea is to automatically compute the optimum allocation and placement of IFBs with the aim to construct an *optimized communication infrastructure*. The presented approach supports the designer by creating all IFBs of a non-reconfigurable system automatically within one iteration through the Interface Synthesis Design Flow, producing a minimum of implementation - and communication costs. The details of this optimization approach have been published in [11, 37].

Optimized Distribution of IFBs

The optimization approach copes with a System Architecture that comprises a fixed binding of tasks to chips and a static communication structure, which is physically restricted to a given set of interconnections. The assignment of mapping functions to IFBs as well as the allocation and binding of the IFBs to chips, next to the reservation of the necessary interconnections characterizes our *design space*. The communication graph, depicted in Figure 4.13 visualizes this design space. As we will see, the location of an IFB in the System Architecture highly determines the number of required interconnections.

Obviously the *communication costs* between and the *implementation costs* for the IFBs vary depending on the allocation and the binding. Therefore, it is our goal to compute an *optimal allocation* and *binding* ($\Leftrightarrow$ placement) of IFBs and mapping functions. To achieve this goal, we define *cost functions* for the communication and the implementation of an IFB.

Cost Functions for Clustering

The definition of cost functions, for example, with the help of Integer Linear Programming, as well as a detailed overview about partitioning concepts and algorithms are presented in [232]. The total costs of the System Architecture $C_{SysArch}$ are defined as the sum of all communication and implementation costs, which have to be minimized:

$$C_{SysArch} = C_{Tasks} + C_{StaticIO} + \sum_{i=1}^{k} C(IFB_i) \tag{5.1}$$

k stands for the number of allocated IFBs. As static part of the system, the costs arising from the implementation of tasks (C_{Tasks}) and from fixed direct interconnections ($C_{StaticIO}$) are negligible for the optimization. Therefore, we do not care for these items.

Only the costs resulting from the IFBs are variable: On one hand, an IFB requires resources in terms of CLBs or memory for its implementation (*implementation costs,* C_{Impl}). On the other hand, an IFB reserves interconnections for establishing the communication between the connected tasks (*interconnection costs,* $C_{Interconnect}$).

$$C(IFB_i) = C_{Impl}(IFB_i) + C_{Interconnect}(IFB_i) \tag{5.2}$$

$$n = \text{number of mapping functions bound to } IFB_i \tag{5.3}$$

The *implementation costs* C_{Impl} are defined as follows:

$$C_{Impl}(IFB_i) = (1 + n \cdot IFB_{MappingCosts}) \cdot IFB_{Body} + \sum_{x=1}^{\#\text{PH-Modes}} C(\text{PH-Mode}_x) + \sum_{x=1}^{\#\text{SH-Modes}} C(\text{SH-Modes}_x) \tag{5.4}$$

Equation 5.4 consists of three items: the costs for the PH-Modes, those for the SH-Modes, and a term that estimates the overhead of the IFB skeleton. The 3rd term comprises a fixed overhead and one that increases linearly with the number of processed mapping functions.

The *communication costs* $C_{Interconnect}$ are defined as follows:

$$C_{Interconnect}(IFB_i) = \sum_{m=1}^{n} \#\,interconnections_m \cdot lenght_m \tag{5.5}$$

As we get by with primitive electrical interconnections, the communication costs arise from the product of the required interconnections with their length, which is defined as the number of passed edges through the architecture graph. With the aid of the given equations the IFB costs can be estimated precisely.

Definition of Optimization Strategies

Due to the huge amount of available logic, the communication resources became more and more the critical aspect of SoC-design. Especially the off-chip communication across the System Architecture has to be minimized. Therefore, the reduction of the interconnection costs is one main goal for the distribution of IFBs. However, a minimization of the interconnection costs leads to an increased number of IFBs resulting in higher implementation costs. Depending on the given System Architecture and the current design aim, different strategies can be approached as presented in Figure 5.49:

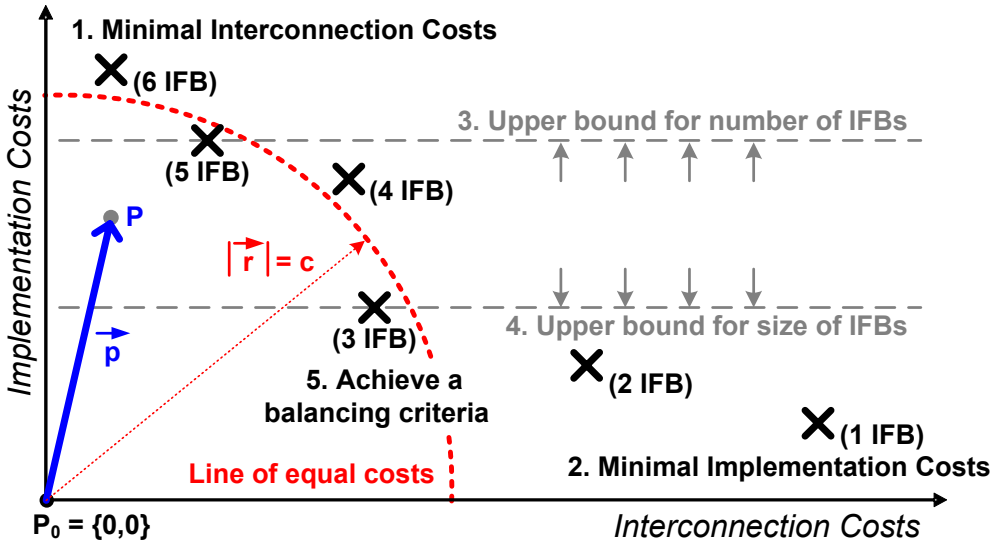


Figure 5.49: Goals for the distribution of IFBs

1. **Minimal interconnection costs**
   In this strategy, we allocate one IFB for each mapping function and bind it to that chip, which produces a minimum of interconnection costs by implementing this IFB. This is the chip with the highest communication effort with respect to the adapted task set. The disadvantage of this strategy are high implementation costs.
2. **Minimal implementation costs**
   The number of IFBs is reduced to a minimum by maximizing the number of mapping functions per IFB. This strategy increases the utilization of the particular IFBs whereas the implementation costs are reduced to a minimum. As disadvantage, this strategy leads to high interconnection costs.
3. **Upper bound for number of IFBs**
   This strategy optimizes the interconnection costs in respect to a given upper bound for the number of IFBs.
4. **Upper bound for implementation costs**
   This strategy decreases the local implementation costs per chip regarding an upper bound for the size of an IFB.
5. **Achieve a balancing criteria**
   A balancing criteria can be useful to avoid extremal values for implementation- or interconnection costs. An example for this goal could be the average utilization of IFBs (load balancing) or a balanced traffic on the communication links.

Each cross in Figure 5.49 represents a pareto optimal solution for a fixed number of IFBs. The absolute value $|\overrightarrow{p}|$ defines the costs of point P including the sum of all IFB costs. Thus, the circular arc defines the line of equal costs c according to equation (5.1). The optimization strategies can be integrated into the cost function by adding quantifiers to the implementation- and interconnection terms in equation (5.2). In that case the total costs becomes a weighted sum, which allows us to prefer different strategies.

Solving the Optimization Problem

The problem of distributing a number of IFBs over the System Architecture and deciding which mapping function to implement in which IFB can be seen as a *partitional clustering problem*. To handle this problem, we developed a *bottom up partitional clustering algorithm* that uses heuristics to examine only a small number of partitions. The result is a number of disjoint clusters without a hierarchy, where each cluster represents one IFB. The clustered patterns are mapping functions which are implemented inside the IFBs.

First of all, the optimization algorithm constructs an initial distribution that satisfies the strategy "*minimal interconnection costs*". Afterwards, the clustering algorithm migrates and merges mapping functions and IFBs in order to fulfill the selected optimization strategy. Thereby, the clustering algorithm tries to reduce the implementation costs by migrating mapping equations from one IFB to another until the selected IFB has no mappings assigned. The empty IFBs are then removed from the system. A lower number of IFBs results in decreased implementation costs and increased interconnection costs. If the selected strategy is satisfied, or no further reduction of the number of IFBs is possible, the clustering algorithm stops. The precise algorithm is explained in [11] and [37].

5.6 Summary

This chapter highlighted the three design phases of the Interface Synthesis Design Flow in detail: the *modeling phase*, the *synthesis phase*, and the *integration phase*. To evaluate our methodology and give a proof of concept, we implemented the Interface Synthesis Design Flow as an EDA tool, the IFS-Editor.

In addition, we presented an extension of the Interface Synthesis Design Flow that tackles the generation of a globally optimized communication infrastructure resulting in minimum communication - and implementation costs.

Modeling Phase

After having presented our modeling concept in the previous chapter, we now introduced our UML 2.0 profile. It supports the modeling of the structural and the behavioral aspects of the IFS System Architecture in an intuitive and abstract way. To define the UML 2.0 profile, we extended the UML 2.0 meta model. We transferred the profile to the CASE tool Fujaba and implemented the model transformation according to our modeling concept. Thanks to the UML 2.0 profile, we are now able to provide an integrated design flow that accomplishes the complete design process, from high level UML down to synthesizable VHDL.

Synthesis Phase

Every time we have to interconnect incompatible interfaces in the System Architecture we generate an IFB. The automated IFB generation is a central topic of the synthesis phase that we divide into the construction of a language independent intermediate representation (IFB model) and the generation of the final IFB target code. To construct the IFB model, we preprocess the selected input (IFDs, TPD) from the System Architecture and define the IFB internal data processing (IFD-Mapping). Afterwards, we assemble the hierarchical data structure of the IFB model composed of EIFDs, created by the EIFD-Factory.

As this thesis focusses on communication synthesis, we explained the protocol synthesis in greater detail. The presented approach constructs the protocol state machines (PSM) that are required for the implementation of the PH-Modes from the given waveform state machines (WSM). To describe the protocol synthesis algorithm, we introduced the abstract data types *basic block*, *protocol matrix*, *package*, and *frame* including their construction and their meaning for the synthesis algorithm.

The second major aspect referred to the definition of the IFB internal data processing. We developed the *IFD-Mapping Language* for the definition of *data processing operations* in the form of *mapping functions* that support the assignment of constants, the mapping of incoming to outgoing data bits, the application of boolean functions and guarded assignments, and the use of linear bounded automata for the processing of data. For the integration of the IFD-Mapping into the IFS-Editor, we implemented a syntax and a semantic check, including a test for the package causality and deadlocks.

With the help of the IFD-Mapping we optimize our interface models (shrink data packages) for further processing. The EIFD Factory instantiates and personalizes all EIFDs which are required to compose the structural and behavioral aspects of the IFB model based on the IFD-Mapping. These EIFDs are then assembled to the hierarchical IFB model in the form of an EIFD-tree in consistency to the IFB Macro-Structure.

The succeeding code generation, the second design step of the synthesis phase, translates the IFB model into a target language, in our case, into VHDL. In order to implement the related compiler backend we adapted the well known code generation technique *frame processing*. The two main elements are VHDL Gen, which realizes the iteration of the meta model and the assembly of the code fragments, and VHDL Factory that provides the particular frames, which create the actual VHDL code by instantiating code snippets.

Next to the target code generation of an IFB-IP, the developed code generator supports the creation of *Configware*, which is the information required to configure a chip and the generation of complete systems, which can be used for simulation purposes.

Integration Phase

After explaining the central Interface Synthesis Design Flow, we provided an insight into the concluding integration phase. Due to the high dependability to the applied tools and target languages, the integration phase has been attached to the Interface Synthesis Design Flow as an optional and user specific process. One example for the integration of the IFB VHDL code is *Part-Y*, an EDA tool for the partial runtime reconfiguration with a focus on FPGAs that is developed in our working group.

Extension of the Interface Synthesis Design Flow

Finally, we presented an extension of the Interface Synthesis Design Flow. The idea is to automatically compute the optimum allocation and placement of IFBs in order to create a globally optimized communication infrastructure. Thereby, the presented approach supports the designer to create all IFBs of a non-reconfigurable system within one iteration through the Interface Synthesis Design Flow, producing a minimum of implementation - and interconnection costs. To solve this challenge, we defined cost functions for the implementation of an IFB and the communication in relation to the System Architecture. With the help of a bottom up partitional clustering algorithm we try to satisfy one of the five different optimization strategies from which the user can select.

CHAPTER 6

The Interface Block (IFB)

In this chapter we introduce the *IFB Hardware Template*. We developed the *schematic* as a detailed construction plan of the *IFB hardware version*, which facilitates the creation of IFB implementations in the form of hardware description languages, like VHDL. As mentioned in the previous chapter, we applied the IFB Hardware Template to implement the VHDL code generator within the Interface Synthesis Design Flow.

Next to the code generation, the IFB Hardware Template provides the basis for a *cycle accurate analysis* of the IFB, which is essential for a *schedulability analysis* of the adapted protocols. In this chapter, we explain how to evaluate the IFB description in order to derive the information that is required as input for standard scheduling approaches.

Finally, we introduce two orthogonal optimization approaches, which allow us either to minimize the latency of the protocol adaptation or to optimize the required resources (chip area) for the hardware implementation.

6.1 IFB Hardware Template

As presented in Figure 6.1 and 6.2, we modeled the IFB Hardware Template in the form of a *schematic* on the *RT*- and the *Algorithm level* (see Figure 3.2). Thereby, we transferred the abstract System Architecture consistently into an efficient clock-synchronous hardware circuit. Therefore, each component is provided with a clock (`CLK`) and a reset (`Reset`) signal. Asynchronous communication can be handled only by synchronizing it to the IFB clock.

The IFB Hardware Template represents the *component hierarchy* with the help of graphically nested objects. The available *state machines* (PH-Modes, SH-Modes, SH-Recon-Modes, DataReader, DataWriter, Ctrl_{in}, Ctrl_{out}, and the Reconfiguration Unit) are specified as *automata graphs* on the algorithm level. The nodes inside these graphs represent the states; the edges stand for the transitions. A text, depicted close to a state, represents its Moore output, while the annotations next to the transitions define the transition conditions.

The composition of the other elements (CU, PH-Switch, SH-Switch, and Scoreboard) is given as an *RTL design*, consisting of *gates*, *registers*, and *lookup tables* (LUT).

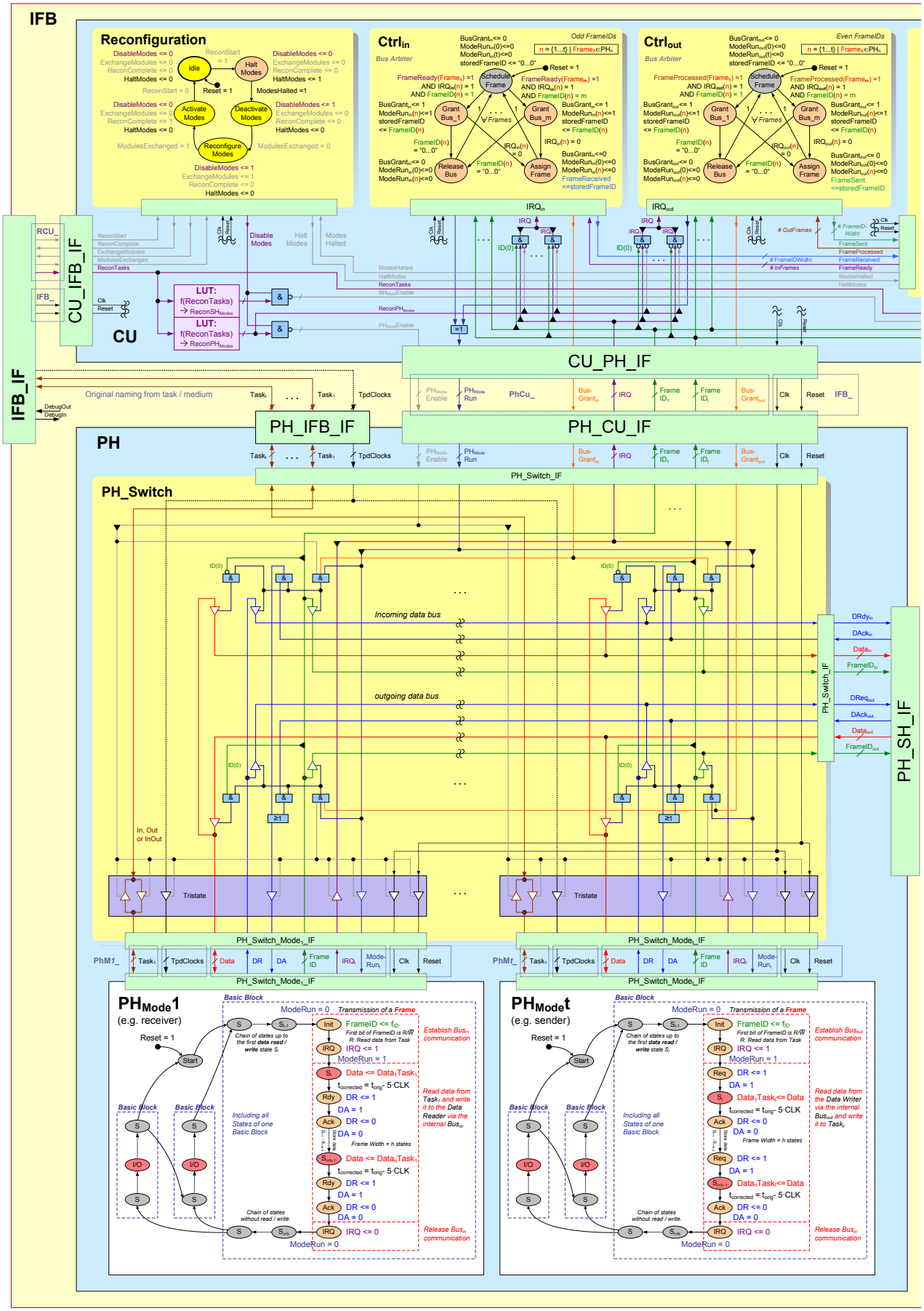


Figure 6.1: IFB Hardware Template (left side)

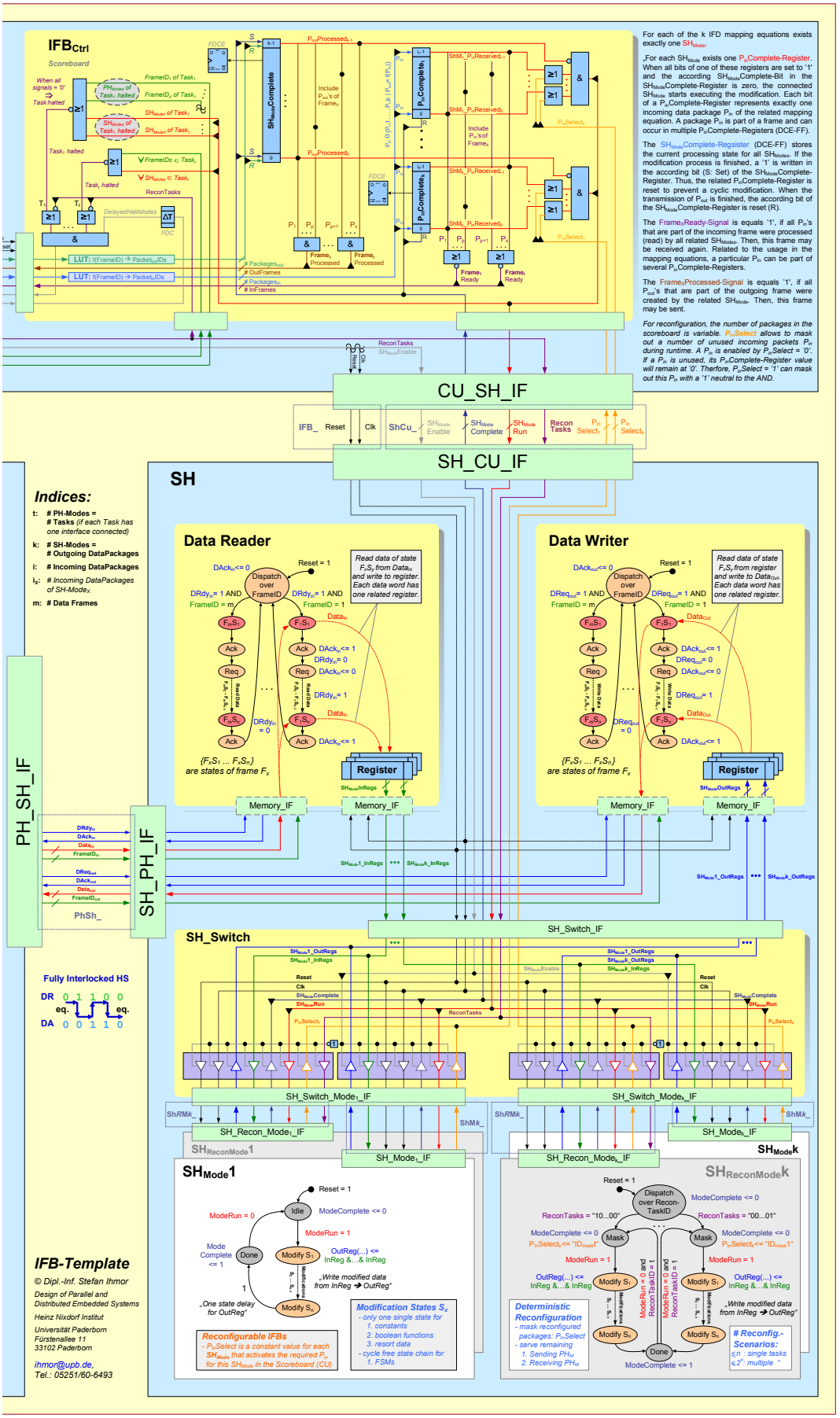


Figure 6.2: IFB Hardware Template (right side)

6.1.1 Protocol Handler

In the following section we discuss the particular modules of the IFB Hardware Template. On the bottom of Figure 6.1 we find the PH including the PH-Modes and the PH-Switch.

PH-Modes

The composition of the PH-Modes has already been dealt with when we introduced the EIFD-Factory (see Section 5.2.8, Figure 5.42). Each PH-Mode comprises a *synthesized PSM* that has been optimized, and provided with the *frame structure* and the *handshaking states* for the IFB internal communication. All signals, which are required by a PH-Mode are routed to the PH-Switch. To save resources, we used only one data bus to exchange incoming and outgoing data with the internal memory. As we mentioned before, this design is sufficient for simplex communication based on pure frames (see Section 5.2.4). In the case of duplex communication, all signals related to the data bus (`Data`, `DA`, `DR`), and the memory bus arbitration signals (`FrameID`, `IRQ`, `ModeRun`) would have to be doubled.

PH-Switch

The PH-Switch is a dedicated circuit that implements a *memory bus* for the *bus access* of the PH-Modes, which is connected to the *memory interface*. To establish the connection to the memory bus, the PH-Switch applies tri-state buffers in combination with a selection logic that is controlled by the coincidence of the `BusGrant` and the `ModeRun` signals.

In the case of an RTR-IFB, the PH-Switch comprises an additional *tri-state bank*, which represents the *bus macros* that are required to reconfigure the PH-Modes (see Section 3.3.1). The tri-states are controlled by the `PhModeEnable` signals coming from the Reconfiguration Unit in the CU.

6.1.2 Sequence Handler

Similar to the PH, we developed the SH including the SH-Switch, the SH-Modes and SH-Recon-Modes, as well as the DataReader and the DataWriter (the IFB internal memory). The SH is depicted on the bottom of Figure 6.2.

SH-Modes & SH-Recon-Modes

The synthesis of the SH-Modes has been already handled in Section 5.2.8. Each SH-Mode comprises a state machine that implements the related mapping function. In Figure 5.43 we illustrated an exemplary SH-Mode which realizes a FSM based mapping function.

When a SH-Mode is inactive, it remains in its *idle* state. The `ShModeRun` signal, which is managed by the scoreboard, starts the processing procedure. After the processing is finished, the SH-Mode activates the related `ShModeComplete` signal to notify the scoreboard.

In the case of the RTR-IFB, we create an additional SH-Recon-Mode for each SH-Mode. In Section 4.4.1, we presented the modeling of reconfiguration behavior (see Figure 4.19) and the construction of SH-Recon-Modes for dedicated reconfiguration scenarios (see Figure 4.18).

The IFB transfers the incoming data from the internal memory to the SH-Modes (and the SH-Recon-Modes) and the processed data back again via the SH-Switch.

SH-Switch

In the presented IFB Hardware Template we assume a fully parallel access of all SH-Modes to the internal memory. Therefore, an explicit memory interface, as it is available for the PH-Modes, is not required. Depending on the kind of memory (for example, if we would employ a Block RAM), we would have to add a memory interface into the SH-Switch that sequentializes the memory access of the SH-Modes. Similar to the PH-Switch, the SH-Switch comprises a tri-state bank, which is added in the case of an RTR-IFB.

IFB Internal Memory: Data Reader & Data Writer

The *DataWriter* realizes the outgoing *IFB internal memory* as visualized in Figure 6.3. On one hand, the memory is connected to the incoming memory bus (left `Memory_IF`); on the other hand, it is interlinked with the SH-Modes via the SH-Switch (right `Memory_IF`). The DataReader is implemented in a similar manner to the DataWriter.

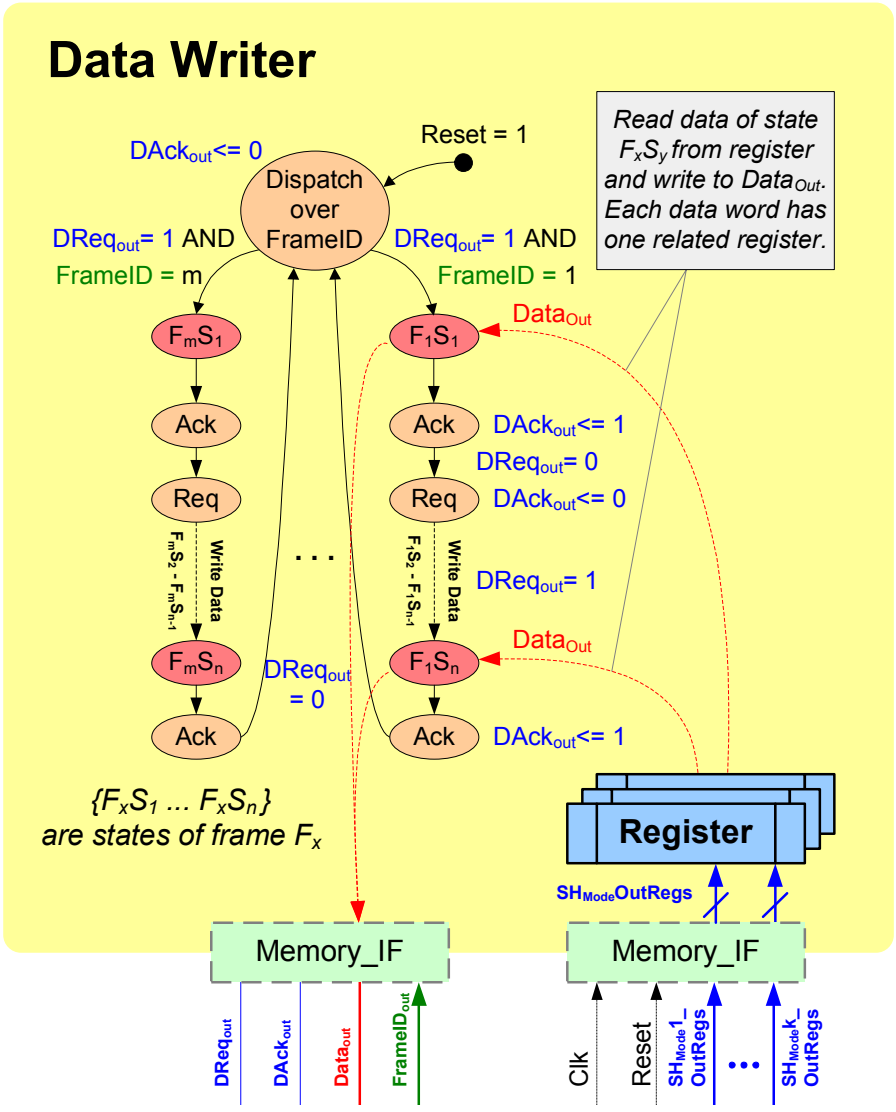


Figure 6.3: The *DataWriter* realizes the outgoing *IFB internal memory*

The DataWriter implements a dedicated memory. Therefore, each data word has it own register where it is stored. Depending on the currently transmitted *frame* (`FrameID`), the *dedicated memory interface* knows exactly which data words have to be read or written. Not to lose data, the memory interface uses a *fully interlocked handshake*. For this reason we inserted the additional *I/O states* inside the frames during the synthesis of the PH-Modes.

6.1.3 Control Unit

To recapitulate, the CU handles three different roles: The arbitration of the memory bus, the memory management, and the control of the reconfiguration process. Therefore, it comprises the components Ctrl_{in}, Ctrl_{out}, the scoreboard, and the Reconfiguration Unit.

Ctrl_{in} and Ctrl_{out} – The Memory Bus Arbiters

We implemented the memory bus arbitration in the form of two parallel highest-priority-first schedulers inside the CU. Due to the use of pure frames, we can implement the schedulers independent of each other. In the case of duplex communication, the two schedulers would have to operate interlocked, as incoming and outgoing frames could overlap (mixed frames).

As presented in Figure 6.4, Ctrl_{in} comprises one `GrantBus` state for each incoming frame (Ctrl_{out} one for each outgoing frame). When a valid `FrameID` and the related `IRQ` are set, the arbiter goes to the respective `GrantBus` state to establish the bus connection inside the PH-Switch. Because of the transition's evaluation sequence in the implementation, the scheduler works as a highest priority first scheduler. When the transmission of the current frame finishes properly, the `IRQ` signal is reset by the PH-Mode and the arbiter goes to the `AssignFrame` state, where it notifies the scoreboard to register the last transmitted packages. Otherwise, in the case of a transmission error, the `FrameID` returns to zero (`"0...0"`) in addition to a reset `IRQ` signal, which forces the arbiter to abort the current transmission. This is done in the state `ReleaseBus`. In Section A.6, we explain the function of the control signals `IRQ` and `FrameID` in detail.

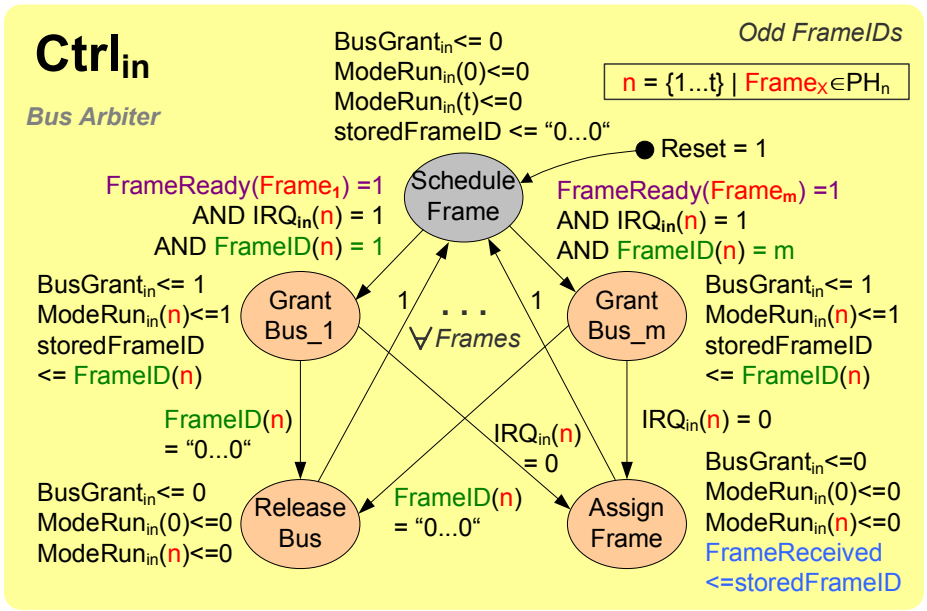


Figure 6.4: The memory bus arbitration implemented as scheduler

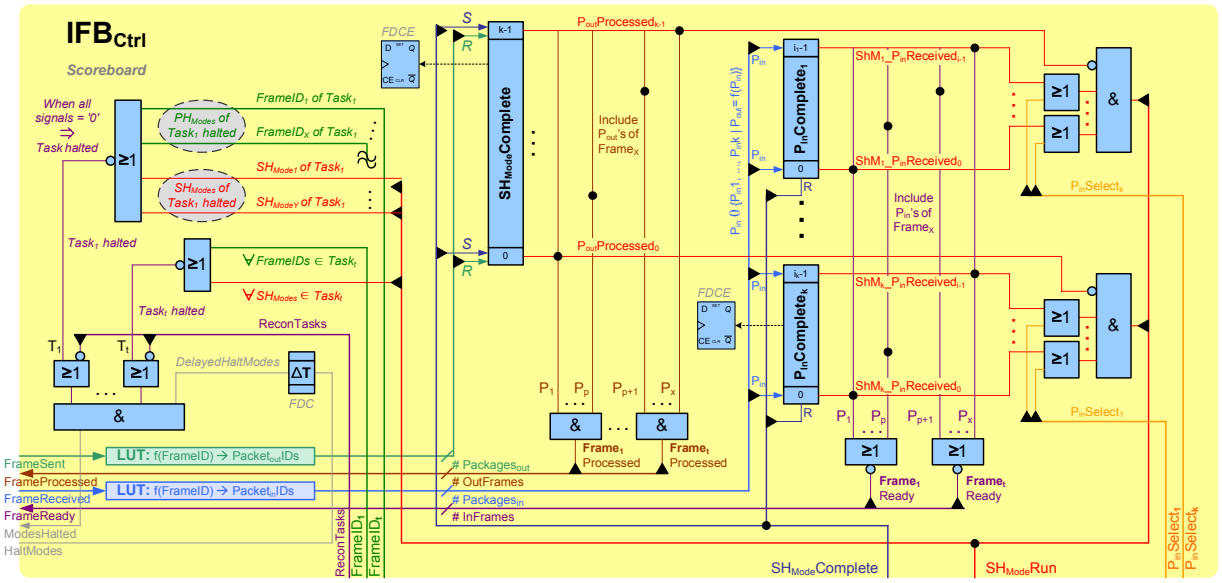


Figure 6.5: The memory management unit, implemented as a scoreboard

IFB$_{Ctrl}$ (Scoreboard) – The Memory Management Unit

The scoreboard manages the internal memory to abide the causality within the pipelined I - P - O process. Therefore, it organizes the status of incoming and outgoing data words in combination with Ctrl$_{in}$ and Ctrl$_{out}$. The scoreboards knows which data words are currently present and if they may be overwritten (after they have been processed), and provides this information to Ctrl$_{in}$ and Ctrl$_{out}$.

To implement this function, there exists a `P`$_{in}$`Complete` register for each SH-Mode. Each bit of a `P`$_{in}$`Complete` register represents exactly one incoming data package P$_{in}$ of the related mapping function. Depending on the mapping functions, a particular package can occur in multiple `P`$_{in}$`Complete` registers, which are realized by Data - Clock - Enable flip flops (DCE-FF). A bit in the `P`$_{in}$`Complete` register is set when the related package has been received (see Ctrl$_{in}$). When all bits of a `P`$_{in}$`Complete` register are set to `'1'` and the according `ShModeComplete`-Bit in the `ShModeComplete` register is equal to zero, the related SH-Mode is started by setting the related `ShModeRun` signal to `'1'` (see SH-Modes). After the SH-Mode has finished processing, the `P`$_{in}$`Complete` register is reset.

The `ShModeComplete` register (DCE-FF) stores the execution status of all SH-Modes. If the data processing of a SH-Mode is finished, the according bit of the `ShModeComplete` register is set. At the same moment the related `P`$_{in}$`Complete` register is reset to prevent a cyclic data processing. When the transmission of an outgoing package P_{out} is finished, the according bit of the `ShModeComplete` register is reset, which is triggered by Ctrl$_{out}$.

The `Frame`$_X$`Ready` signal is set to `'1'`, if all incoming packages that belong to an incoming frame were completely processed by all related SH-Modes. Then, this frame may be received again, i. e., the data word can be overwritten in the DataReader. Depending on the mapping functions a particular package can be part of several `P`$_{in}$`Complete` registers.

The `Frame`$_X$`Processed` signal is set to `'1'` if all outgoing packages that belong an outgoing frame were completely generated by the related SH-Modes. Then, this frame can be sent.

The scoreboard of the RTR-IFB comprises two additional features. First, the number of processed data packages has to be flexible to be able to serve various tasks. Since the CU is not reconfigured, our RTR-IBF design supports a maximum number of adapted interfaces, including a limited number of packages per mapping function. The upper limits have to be specified a-priori at design time. When a mapping function requires less than the maximum number of packages, the missing ones are masked out (at runtime) by the SH-Modes with the help of the P_{in}Select signals. A P_{in} is enabled by setting `P`$_{in}$`Select` to `'0'`. The same technique is used by the SH-Recon-Modes to disable packages which are temporarily not available because of the reconfiguration of the related tasks. Secondly, the scoreboard must be able to detect if all reconfigured modes are halted. The affected PH-Modes are idle when their `FrameID` is equal to `"0...0"`; the SH-Modes when the related `ModeRun` signal is zero. We located the dedicated circuitry (gate logic) on the left side the scoreboard.

Reconfiguration Controller

The Reconfiguration Unit organizes the *IFB internal reconfiguration*. Therefore, we create this unit only for the RTR-IFB. It is the job of the reconfiguration controller to halt and to disconnect the PH-Modes and the SH-Modes that are going to be reconfigured. Therefore, the RU interacts with the RCU, the scoreboard, the memory bus arbiters, and the switches. In Section A.5 we introduce a template for the RCU (see Figure A.3) and describe the communication with the RU. The RU implements the reconfiguration flow as presented in Section 4.4.1. After the RU detects the `ReconStart` signal in the `idle` state, it stalls the affected PH-Modes by masking out their `IRQ` signal (state `HaltModes`). When both the transmission of all frames inside the affected PH-Modes, and the depending execution of the SH-Modes has finished, the scoreboard activates the `ModesHalted` signal and the RU cuts off the modes inside the `DeactivateModes` state. To decode the affected modes we integrated two LUTs in the CU. Then, the CU goes directly to `ReconfigureModes` where it notifies the RCU to perform the reconfiguration. After this is finished the RU reactivates the replaced modes in state `ActivateModes` and goes back to the `idle` state.

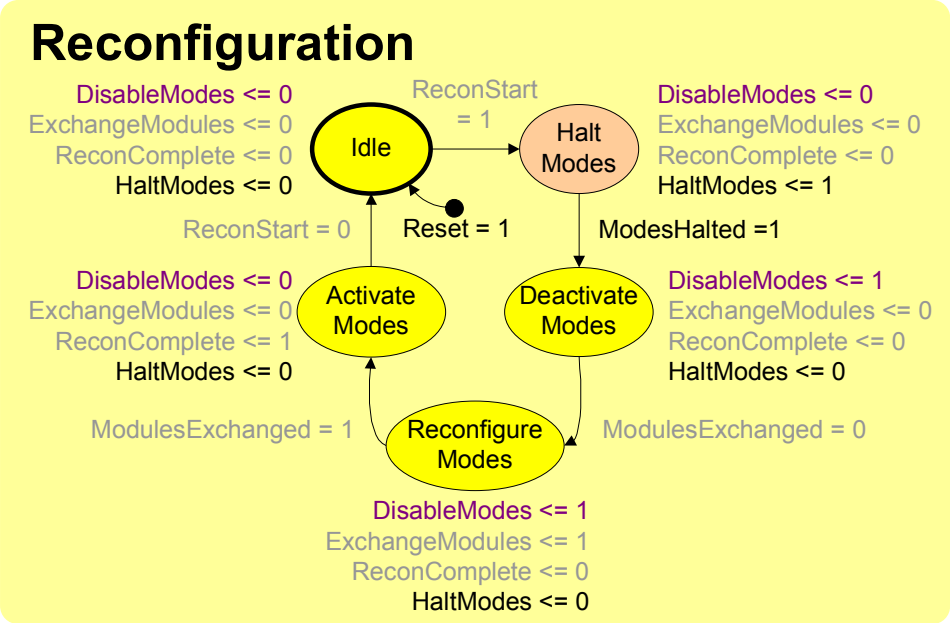


Figure 6.6: The Reconfiguration Unit

6.2 Cycle Accurate Analysis of an IFB

After explaining the IFB Hardware Template we can now refine the I - P - O process that we introduced in Section 4.3.1. Table 6.1 presents the minimum sequence of actions to process one data bit including the internal *fully interlocked handshake* protocol. The detailed I - P - O process is relevant for the timing analysis and the optimization approaches.

Clock cycle	Phase	Action	Related Component
1	1	Set FrameId	PH-Mode (in)
2	1	Set IRQ	PH-Mode (in)
3	1	Grant incoming memory bus access	CTRL_{In}
4	2	Receive data	PH-Mode (in)
5	2	Set DR (data ready)	PH-Mode (in)
6	2	Transfer data into memory	Data Reader
7	2	Set DA (data acknowledge)	Data Reader
8	2	Reset DR	PH-Mode (in)
9	2	Reset DA	Data Reader
10	3	Reset IRQ	PH-Mode (in)
11	3	AssignFrame / ReleaseBus	CTRL_{in}
12	3	Update PInComplete registers	Scoreboard
13	4	Process data (minimum sequence)	SH-Mode
14	4	Set ShModeComplete signal	SH-Mode
15	4	Set ShModeComplete register	Scoreboard
16	5	Set FrameId	PH-Mode (out)
17	5	Set IRQ	PH-Mode (out)
18	5	Grant outgoing memory bus access	CTRL_{out}
19	6	Set DR (data request)	PH-Mode (out)
20	6	Put data on internal bus	Data Writer
21	6	Set DA (data acknowledge)	Data Writer
22	6	Transmit data	PH-Mode (out)
23	6	Reset DR	PH-Mode (out)
24	6	Reset DA	Data Writer
25	7	Reset IRQ	PH-Mode (out)
26	7	AssignFrame / ReleaseBus	CTRL_{out}
27	7	Reset ShModeComplete register	Scoreboard

Table 6.1: Refined I - P - O sequence

The refined I - P - O sequence is a *cycle accurate* representation of the IFB internal data processing procedure. We cluster the 27 operations into seven consecutive *phases*:

- **Phase 1 & Phase 5 :** Establish (incoming/outgoing) memory bus connection
- **Phase 3 & Phase 7 :** Release (incoming/outgoing) memory bus connection
- **Phase 2 :** Receive data
- **Phase 4 :** Process data
- **Phase 6 :** Transmit data

IPO pipeline	Phase	Operation	IFB component	Clock cycles
I	1)	Establish	PH, Ctrl	3
	2)	Receive Data	PH, SH	$6 \cdot \alpha$
	3)	Release	PH, Ctrl, Scoreboard	3
P	4)	Process	SH, Score.	$2 + \gamma$
O	5)	Establish	PH, Ctrl	3
	6)	Transmit Data	PH, SH	$6 \cdot \beta$
	7)	Release	PH, Ctrl, Scoreboard	3

α : Number of successive incoming data bits
β : Number of successive outgoing data bits
γ : States required for the protocol transformation

Table 6.2: I - P - O clock cycles, required to process multiple data bits

Table 6.2 depicts the cycle accurate clock cycles required to transform α successive input bits into β successive output bits with the help of γ processing operations. Thereby, we define γ as the maximum number of sequential modification states in the related SH-Mode. As presented in Table 6.1 it takes three clock cycles to establish and to release the memory bus connection when the memory bus is idle. The timing analysis, presented in the succeeding section, has to assure that this condition is always met in order to avoid resource conflicts, which could invalidate the deadline conform data processing. Based on the fully interlocked handshake, it takes six clock cycles to transfer a bit between a PH-Mode and the memory.

6.3 Timing Analysis

The IFB is a transparent protocol converter. Therefore, its operation must not disturb or delay the communication between timed protocols. In order to fulfill this condition, the IFB internal data processing speed (clock rate) has to be fast enough to serve the particular protocols. In the following section, we *formalize* this aspect and present a *feasibility analysis*, which determines the minimum IFB clock rate for the processing of a given protocol. A successful feasibility test for all adapted protocols is a necessary criteria for the protocol adaption in real-time.

When multiple interfaces are connected to an IFB, the feasibility of the particular protocols is not sufficient to guarantee the proper execution of multiple timed protocols. Therefore, we developed a *schedulability analysis* which guarantees the deadline conform execution of the adapted real-time protocols.

6.3.1 Feasibility Analysis

From the viewpoint of the task, the pipelined protocol adaptation, performed by the IFB, is comparable to a pipelined processor architecture, that requires several internal operations (clocks cycles) to process an instruction word inside one of its stages. The instruction fetch process, for example, is controlled by a micro programm that comprises several states.

Our *feasibility test* is *successful* when the platform delivers a clock that is fast enough to trigger the IFB. Afterwards, we explain how to determine the respective IFB clock rate.

In Section 2.2.2 we introduced the term *Cycles-Per-Bit* (CPB) that defines the maximum number of clock-cycles which are required to *process one bit* inside the IFB. To perform the protocol adaptation for a given protocol frequency f_{prot}, the selected IFB clock frequency f_{IFB} that triggers the synchronous IFB has to fulfill:

$$f_{IFB} \geqslant CPB \cdot f_{prot}$$

In general $f_{prot} = \frac{1}{t_{prot}}$, where t_{prot} is the shortest time interval of the adapted protocol. In synchronous protocols, f_{prot} is usually defined as: $f_{prot} = \textit{clock rate}$ [Hz].

The cycle accurate analysis of the IFB Hardware Template indicates that we require a minimum of 27 clock cycles to process a particular bit, which is a non-satisfying result, since the IFB clock has to be at least 27 times faster than the one of the demanded protocol.

However, we did not yet consider the pipelined I - P - O processing. With the help of the previously introduced *phases* we can determine a realistic CPB value. This value is derived from the most time consuming operation of the I - P - O pipeline – the handling of the first data bit inside a frame, which implies the establishment of the memory bus connection:

$$f_{IFB} \geqslant (3+6) \cdot f_{prot} \Rightarrow CPB = 9 \quad \text{(Including the “establish bus” states)}$$

As we can see in Table 6.2, the phases 2) and 6) require exactly 6 states (clock cycles) to read or write a bit. Additionally, we have to establish the memory bus connection for the first bit (phase 1) and 5)), which requires 3 clock cycles. Therefore, $CPB = 9$ for the pipelined IFB version. The 3 clock cycles required to release the bus (phase 3) and 7)) can be neglected, as they account for the following control state. If we can integrate the establishment of the memory bus connection into two existing control states of the synthesized PSM right before the frame, the CPB value decreases to 6. At the same time, a CPB value of 6 is also the minimum possible value, as long as we insist on the fully interlocked memory I/O.

$$f_{IFB} \geqslant 6 \cdot f_{prot} \Rightarrow CPB = 6 \quad \text{(Optimum I - P - O pipeline)}$$

I - P - O Pipeline Clock Rate

In accordance to Table 6.2, we define the *I - P - O pipeline clock rate* as f_{IFB} divided by the maximum number of states in one of the I - P - O pipeline stages ($\#S$: number of states) :

$$\begin{aligned} f_{IPO} &= min(f_I, f_P, f_O) = \frac{f_{IFB}}{max(\#S_I, \#S_P, \#S_O)} \\ &= \frac{f_{IFB}}{max(6+6\alpha\,,\ 2+\gamma\,,\ 6+6\beta)} = \frac{f_{IFB}}{2+max(4+6\alpha\,,\ \gamma\,,\ 4+6\beta)} \end{aligned}$$

For a very large number of processed data bits (here we assume : 1 kBit) and a restricted number of sequential modification states (here : 5), the pipeline clock converges to:

$$f_{IPO_{1024}} = \frac{f_{IFB}}{2+max(4+6\cdot 1024\,,\ 5\,,\ 4+6\cdot 1024)} \approx \frac{f_{IFB}}{6\cdot 1024} = \frac{f_{prot}}{max(\#S_I, \#S_O)}$$

As we could expect, the I - P - O pipeline clock rate of the optimum IFB ($CPB = 6$) converges to the protocol's clock rate divided by the maximum number of successively transmitted bits.

6.3.2 Schedulability Analysis

If we adopt only one sending and one receiving interface, a successful feasibility analysis is also a valid schedulability test, as the executed components run parallel and do not share any resources. Otherwise, we require an additional *schedulability analysis* to assure that a *valid schedule* for the *memory bus arbitration* exists.

Our idea for the schedulability analysis is to evaluate the protocol descriptions of the adapted interfaces in order to process the *input* for a *standard scheduling approach*. Therefore, we *reduce* each protocol to a *representative worst case basic block* BB' that we interpret as a task with the attributes deadline, period, and execution time. To construct a *static schedule* for this task set, we utilize the standard scheduling policy *cyclic scheduling*, which is applicable for non-preemptive tasks. The successful construction of a schedule passes also for a valid *schedulability test*. A detailed explanation of the presented schedulability analysis approach has been published in [25].

To compute the *worst case basic block* BB', we developed a worst case execution time (WCET) analysis. Thereby, we make use of the fact that a PH-Mode operates continuously and parallel to the other PH-Modes. The first aspect means that there is exactly one basic block executed at each point of time. The second one indicates that all PH-Modes are independent of each other and thus can be handled separately. To apply our WCET analysis, we make the following assumptions:

1. There exists a successful feasibility test for each protocol
2. Neither a task nor a PH-Mode stalls the protocol execution
3. A valid IFD-Mapping has been entered (no deadlocks etc.)
4. The protocol must not possess dead states
5. The transmission of a frame cannot be interrupted $\Leftrightarrow$ Frames are non-preemptive
6. Deadlines and periods have been modeled by the designer
7. The protocol must not possess more that one infinite loop (constant ones are flattened)

The scheduling of incoming and outgoing frames can be handled separately, since our schedulers are working independently. Therefore, the scheduling of incoming frames is equivalent to the scheduling of the outgoing frames. As depicted in Figure 6.7 we define the *real-time specific parameters*: *deadline*, *period*, and *execution time* to compute the BB' [70]:

$$\begin{array}{lll} \text{Period}: & t_P & \Leftrightarrow \quad \text{Frequency}: \; f_P = \frac{1}{t_P} \\ \text{Deadline}: & t_D & \\ \text{Execution time}: & t_{ex} & \end{array}$$

The frequency f_P defines the period of the cyclic execution of a BB. It has to be specified for each basic block. We model f_P by a timed self transition (timerOrDeadline). The relative deadline t_D represents the maximum time until the transmission (execution) of a frame has to be completed. We model t_D as a timerOrDeadline that spans the related frame. The execution time t_{ex} specifies the time to transmit a frame. It does not have to be modeled explicitly as we can compute its value by the frameWidth divided by the clock rate f_{IFB}:

$$t_{ex} = \frac{\text{frameWidth}}{f_{IFB}} = \frac{\#S_{establish} + \#S_{data/handshake} + \#S_{release}}{f_{IFB}} = \frac{6 \cdot \#S_{data} + 6}{f_{IFB}}$$

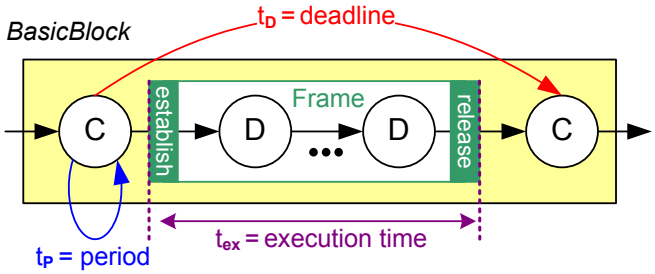


Figure 6.7: Definition of the real-time attributes period, deadline and execution time

Afterwards, we present the developed algorithm for the WCET analysis. It systematically *reduces* the given protocol to construct the BB' by applying *graph operations* on the basic block CFG (for the construction of the CFG see Section 5.2.2).

WCET Analysis Algorithm

As mentioned previously, one condition to apply our approach is that all inner loops are restricted. In that case, we can flatten all loops by unrolling the covered basic blocks. As you can see in Example Code 6.1 in line two, we perform the loop unrolling as the first operation of the protocol reduction procedure (see Section 5.2.1). Afterwards, we reduce the flattened protocol by merging *sequential* and *parallel basic blocks* until we obtain a single BB. This BB is the requested BB', which is returned by the method `getWCETBB()`.

Example-Code 6.1: Protocol reduction pseudo code

```
01  public BasicBlock reduceProtocol (Protocol p) {
02    flattenLoops(p);
03    while(p.numberOfBBs > 1) {
04      mergeSequentialBBs(p);
05      mergePrallelBBs(p);
06    }
07    return p.getWCETBB();
08  }
```

Figure 6.9 depicts how to merge a set of *sequential basic blocks*. Thereby, BB' has to comprise the worst case for each attribute. The following equations present the merging functions for sequential basic blocks, implemented in `mergeSequentialBBs(p)`:

$$\text{Sequential BB:} \qquad t'_{ex} = \sum_{i=1}^{k} t_{ex_i} \qquad t'_D = \sum_{i=1}^{k-1} t_{ex_i} + t_{D_k} \qquad f'_P = max_i(\, f_{P_i}\,)$$

To compute the worst case execution time t'_{ex}, we sum up the sequential execution times of the frames inside the succeeding basic blocks. As the resulting BB' is non-preemptive and our deadlines are specified as relative deadlines, we first sum up $k-1$ execution times plus the last deadline to obtain t'_D. Thereby, the worst case period f'_P in a sequence of frames depends on the maximum available period. In this way, BB' provides the worst case properties of the frames inside the substituted basic blocks.

Figure 6.9 illustrates how to *substitute* a set of *parallel basic blocks*. The substituted attributes are computed by the following equations, implemented in `mergeParallelBBs(p)`:

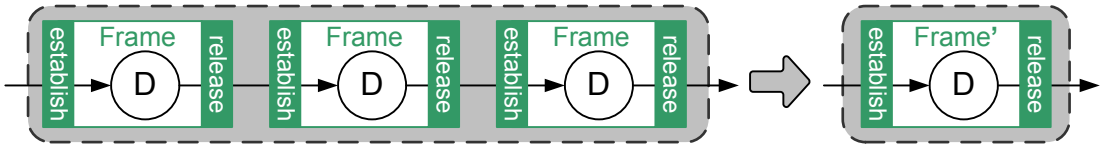


Figure 6.8: Reduction of serial frames

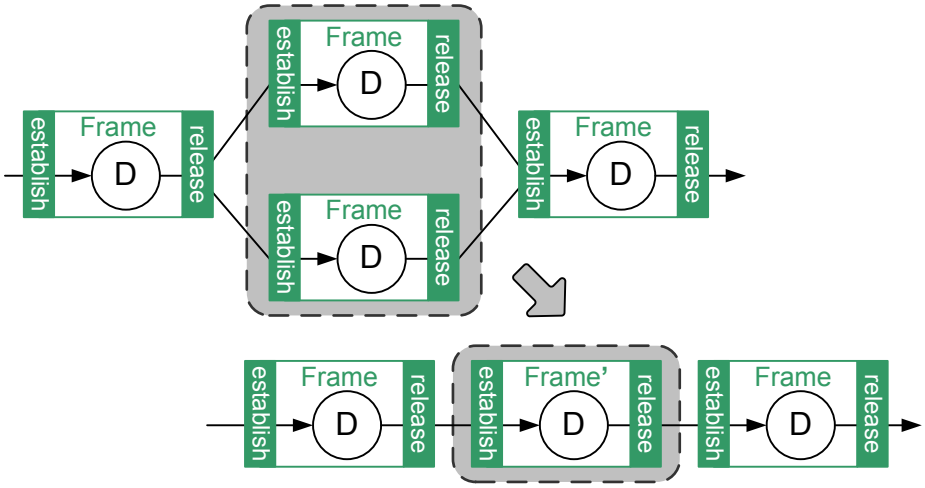


Figure 6.9: Reduction of parallel frames

$$\text{Parallel BB}: \qquad t'_{ex} = max_i(\, t_{ex_i}\,) \qquad t'_D = min_i(\, t_{D_i}\,) \qquad f'_P = \sum_{i=1}^{k} f_{P_i}$$

For the worst case execution time t'_{ex} we select the maximum value of the parallel basic blocks. In contrast, we have to take the minimum deadline to obtain t'_D. To be able to compute all substituted BBs within the specified periods, we sum up the given periods to f'_P. We could apply weighted sums and thus make a more accurate WCET estimation, if we would possess the probability distribution for the execution of the alternative BBs.

If all parameters were properly specified by the designer, the algorithm delivers the *WCET* BB' for each protocol. We treat these BB' as a *periodic* and *non-preemptive task set* that can be scheduled, for example, by the *cyclic scheduling policy*. When a scheduling can be constructed, the task set can be scheduled, which means, the protocols can be executed deadline conform. Otherwise, the feasibility cannot be proved although it might be possible. If any of the parameters *deadline* or *period* is *missing*, we cannot perform the schedulability analysis. In this case, the schedulers could implement *priority based strategies* like "Highest Priority First" or "First Come First Serve", which are not yet real-time capable.

Example of the WCET Analysis

Figure 6.9 demonstrates an example for the WCET algorithm. The exemplary protocol comprises three BBs. In the first and the third step we substitute sequential basic blocks. Thereby, we have to merge the sequential frames inside a basic block in the first iteration of `mergeSequentialBBs(p)`, before we can substitute complete basic blocks. In step two and step four the algorithm substitutes the parallel BBs, whereas there are no more parallel BBs to merge in step four. Then the exit condition is fulfilled and the algorithm terminates.

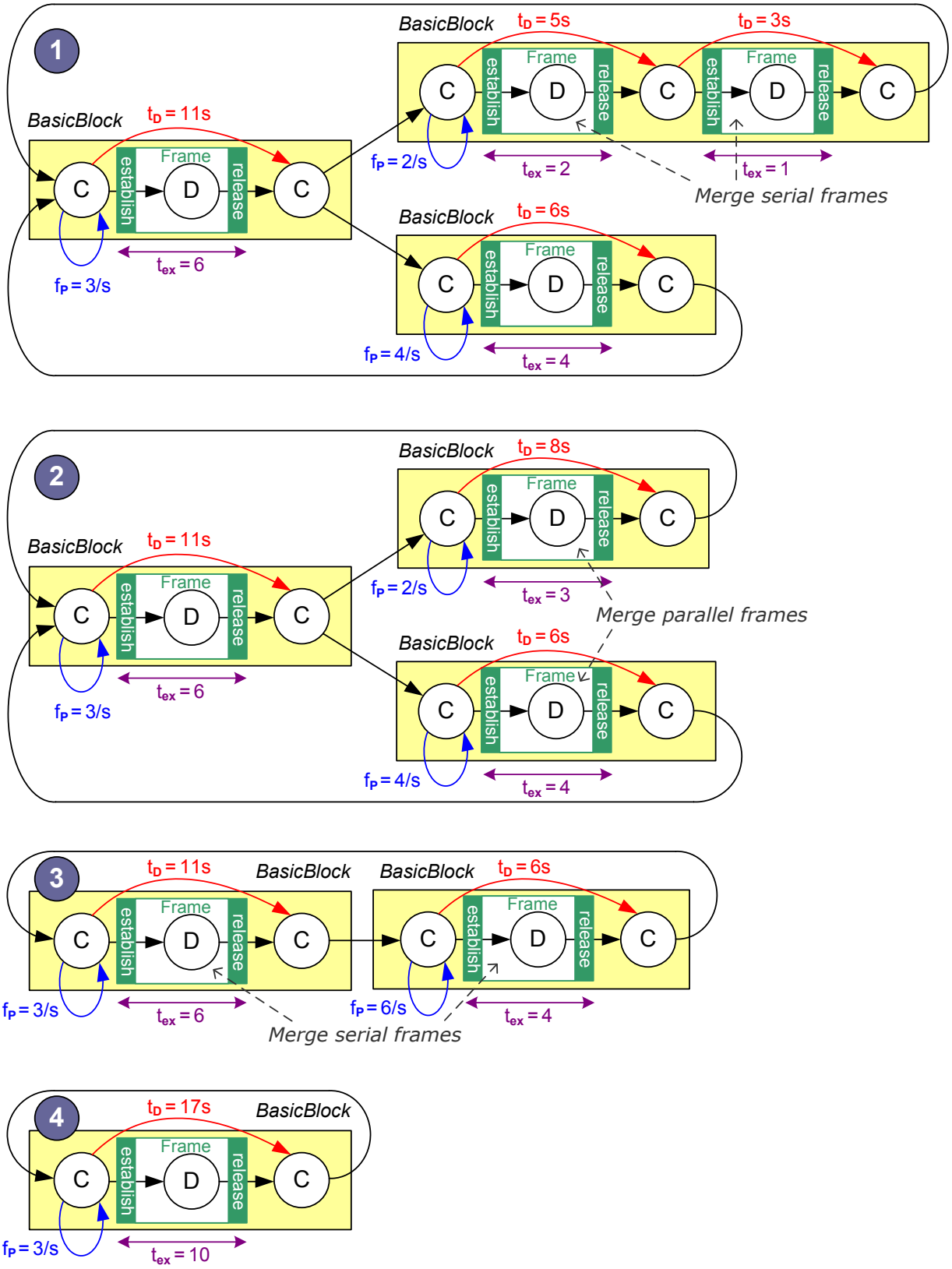


Figure 6.10: Exemplary reduction of a protocol

6.4 IFB Optimization

In this chapter, we present two *optimizing approaches* for the hardware design of the IFB. The first one reduces the *latency* caused by the IFB. The second approach focusses on the *reconfigured execution* of an IFB, based on the *micro reconfiguration* technique. Thereby, we utilize *caching strategies* to optimize the *required chip area*. However, as both optimization approaches exclude each other, we cannot apply them simultaneously. The details of the IFB optimization have been published in [5, 1, 26].

6.4.1 Data Flow (Latency) Optimization

The data flow optimization takes advantage of the pipelined IFB execution. Therefore, it is our goal to *maximize* the *I - P - O parallelization*, which can be achieved by starting the three stages of the I - P - O pipeline as soon as possible. To explain the developed pipeline optimization, let us recapitulate the standard I - P - O pipeline. As shown earlier, the protocol adaptation is executed repeatedly in the form of communication cycles (see Section 4.4.1). To guarantee a correct *scheduling* of the *I - P - O pipeline stages* within the communication cycle, the CU follows the *causality conditions*, defined in Section 4.3.1.

The minimum scenario for the standard I - P - O pipeline (called *simple schedule*) consists of one input - and one output frame. In Figure 6.11 we illustrate two communication cycles of the simple schedule in the form of a *Gantt diagram*. Further cycles are generated analogous. The input frame is mapped to *input stage* `I`, the protocol transformation to *processing stage* `P`, and the output frame to *output stage* `O`. The numbers (e.g. in `I1`) characterize the iteration of the communication cycle. In the standard I - P - O pipeline, the earliest moment to receive `I2` is after `P1` has been completed. Otherwise, `I2` would overwrite currently processed data in the internal memory. Similarly, `P2` may be started only after `O1` and `I2` have finished.

In the simple schedule, both input and output stages comprise exactly one frame. Therefore, it is not possible in this scenario to optimize the latency by advancing the execution of a stage as the execution of a frame may not be interrupted. However, if an outgoing frame is a *super-frame*, including multiple *package instances*, an effective optimization is possible.

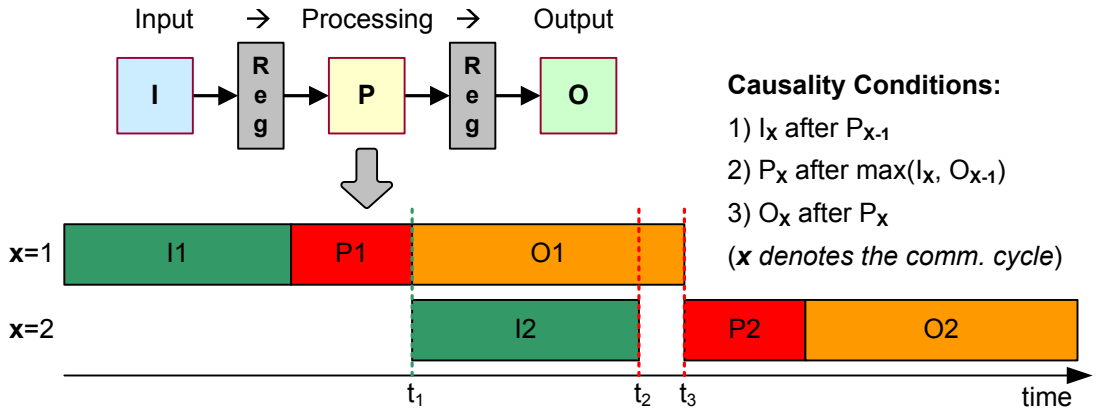


Figure 6.11: Two communication cycles (simple schedule) depicted as Gantt diagram

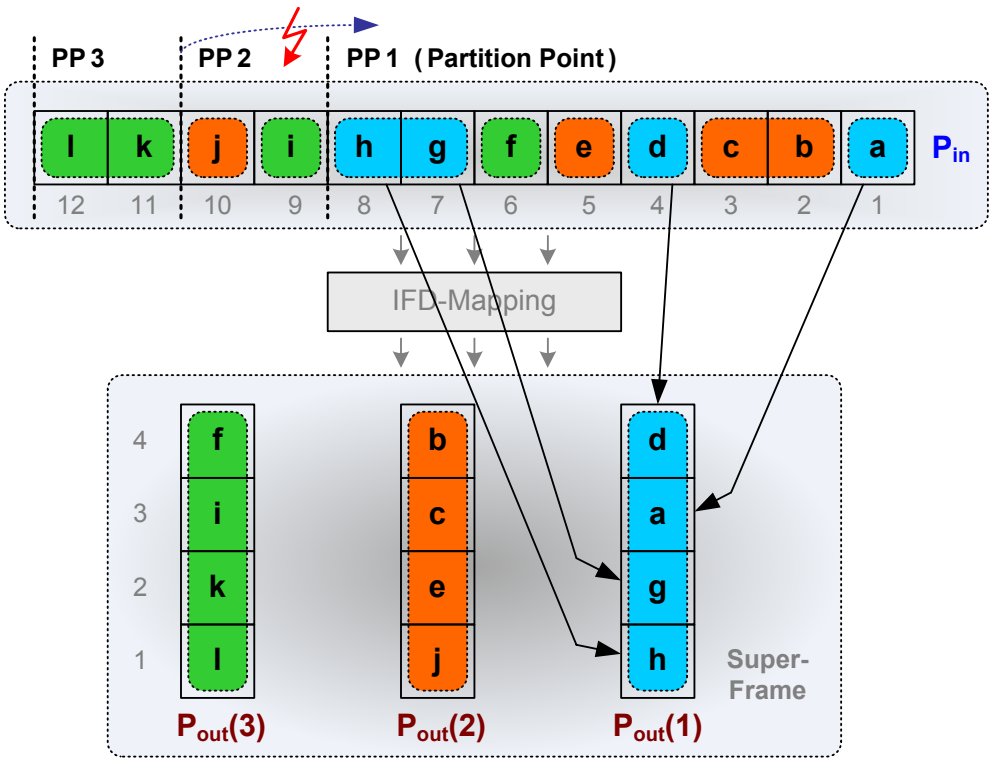


Figure 6.12: Advanced execution of the processing- and the output stage

Instead of receiving complete incoming frames, which might also be super-frames, we start processing the incoming data when all information related to the first (second, ...) package instance of the outgoing super-frame has been received and stored inside the internal memory. By analyzing the protocol description and the scenario based IFD-Mapping, we can identify all points in time when an *advanced stage execution* is possible.

Sub-Frames and Partition Points

We call these moments "*partition points*", as they divide the incoming frames into *partitions*, the so called *sub-frames*. As we can see in Figure 6.12, we insert a partition point $\texttt{PP}_n$ into the incoming frame whenever the information for the package instance $\texttt{P}_{out}\texttt{(n)}$ of the outgoing super-frame is ready for its transformation. Thereby, we insert the partition point $\texttt{PP}_n$ in such a way that it is scheduled at the earliest in combination with $\texttt{PP}_{n-1}$ (for $n > 1$). This means we delay the processing of a package – although its required input is completely available – until the data of the preceding package instances has been completely received. For this reason, our optimization technique is only reasonable when the information related to the first instances is located "not too far" at the end of the incoming frame.

We do not alter the transformation sequence as the output sequence is fixed, anyway. In our hardware architecture, the data processing works fully parallel and takes only a few clock cycles (modify states). Therefore, we can achieve a gain by reorganizing the data processing sequence, which requires an additional control, only if complex data processing operations in the form of large state machines have to be executed.

The partition points and the sub-frames are relevant for the optimization approaches presented in the next passages: the *frame-based* and the *sub-frame-based schedule*.

Frame-based Schedule

The frame-based schedule retains the given frame structure, including the request - and the release operations. To mark the particular sub-frames inside a frame, we assign *sub-frame IDs* as presented in Figure 6.14 on the left side. For the automated evaluation within the IFB synthesis, we have to integrate the sub-frame IDs as an additional information into the protocol description.

Figure 6.13 depicts the frame-based schedule that results from the combination of the simple schedule and the previously discussed IFD-Mapping. In accordance to the three partition points, the incoming frame `I1` has been divided into the sub-frames `I1.1`, `I1.2`, and `I1.3`. The same is true for `P1` and `O1`. In this way, we obtain a maximization of the *I - P - O parallelization* by subdividing the pipeline stages into the smallest reasonable sections.

Similar to the simple schedule, we execute the input stage non-preemptive. The processing stage assembles the outgoing package instance $\texttt{P}_{out}\texttt{(n)}$ when the succeeding sub-frame ID, identified by $\texttt{PP}_{n+1}$, becomes valid. This means, the incoming frame advanced to the next partition and the information required to process $\texttt{P}_{out}\texttt{(n)}$ is now available. Due to the stall effect, there might be even more package instances ready to be processed. After the processing stage $\texttt{P}_{X.Y}$ finished the transformation of these packages, the corresponding output stage $\texttt{O}_{X.Y}$ is allowed to transmit the completed packages. We are allowed to stall the transmission of particular package instances inside a super-frame as they belong to self-containing BBs. The transmission of a BB itself may not be preempted.

To implement the frame-based schedule, which means to handle sub-frames and sub-frame IDs in the IFB, we have to provide additional signals to transmit and control states to handle this information. One solution could be to insert a sub-frame ID signal (PH to Ctrl_{in}) and to extend the `ShModeRun` (CU to SH) as well as the `FrameProcessed` signals (scoreboard to Ctrl_{out}). With the help of the additional information, the SH-Modes can be enabled to process super-frames stepwise. Similarly, we can extend the PH-Modes to separately transmit outgoing package instances of a super-frame. Another way that goes without modifying the CU is a direct coupling of the PH-Modes and the SH-Modes with the help of a bypass that forwards the information related to the sub-frame IDs.

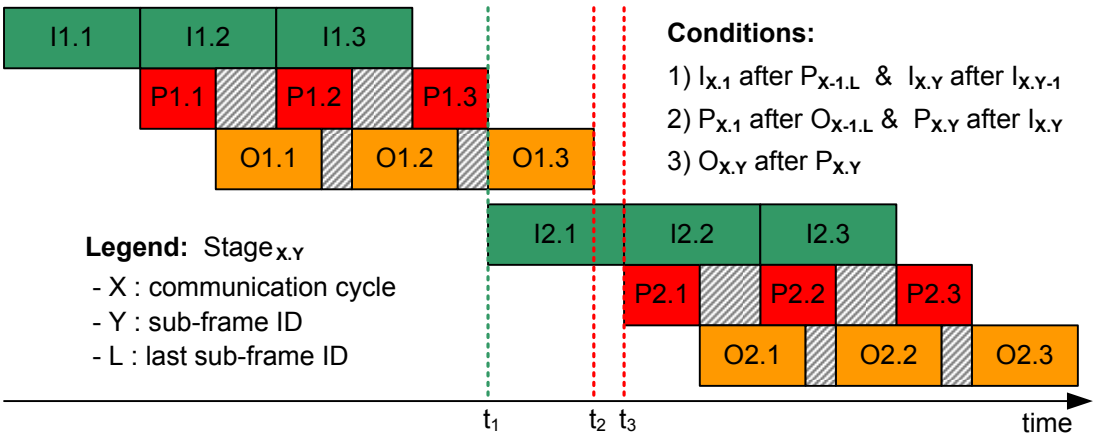


Figure 6.13: Latency optimization approach I) – The frame-based schedule

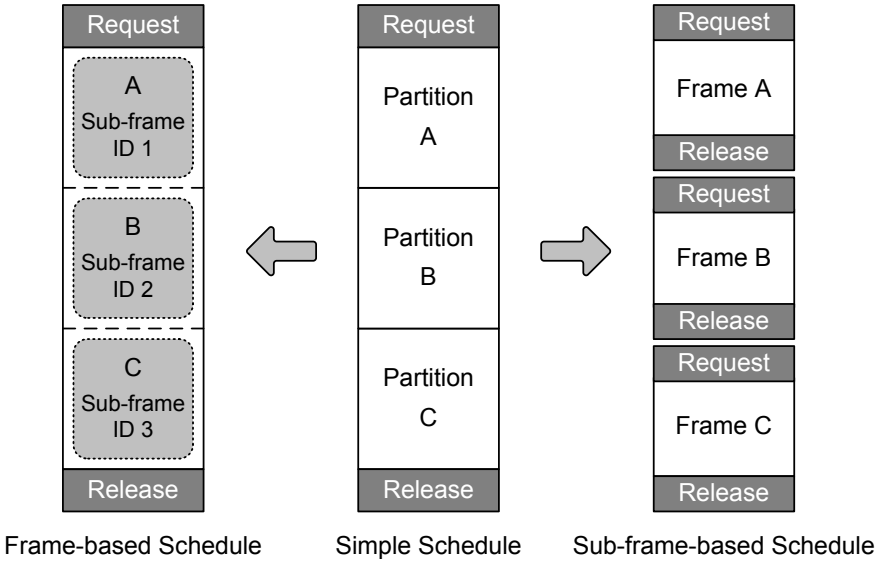


Figure 6.14: Integration of the optimization approach into the control unit

Sub-frame-based Schedule

In contrast to the frame-based schedule, the sub-frame-based schedule modifies the original frame structure. As depicted on the right side of Figure 6.14, it divides the incoming frame into one self-contained frame for each partition, which is labeled with the sub-frame ID. To receive a frame, the incoming memory bus has to be requested for the transmission of each created (sub-) frame. Due to the fact that we permit the interruption of an ongoing transmission, this strategy cannot be applied for timed protocols in general.

The detailed scheduling conditions for the sub-frame-based schedule are given in Figure 6.15. As an advantage of this approach, the input stage $\texttt{I}_{X.Y}$ does not have to wait until the last processing stage $\texttt{P}_{X-1.L}$ of the previous communication cycle has finished. Due to the division into separate frames, it has already started when $\texttt{max(I}_{X-1.L}\texttt{, P}_{X-1.Y}\texttt{)}$ has been computed. Therefore, the pipeline stages are even more interlocked than in the frame-based approach.

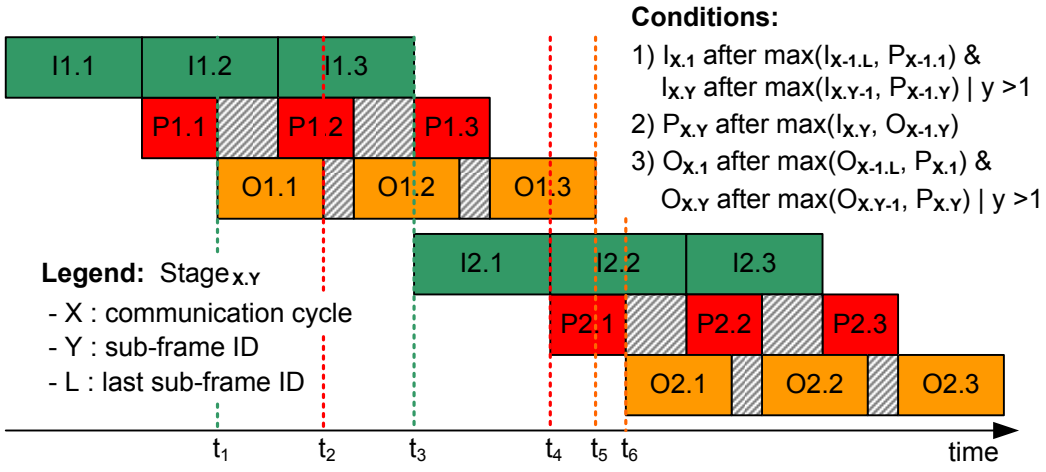


Figure 6.15: Latency optimization approach II) – The sub-frame-based schedule

To implement the sub-frame-based schedule, the IFB Macro-Structure does not have to be modified. The only effort is the generation of the new frame structure based on the identified sub-frames in a preprocessing step. However, this optimization generates additional overheads: each generated frame includes the states which are required to request and to release the internal memory bus. In the following section, we compare the two optimization approaches and show how far the additional overheads reduce the achievable yield.

Results

To evaluate the optimization approaches, we have performed a *cycle accurate simulation* of multiple communication cycles. This means an ASAP (as soon as possible) scheduling of the 27 I - P - O process steps according to the presented schedule conditions. The detailed schedules are given in [1, 26]. As the results depend on the scenario based IFD-Mapping, the simulation has to be performed for each scenario to obtain the dedicated schedules.

To compare the performance of the three approaches we refer to the optimum scenario for this purpose, the trivial forwarding of data. Therefore, we assume a serial incoming and a serial outgoing frame of the same size, whereas the data is ordered in such a way that the first received bit is also transmitted first. At the top of Table 6.16 we depicted the simulation results for an 8-bit data word, below for 1 kBit. The tables deliver the necessary clock cycles that are required for the execution of two communication cycles when the incoming protocol frame is consecutively divided into one, two, four, and eight sub-frames.

The simple schedule adds up to 166 clock cycles for the 8-bit and to 18.455 clock cycles for the 1024-bit word. This is equal to the results of the two optimization approaches in case of one sub-frame. Including multiple sub-frames, the frame-based schedule performs always better than the sub-frame-based or the simple schedule. In general, we can say: the smaller the sub-frames, the higher the achieved optimization. However, the sub-frame-based schedule improves continuously only until the sub-frame size becomes less than three bits. Below this critical point, the overhead for the additional request and release states exceeds the achieved yield. In this case, the sub-frame-based schedule becomes even worse than the simple schedule. Despite the fact that the frame-based scheduling performs better than the sub-frame-based approach, it requires modifications of the IFB architecture, while the sub-frame-based schedule gets along with a preprocessing step that subdivides the frames.

Approach	8 Bits	2x4 Bits	4x2 Bits	8x1 Bit
Simple Schedule	166	-	-	-
Frame-based Schedule	166	139 (83,7%)	124 (74,7%)	118 (71,1%)
Sub-frame-based Schedule	166	151 (91,0%)	199 (119,9%)	205 (123,5%)

Approach	1024 Bits	2x512 Bits	4x256 Bits	8x128 Bits
Simple Schedule	18.455	-	-	-
Frame-based Schedule	18.455	15.377 (83,3%)	13.840 (75,0%)	13.060 (70,8%)
Sub-frame-based Schedule	18.455	15.391 (83,4%)	13.878 (75,2%)	13.150 (71,3%)

Figure 6.16: A qualitative evaluation of the optimization approaches

6.4.2 Area Optimization

The second optimization approach minimizes the required *chip area* of the IFB in order to decrease the *IFB implementation costs*. Thereby, we make use of the property that each stage of the I - P - O pipeline implies an idle-time where it can be replaced by another stage. Exchangeable stages can share the same execution resources, which provides us with the potential to minimize the required chip area. We use *caching* to optimize the displacement of I - P - O stages, which highly determines the efficiency of our approach.

To replace the I - P - O stages – implemented as PH-Modes and SH-Modes – we demand a reconfigurable execution platform. In order to exchange I - P - O stages at runtime, we exploit the *micro reconfiguration*, which leads to a *reconfigured IFB execution* (see Section 4.4). The details about the area optimization have been published in [5, 1, 26].

The Runtime Reconfigurable I - P - O Pipeline

As presented in Section 4.3.1 the I - P - O scheme involves a three staged pipeline. In order to consider the *hardware reconfiguration* aspect, we *subdivide each stage* into a reconfiguration step (L_X : load stage X) and an execution step (E_X : execute stage X). Thus, we enhance the three basic I - P - O stages to a total number of six stages as depicted in Figure 6.17. The load step is optional, as there is no need to load stages which are already present on the execution platform. Therefore, the runtime reconfigurable I - P - O pipeline provides bypasses which are able to skip needless load stages. The registers, which are located between the I - P - O stages, represent the IFB internal memory. Similar to common pipeline architectures, the I - P - O pipeline knows three kinds of pipeline conflicts:

1. **Structural Conflicts**: All PH-Modes have to apply for the same memory bus interface to exchange data with the internal memory. Therefore, input and output stages are *mutual exclusive*. This means, a maximum of one input and one output stage can be executed simultaneously although several of them might be present on the FPGA.
2. **Control Conflicts**: The macro reconfiguration, used to reconfigure a task, represents a control conflict. As effect of this conflict, the pipeline is stalled for the affected stages.
3. **Data Conflicts**: External delays in the protocol execution are treated as data conflicts.

These conflicts arise in the standard pipeline as well as in the reconfigurable I - P - O pipeline. However, the reconfigurable pipeline can imply additional control conflicts, for example, a reserved reconfiguration port that interrupts the micro reconfiguration of the I - P - O stages. All these conflicts have to be avoided to assure a deadline conform protocol adaptation.

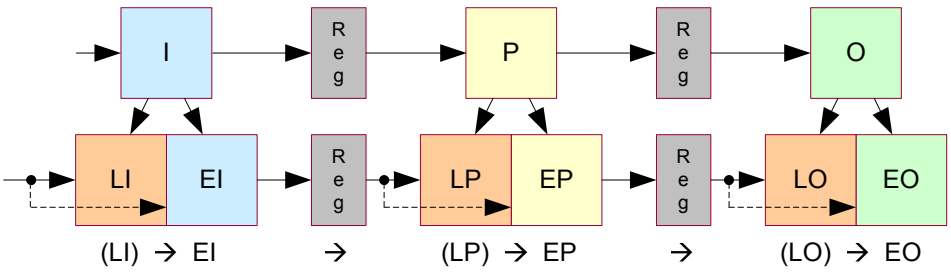


Figure 6.17: The reconfigured IFB execution pipeline

Execution Platform of the Runtime Reconfigurable I - P - O Pipeline

To execute the runtime reconfigurable I - P - O pipeline we require a *runtime environment* which supports *runtime reconfiguration* (see Section 4.4). As presented in Figure 6.18, we use a *slotted FPGA architecture*, which provides one fixed slot (here depicted on the right side) and a number of reconfigurable slots (here: `Slot 1` ... `Slot 6`). The reconfigurable slots turn into the I - P - O pipeline stages by means of reconfiguration. Due to the spatial execution capabilities of the FPGA, we can process n stages (PH-Modes and SH-Modes) truly parallel, where n is the number of available slots.

As mentioned in Section 4.4.1, we need the fixed slot to implement the *IFB skeleton* which comprises the non-reconfigurable parts of the IFB. It offers, for example, the docking ports to exchange PH-Modes and SH-Modes, serves as intermediate data storage between the different stages, and establishes the connection to the adapted tasks.

In contrast to the pipeline architecture of general purpose processors, which is composed of static hardware, the stages of our pipeline are built on demand and thus are not static, neither in place nor in functionality. This is what we define as a *multi-functional pipeline stage*. We consider each slot of our runtime environment as one possibility to place a stage of the I - P - O pipeline. Whenever a currently not present stage is requested during runtime, we download the required stage on demand. After a stage finishes processing, it remains idle until it is replaced by another stage or it is reactivated for the succeeding communication cycle. As usual for pipelining, we consider all processing steps of equal duration, resulting from the most time-consuming stage. Therefore, the shortest period of time required for the stage execution or the reconfiguration appoints the overall stage execution time.

We derive the scenario specific pipeline stages from the I - P - O graph (see Figure 4.16). If all stages can be loaded completely onto the FPGA, dynamic reconfiguration gets obsolete. Therefore, we address only those scenarios, where the number of stages is greater or equal than the number of available slots ($\#stages \geqslant \#slots$). Such a scenario occurs when an IFB, resulting from a large task graph, does either not fit into the FPGA due to limited resources, or it originates in a stage utilization that is too low according to our demands.

The area optimization approach is a *trade-off* between the number of *engaged slots* and the resulting *computation time* of the runtime reconfigurable I - P - O pipeline. The maximum number of slots ($\#\,slots = \#\,stages$) guarantees the shortest computation times, while the minimum number ($\#\,slots = 1$) leads to a sequential execution of all stages. In this case, the computation time calculates from the sum of the particular stage execution times.

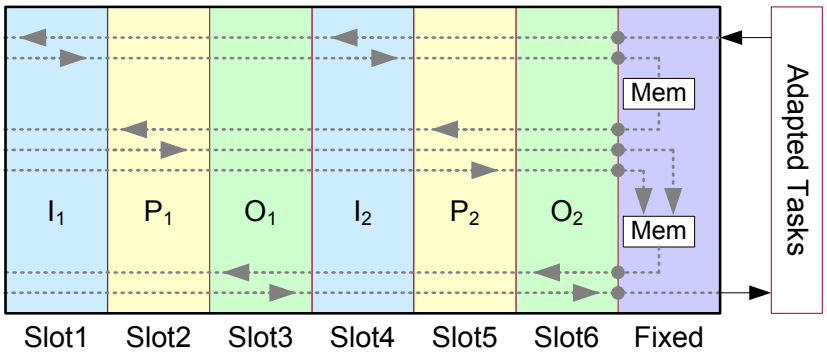


Figure 6.18: RTR pipeline architecture providing multi-functional pipeline stages

Caching of Runtime Reconfigurable Pipeline Stages

If we treat the slots of our runtime architecture as the *cache lines* of a *fully associative cache* and the stages as the *cached elements*, we can apply *caching theory* to minimize the number of stage reconfigurations. This is possible, because we can skip the loading of a stage that is still present on the FPGA. In this way, we optimize the displacement of the I - P - O stages. The more slots are idle, the better is the attainable caching effect. Thereby, it does not make sense to apply caching to the one-slot solution, which is never idle, or to the two-slot solution, where one slot is executing while the other one is reloading the succeeding stage. For the rest of this section, we assume the I - P - O stages to be homogenous, i. e., the stages consume the same area and require the same computation - and reconfiguration time.

To create an *efficient schedule* for the *pipeline stages*, we evaluate the scenario based I - P - O graph (see Figure 4.16) in combination with the causality conditions (see Section 4.3.1). As presented in Section A.1, the I - P - O graph defines the repeated execution of the static communication cycle, which consists of the I - P - O nodes. Therefore, we know the precise occurrence of each node in advance. For this reason, we can apply the *optimum caching strategy* (replace the element that is required last), which allows us to compute an optimal schedule for the pipeline stages.

The traversing of the I - P - O graph offers a specific degree of freedom, depending on the available I-P-O nodes. However, with the help of the *I - P - O progress line* (see Figure 4.16), we know at all times which nodes inside the communication cycle might be executed next. Independent from the particular schedule, the *causality* of the I - P - O scheme must not be violated. To initialize the cache we use a first-fit algorithm that downloads the first stages.

The developed *caching strategy* is defined as follows: If the stage, which is related to the next executed node of the I - P - O graph, is not present on the FPGA, it has to be loaded into a free slot (cache initialization) or, if no free slot is available, it replaces an idle stage. To ensure that always the last required stage is replaced (optimum caching strategy), we define the following *replacement policies*:

1) If no stage is idle, the replacement is stalled.
2) Otherwise, we replace the idle stages by the following sequence:
   - 2.1) If there exists exactly one idle stage, we replace this stage.
   - 2.2) In case of several idle stages, we replace those stages which depend on the downloaded one. Thereby, we select the node with the *greatest distance* to the *I - P - O progress line*. The distance is defined as number of intermediate edges.
   - 2.3) If no dependent idle stage exists, we choose that one (from all idle stages) with the *greatest distance* to the *I - P - O progress line*.

Example

We tested our replacement strategy for several random scenarios. In Figure 6.19, we display three iterations of the communication cycle, related to the I - P - O graph depicted in Figure 4.16. As we can see, the example reveals a considerable potential for optimization.

The diagram shows the consecutive *allocation* of the *six slots*. After the initialization phase, i. e. the downloading of the first six stages onto the FPGA, we start the execution. In the first iteration, we need six steps to process all communication requests. Slot 1 and 2 are reconfigured. In the second iteration, we process I_2 first, as the stage is still present

on the FPGA. In the meantime we reload I_1, which allows us to process seamlessly. The example continues like this for the succeeding communication cycles. It demonstrates that we can execute two input -, three processing - and three outputs stages inside six slots with a constant performance of three pipeline clock cycles per communication cycle.

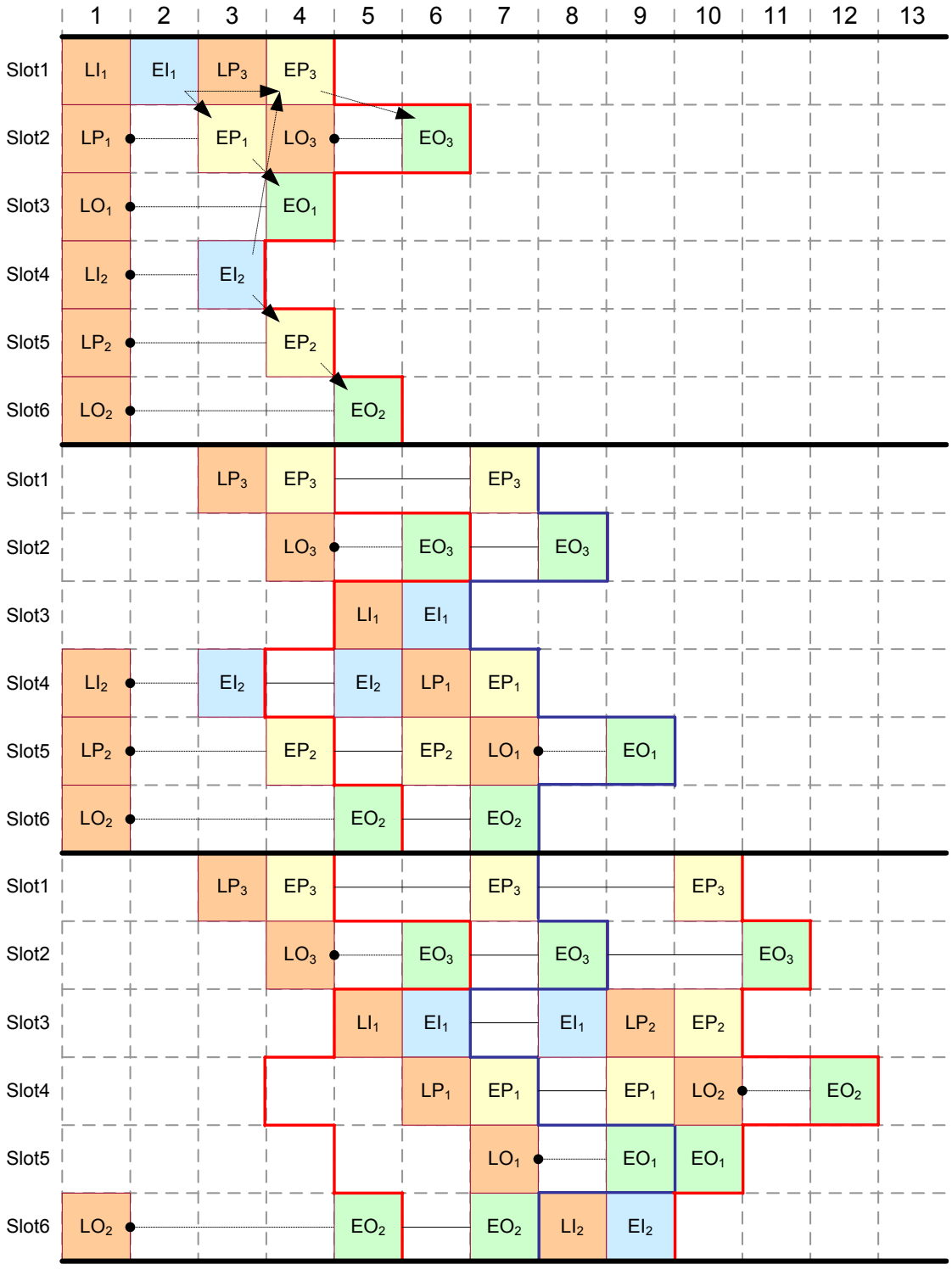


Figure 6.19: Reconfigured execution of an exemplary I - P - O graph

Timing Estimations

In the previous example we have obtained the timing information by a simulation of the I-P-O pipeline. It is even more valuable to dispose of an analytic method to determine the timing. Unfortunately, we cannot *compute* the *best case computation time* for the runtime reconfigurable pipeline with the help of an equation. However, we can estimate the *worst case computation time* T_{WCCT}.

The computation time is given by $T_{comp} = T_{RTR} + T_{Ex}$, whereas the execution time T_{Ex} is the processing time required to compute all nodes of an I-P-O graph consecutively. Therefore, $T_{Ex} = t_{Ex} \cdot \#nodes$, which is constant for a given I-P-O graph. T_{RTR} is the time needed for the reconfiguration. It depends on the ratio $\#nodes/\#slots$. Each slot that does not initially fit into the device has to be reconfigured, which takes the reconfiguration time $T_{RTR} = \max\{0, \#nodes - \#slots\} \cdot t_{RTR}$ for one iteration. In the worst case we have to replace a stage, which is a succeeding node inside the communication cycle. In the next step, this stage has to be reloaded to the device. If this procedure continues for all pipeline stages, we obtain a multiplication of the reconfiguration time by the number of succeeding pipeline stages. In the case of the I-P-O pipeline, the factor is three. Therefore, the *worst case computation time* T_{WCCT} is defined as follows:

$$T_{WCCT} = \#nodes \cdot t_{Ex} \; + \; 3 \cdot \max\{0, \#nodes - \#slots\} \cdot t_{RTR}$$

T_{WCCT} defines the upper bound for T_{comp} of one communication cycle without pipelining. Using the I-P-O pipeline in combination with the optimum caching strategy enormously reduces the effective computation. This is in our example ($t_{Ex} = 1/f_{IPO}$)

$$T_{WCCT} \; = \; 8 \cdot \frac{1}{f_{IPO}} + 3 \cdot \max\{0, 8-6\} \cdot t_{RTR} \; = \; \frac{8}{f_{IPO}} + 6 \cdot t_{RTR} \; = \; \frac{14}{f_{IPO}}$$

whereas $t_{RTR} = 1/f_{IPO}$, which is true when the pipeline clock rate is dominated by the reconfiguration time. Remember, the simulation required three pipeline clock cycles after the initialization to complete one communication cycle. Therefore, the optimized value ($\frac{3}{f_{IPO}}$) is 4,67 times better than the estimated worst case computation time.

Pipeline Utilization

Next to the required pipeline clock cycles, the *utilization* of the *slots* is an important metric of our runtime reconfigurable pipeline. With the help of our simulations we can determine the *pipeline utilization* U by the following equation (PCC = pipeline clock cycles):

$$U = \frac{\sum_{PCC} \; \#\, active\, Slots}{\#\, Slots \cdot \#\, PCC}$$

We simulated 10 communication cycles of different communication scenarios on our slotted architecture with 1 to 10 slots. Figure 6.20 depicts the required *pipeline clock cycles* (top) and the *pipeline utilization* (bottom) of our standard example, including 2 input and 3 output stages based on *multi-slot* and *single-slot reconfiguration*. Single-slot means that only one slot is reconfigurable whereas in multi-slot devices multiple slots can be reconfigured at the same time. The diagram is divided into three areas: The one- and the two-slot solution (area 1) deliver an equal result for both variants, since a maximum of one slot is reconfigured at the same time. In the two-slot solution this results from an alternating execution of the slots due to dependencies (causality conditions) between the I-P-O stages.

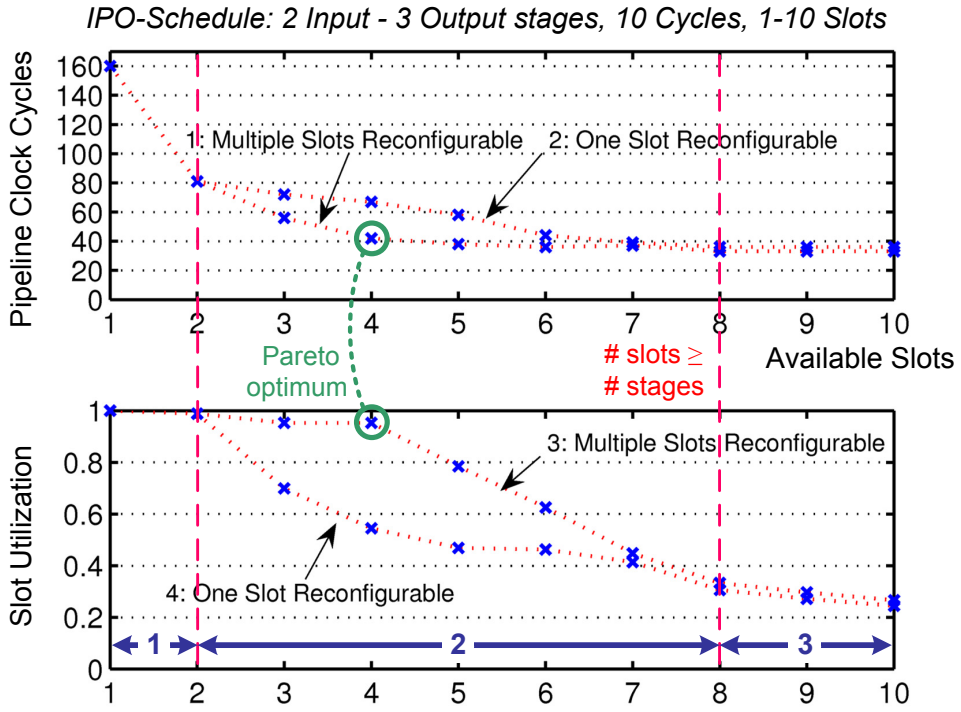


Figure 6.20: Required pipeline clock cycles and slot utilization

For $\#slots \geqslant \#stages$ (3), the two solutions converge towards each other. The remaining difference results from the unequal pipeline initialization phase. In the area in-between (2), the two reconfiguration variants differ from each other. This is due to the fact that especially the single-slot reconfigurable devices suffer from an increasing cache-miss rate, which results from the reduction of the available slots. Form and size of the "eye-opening" depend on the respective communication scenario. The diagram allows us to find *pareto optimal solutions* like the schedules based on four slots. Figure 6.21 depicts these schedules, which have been automatically generated by a scheduler implemented in the IFS-EDITOR. The slot utilization and the required pipeline clock cycles differ remarkably between *single-* and *multi-slot* solution. This is the result of the compact schedule for the multi-slot device, while the single-slot schedule is protracted by the continuous stage reconfigurations.

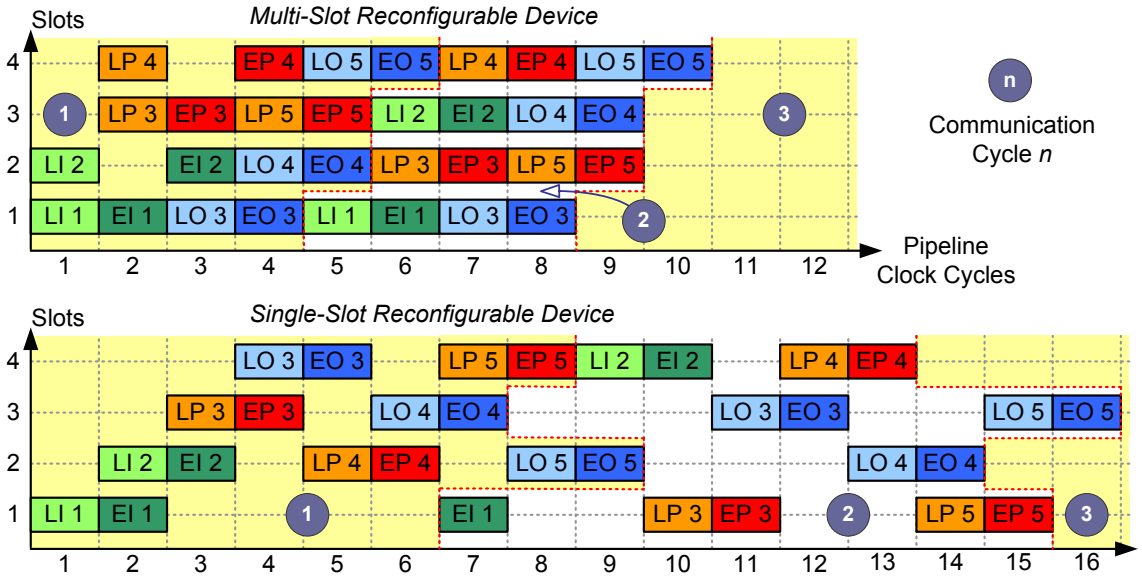


Figure 6.21: Multi-slot vs. single-slot reconfiguration given four slots

6.5 Summary

In this chapter we focussed on the hardware implementation of the IFB. Therefore, we introduced the IFB Hardware Template which specifies the implementation details of the IFB. This enabled us to perform a precise analysis of the timing and the required resources, which resulted in two orthogonal optimization approaches: one to optimize the latency, the other to minimize the chip area. A feasibility- and a schedulability analysis guarantee the deadline conform execution of real-time protocols.

The IFB Hardware Template

First of all, we presented the IFB Hardware Template, a schematic on the RT- and the algorithm level, applicable as template for IFB hardware implementations. The RT elements model the structure and the dedicated logic parts, whereas the algorithm parts model the behavior in the form of automata graphs. We discussed the PH including the PH-Modes and the PH-Switch as well as the SH with the SH-Modes, the SH-ReconModes, the SH-Switch, and the IFB internal memory. To provide a fully parallel memory access to the SH-Modes, we synthesize the internal memory in the form of dedicated registers. The switches handle the memory bus access and provide the bus macros for the reconfiguration of the PH-Modes and SH-Modes.

Furthermore, we discussed the detailed functionality of the CU including the memory bus arbiters, the memory management unit (scoreboard) and the Reconfiguration Unit. With the help of the bus arbiters, we implement the scheduling strategy for the memory bus access of the PH-Modes. The scoreboard ensures that the stage execution follows the causality conditions of the pipelined I-P-O processing.

In the case of the RTR-IFB, the Reconfiguration Unit copes with the replacement of the affected PH-Modes and SH-Modes. Therefore, it halts and deactivates the reconfigured modes in cooperation with the external reconfiguration control unit. When the reconfiguration is finished, the Reconfiguration Unit reactivates the exchanged modes again.

Timing Analysis

To perform a precise timing analysis of the IFB, we presented a cycle accurate evaluation of the I-P-O pipeline. To process a particular bit, the IFB requires 27 clock cycles that we divided into seven phases. These phases determine the precise clock cycles to establish and to release the memory bus, as well as to receive and to transmit a data bit.

The succeeding timing analysis comprises a feasibility- and a schedulability analysis. The feasibility test is a necessary condition to process data in real-time. Depending on the given protocols, the IFB requires six or nine clock cycles to process a particular bit (CPB). Furthermore, we delivered an equation to estimate the clock rate of the I-P-O pipeline.

When multiple interfaces are adapted by the IFB, an additional schedulability analysis has to be performed that determines if all protocols can be processed deadline conform. Therefore, we compute a worst case basic bock BB' for each adapted protocol, which comprises the attributes deadline, period, and execution time. To process BB', we developed a WCET analysis utilizing a graph based protocol reduction. We interpret the computed worst case basic blocks as a non-preemptive task set which can be scheduled with the help of a standard scheduling approach. One appropriate schedulability policy is the cyclic scheduling.

Afterwards, we presented two optimization approaches, based on the IFB Hardware Template. The first approach optimizes the IFB internal latency; the second one focusses on the minimization of the required chip area. As both approaches make different assumptions, they cannot be combined.

Latency Optimization

To reduce the latency, we improve the IFB internal data flow. The idea is to maximize the I - P - O parallelization by advancing the execution of the I - P - O stages. Thus, we do not have to wait until complete packages have been received, as we can start processing, when the information for the first outgoing package is available. For this reason, we analyze the IFD-Mapping in combination with the given protocols. Wherever an advanced execution is possible, we annotate the protocol of the sending interface with division points which partition the incoming frames into consecutive sub-frames.

The frame-based scheduling approach generates sub-frame IDs from the division points, which allow the advanced start of the related processing stage and the advanced transmission of completed outgoing packages. In contrast, the sub-frame-based scheduling creates small self-containing frames from the sub-frames that can be processed separately. On one hand, the frame-based schedule provides better optimization results; on the other hand, it requires an extension of the IFB architecture. The sub-frame-based schedule, which can even worsen the latency for sub-frames including less than three data bits due to the introduced overhead, goes with a simple preprocessing of the frame structure.

Area Optimization

Finally, we presented our area optimization approach. We defined the runtime reconfigurable I - P - O pipeline that is executed on a slotted FPGA architecture. The slots are utilized as multi-functional pipeline stages, which can hold any of the I - P - O pipeline stages.

To minimize the stage displacement, we apply caching theory. Due to the static structure of the I - P - O graph, which defines the repeated execution of the fixed communication cycle, we know the precise occurrence of each I - P - O stage in advance. Thereore, we can apply the optimum caching strategy. The fewer slots are provided, the higher is the cache miss rate, which leads to an increased reconfiguration of slots.

For the analytic evaluation of the runtime reconfigurable I - P - O pipeline we defined equations to estimate the worst case computation time and the pipeline utilization. We presented simulation results for our default I - P - O example which demonstrate the efficiency of the combination of pipelining and caching in comparison to the worst case computation time. Furthermore, we illustrated the required pipeline clock cycles and the slot utilization for a different number of slots with respect to single-slot and multi-slot reconfigurable devices and explained the characteristics of the presented diagrams and stage schedules.

CHAPTER 7

Results

To conclude this work, we give a brief insight into the IFS-Editor. Therefore, we offer characteristic values and show some interesting screen-shots of our EDA tool. Afterwards, we present the results of a case-study in which the IFB adapts RFID to I^2C. Finally, we compare some relevant characteristics of our approach with related work.

7.1 The IFS Design Environment: IFS-Editor

The IFS-Editor has been implemented in Java. It took us four years to complete this EDA tool, comprising about 120.000 lines of code (LOC). Basis for the implementation is the Model-View-Controller concept which proposes a strict separation of the data model (package *IFDS*, which stands for *interface data structure*, and package *XMLElementAPI*), the view (package *Editor*), and the control (package *Synthesis* and *CodeGen*).

As presented in Figure 7.1, the tool supports the component based modeling of the System Architecture in order to describe the communication infrastructure of distributed embedded systems. Furthermore, it provides the connection to the CASE tool Fujaba, including the model transformation, required to transfer the UML models into the IFS-Data-Structure.

To perform the two IFB synthesis steps the, IFS-Editor offers the *Synthesis Wizard* and the *CodeGen Wizard*, which handle the user interaction and encapsulate the developed synthesis algorithms. The Synthesis Wizard creates the target language neutral IFB model, whereas the CodeGen Wizard generates the final IFB target code – in our case VHDL.

We have implemented a *protocol visualization* that illustrates the modeled protocols in the form of waveform diagrams. To depict PSMs, the tool can also visualize transition conditions. This allows us to examine the generated PSMs of the PH-Modes and the SH-Modes.

Figure 7.2 visualizes the IFD-Mapping Editor. The designer enters the IFD-Mapping in the text field at the top. The protocol-visualization has been extended to depict also incoming and outgoing packages to facilitate the mapping process. The result of the syntax- and the semantic check are printed in the tab next to the protocol visualization (*Console*).

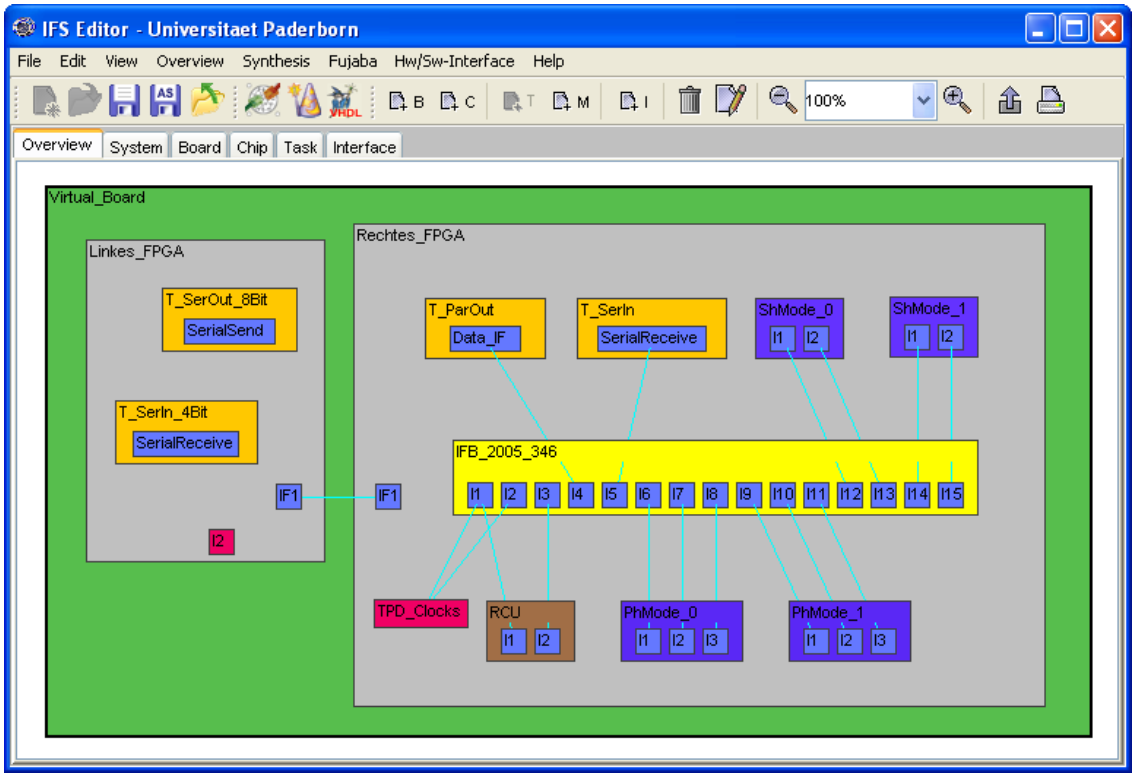


Figure 7.1: IFS-Editor – Component based System Architecture view

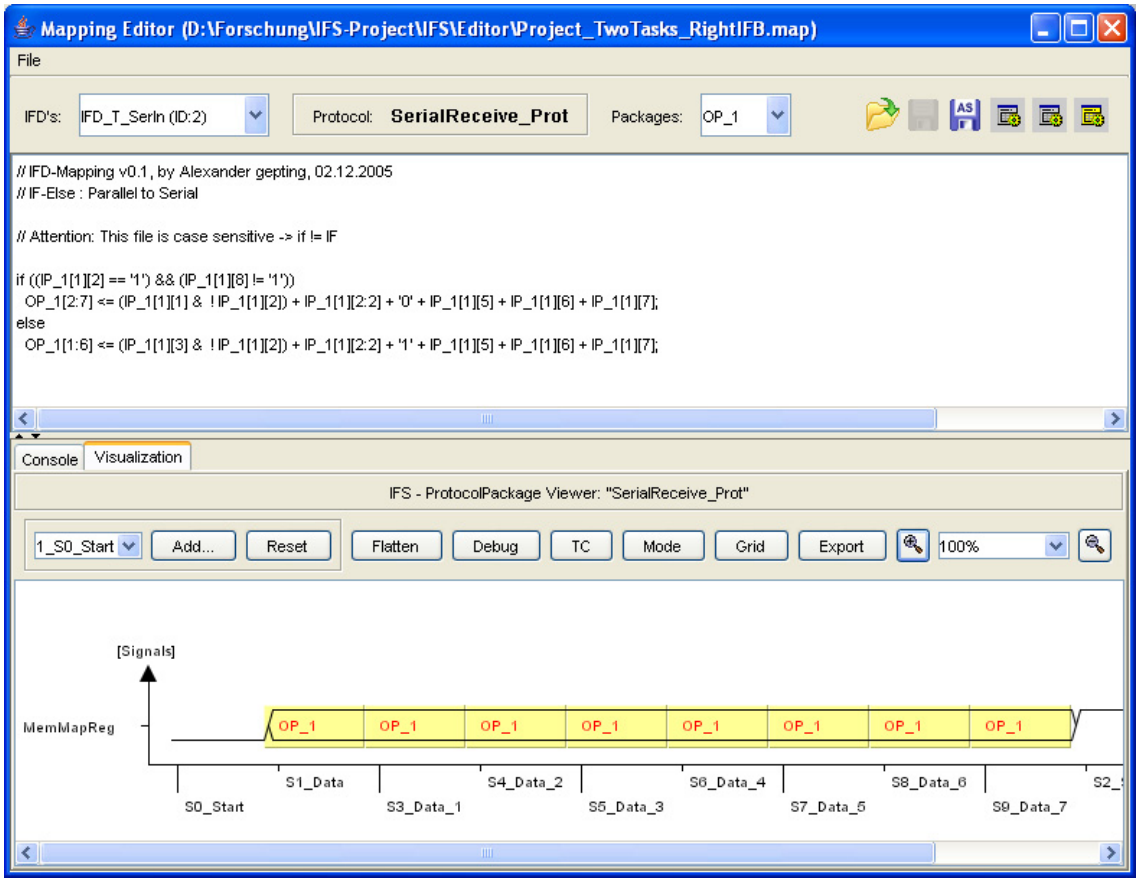


Figure 7.2: IFS-Editor – The IFD-Mapping Editor

7.2 Case-Study : Adaptation of RFID to I^2C

Our case-study is a toy-train scenario including several incompatible interfaces. The recognition and identification of the trains is realized by means of the RFID technology. We use an IFB as central interface adapter module to convert the RFID protocol (RS232) into the I^2C bus system of the train controller. The case-study has been presented in [28, 73].

A micro controller evaluates the RFID transponder signals and sends 5 bytes in the form of five RS232 packages at a baud rate of 9600 baud/s without parity bits via its TTL interface. The Inter-Integrated-Circuit-Bus (IIC = I^2C) is a defacto standard for integrated circuits. It is a bidirectional master-slave data bus consisting of two signals (SCL: serial clock line and SDA: serial data line).

To evaluate the quality of an *automatically generated* IFB, we compare the generated VHDL code with a *hand-coded* version. Therefore, with the help of the IFS-Editor, we modeled the interfaces of the RFID controller and the I^2C bus, including the interface topology and the related protocol descriptions. To ensure the electrical compatibility, we applied specific transducers for the voltage adaptation.

Quantitative Evaluation

Afterwards, we present some interesting characteristics of the generated VHDL code and the low-level synthesis. The design consists of two PH-Modes (one for the interaction with the RFID controller and one for the I^2C bus) and two SH-Modes for the conversion of the RS 232 into the I^2C protocol. The generated VHDL code includes 16 VHDL files comprising a total number of 7626 LOC (lines of code). The low level synthesis for the *Spartan 2 200e* FPGA using Xilinx ISE delivered the results presented in Figure 7.3. As we can see, the placed and routed IFB requires only a small number of the available resources provided by the *low-end device Spartan 2 200e*.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	472	4,704	10%	
Numbers used as Flip Flops	467			
Numbers used as Latches	5			
Number of 4 input LUTs	746	4,704	15%	
Logic Distribution				
Number of occupied Slices	425	2,352	18%	
Number of Slices contaning only related logic	425	425	100%	
Number of Slices contaning unrelated logic	0	425	0%	
Total Number 4 input LUTs	775	4,704	16%	
Number used as logic	746			
Number used as route-thru	29			
Number of bonded IOBs	4	142	2%	
IOB Latches	1			
Number of GCLKs	1	4	25%	
Number of GCLKIOBs	1	4	25%	
Total equivalent gate count for design	8,938			
Additional JTAG gate count for IOBs	240			

Figure 7.3: Low level synthesis results generated with Xilinx ISE

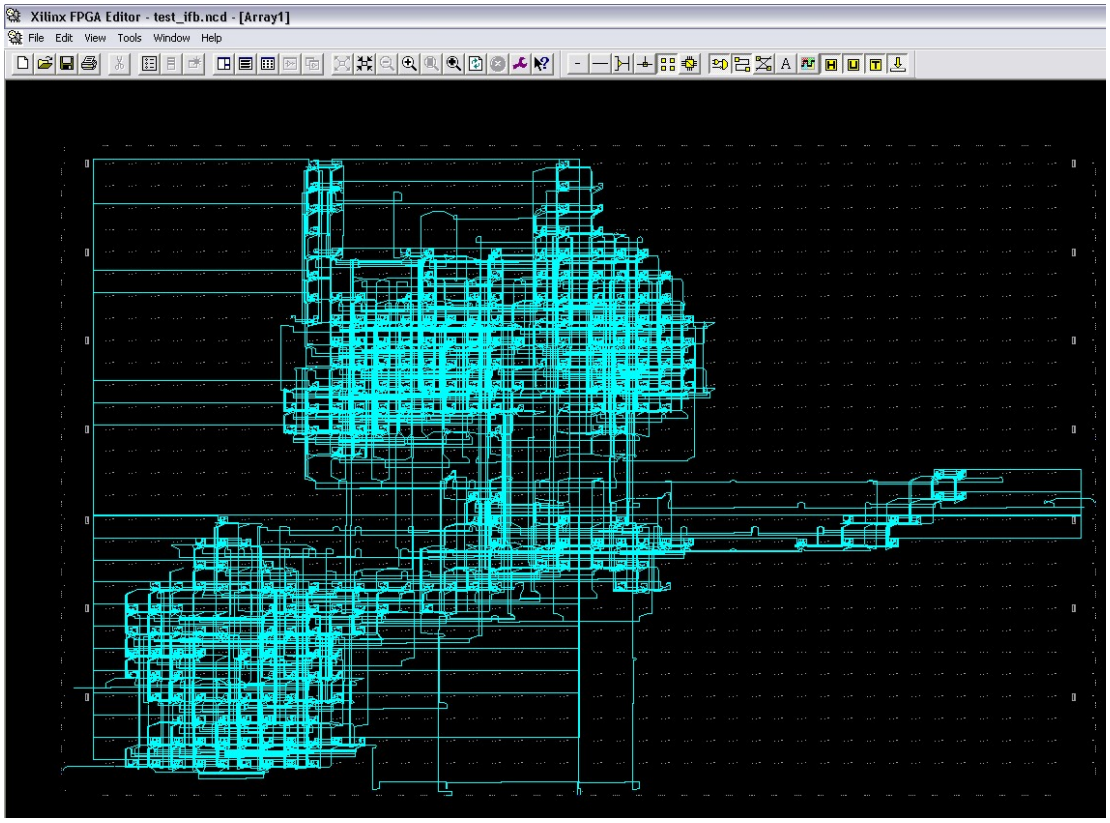


Figure 7.4: The RFID – I^2C design, illustrated by the Xilinx FPGA Editor

Similar to the controller design in Section 4.4.4, we illustrate the RFID – I^2C design with the help of the Xilinx FPGA Editor (see Figure 7.4) and the Xilinx Floorplanner (see Figure 7.5). As we can see, only a few number of well placed resources have been allocated. Thereby, the placement has been mainly determined by the interconnections to the external I/Os.

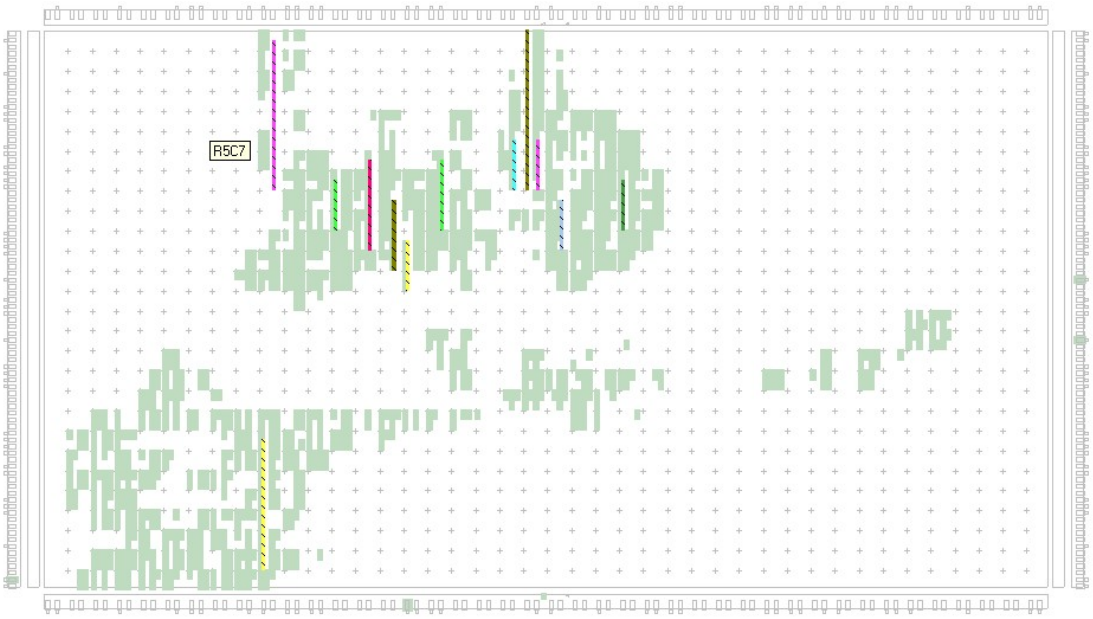


Figure 7.5: The RFID – I^2C design, illustrated by the Xilinx Floorplanner

7.3 Comparison With Other Approaches

Our Interface Synthesis approach is unique in its field of application. Therefore, we cannot directly compare the Interface Synthesis methodology and the Interface Block with existing approaches. However, we can draw a conclusion by referring to some relevant related domains: *protocol adaptation*, *reconfiguration*, *IP integration*, and *EDA tools*.

The most domain specific interface synthesis approaches are highly specialized and thus deliver highly optimized results. In contrast, the IFS methodology has been developed to cover a wide range of protocols. Therefore, it cannot always deliver such efficient results as the domain specific approaches. However, it is the only EDA tool that copes with the runtime reconfiguration of adapted tasks.

With respect to the protocol adaption aspect, the IFB is a rather complex protocol adapter module. Therefore, it does not really make sense to apply it for "trivial" protocol adaptations like a serial-to-parallel conversion, which can be handled by a simple shift register. However, the IFB supports the adaptation of multiple real-time interfaces including complex data processing operations in contrast to most approaches, which allow only the forwarding of information without an active modification. The IFB was not primarily designed to process streaming protocols, but to cope with several different kinds of communication, like network traffic, or the interaction of embedded IPs. However, due to the pipelined I - P - O processing, the IFB offers a short latency, which also allows the processing of streamed data.

The IFS approach applies runtime reconfiguration to support the transparent exchange of tasks at runtime. All other approaches have to inform the unaffected tasks about the ongoing reconfiguration to avoid communication gaps. In our Interface Synthesis approach we hide these reconfiguration based problems completely from the system designer. Furthermore, we utilize micro reconfiguration in combination with caching to minimize the reconfiguration effort, which is a well studied problem. However, our approach models communication cycles as I - P - O graph, which allows us to implement an optimum online caching strategy.

From the viewpoint of IP integration, the Interface Synthesis methodology is specialized on the IPQ format, compliant to VSIA's VCT standard. Highly specialized tools which provide support for the IP integration, like Xilinx ISE, create additional interface IPs to interconnect IPs. Therefore, these tools provide an automated user support for the personalization of the interface IP. However, these tools are usually restricted to specific architectures and IPs. Xilinx ISE, for example, can only adapt Xilinx specific IPs to the proprietary OPB bus. This is also true for other FPGA vendors, like Altera with relation to their SoC bus called "Avalon". Our approach handles all IPs as black boxes and goes without a protocol semantics to maximize the diversity and therefore, the field of application.

Our EDA tool provides the complete functionality from the abstract modeling in UML 2.0 to the code generation of the IFB. Only few academic approaches prove the related methodology by an integrated design flow of this complexity. To improve the usability of the IFS-Editor, we concentrate on a high reusability of the created models and a maximum degree of automatization. The implementation (Java code) is a well organized object oriented design, partially documented and can be extended after a short period of vocational adjustment.

CHAPTER 8

Conclusion and Outlook

8.1 Conclusion

In this thesis, we focused on the communication-based design of runtime reconfigurable embedded systems. As presented in Figure 8.1, we developed an integrated design flow, called Interface Synthesis Design Flow, that consolidates interface synthesis techniques with reconfigurable computing concepts. We implemented the developed concepts in the form of an EDA tool, the IFS-EDITOR. Thereby, our main objective was the automated generation of an interface adapter module, called Interface Block, that interconnects tasks and media comprising incompatible interfaces.

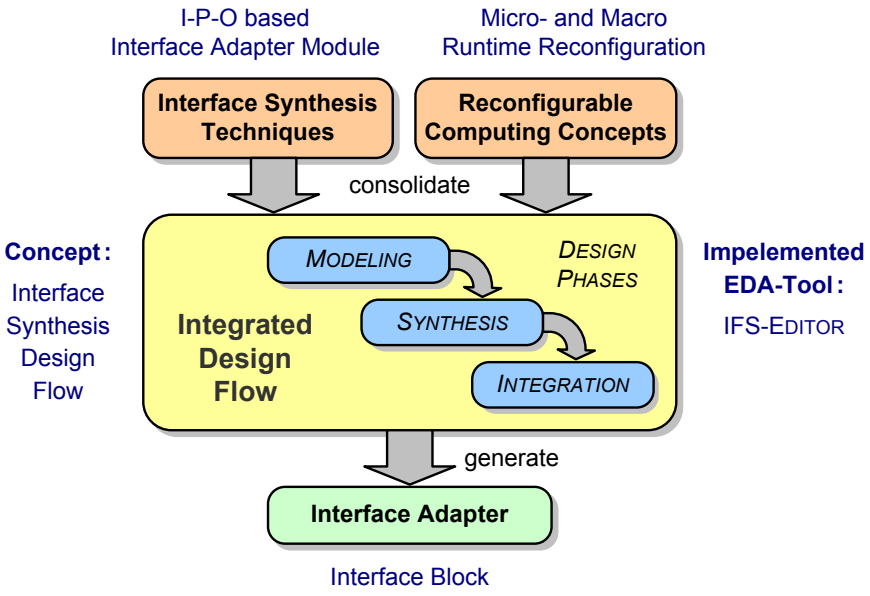


Figure 8.1: The resulting integrated design flow

Interface Synthesis Design Flow

The Interface Synthesis Design Flow is an integrated design flow, comprising three consecutive design phases: modeling, synthesis, and integration. The modeling phase provides an adequate way for the model-based design of real-time communication in embedded systems. We apply the created models as input for the synthesis phase, which is specialized on generating our reconfigurable interface adapter module, the IFB. The created IFB can be integrated into an existing design within the optional integration phase.

Central part of the modeling phase is our modeling concept, involving XML, Java, and UML. Based on an XML scheme, we defined the IFS System Architecture, a system-level model for the specification of communication infrastructures in complex communication scenarios. The System Architecture is a hierarchical structure composed of architectural components (system, board, chip) and interacting communication components (task, medium). We offer a UML based modeling of the System Architecture to provide a standardized and intuitive graphical user interface. Therefore, we developed a UML 2.0 profile that we transferred to the CASE tool Fujaba. To close the tool chain, we implemented a model transformation of the dedicated UML 2.0 models into the data structure of the IFS-Editor.

The System Architecture defines the static input for the automated IFB synthesis in the form of interface descriptions (IFD) and target platform descriptions (TPD). Additionally, the designer has to provide an IFD-Mapping in the beginning of the synthesis phase, which specifies the scenario based data processing inside the IFB. In this way, we avoid defining a global data semantics for IFDs and earn a maximum degree of freedom in interconnecting heterogeneous components, which is especially relevant for the IP based design.

The modeling phase is followed by an automated IFB synthesis comprising two synthesis steps: The first one results in a target language independent IFB model. The second step comprises a code generation, which generates the final IFB implementation in a dedicated target language. To construct the IFB model, we preprocess the selected input (IFD, TPD) from the System Architecture. This implies the protocol synthesis that transforms the given behavior descriptions in the form of waveform state machines into protocol state machines. The PSMs implement the complementary behavior of the adapted component interfaces. We derive the available protocol packages from these automata as input for the scenario based IFD-Mapping. The designer can specify the protocol adaptation based on four basic data processing operations. Therefore, we developed the IFD-Mapping-Language including a syntax- and sematic analysis to evaluate the correctness of the entered IFD-Mapping. Afterwards, we optimize the PSMs based on the IFD-Mapping to construct the IFB model. The IFB model is a hierarchical structure conform to the System Architecture, assembled from EIFDs which have been created by the EIFD Factory. It is also referred to as the intermediate representation of the IFB and is taken as input for the code generation. We developed an adequate VHDL code generator (compiler backend) based on the frame processing technique. The code generator produces synthesizable VHDL code for an individual IFB (IFB-IP), a chip (ConfigWare), or the complete system (system simulation).

Standardized tools for the low-level hardware synthesis, for example Xilinx ISE, or compiler like the GNU C-compiler (gcc), are supposed to be used to create the final configuration bit-streams or the executable files within the concluding integration phase.

As first approach in the interface synthesis/reconfiguration domain, we present an integrated high level synthesis design flow from UML 2.0 down to synthesizable VHDL.

Interface Block

The IFB is a runtime reconfigurable protocol adapter module which affords the deterministic and transparent reconfiguration of tasks at runtime due to the synergy of interface synthesis concepts and reconfiguration techniques. To interconnect multiple tasks and media, which we treat as black boxes, the IFB is a self-contained and transparent protocol adapter. This is of particular importance for the adaptation of protected IPs, where the interface description is the only information available. We support signal based communication as well as memory mapped I/O. Depending on the code generation, we can implement the IFB in hardware or software. In general, the IFB Macro-Structure allows us the handling of synchronous and asynchronous communication. However, the presented IFB Hardware Template specifies a clock synchronous hardware realization of the IFB.

The IFB Macro-Structure defines the structure and the functionality of the IFB and its main components: the protocol handler (PH), the sequence handler (SH), and the control unit (CU). The PH consists of the PH-Modes, which act as stubs and perform the interaction with the component interfaces. They pack and unpack the adapted information and provide it to the SH. Dedicated SH-Modes are responsible for the protocol conversion according to the IFD-Mapping. With the help of the CU, we manage the pipelined protocol translation comprising the stages Input - Processing - Output and the IFB internal reconfiguration.

To guarantee a deterministic runtime reconfiguration of tasks and therewith a predictable behavior of the remaining system, the RTR-IFB comprehends a Reconfiguration Unit. Therefore, whenever a connected task is exchanged, the affected PH- and SH-Modes are reconfigured as well, and a predefined behavior, specialized on different scenarios, is executed during the reconfiguration process to avoid communication gaps. In combination with a reconfiguration control unit, the IFB provides a basis technology to compose embedded systems supporting the deterministic (macro -) reconfiguration of tasks at runtime.

Starting from the modular IFB Macro-Structure we developed the IFB Hardware Template, a schematic on the RT and the algorithm level, which defines a precise construction pattern for hardware implementations of the runtime reconfigurable IFB. Furthermore, the schematic allows us to perform a cycle-accurate evaluation of the IFB. Based on this information, we accomplished a precise feasibility and schedulability analysis, utilizing a worst case execution time analysis to evaluate the adapted protocols. In addition, we explained how to determine the clock-cycles-per-bit value and the I - P - O pipeline clock rate.

Furthermore, we presented two optimization approaches for the IFB. The first approach maximizes the I - P - O parallelization by advancing the stage execution in order to minimize the IFB latency. Thus, we can start the processing and the succeeding transmission of completed data packages although the related input stage did not finish yet. We demonstrated that we can effectively reduce the latency in this way. The second approach utilizes micro reconfiguration to minimize the required chip area for the RTR-IFB implementation. As runtime environment for the reconfigurable I - P - O pipeline, we specified a slotted FPGA architecture that provides several slots which act as multi functional pipeline stages. In combination with the optimum caching strategy, we were able to increase the slot utilization and reduce the number of I - P - O stage reconfigurations.

As we have seen, the IFB is a powerful and flexible protocol adapter, comprising a high potential for optimization. The realization of the IFB Macro-Structure by the IFB Hardware Template leads to a resource efficient and real-time capable hardware implementation.

EDA Tool: IFS-Editor

To evaluate our Interface Synthesis methodology, we developed the IFS-Editor. This EDA tool implements the Interface Synthesis Design Flow and provides therewith the complete functionality to generate static and runtime reconfigurable IFBs. The three design phases modeling, synthesis, and code generation have been implemented as functional entities that are closely linked. Interactive wizards guide the designer through the synthesis and the code generation phases. In this way, we provide a maximum degree of automatization for the generation of an IFB.

Designated fields of application for the Interface Synthesis methodology and the generated Interface Blocks are the reconfigurable computing domain, the rapid system prototyping of heterogenous and distributed embedded systems, and the IP-based System-on-Chip design.

A collection of my publications and the advised Studien - and Diplomarbeiten are given afterwards in two separate bibliographies. As one highlight, we composed two books from most relevant publications, published in the "Wissenschaftliche Schriftenreihe: Eingebettete, Selbstorganisierende Systeme" [7, 1].

8.2 Outlook

Within the scope of this thesis, we developed the Interface Synthesis Design Flow that has been implemented by our EDA tool, the IFS-Editor. Furthermore, we presented the hardware realization of the Interface Block, including efficient optimization approaches and analysis techniques. In the following, we introduce some interesting challenges concerning future work wiht the capability to improve the presented work.

One way to increase the usability and the acceptance of the IFS methodology by the industry would be to create a well tested data base of relevant IPs. It would also help to turn the IFS-Editor into a foolproof and highly optimized EDA tool, which demands the integration of additional check methods and optimization approaches. Furthermore, it would be helpful to improve the available functionality by extending additional service methods, for example the code generation for different languages.

A first step to improve the IFS-Editor would be to implement the presented theoretical concepts for the analysis and the optimization of the Interface Synthesis Design Flow and the IFB, which have not yet been integrated. Beyond this, it makes sense to develop further concepts to perfect our methodology and to fine tune the IFB architecture.

An important aspect that we are already working on is the *method based communication* [27]. There, the major challenge is to handle abstract data types and method signatures. The extended IFS methodology will be able to adapt methods with incompatible parameters (types and identifiers), which still differentiates us from approaches like CORBA. Therefore, we have to extend our definition of hardware/software interfaces, which is not restricted to memory mapped I/O anymore.

Another goal is to apply the IFB as a flexible interface adapter in dynamic distributed embedded systems. To perform a fully automated interface adaption in such a system online, we are researching domain specific *ontologies*, that utilize our mapping functions to express the mapping of incompatible function classes within a classification scheme.

APPENDIX A

Extensions to the Interface Synthesis

A.1 Communication Cycles

To append one communication cycle to another, we insert the necessary edges from the output nodes to the input nodes. Therefore, we create an outgoing edge for each output node to the input nodes of the next communication cycle that deliver the required input to process the output for the current node.

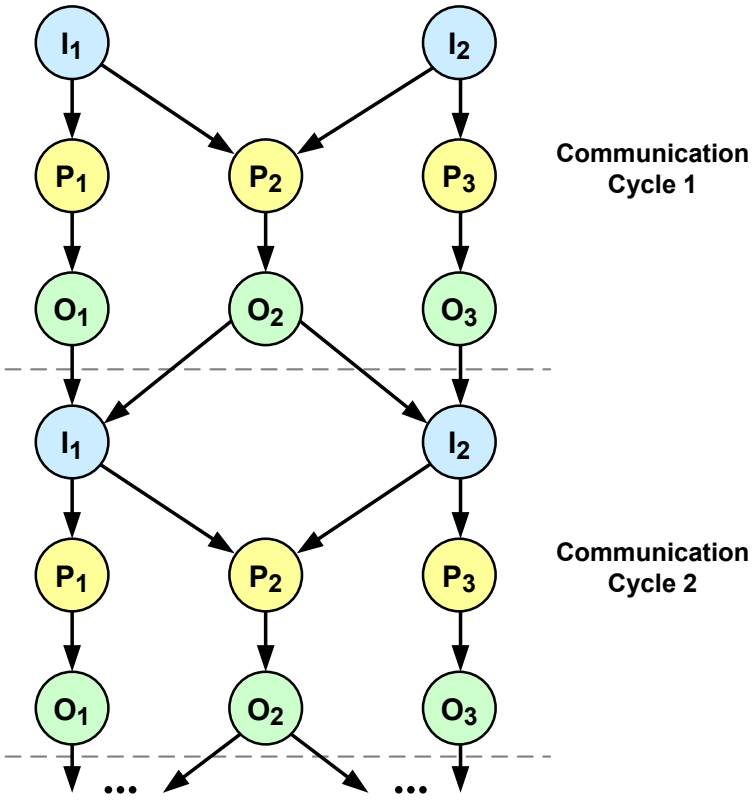


Figure A.1: Two succeeding communication cycles

A.2 Generating Basic Blocks

A basic block begins in the first state, after a branching state, or in a merging state and has the maximum size. Figure A.2 depicts an example for the identification of basic blocks. In step 1 we start with BB_1. That covers only the first state. We enlarge BB_1 in step 2 until we reach the first branch, where we create BB_2 and BB_3. In step 3 we enlarge BB_2 to its maximum size. Afterwards, BB_3 is processed. Thereby, we notice in step 5 that BB_2 contains the merging state S_4. Therefore, BB_2 is split into BB_2 and BB_4. The states S_6 and S_9 remain uncovered as dead states.

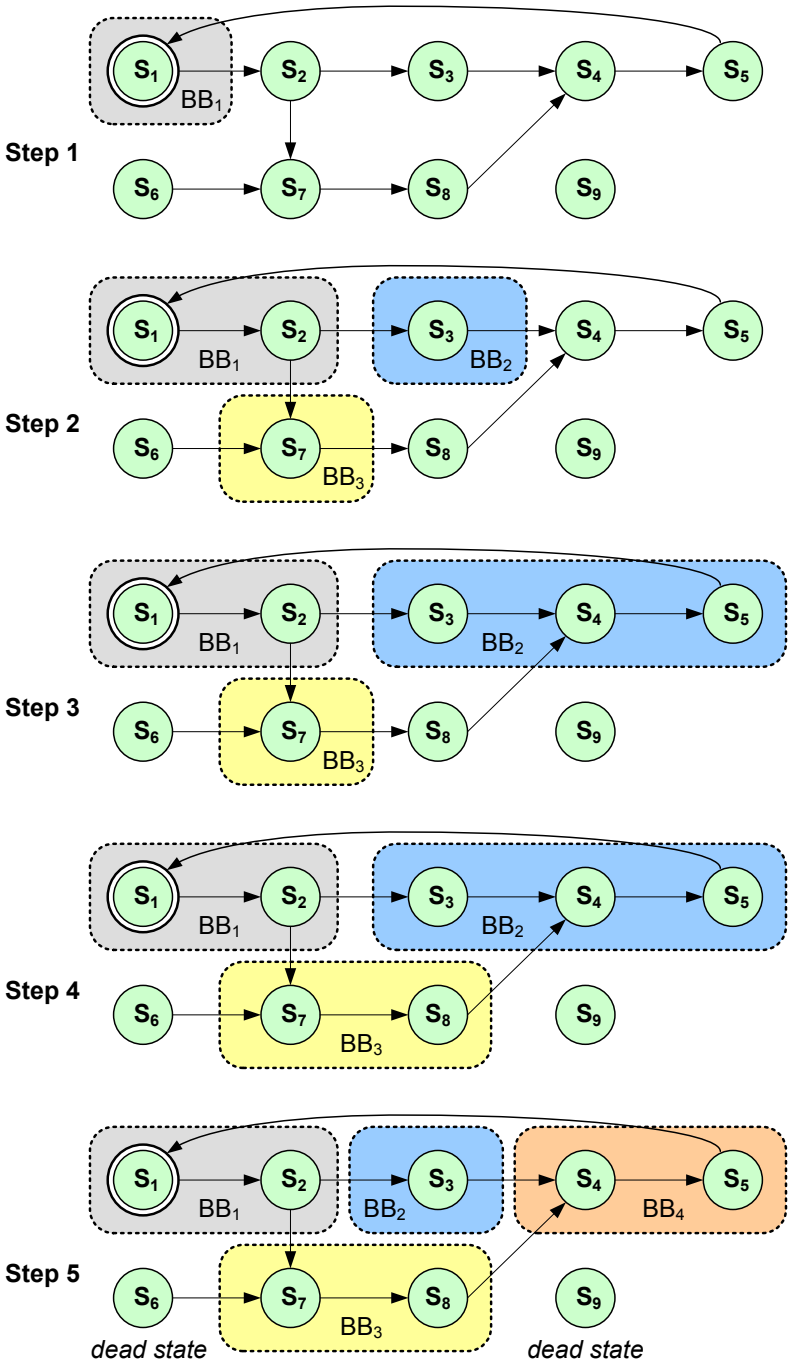


Figure A.2: Example for the generation of basic blocks

A.3 Grammar of the IFD-Mapping Language

This section delivers the complete grammar of the IFD-Mapping language. The grammar defines the syntax of valid IFD-Mapping instances and is specified in the *EBNF* style.

```
MappingFunction          = {ImportDeclaration} {MappingDeclaration};
ImportDeclaration        = 'import' Identifier;
Identifier               = Letter {Letter | Digit};
Letter                   = 'A' | 'B' | 'C' | ··· | 'Z' | 'a' | 'b' | 'c' | ··· | 'z';
Digit                    = '0' | '1' | '3' | ··· | '9';
MappingDeclaration       = VariableDeclarator ';' | Statement;
VariableDeclarator       = VariableType Identifier ['=' ConditionalOrExpression];
VariableType             = ('boolean' | 'int' | 'bit') ['[]'] ['[]'];
Assignment               = PackageOrVariable AssignmentOperator
                           ConditionalOrExpression;
AssignmentOperator       = '<=' | '=';
ConditionalOrExpression  = ConditionalAndExpression {'||' ConditionalAndExpression};
ConditionalAndExpression = InclusiveOrExpression {'&&' InclusiveOrExpression};
InclusiveOrExpression    = ExclusiveOrExpression {'|' ExclusiveOrExpression};
ExclusiveOrExpression    = AndExpression {'^' AndExpression};
AndExpression            = EqualityExpression {'&' EqualityExpression};
EqualityExpression       = AdditiveExpression {('==' '!=') AdditiveExpression};
AdditiveExpression       = UnaryExpression {'+' UnaryExpression};
UnaryExpression          = '~' UnaryExpression | '!' UnaryExpression |
                           | PrimaryExpression;
PrimaryExpression        = PackageOrVariable | Literal | ConstantArray |
                           | '{' ConstantArray {',' ConstantArray} '}' |
                           | '(' ConditionalOrExpression ')';
PackageOrVariable        = ProtocolPackage ['(' IntegerLiteral ')']
                           [ProtocolPackageSuffix] | Identifier;
ProtocolPackage          = ('IP_' | 'OP_') IntegerLiteral;
ProtocolPackageSuffix    = '[' IntInterval ']' [ '[' IntInterval ']' ];
Literal                  = IntegerLiteral | BooleanLiteral | BitLiteral;
ConstantArray            = '{' Literal {',' Literal} '}';
IntInterval              = IntegerLiteral [':' IntegerLiteral];
IntegerLiteral           = (Digit - '0') {Digit};
BooleanLiteral           = 'true' | 'false';
BitLiteral               = "'('1' | '0')'";
Statement                = Block | IfStatement | Assignment ';' ;
Block                    = '{' {MappingDeclaration}' }';
BlockStatement           = VariableDeclarator ';' | Statement;
IfStatement              = 'if' '('ConditionalOrExpression')' Statement
                           ['else' Statement];
```

A.4 VHDL Examples for the Created IFB Target Code

The developed code generator creates VHDL code from the synthesizable VHDL subset. In Example-Code A.1 (register file) and A.2 (SH-Mode) we present two VHDL descriptions that have been created automatically with the help of the IFS-Editor.

Register File

The register file (Example-Code A.1) defines the abstract data structure of the internal memory (Data Writer and Data Reader). In the presented example, the incoming memory (`datareadermem`) comprises one frame (`frame_1`), which provides one package (`IP_1_SP0`), including four parallel bits (`vec0, ..., vec6`). The illustrated data structure is derived from a parallel sender that provides eight bits in parallel. Four of these bits have been considered in the IFD-Mapping, therefore, the related package was reduced to four bits.

The outgoing memory (`datawritermem`) also provides one frame (`frame_2`). As the given IFB writes to a serial receiver, the frame consists of one package (`OP_1_SP0`), providing a four-bit serial data word (`vec0`). We can apply this register file as a VHDL library that allows an abstract notation of the internal data types.

Example-Code A.1: Generated VHDL Register File

```
01 library IEEE;
02 use IEEE.STD_LOGIC_1164.all;
03
04 package RegFile is
05
06   type OP_1_SP0 is record
07      vec0 : std_logic_Vector(3 downto 0);
08   end record;
09
10   type frame_2 is record
11      p2 : OP_1_SP0;
12   end record;
13
14   type datawritermem is record
15      f2 : frame_2;
16   end record;
17
18   type IP_1_SP0 is record
19      vec0 : std_logic;
20      vec1 : std_logic;
21      vec2 : std_logic;
22      vec3 : std_logic;
26   end record;
27
28   type frame_1 is record
29      p1 : IP_1_SP0;
30   end record;
31
32   type datareadermem is record
33      f1 : frame_1;
34   end record;
35
36 end package RegFile;
```

Entity and Architecture Description of a SH-Mode

Example-Code A.2 presents the generated VHDL code of a SH-Mode, which realizes a mapping function that assigns incoming to outgoing data. The VHDL code comprises exactly one *entity* and one *architecture* description. Therefore, it is unnecessary to add a *configuration* for the explicit mapping of entity and architecture, as it is already given implicitly.

The entity description comprises a clock- (`ShM1_IFB_Clk`) and a reset (`ShM1_IFB_Reset`) signal, which are required for the synchronous design. Furthermore, it provides the control signals, which are routed to the CU (`ModeRun`, `ModeComplete`, and `PInComplete`). We add a prefix (here: `ShM1_Sw_` that stands for "*SH-Mode 1 to Switch*") in front of each identifier, as signal names have to be unique. The data signals to the internal memory are written as complex data type (here: `op_1_sp1`, which stands for "*outgoing package 1 – sub-package 1*"[1]). The definition of complex data types has been presented in Example-Code A.1.

The architecture description starts with the *declaration* of the *local signals*, which originate from the ProtocolPins inside the IFDs. Based on the ProtocolMap we map the local signals to the related entity signals. As shown in the example, the ProtcolPin identifiers can be equal to the entity signal names[2]. For this reason, we extend each local signal identifier with the prefix "`_loc`".

Furthermore, we define an abstract data type (`StateType`), which specifies the required states of the synthesized state machine. In the given example, it comprises three states: `Idle` and `Done`, which are default states given by the IFB Hardware Template, and the assignment state `Assignment_OP_1_SP1_StateID_3`. In this state, we assign the incoming data word to the outgoing data word . We divide the generated state machines into three processes (`SH_Mode_1_TransitionAndOutput`, `SH_Mode_1_FF_Out`, and `SH_Mode_1_Synchronize`):

`SH_Mode_X_TransitionAndOutput:` Within this process, we implement the *state transition function* and the *output function* (Moore output). Thereby, we include only those outputs, which are defined for all states. Typical signals, which are handled by this process, are the IFB internal control signals. The outputs that represent assignments to registers are usually not specified for every state, otherwise they could be replaced by combinational logic. Therefore, we implement all register assignments in an extra process, called `SH_Mode_X_FF_Out`.

`SH_Mode_X_FF_Out:` In dependency of the process `SH_Mode_X_TransitionAndOutput`, we define "incomplete" assignments to signals and registers here. The low level synthesis in the integration phase synthesizes registers in the form of Data-Clock-Enable flip flops (DCE-FF), based on this process. Thereby the conditions inside the `if`-statements (here: `if (CS = Assignment_OP_1_SP1_StateID_3) then`) are transformed into the combinatorial enable logic of the DCE-FF.

`SH_Mode_X_Synchronize:` The synchronize process is responsible for the *stepping* of the *current state* (`CS`) to the *next state* (`NS`). The resulting design is a *synchronous circuit* that performs its state transition at the rising edge of the clock signal (`ShM1_IFB_Clk`).

[1] The sub-packages result from the optimization step after the IFD-Mapping has been entered.

[2] The IFS-Editor provides the functionality to automatically create the ProtocolPins and the ProtocolMap based on an interface description. Thereby, the algorithm copies the signal identifiers of the interface to the created ProtocolPins

Example-Code A.2: Generated SH-Mode in VHDL

```
001 Library IEEE,IFB;
002 use IEEE.STD_LOGIC_1164.all;
003 use IFB.RegFile.all;
004 --                      +--/-- Handler
005 --                      |
006 --             +--------+-------+
007 --             |  Handler_Mode   |
008 --             |   +--------+    |
010 --             |   |  FSM   |    |
011 --             |   +--------+    |
012 --             +-----------------+
013
014 Entity SH_Mode_1 is
015 Port(
016    ------------------------------------------------------------
017    -- IFSInterface: ShM1_Internal_Clock_And_Reset           --
018    ------------------------------------------------------------
019       ShM1_IFB_Clk    : in std_logic;
020       ShM1_IFB_Reset  : in std_logic;
021    ------------------------------------------------------------
022    -- IFSInterface: ShM1_SH_Mode_1                          --
023    ------------------------------------------------------------
024       ShM1_Sw_ModeRun      : in std_logic;
025       ShM1_Sw_ModeComplete : out std_logic;
026       ShM1_Sw_PInSelect    : out std_logic;
027       ShM1_p24             : out op_1_sp0;
028       ShM1_p31             : in ip_1_sp0
029    );
030 end SH_Mode_1;
031
032 Architecture SH_Mode_1_Behavior of SH_Mode_1 is
033 ------------------------------------------------------------
034 -- Definition of Local Signals                            --
035 ------------------------------------------------------------
036 signal ShM1_Sw_ModeRun_loc   : std_logic;
037 signal ShM1_Sw_ModeComplete_loc : std_logic;
038 signal ShM1_Sw_PInSelect_loc : std_logic;
039 signal ShM1_p24_loc          : op_1_sp0;
040 signal ShM1_p31_loc          : ip_1_sp0;
041
042 ------------------------------------------------------------
043 -- Declaration of State_Type Definition                   --
044 ------------------------------------------------------------
045 type StateType is (
046                     Idle,               -- Idle State of SH-Mode
047                     Done,               -- Final State of SH-Mode
048                     Assignment_OP_1_SP1_StateID_3 -- Assign OP_1_SP1
049                    );
050 signal CS, NS : StateType;
051
052 begin
053 ------------------------------------------------------------
054 -- Mapping: Entity <=> Local Signals (Prot-Map)           --
055 ------------------------------------------------------------
056    ShM1_Sw_ModeRun_loc  <= ShM1_Sw_ModeRun;
057    ShM1_Sw_ModeComplete <= ShM1_Sw_ModeComplete_loc;
```

```
058    ShM1_Sw_PInSelect     <= ShM1_Sw_PInSelect_loc;
059    ShM1_p24              <= ShM1_p24_loc;
060    ShM1_p31_loc          <= ShM1_p31;
061
062 -------------------------------------------------------------
063 -- FSM Processes                                            --
064 -------------------------------------------------------------
065 SH_Mode_1_TransitionAndOutput: process (CS, ShM1_Sw_ModeRun_loc)
066 begin
067    case CS is
068       when Idle =>
069          ShM1_Sw_ModeComplete_loc <= '0';
070          ShM1_Sw_PInSelect_loc <= '0';
071          if (ShM1_Sw_ModeRun_loc = '1') then
072             NS <= Assignment_OP_1_SP1_StateID_3;
073          else
074             NS <= Idle;
075          end if;
076
077       when Done =>
078          ShM1_Sw_ModeComplete_loc <= '1';
079          ShM1_Sw_PInSelect_loc <= '0';
080          if (ShM1_Sw_ModeRun_loc = '0') then
081             NS <= Idle;
082          else
083             NS <= Done;
084          end if;
085
086       when Assignment_OP_1_SP1_StateID_3 =>
087          ShM1_Sw_ModeComplete_loc <= '0';
088          ShM1_Sw_PInSelect_loc <= '0';
089          NS <= Done;
090    end case;
091 end process;
092
093 SH_Mode_1_FF_Out: process (CS, ShM1_IFB_Clk, ShM1_IFB_Reset)
094 begin
095    if (ShM1_IFB_Reset = '1') then
096       ShM1_p24_loc.vec0 <= "0000";
097    elsif (ShM1_IFB_Clk'event and ShM1_IFB_Clk = '1') then
098       if (CS = Assignment_OP_1_SP1_StateID_3) then
099          ShM1_p24_loc.vec0 <=   ShM1_p21_loc.vec0 & ShM1_p21_loc.vec1
100                               & ShM1_p21_loc.vec2 & ShM1_p21_loc.vec3;
101       end if;
102    end if;
103 end process;
104
105 SH_Mode_1_Synchronize: process (ShM1_IFB_Clk, ShM1_IFB_Reset)
106 begin
107    if (ShM1_IFB_Reset = '1') then
108       CS <= Idle;
109    elsif (ShM1_IFB_Clk'event and ShM1_IFB_Clk = '1') then
110       CS <= NS; -- Advance Current State
111    end if;
112 end process;
113
114 end SH_Mode_1_Behavior;
```

A.5 Template of the Reconfiguration Control Unit

Figure 6.5 depicts our template for the Reconfiguration Control Unit. The design provides the interface and the behavior (yellow states) to interact with the Reconfiguration Unit. To implement a complex behavior, like self reconfiguration, the design has to be extended by this functionality (blue states).

The necessary signals to interact with the Reconfiguration Unit inside the Interface Block are `ReconStart`, `ReconComplete`, `ExchangeModules`, `ModulesExchanged`, and `ReconTask`. To determine which tasks are reconfigured, the RCU provides the `ReconTask` bus (one signal per task). The signals, which are related to the reconfigured tasks, are set to `'1'`. After the `ReconTask` signal has been set, the IFB is notified to start the reconfiguration by setting the signal `ReconStart`. When the IFB halted all affected modes, it responses by setting the `ExchangeModules` signal. Now, the RCU can reconfigure all tasks in combination with the affected IFB components (PH-Modes and SH-Modes). After this process has been finished, the RCU rises the `ModulesExchanged` signal. Then, the IFB reactivates the reconfigured modes and completes the reconfiguration process by activating the `ReconComplete` signal.

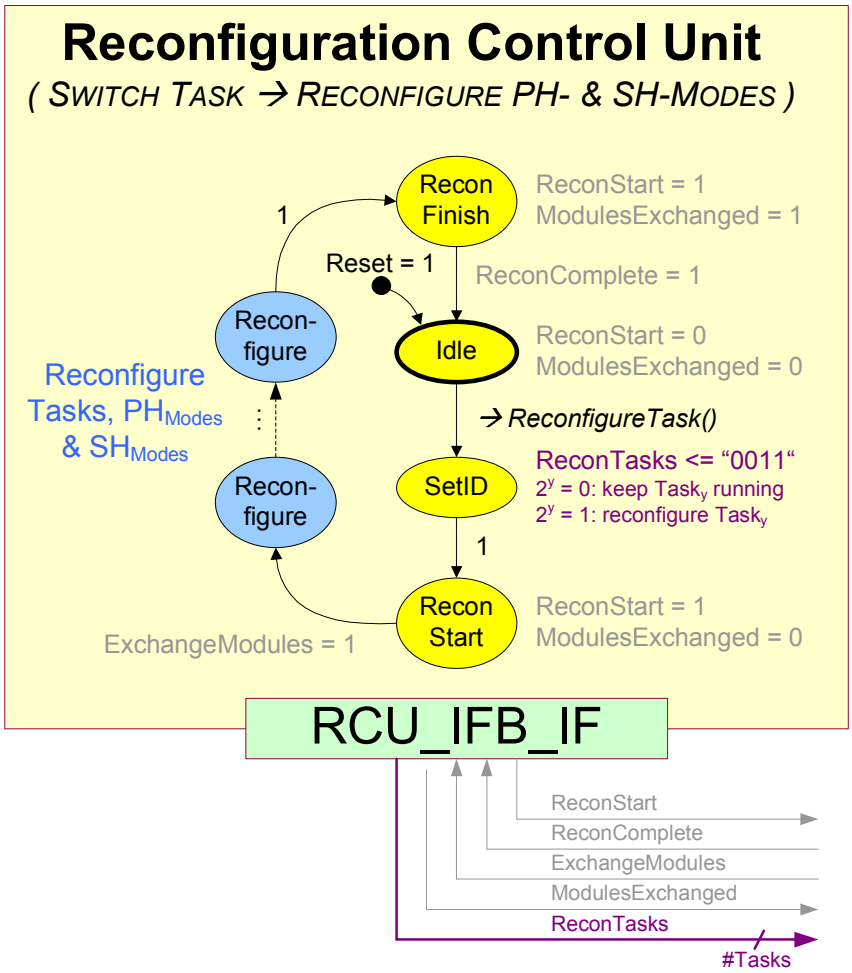


Figure A.3: Template of the Reconfiguration Control Unit

A.6 Validity Period of Control Signals inside Protocol Frames

Figure A.4 illustrates the *validity period* of the control signals `IRQ` and `FrameID`. As we can see, the FrameID is set one state in advance to the IRQ to guarantee a stable FrameID signal when IRQ is set to `'1'`. At the end of the frame, the IRQ is reset to `'0'`, one state before the FrameID is invalidated (`"0...0"`). We implemented this effect to detect if a frame has been successfully processed, or if a timing violation forced the termination of the ongoing frame transmission.

A timing violation is modeled in the form of a transition that provides a `TimerOrDeadline`, which leads from a data state inside to a state outside of the current frame. When a deadline expires, the related transition fires and therefore, we do not pass the $\overline{IRQ}$ state (`IRQ = '0'` and `FrameID` $\neq$ `"0...0"`). In this way, the CU determines whether a frame has been successfully sent or not (see Section 6.1.3).

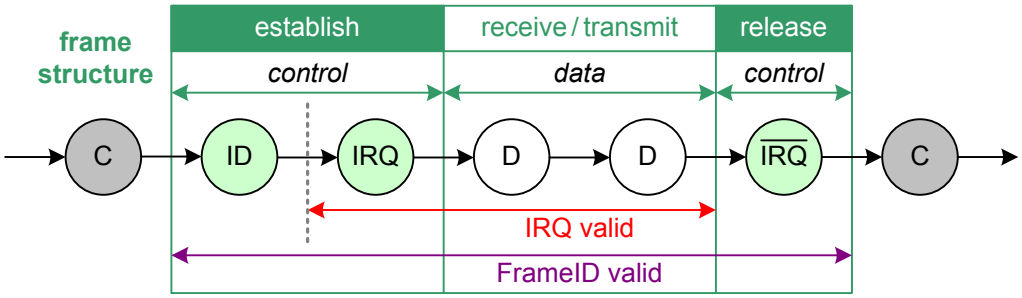


Figure A.4: Validity period of the control signals inside a protocol frame

Own Previous Work

[1] Christian Behler, Gilles Bertrand Gnokam Defo, Michel Camel Kouamo Sime, Tobias Loke, Wolfram Hardt (Hrsg.), and Stefan Ihmor (Hrsg.). *Schnittstellensynthese : Modellierung – Optimierung – Codegenerierung*, volume 2 of *Wissenschaftliche Schriftenreihe: Eingebettete, Selbstorganisierende Systeme*. TUDpress, Bergstr.70, 01069 Dresden, Germany, May 2006.

[2] Wolfram Hardt, Markus Visarius, and Stefan Ihmor. Rapid Prototyping of Real-Time Interfaces. In *Field Programmable Logic (FPL) - Poster Session*, Belfast, Northern Ireland, UK, October 2001.

[3] Stefan Ihmor. Entwurf von Echtzeitschnittstellen am Beispiel interagierender Roboter. Master's thesis, Universität Paderborn, Warburger Str. 100, 33098 Paderborn, November 2001.

[4] Stefan Ihmor, Nilson Bastos Jr., Rafael Cardoso Klein, Markus Visarius, and Wolfram Hardt. Rapid Prototyping of Realtime Communication – A Case Study: Interacting Robots. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, pages 186–192, San Diego, CA, June 2003. IEEE Computer Society.

[5] Stefan Ihmor and Florian Dittmann. Optimizing Interface Implementation Costs Using Runtime Reconfigurable Systems. In Toomas Plaks, editor, *Engineering of Reconfigurable Systems and Algorithms (ERSA'05)*, pages 85–91, Monte Carlo Resort, Las Vegas, Nevada, USA, 2005.

[6] Stefan Ihmor, Marcel Flade, and Wolfram Hardt. Integrating Error Processing Into Automated Communication Synthesis. Technical Report Technical Report, Heinz Nixdorf Institute, University Paderborn, Fuerstenallee 11, 33102 Paderborn, Germany, January 2004.

[7] Stefan Ihmor, Marcel Flade, and Wolfram Hardt (Hrsg.). *Rekonfigurierbare Schnittstellen*, volume 1 of *Wissenschaftliche Schriftenreihe: Eingebettete, Selbstorganisierende Systeme*. TUDpress, Bergstr.70, 01069 Dresden, Germany, June 2005.

[8] Stefan Ihmor and Wolfram Hardt. Runtime Reconfigurable Interfaces - The RTR-IFB Approach. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW'04)*, Santa Fe, USA, April 2004. IEEE Computer Society.

[9] Stefan Ihmor and Wolfram Hardt. Runtime Reconfigurable Interfaces - The RTR-IFB Approach. *International Journal of Embedded Systems (IJES), Inderscience Publisher*, Issue 5/6(Article 1), 2005.

[10] Stefan Ihmor, Michel Camel Kouamo Sime, and Wolfram Hardt. A UML2.0 Profile for Model-Based Design of Real-Time Interfaces in Embedded Systems. Technical Report Technical Report, Heinz Nixdorf Institute, University Paderborn, Fuerstenallee 11, 33102 Paderborn, Germany, March 2005.

[11] Stefan Ihmor, Tobias Loke, and Wolfram Hardt. Synthesis of Communication Structures and Protocols in Distributed Embedded Systems. In *RSP, 16th International Workshop on Rapid System Prototyping*, Montreal, Canada, 2005. IEEE Computer Society Press.

[12] Stefan Ihmor, Markus Visarius, and Wolfram Hardt. A Consistent Design Methodology for Configurable HW/SW-Interfaces in Embedded Systems. In *Proc. of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems (DIPES)*, pages 237–246, Montreal, Canada, August 2002. Kluwer Academic Publishers.

[13] Stefan Ihmor, Markus Visarius, and Wolfram Hardt. A Design Methodology for Application-specific Real-Time Interfaces. In *Proceedings of 2002 IEEE International Conference on Computer Design (ICCD)*, IEEE International Conference on Computer Design, Freiburg, Germany, September 2002.

[14] Stefan Ihmor, Markus Visarius, and Wolfram Hardt. Modeling of Configurable HW/SW-Interfaces. In Rolf Drechsler, editor, *Proc. of the 6. GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 51–60, Bremen, Germany, February 2003. Shaker Verlag.

[15] André Meisel, Markus Visarius, Wolfram Hardt, and Stefan Ihmor. Self-Reconfiguration of Communication Interfaces. In *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping*, pages 144–150, Geneva, Switzerland, June 2004. IEEE Computer Society.

[16] Martin Schaaf, Markus Visarius, Ralph Bergmann, Rainer Maximini, Marco Spinelli, Johannes Lessmann, Wolfram Hardt, Stefan Ihmor, Wolfgang Thronicke, Jasmin Franz, Carsten Tautz, and Ralph Traphöner. IPCHL - A Description Language for Semantic IP Characterization. In Eugenio Villar and Jean P. Mermet, editors, *Best of FDL'02, System Specification & Design Languages*, September 2002.

[17] Markus Visarius, Wolfram Hardt, Stefan Ihmor, and Johannes Lessmann. IPQ Format: Concepts and Implementation. In *Proc. of the International Workshop of the MEDEA+ Project A-511 ToolIP*, Madrid, Spain, January 2002.

[18] Markus Visarius, Johannes Lessmann, Carsten Amelunxen, Stefan Ihmor, and Wolfram Hardt. Initial IPQ Toolbox Implementation. TechReport 01, University of Paderborn, Informatik- und Prozesslabor, Warburger Strasse 100, 33098 Paderborn, Germany, January 2003.

[19] Markus Visarius, Johannes Lessmann, Stefan Ihmor, and Wolfram Hardt. Definition of the IP Qualifying Format. TechReport 01, University of Paderborn, Informatik- und Prozesslabor, Warburger Strasse 100, 33098 Paderborn, Germany, January 2002.

[20] Markus Visarius, Johannes Lessmann, Stefan Ihmor, and Wolfram Hardt. IP Qualifying Format. In *Poster Session of the German Workshop on Entwurfsplattformen komplexer angewandter Systeme und Schaltungen (EkompaSS)*, Bonn, Germany, April 2002.

[21] Markus Visarius, Johannes Lessmann, Stefan Ihmor, and Wolfram Hardt. IPQ Format based Retrieval. In *MEDEA+ Design Automation Conference - Demonstration and Poster Exhibition*, Stresa, Italy, October 2002.

[22] Markus Visarius, Johannes Lessmann, Stefan Ihmor, and Wolfram Hardt. Validation of the Requirements on the IP Qualifying Format. TechReport 01, University of Paderborn, Informatik- und Prozesslabor, Warburger Strasse 100, 33098 Paderborn, Germany, January 2002.

[23] Markus Visarius, Johannes Lessmann, Stefan Ihmor, Wolfram Hardt, and Wolfgang Thronicke. Specification of a Toolbox for the IP Qualifying Format. TechReport 01, University of Paderborn, Informatik- und Prozesslabor, Warburger Strasse 100, 33098 Paderborn, Germany, January 2002.

[24] Markus Visarius, Johannes Lessmann, Frank Kelso, Wolfram Hardt, Carsten Amelunxen, Andreas Scholand, and Stefan Ihmor. Definition of the IPQ Format with Respect to Specialized Description Features. TechReport 08, University of Paderborn, Informatik- und Prozesslabor, Warburger Strasse 100, 33098 Paderborn, Germany, August 2002.

Advised Bachelor Thesis and Diploma Thesis

[25] D. Averberg. Synthese von deadline-konformen Protokollkonvertern für heterogene verteilte Anwendungen. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, Mar. 2005.

[26] C. Behler. Datenflussoptimierung in rekonfigurierbarer Hardware durch Pipelining. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, May 2005.

[27] C. Behler. Methodenbasierte Schnittstellensynthese basierend auf lokalen Ontologien für verteilte eingebettete Systeme. Laufende Diplomarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, Dec. 2006.

[28] A. Daubner. Evaluierung des IFS-Editors am Beispiel der Protocoladaptierung von RFID und I^2C. Laufende Studienarbeit, Technische Universität Chemnitz, Straße der Nationen 62 09111 Chemnitz, Sept. 2006.

[29] O. Fick. Visualisierung von Parametern komplexer Schnittstellen für eingebettete Systeme auf Basis von XML. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, June 2003.

[30] O. Fick. Testmustergenerierung mit Hilfe von Wahrscheinlichkeitsverteilungen im Rahmen des modellbasierten Tests eingebetteter Systeme. Master's thesis, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, June 2005.

[31] S. Finke. Schnittstellensynthese für HW/SW-migrierbare Dienste eines Realzeitbetriebssystems. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, Aug. 2006.

[32] M. Flade. Modellbasierte Spezifikation von Schnittstellen im Hardware Entwurf. Studienarbeit, Technische Universität Chemnitz, Straße der Nationen 62 09111 Chemnitz, Nov. 2003.

[33] M. Flade. Automatische Adaption von Hardware-Acceleratoren für Verhaltenssimulation. Master's thesis, Technische Universität Chemnitz, Straße der Nationen 62 09111 Chemnitz, Sept. 2004.

[34] A. Gepting. Szenariobasierte Datentransformation semantikfreier Kommunikationsprotokolle für die automatische Synthese in eingebetteten Systemen. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, June 2006.

[35] G. B. Gnokam Defo. VHDL Codegenerierung für rekonfigurierbare Schnittstellen in eingebetteten Systemen. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, May 2005.

[36] M. C. Kouamo Sime. Modellbasierte Spezifikation von Schnittstellen im Hardware Entwurf. Master's thesis, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, Feb. 2005.

[37] T. Loke. Synthese von Kommunikationsstrukturen in verteilten eingebetteten Systemen. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, Mar. 2005.

[38] J. Twiefel. Sichere internetbasierte echtzeitfähige Robotersteuerung. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, May 2003.

Bibliography

[39] S. Abdi, D. Shin, and Daniel D. Gajski. Automatic Communication Refinement for System Level Design. In *DAC*, Center for Embedded Computer Systems University of California Irvine CA 92697 USA, June 2003. Copyright 2003 ACM.

[40] Anat Agarwal. Raw Computation. *Scientific American*, 281(2):44–47, August 1999.

[41] A. Ahmadinia, J. Ding, C. Bobda, and J. Teich. Design and Implementation of Reconfigurable Multiple Bus on Chip (RMBoC). Technical Report 02-2004, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, November 2004.

[42] Ali Ahmadinia, Christophe Bobda, Dirk Koch, Mateusz Majer, and Jürgen Teich. Task scheduling for heterogeneous reconfigurable computers. In *SBCCI*, 2004.

[43] Janaki Akella and Kenneth L. McMillan. Synthesizing Converters Between Finite State Protocols. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 410–413, Washington, DC, USA, 1991. IEEE Computer Society.

[44] Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 2006.

[45] Virtual Socket Interface Alliance. *Architecture Document*. Los Gatos, California, 1997.

[46] Virtual Socket Interface Alliance. *System Level Design, Model Taxonomy*. Los Gatos, California, 1998.

[47] VSI Alliance. Official VSIA Homepage. http://www.vsi.org/.

[48] Don Anderson. *FireWire System Architecture*. Addison Wesley, Mindshare Inc., second edition, 1999.

[49] Automotive Opens System Architecture. Autosar Homepage. http://www.autosar.org/.

[50] P.J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, San Diego, CA, second edition, 2002.

[51] Forschungszentrum Informatik (FZI) at the University of Karlsruhe and Department of Technical Computer Science University of Tübingen. *Spyder - Virtex - X2, User's Manual*, July 2000.

[52] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts. *Handel-C Language Reference Guide*. Oxford University Computing Laboratory, August 1996.

[53] John Aynsley and David Long. *Draft Standard SystemC Language Reference Manual*. Open SystemC Initiative (OSCI), August 2005.

[54] M. Bedford Taylor and et.al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of International Symposium on Computer Architecture*. IEEE Computer Society, June 2004.

[55] M. Bednara, Klaus Danne, Markus Deppe, Oliver Oberschelp, Frank Slomka, and Jürgen Teich. Design and Implementation of Digital Linear Control Systems on Reconfigurable Hardware. *EURASIP Journal on Applied Signal Processing*, 2003(6):594–602, 2003.

[56] L. Benini and G. De Micheli. Network on Chips: A New SoC Paradigm, 2001.

[57] Gérard Berry. *The Esterel v5 Language Primer*. Centre de Math´ematiques Appliqu´ees, Ecole des Mines and INRIA, Sophia-Antipolis, version 5.21 release 2.0 edition, April 1999.

[58] C. Bobda. Temporal Partitioning and Sequencing of Dataflow Graphs on Reconfigurable Systems. In *International IFIP TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002)*, pages 185–194, Montreal, Canada, 2002. IFIP.

[59] Christophe Bobda. *Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement*. PhD thesis, University Paderborn, Heinz Nixdorf Institute, 2003.

[60] Christophe Bobda and Ali Ahmadinia. Design and Implementation of Digital Linear Control Systems on Reconfigurable Hardware. *IEEE Design & Test of Computers, Special Issue on Networks on Chip*, September 2005.

[61] Christophe Bobda, Ali Ahmadinia, and Rajesh Kurapati. DyNoC : A Communication Infrastructure for Dynamic Communication on Reconfigurable Devices. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, Tempere, Finland, August 2005.

[62] Christophe Bobda, Klaus Danne, Ali Ahmadinia, and Jürgen Teich. A New Approach for Reconfigurable Massively Parallel Computers. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'03)*. IEEE, December 2003.

[63] Christophe Bobda, Mateusz Majer, Dirk Koch, Ali Ahmadinia, and Jürgen Teich. A Dynamic NoC Approach for Communication in Reconfigurable Devices. In *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL)*, volume 3203 of *Lecture Notes in Computer Science (LNCS)*, pages 1032–1036, Antwerp, Belgium, August 2004. Springer.

[64] Borland. Together J, the Together J case tool. http://www.togethersoft.com/.

[65] G. Borriello and Randy Katz. Synthesis and Optimization of Interface Transducer Logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 274–277, November 1987.

[66] Gaetano Borriello. *A new interface specification methodology and its application to transducer synthesis*. PhD thesis, University of California, 1988. Chair-Randy H. Katz.

[67] G. Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Int. Workshop on Field-Programmable Logic and Applications (FPL)*, 1996.

[68] A. Brinkmann, D. Langen, and U Rückert. A Rapid Prototyping Environment for Microprocessor based System-on-Chips and its Application to the Development of a Network Processor. In *10th International Conference on Field Progammable Logic and Applications (FPL'2000)*, pages 838–841, Villach, Austria, August 2000.

[69] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, August 2004.

[70] Giorgio C. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston [u.a.], 3 edition, 2000.

[71] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Donald E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 55–64. ACM Press, 1998.

[72] L. Cai, D. Gajski, P. Kritzinger, and M. Olivares. Top-Down System Level Design Methodology Using SpecC. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*. IEEE Press, 2002.

[73] Mirko Caspar, Markus Visarius, and André Meisel. Spezifikation und Entwurf einer Multi-Port-Schnittstelle. In *Proc. of the German Workshop Dresdner Arbeitstagung Schaltungs- und Systementwurf*, pages 59–63, Dresden, Germany, April 2005.

[74] K.C. Chang. *Digital Systems Design with VHDL and Synthesis*. IEEE Computer Society, Los Alamitos, CA, 1999.

[75] P. Chou, R. Ortega, and G. Borriello. Synthesis fo the hardware/software interface in microcontroller-based systems. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 488–495, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[76] P. Chou, R. Ortega, and G. Borriello. Interface co-synthesis techniques for embedded systems. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 280–287, Washington, DC, USA, 1995. IEEE Computer Society.

[77] P. M. Chu and M. T. Liu. Synthesizing protocol specifications from from service specification in the FSM model. In *Proceedings of Computer Networking Symposium*, pages 173–182, Washington, DC, USA, 1988. IEEE Computer Society.

[78] System C Community. System C Homepage. http://www.systemc.org/.

[79] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computer Surveys*, 34(2), 2002.

[80] ASN Consortium. ASN.1 Information Site. http://asn1.elibel.tm.fr/.

[81] Spirit Consortium and Philips Electronics. Official Spirit Homepage, Structure for Packaging, Integrating and Re-using IP within Tool-flows. http://spiritconsortium.org/.

[82] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press Cambridge, Massachusetts London, England, 2001.

[83] L. Cristiano, G. Reuber, J. N. Claudionor, C. da Silva Jr. Diógenes, O. Antonio, L. Luciana, and H. Luciana. JADE: An Embedded Systems Specification, Code Generation and Optimization. In *JADE*. LOCOM/DCC/UFMG - Federal University of Minas Gerais - Brazil, 2000.

[84] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the Design Automation Conference*, pages 684–689, Las Vegas, NV, June 2001.

[85] Klaus Danne. Operating Systems for FPGA Based Computers and Their Memory Management. In *ARCS 2004 Organic and Pervasive Computing, Workshop Proceedings*, volume P-41 of *GI-Edition Lecture Notes in Informatics (LNI)*, Bonn, March 2004. Köllen Verlag.

[86] Klaus Danne, Christophe Bobda, and Heiko Kalte. Increasing Efficiency by Partial Hardware Reconfiguration: Case Study of a Multi-Controller System. In Toomas Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, Nevada, USA*, June 2003.

[87] Klaus Danne, Christophe Bobda, and Heiko Kalte. Run-time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL2003), Lisbon, Portugal*, September 2003.

[88] J.-M. Daveau, G. F. Marchioro, C. A. Valderrama, and A. A. Jerraya. VHDL generation from SDL specifikations. In *VHDL*. Chapmann & Hall, 1996.

[89] P. Delforge. IP Business Models, IP reuse. In *In Proc. of Intellectual Property in Electronics Conference (IP98)*, pages 11–16, March 1998.

[90] Design and Reuse. D & R Homepage. http://www.us.design-reuse.com/.

[91] Florian Dittmann and Achim Rettberg. Design of Partially Reconfigurable Systems: From Abstract Modeling to Practical Realization. In *1st International Workshop on Reconfigurable Computing Education*, Karlsruhe, Germany, 1March 2006.

[92] Florian Dittmann, Achim Rettberg, and Fabian Schulte. A Y-Chart Based Tool for Reconfigurable System Design. In *Workshop on Dynamically Reconfigurable Systems (DRS) 2005*, Innsbruck, Austria, 17 March 2005.

[93] R. Dömer. *System-level Modeling and Design with the SpecC Language*. PhD thesis, University of Dortmund, Germany, April 2000.

[94] R. Dömer and D. Gajski. Reuse and Protection of Intellectual Property in the SpecC System. In *Asia and South Pacific Design Automation Conference 2000, Yokohama, Japan*, January 2000.

[95] R. Dömer, D. Gajski, and J. Zhu. IP-centric Methodology and Design with the SpecC Language. In *NATO-ASI Workshop on System Level Synthesis for Electronic Design, Il Ciocco, Lucca, Italy*, August 1998.

[96] R. Dömer, D. Gajski, J. Zhu, G. Aggarwal, E. Chang, T. Ishii, J. Kleinsmith, and J. Zhu. IP-Centric Methodology. In *Proceedings of the 1st International Workshop on Design, Test and Applications, Dubrovnik, Croatia*, June 1998.

[97] R. Dömer, D. Gajski, J. Zhu, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston, University of California, Irvine, March 2000.

[98] R. Dömer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual*. SpecC Technology Open Consortium, 2002.

[99] Bruce Powel Douglas. *Real-time UML: Developing efficient objects for embedded systems*. Addison-Wesley, Reading, Massachusetts [u.a.], first edition, 1998. The Addison-Wesley object technology series.

[100] Bruce Powel Douglas. *Doing hard time : developing real-time systems with UML objects, frameworks and patterns*. Addison-Wesley, Reading, Massachusetts [u.a.], third edition, 2000. The Addison-Wesley object technology series.

[101] Bruce Powel Douglass. Custom Embedded Communication Protocols. Technical report, I-Logix Inc., 2002.

[102] Bruce Powel Douglass. Dr. Douglass' Guided Tour Through the Wonderland of Systems Engineering, UML and Rhapsody. Technical report, I-Logix Inc., 2002.

[103] Bruce Powel Douglass. Rhapsody for Systems Architecture - Better Architecture With the UML. Technical report, I-Logix Inc., June 2002.

[104] Bruce Powel Douglass. Breakthroughs in Software and Systems Engineering. Technical report, I-Logix Inc., 2004.

[105] Olivier Dubuisson. *ASN.1 - Communication between heterogeneous systems*. Morgan Kaufmann Publishers, September 2000.

[106] On-Chip Bus (OCB) Development Working Group (DWG). *Virtual Component Interface (VCI) Standard (OCB 2 1.0)*. Los Gatos, California, 2000.

[107] Michael Eisenring and Jürgen Teich. Interfacing Hardware and Software. In *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, pages 520–524, London, UK, 1998. Springer-Verlag.

[108] Wilfried Elmenreich, Günther Bauer, and Hermann Kopetz. The Time-Triggered Paradigm. In *Proceedings of the Workshop on Time-Triggered and Real-Time Communication*, Manno, Switzerland, December 2003.

[109] Rolf Ernst, Jorg Henkel, and Thomas Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Des. Test*, 10(4):64–75, August 1993.

[110] G. Estrin, B. Bussel, R. Turn, and J. Bibb. Parallel processing in a restrucurable computer system. *IEEE Transactions on Electronic Computers*, 12(5), 1963.

[111] G. Estrin and R. Turn. Automatic assignment of computations in a variable structure computer system. *IEEE Transactions on Electronic Computers*, 12(5), 1963.

[112] Virtual Component Exchange. Virtual Component Exchange - VCX facilitates transactions in virtual components (VCs), within an efficient, international and open market infrastructure. http://www.thevcx.com/, December 2002.

[113] S. Fekete, M. Köhler, and J. Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Proc. DATE 2001, Design, Automation and Test in Europe*, Munich, Germany, March 2001.

[114] Ltd. Fujitsu Ltd., Fujitsu Laboratories Ltd. & Hitachi. Component Wrapper Language: A New Language for Interface Specification. Technical report, Fujitsu Laboratories, 9/1 2002.

[115] Ltd. Fujitsu Ltd., Fujitsu Laboratories Ltd. & Hitachi. *Component Wrapper Language Specifications*, 9/1 2002.

[116] Ltd. Fujitsu Ltd., Fujitsu Laboratories Ltd. & Hitachi. *Component Wrapper Language User´s Guide*, 9/1 2002.

[117] Daniel D. Gajski and R. H. Kuhn. Guest Editor's Introduction: New VLSI Tools. *IEEE Computer*, December 1983.

[118] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[119] Michael Gasteier and Manfred Glesner. Bus-Based Communication Synthesis on System-Level. In *ISSS '96: Proceedings of the 9th international symposium on System synthesis*, page 65, Washington, DC, USA, 1996. IEEE Computer Society.

[120] A. Gerstlauer, D. Shin, R. Dömer, and D. Gajski. System-Level Communication Modeling for Network-on-Chip Synthesis. In *Asia and South Pacific Design Automation Conference 2005, Shanghai, China*, January 2005.

[121] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, University of California, Irvine, 2001.

[122] L. Ghanmi, A. Ghrab, M. Hamdoun, B. Missaoui, G. Saucier, and K. Skiba. E-Design Based on the Reuse Paradigm. In *Proceedings of the Design, Automation and Test in Europe*, pages 214–220, Paris, France, March 2002.

[123] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[124] Maya B. Gokhale and Paul S. Graham. *Reconfigurable Computing*. Springer, The Netherlands, 2005.

[125] Hassan Gomaa. *Designing concurrent, distributed, and real-time applications with UML*. Addison-Wesley, Boston [u.a.], 2000. The Addison-Wesley object technology series.

[126] Prof. Dr. Winfried Görke. *Fehlertolerante Rechensysteme*. R. Oldenbourg Verlag GmbH, München, 1989.

[127] M. G. Gouda and Y. T. Yu. Synthesis of communicaiton finite state machines with guaranteed progress. *IEEE Trans. Commun.*, 32(7):779–788, 1984.

[128] Mentor Graphics and Synopsys. OpenMORE Assessment Program. http://www.openmore.com/, August 2002.

[129] Björn Griese, Erik Vonnahme, Mario Porrmann, and Ulrich Rückert. Hardware Support for Dynamic Reconfiguration in Reconfigurable SoC Architectures. In *Proceedings of the International Conference on Field Programmable Logic and its Applications (FPL2004)*, Antwerp, Belgium, August 2004.

[130] System-Level Design Development Working Group. *System-Level Interface Behavioral Documentation Standard (SLD 1 1.0)*. Los Gatos, California, 2000.

[131] Virtual Component Transfer Development Working Group. *Virtual Component Attributes (VCA) With Formats for Profiling, Selection, and Transfer Standard 2.3 (VCT 2 2.3)*. VSI Alliance, Los Gatos, California, 2001.

[132] Virtual Component Transfer Development Working Group. *Virtual Component Transfer Specification 2 (VCT 1 2.1)*. VSI Alliance, Los Gatos, California, 2001.

[133] P. Gutberlet and W. Rosenstiel. Specification of interface components for synchronous data paths. In *ISSS '94: Proceedings of the 7th international symposium on High-level synthesis*, pages 134–139, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[134] J. Haase. Design Methodology for IP Providers. In *In Proc. of the Design, Automation and Test in Europe*, pages 728–732, March 1999.

[135] M. Hamdoun, A. Ghrab, P. Hernandez, and G. Saucier. IP XML Encapsulation Portal. In *International Workshop on IP-Based SoC Design*, December 2001.

[136] Wolfram Hardt. *Integration von Verzögerungszeit-Invarianz in den Entwurf eingebetteter Systeme*. Phdthesis, Universität Paderborn, Fachbereich Mathematik und Informatik, Warburger Str. 100, 33098 Paderborn, Deutschland, 2000. Habilitationsschrift.

[137] Wolfram Hardt, André Meisel, and Markus Visarius. Automatische Rekonfigurierung von Hardware-Schnittstellen. In *Proc. of the German Workshops "Dresdner Arbeitstagung Schaltungs- und Systementwurf" and "Intellectual Property Prinzipien"*, Dresden, Germany, April 2004.

[138] Wolfram Hardt, Franz Rammig, Carsten Böke, Joachim Stroop, Achim Rettberg, G. Del Castillo, Bernd Kleinjohann, and Jürgen Teich. IP-based System Design within the PARADISE Design Environment. *Journal of Systems Architecture – the Euromicro Journal*, 2001.

[139] Wolfram Hardt, Franz Rammig, Achim Rettberg, Bernd Kleinjohann, Markus Visarius, and Marc André Wiese. The PARADISE Design Environment - Design Environment for parallel & distributed embedded RT systems. In *Design Automation Conference (DAC) University Booth - Demonstration and Poster Exhibition*, Los Angeles, CA, USA, June 2000.

[140] J. Harkin, T. M. McGinnity, and L. P. Maguire. Modeling and optimizing run-time reconfiguration using evolutionary computation. *Transactions on Embedded Computing Sys.*, 3(4):661–685, 2004.

[141] Matthew Hause and F.J. Thom. Building Embedded Systems with UML 2.0/SysML. Technical report, ARTiSAN Software Tools GmbH, 2004.

[142] Matthew Hause, Francis Thom, and Alan Moore. The Systems Modelling Language (SysML). Technical report, ARTiSAN Software Tools GmbH, 2005.

[143] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on chip: An architecture for billion transistor era. In *Proceeding IEEE NorChip Conference*, 2000.

[144] John L. Hennessy and David A. Patterson. *Computer organization and design: The hardware software interface*. Morgan Kaufmann, San Francisco, Calif., 1994. With a contrib. by James R. Larus.

[145] John L. Hennessy and David A. Patterson. *Computer Architecture*. Morgan Kaufmann, San Francisco, California, 2003.

[146] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Formal Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[147] Michael Hübner, Michael Ullmann, Lothar Braun, A. Klausmann, and Jürgen Becker. Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems. In *Int. Workshop on Field-Programmable Logic and Applications (FPL)*, pages 1037–1041. Springer, 2004.

[148] I-Logix. I-Logix Homepage. http://www.ilogix.com/.

[149] I-Logix. Reuse of IP with UML2.0 based Model-Driven Developement. Technical report, I-Logix Inc., 2004.

[150] Department YM5A IBM Corporation. *32-Bit Device Control Register Bus, Architecture Specifications, Version 2.9, SA-14-2525-00*, June 2000.

[151] IEEE and Dennis Brophy. System Verilog Homepage. http://www.systemverilog.org/.

[152] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, June 2004.

[153] 4.0 JavaCC˙ *Java Compiler Compiler (JavaCC^{TM}) - The Java Parser Generator.* https://javacc.dev.java.net/, 2006.

[154] Y. Kakuda and Y. Wakahara. Component-based synthesis of protocols for unlimited number of processes. In *Proceedings of IEEE COMPSAC'87*, pages 721–730. IEEE Computer Society, 1988.

[155] Yoshiaki Kakuda and Hironori Saito. An Integrated Approach to Design of Protocol Specifications Using Protocol Validation and Synthesis. *IEEE Trans. Comput.*, 40(4):459–467, 1991.

[156] H. Kalte, M. Porrmann, and U. Rückert. Rapid Prototyping System für dynamisch rekonfigurierbare Hardwarestrukturen. In *AES2000*, pages 150–157, Karlsruhe, January 2000.

[157] H. Kalte, M. Porrmann, and U. Rückert. A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*, 2002.

[158] M. Keating and P. Bricaud. *Reuse Methodology Manual for System-On-A-Chip Designs.* Kluwer Academic Publishers, Boston Dordrecht London, June 1998.

[159] M. Keating and P. Bricaud. *Reuse Methodology Manual for System-On-A-Chip Designs.* Kluwer Academic Publishers, Boston Dordrecht London, second edition edition, June 1999.

[160] M. Keating, P. Bricaud, and R. J. Rickford. *Reuse Methodology Manual for System-On-A-Chip Designs.* Kluwer Academic Publishers, Boston Dordrecht London, third edition edition, June 2002.

[161] Paul W. King. Formalization of Protocol Engineering Concepts. *IEEE Transactions on Computers, Special Issue On Protocol Engineering*, 40(4):387–403, 1991.

[162] Akira Kitajima, Keiichi Yasumoto, Teruo Higashino, and Kenichi Taniguchi. Deriving Concurrent Synchronus EFSMs from Protocol Specifications in LOTOS. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E82-A(3):487–494, 1999.

[163] Peter Voigt Knudsen and Jan Madsen. Integrating Protocol Selection with Hardware/Software Codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1077–1095, August 1999.

[164] M. Koegst, P. Conradi, D. Garte, and M. Wahl. A Systematic Analysis of Reuse Strategies for Design of Electronic Circuits. In *Proceedings of the Design, Automation and Test in Europe*, pages 292–296, March 1998.

[165] Hermann Kopetz, Michael Holzmann, and Wilfried Elmenreich. A Universal Smart Transducer Interface: TTP/A. In *3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2000)*, March 2000.

[166] Hermann Kopetz and Neeraj Suri. Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces. *6th IEEE International Symposium on Object-Oriented Real-Time Computing (ISORC03), May 14 - 16, 2003, Hokkaido, Japan*, May 2003.

[167] Herrmann Kopetz. The Time-Triggered Architecture. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 22–29, Kyoto, Japan, April 1998.

[168] Herrmann Kopez. *Design Principles for Distributed Embedded Applications*. Kluwer Academic Publ., Boston [u.a.], fourth edition, 2001.

[169] J. Lessmann. Integration of Tools for IP based Design with a Generic XML Format and Web Services. Master's thesis, University Paderborn, June 2003.

[170] Davin Lim and Mike Peattie. *Xilinx Application Note XAPP290: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, May 2002.

[171] Roger Lipsett, Carl Schaefer, and Cary Ussery. *Hardware Description and Design*. Kluwer Academic Publishers, Boston/Dordrecht/London, first edition, 1989.

[172] Ming T. Liu. Introduction to Special Issue on Protocol Specification. *IEEE Transactions on Computers, Special Issue On Protocol Engineering*, 40(4):373–375, 1991.

[173] A Mäder. *Guide to HDL Coding Styles for Synthesis*. Universität Hamburg – Fachbereich Informatik, Hambug, Germany, 1999. Arbeitsbereich Technische Grundlagen der Informatik.

[174] Andreas Mäder. *VHDL Kurzbeschreibung*. Universität Hamburg – Fachbereich Informatik, Hambug, Germany, 2001. Arbeitsbereich Technische Grundlagen der Informatik.

[175] Andreas Mäder. *VHDL Kompakt*. Universität Hamburg – Fachbereich Informatik, Hambug, Germany, 2005. Arbeitsbereich Technische Grundlagen der Informatik.

[176] Theodore Marescaux, Andrei Bartic, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins, editors. *Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs*. IMEC, Springer Verlag, September 2002.

[177] Theodore Marescaux, J-Y. Mignolet, Andrei Bartic, W. Moffat, Dideriek Verkest, Serge Vernalde, and Rudy Lauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable Systems. In *Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, September 2003.

[178] Markus Voelter. *Program Generation, A Survey of Techniques and Tools*. http://www.voelter.de/data/presentations/ProgramGeneration.zip, 2002.

[179] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers, The Netherlands, 2003.

[180] Stanley Mazor and Patricia Langstraat. *A Guide to VHDL*. Kluwer Academic Publishers, Berlin Heidelberg, first edition, 1992.

[181] Spirit Schema Working Group Membership. *Spirit-User Guide*, 06 2005.

[182] Spirit Schema Working Group Membership. *Spirit XML-Schema Documantation*, 06 2005.

[183] Philip Merlin and Gregor von Bochmann. On the Construction of Submodule Specifications and Communication Protocols. Technical Report 1, Univ. of Montreal, Dep. Inform. Oper. Res., New York, NY, USA, 1983.

[184] Sanjiv Narayan and Daniel D. Gajski. Protocol generation for communication channels. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 547–551, New York, NY, USA, 1994. ACM Press.

[185] Sanjiv Narayan and Daniel D. Gajski. Interfacing incompatible protocols using interface process generation. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 468–473. ACM Press, 1995.

[186] Scott Niemann. Executable Systems Design with UML 2.0. Technical report, I-Logix Inc., 2004.

[187] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *IPDPS*, 2003.

[188] Johnny Öberg, Anshul Kumar, and Ahmed Royal. Grammar-based Hardware Synthesis of Data Communication Protocols. In *ISSS '96: Proceedings of the 9th international symposium on System synthesis*, page 14, Washington, DC, USA, 1996. IEEE Computer Society.

[189] Simon Oberthür and Carsten Böke. Flexible Resource Management - A framework for self-optimizing real-time systems. In Bernd Kleinjohann, Guang R. Gao, Hermann Kopetz, Lisa Kleinjohann, and Achim Rettberg, editors, *Proceedings of IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'04)*. Kluwer Academic Publishers, 2004.

[190] Simon Oberthür, Carsten Böke, and Björn Griese. Dynamic Online Reconfiguration for Customizable and Self-Optimizing Operating Systems. In *Proceedings of the 5th ACM international conference on Embedded software (EMSOFT'2005)*, pages 335–338, 2005. Jersey City, New Jersey.

[191] Object Management Group (OMG). *Common Object Request Broker Architecture: Core Specification*, March 2004.

[192] Object Management Group (OMG). *Unified Modeling Language, Superstructure Version 2.0*. http://www.omg.org/docs/ptc/03-08-02.pdf, April 2004.

[193] Mattias O'Nils. *Specification, Synthesis and Validation of Hardware/Software Interfaces*. PhD thesis, Royal Institute of Technology, April 1999.

[194] Frank Oppenheimer, Dongming Zhang, and Wolfgang Nebel. COHSID: ComiX HW/SW Interface Designer, 2001.

[195] Frank Oppenheimer, Dongming Zhang, and Wolfgang Nebel. Modelling communication interfaces with comix. In *Ada Europe '01: Proceedings of the 6th Ade-Europe International Conference Leuven on Reliable Software Technologies*, pages 337–348, London, UK, 2001. Springer-Verlag.

[196] Robert Orfali and Dan Harkey. *Client Server Programming with Java and Corba*. John Wiley and Sons, 2nd edition, 2003.

[197] R. Ortega and G. Borriello. Communication Synthesis for Embedded Systems with Global Considerations. In *Proceedings of the Codes/CACHE 1997, Braunschweig, Germany*, pages 24–26, March 1997.

[198] R. Ortega and G. Borriello. Communication Synthesis for Distributed Embedded Systems. In *International Conference on Computer-Aided Design*, 1998.

[199] David A. Patterson and John L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann, San Francisco, California, 1998.

[200] Robert L. Probert and Kassem Saleh. Synthesis of Communication Protocols: Survey and Assessment. *IEEE Transactions on Computers, Special Issue On Protocol Engineering*, 40(4):468–476, 1991.

[201] C. V. Ramamoorthy, Siyi T. Dong, and Yutaka Usuda. An implementation of an automated protocol synthesizer (APS) and its application to the X.21 protocol. *IEEE Trans. Softw. Eng.*, 11(9):886–908, 1985.

[202] Rational. Rose Rational Software Corporation. The Rational Rose case tool. http://www.rational.com/.

[203] Achim Rettberg, Wolfram Hardt, and Markus Visarius. The PARADISE Design Environment. In *Design, Automation and Test in Europe (DATE) University Booth - Demonstration and Poster Exhibition*, Paris, France, March 2000.

[204] A. Reutter, B. Mößner, and W. Rosenstiel. Design of generic components for efficient reuse in high level designs. In *Workshop on System Design Automation*, pages 61–67, March 1998.

[205] James A. Rowson and Alberto 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 Press.

[206] E. A. Rundensteiner and D. D. Gajski. A Design Representation Model for High-Level-Synthesis. Technical Report 90-27, UCI, September 1990.

[207] Kassem Saleh. Synthesis of communications protocols: An annotated bibliography. *SIGCOMM Comput. Commun. Rev.*, 26(5):40–59, 1996.

[208] Kassem Saleh and Robert L. Probert. Synthesis of Error-Recoverable Protocol Specifications From Service Specifications. In *Proceedings of Second Int. Conf. Comput. Inform.*, pages 428–433, May 1990.

[209] Th. Scharnhorst. AUTOSAR Web Contend. Technical report, AUTOSAR GbR, October 2005.

[210] Th. Scharnhorst, H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, P. Heitkämper, J. Leflour, J.-L. Maté, and K. Nishikawa. AUTOSAR – Challenges and Achievements 2005. Technical Report 1907, AUTOSAR GbR, 2005.

[211] Herman Schmit. Incremental Reconfiguration for Pipelined Applications. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, page 47. IEEE Computer Society, 1997.

[212] Herman Schmit, David Whelihan, Andrew Tsai, Matthew Moe, Benjamin Levine, and R. Reed Taylor. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 63–66, 2002.

[213] S. Schulz. Towards A New System Level Design Language – SLDL. In A. Jerraya and J. Mermet, editors, *System Level Synthesis*. Kluwer Academic Publishers, 1999.

[214] A. Seawright and F. Brewer. Clairvoyant: A synthesis system for production-based specification. *IEEE Transactions on VLSI Systems*, 2:172–185, June 1994.

[215] R. Seepold. Reuse of IP and Virtual Components. In *In Proc. of the Design, Automation and Test in Europe*, March 1999.

[216] Bran Selic. The Real-Time UML Standard: Definition and Application. In *3rd International Symposium on Distributed Objects & Applications*, 2001.

[217] D. Shin and Daniel D. Gajski. Interface Synthesis from Protocol Specification. Technical Report CECS-02-13, Department of Information and Computer Science University of California, Irvine, April 2002.

[218] D. Shin, A. Gerstlauer, R. Dömer, and D. Gajski. Automatic Generation of Communication Architectures. In *International Embedded Systems Symposium, "From Specification to Embedded Systems Application"*, Manaus, Brazil, August 2005. Springer Verlag.

[219] Robert Siegmund, K. Bohn, and Dietmar Müller. Specification and Synthesis of Customizable Interfaces of Soft IP Cores using VHDL+. In *8th IFIP International Workshop on IP Based Synthesis and System Design*. Kluwer Academic Publishers, December 1999.

[220] Robert Siegmund and Dietmar Müller. SystemCSV - an extension of SystemC for mixed multi-level communication modeling and interface-based system design. In

DATE '01: Proceedings of the conference on Design, automation and test in Europe, pages 26–33, Piscataway, NJ, USA, 2001. IEEE Press.

[221] Robert Siegmund and Dietmar Müller. A Novel Synthesis Technique for Communication Controller Hardware from declarative Data Communication Protocol Specifications. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 602–607, New York, NY, USA, 2002. ACM Press.

[222] Robert Siegmund and Dietmar Müller. Automatic Synthesis of Communication Controller Hardware from Protocol Specifications. *IEEE Design and Test of Computers*, 19(4):84–95, 2002.

[223] Douglas J. Smith. *HDL Chip Design*. Doone Publications, Madison, AL, USA, seventh edition, 2000. A Practical Guide For Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog.

[224] James Smith and Giovanni De Micheli. Automated composition of hardware components. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 14–19, New York, NY, USA, 1998. ACM Press.

[225] Christoph Steiger, Herbert Walder, Marco Platzner, and Lothar Thiele. Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices. In *Proceedings of the 24th International Real-Time Systems Symposium (RTSS'03)*, December 2003.

[226] J. Stuyt, N. Moualla, and M. ten Have. *CoReUse Standards Book Medea ToolIP*. Philips Semiconductors ReUse Technology Group, Eindhoven, The Netherlands, September 2001.

[227] J. S. Sun and R. W. Brodersen. Design of system interface modules. In *Proceedings of International Conference on Computer Aided Design*, pages 478–481, 1992.

[228] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog For Design – A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer, 2004.

[229] Sparx Systems. Enterprise Architecht. http://www.sparxsystems.com.au/.

[230] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Upper Saddle River, USA, 2002.

[231] ISO TC97/SC21. *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[232] Jürgen Teich. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer-Verlag, Berlin Heidelberg, New York [u.a.], 1997.

[233] ARTiSAN Software Tools. ARTiSAN Homepage, Real-Time Solutions for Systems and Software Development. http://www.artisansw.com/.

[234] ARTiSAN Software Tools. AUTOSAR and UML – A Natural Fit? Technical report, ARTiSAN Software Tools GmbH, 2005.

[235] ARTiSAN Software Tools. Extending UML to Enable the Definition and Design Realtime Embedded Systems. Technical report, ARTiSAN Software Tools GmbH, 2005.

[236] Matthias Treydte, André Meisel, and Markus Visarius. Bus Macro Generierung für rekonfigurierbare Hardwarekomponenten auf VHDL Ebene. In *Proc. of the German Workshop Dresdner Arbeitstagung Schaltungs- und Systementwurf*, pages 41–45, Dresden, Germany, April 2005.

[237] *XILINX Virtex Data Sheet, DS003(1-4)*, 2002.

[238] Markus Visarius and Wolfram Hardt. The IPQ Format – An Approach to Support IP based Design. In *Proc. of the 7. GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 106–115, Kaiserslautern, Germany, February 2004. Shaker Verlag.

[239] Markus Visarius and Wolfram Hardt. An IPQ Format based Toolbox to Support IP based Design. *it - Information Technology*, 04:98–106, April 2005.

[240] Markus Visarius, Wolfram Hardt, Ralph Bergmann, Ivo Vollrath, Martin Schaaf, Andreas Vörg, Natividad Martínez Madrid, Ralf Seepold, Martin Radetzki, Wolfgang Thronicke, Steffen Rülke, Carsten Tautz, Ralph Traphöner, Vasco Jerinic, Dietmar Müller, and Robert Siegmund. Requirements on the IP Qualifying Format. TechReport 08, University of Paderborn, Informatik- und Prozesslabor, Warburger Strasse 100, 33098 Paderborn, Germany, August 2001.

[241] Markus Visarius, Johannes Lessmann, and Wolfram Hardt. Tool Demonstration on IPQ Format based Retrieval. In *Proc. of the International Workshop of the MEDEA+ Project A-511 ToolIP*, Stresa, Italy, October 2002.

[242] Markus Visarius, André Meisel, Markus Scheithauer, and Wolfram Hardt. Dynamic Reconfiguration of IP based Systems. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, Montreal, Canada, June 2005. IEEE Computer Society.

[243] Gregor von Bochmann. Construction Approaches to Subsystem Specifications. Technical report, School of Information Technology and Engineering (SITE) University of Ottawa, February 2000.

[244] Gregor von Bochmann. Synthesis of Communication Systems. Technical report, School of Information Technology and Engineering (SITE) University of Ottawa, August 2002.

[245] Gregor von Bochmann and Reinhard Gotzhein. Deriving protocol specifications from service specifications including parameters. *ACM Transactions on Computer Systems*, pages 255–283, 1990.

[246] W3C. *Extensible Markup Language (XML) 1.0*, January 2002.

[247] H. Walder, C. Steiger, and M. Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS) / Reconfigurable Architectures Workshop (RAW)*, page 178. IEEE Computer Society, April 2003.

[248] E. Walkup and G. Borriello. Automatic Synthesis of Device Drivers for Hardware/-Software Co-design. Technical Report Technical Report 94-06-04, Department of Computer Science and Engineering University of Washington, Seattle, Box 352350 University of Washington, Seattle, WA 98195-2350, August 1994.

[249] Jacek Wytrebowicz. Hardware specification generated from Estelle. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 435–450, London, UK, 1996. Chapman & Hall, Ltd.

[250] Keiichi Yasumoto, Akira Kitajima, Teruo Higashino, and Kenichi Taniguchi. Hardware synthesis from protocol specification in LOTOS. In *Joint International Conference on 11th Formal Description Techniques and 18th Protocol Specification, Testing, and Verification (FORTE/PSTV'98)*, pages 405–420, 1998.

[251] P. Zafiropulo and et al. Towards analyzing and synthesizing protocols. *IEEE Transaction on Communications*, 28(4):651–661, April 1980.

[252] Y. X. Zhang, K. Takahashi, N. Shiratori, and S. Noguchi. An Interactive Protocol Synthesis Algorithm Using a Global State Transition Graph. *IEEE Trans. Softw. Eng.*, 14(3):394–404, 1988.

List of Abbreviations

IFS Related Abbreviations

BB	Basic Block
CU	Control Unit
EIFD	Extended IFDs
$\mathbf{f}_{Map}$	Mapping Functions
IF	Interface
IFB	Interface Block
IFD	Interface Description
IFD-Format	Interface Description Format
IFS	Interface Synthesis
I - P - O	Input - Processing - Output
PH	Protocol Handler
PH-Mode	Protocol Handler Mode
PMN	Protocol Matrix Node
PP	Protocol-Pin
SH	Sequence Handler
SH-Mode	Sequence Handler Mode
TPD	Target-Platform Description

General Abbreviations

ACK	Acknowledge
ACL	Abstract Communication Layer
ADT	Abstract Data Type
ALU	Arithmetic-Logic Unit
AMBA	Advanced Microprocessor Bus Architecture
API	Application Programmer Interface
ASIC	Application Specific Integrated Circuit

ASN.1	Abstract Syntax Notation One
AVCI	Advanced VCIs
BM	Bus Macros
BRAM	Block RAM
BVCI	Basic VCIs
CASE	Computer-Aided-Software-Engineering
CBC	Cipher-Block Chaining
CFG	Control Flow Graph
CLB	Complex Logic Block
CNI	Communication Network Interface
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
CPB	Cycles-Per-Bit
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRC	Cyclic Redundancy Checksum
CWL	Component Wrapper Language
DCR	Device Control Register
DLL	Dynamic Link Library, Microsoft Windows
DOM	Document Object Model
DPRAM	Dual Ported Random-Access Memory
DSP	Digital Signal Processor
DSR	Data Set Ready
DTR	Data Terminal Ready
ECB	Electronic Codebook
EDA	Electronic Design Automation
ES	Embedded System
ESD	Electronic System Design
ESP	Electronic Stability Program
E/E	Electric/Electronic
FF	Flip-Flop
FFT	Fast Fourier Transformation
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FSM	Finite State Machine
Fujaba	From UML to Java and back again
FW	FireWire, IEEE 1398

gcc	GNU C-Compiler
GIOP	General Inter-ORB Protocol
GND	Ground
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Internet Inter-Orb Protocol
IP	Intellectual Property
ISO/OSI	International Standards Organization / Open Systems Interconnection
LAN	Local Area Network
LSI	Large Scale Integration
LUT	Look-Up Table
MDD	Model-Driven Development
MEDL	Message Descriptor List
MMIF	Memory Mapped Interface
MUX	Multiplexer
M-V-C	Model – View – Controller
NoC	Networks on a Chip
OCB	On-Chip Bus
OCB VCI	On-Chip Bus Virtual Component Interface
OCM	On-Chip Memory
OMG	Object Management Group
OPB	On-Chip Peripheral Bus
OPB IFIP	On-Chip Peripheral Bus Interface IP
OS	Operating System
PDC	Park Distance Control
PE	Processing Element
PLA	Programmable Logic Array
PLB	Processor Local Bus
PPC	Power PC
PU	Processing Unit
PVCI	Peripheral VCIs
QoS	Quality of Service
RAM	Random Access Memory

RB	Reconfiguration Block
RCU	Reconfiguration Control Unit
RDY	Ready
RTL	Register Transfer Level
RTOS	Real Time Operating System
RTR-IFB	Runtime Reconfigurable IFB
RTS	Request to Send
RxD	Read Data
SAP	Service Access Point
SBIF	Signal-Based Interface
SDL	Specification and Description Language
SLD	System-Level Design
SLIF	System-Level Interfaces
SM	Switch Matrix
SOAP	Simple Object Access Protocol
SoC	System-on-Chip
SRAM	Static RAM
SW	Software
TC	Transition Condition
TCP/IP	Transmission Control Protocol / Internet Protocol
TMO	Time-triggered Message-triggered Object
TPD	Target Platform Description
TTA	Time Triggered Architecture
TTP	Time Triggered Protocol
TxD	Transmit Data
UML	Unified Modeling Language
USB	Universal Serial Bus
VC	Virtual Component
VCA	Virtual Component Attribute
VCI	Virtual Component Interfaces
VCT	Virtual Component Transfer
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WCET	Worst Case Execution Time
XER	XML Encoding Rules
XML	Extensible Markup Language