# Model-based specification and design of large-scale embedded signal processing systems

Lemaitre, J.

**Citation**

Lemaitre, J. (2008, October 2). *Model-based specification and design of large-scale embedded signal processing systems*. Retrieved from https://hdl.handle.net/1887/13126

| | |
|---|---|
| Version: | Corrected Publisher's Version |
| License: | Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden |
| Downloaded from: | https://hdl.handle.net/1887/13126 |

**Note:** To cite this publication please use the final published version (if applicable).

# Model-based Specification and Design of Large-Scale Embedded Signal Processing Systems

Jérôme Lemaitre

# Model-based Specification and Design of Large-Scale Embedded Signal Processing Systems

**Proefschrift**

ter verkrijging van de graad van Doctor aan de
Universiteit Leiden, op gezag van de Rector Magnificus
Prof. mr. P.F. van der Heijden, volgens besluit van het
College voor Promoties te verdedigen op Donderdag 2
Oktober 2008 klokke 16:15 uur

door

Jérôme Lemaitre
geboren te Compiegne Frankrijk
in 1979

Samenstelling promotiecommissie:

| | | |
|---|---|---|
| promotor | Prof.dr. E. Deprettere | |
| referent | Dr. M. van Veelen | ASML, Veldhoven |
| | | |
| overige leden: | Prof.dr. M. Boasson | Universiteit van Amsterdam |
| | Prof. R. Weber | PRISME/LESI, Universiteit van Orléans, Frankrijk |
| | Dr. S. Alliot | European Patent Office, Den Haag |
| | Dr. A-J. Boonstra | ASTRON, Dwingeloo |
| | Prof.dr. H. Wijshoff | |

The author was affiliated to NWO at ASTRON.
The work in this thesis was carried out in the MASSIVE project supported by STW.

Model-based Specification and Design of Large-Scale Embedded Signal Processing Systems
Jérôme Lemaitre. -
Thesis Universiteit Leiden. - With index, ref. - With summary in Dutch
ISBN 978-90-9023497-7

Copyright ©2008 by Jérôme Lemaitre.
All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without permission of the author.
Printed in France.

# Contents

**Acknowledgments** **ix**

**1 Introduction** **1**

1.1 Large-scale and hierarchical signal processing systems . . . 3

1.1.1 Large scale and distributed system . . . 3

1.1.2 Hierarchical architecture . . . 4

1.1.3 Hierarchical application . . . 5

1.2 Problem statement . . . 6

1.2.1 General problem context . . . 6

1.2.2 Specific problem statement . . . 7

1.3 Solution approach . . . 9

1.3.1 Approach to the general problem . . . 9

1.3.2 Approach to the specific problem . . . 12

1.4 Research contributions . . . 12

1.5 Related work . . . 13

1.6 Thesis outline . . . 15

**2 Application specification** **17**

2.1 Summary . . . 17

2.2 Introduction . . . 18

2.3 Terminology . . . 19

2.4 Selection of a model to specify the behavior of the signal processing network 21

2.4.1 General requirements and constraints . . . . . . . . . . . . . . . . . 22

2.4.2 Selection of the KPN model . . . . . . . . . . . . . . . . . . . . . . 22

2.5 Selection of a model to specify the behavior of the control/monitoring network 23

2.5.1 General requirements and constraints . . . . . . . . . . . . . . . . . 24

2.5.2 Selection of communicating state machines . . . . . . . . . . . . . . 24

2.6 Superimposing a timing network for the synchronization . . . . . . . . . . . 25

2.6.1 Superimposing pulse trains . . . . . . . . . . . . . . . . . . . . . . . 25

2.6.2 Utilization of the notion of time . . . . . . . . . . . . . . . . . . . . 28

2.7 Modeling of the interfacing between the two networks . . . . . . . . . . . . 33

2.7.1 Representation of a process in the signal processing network . . . . . 34

2.7.2 Functional behavior of a process . . . . . . . . . . . . . . . . . . . . 35

2.8 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

2.9 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**3 Architecture specification 41**

3.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

3.2 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

3.3 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

3.4 Representation of platform components . . . . . . . . . . . . . . . . . . . . 45

3.4.1 White-box and black-box models . . . . . . . . . . . . . . . . . . . 45

3.4.2 Processing units . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

3.4.3 Communication units . . . . . . . . . . . . . . . . . . . . . . . . . . 48

3.4.4 Storage units . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

3.5 Signal processing architecture model . . . . . . . . . . . . . . . . . . . . . . 49

3.5.1 First-order architecture templates . . . . . . . . . . . . . . . . . . . 50

3.5.2 Higher-order architecture template . . . . . . . . . . . . . . . . . . . 52

3.6 Control and monitoring architecture model . . . . . . . . . . . . . . . . . . . 53

3.6.1 First-order architecture template . . . . . . . . . . . . . . . . . . . . 53

3.6.2 Higher-order architecture templates . . . . . . . . . . . . . . . . . . 54

3.7 Interfacing of the two architectures . . . . . . . . . . . . . . . . . . . . . . . 54

3.7.1 Interfacing at station level . . . . . . . . . . . . . . . . . . . . . . . . 54
3.7.2 First-order architecture templates interfacing . . . . . . . . . . . . . 56
3.8 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.9 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

**4 Mapping 59**
4.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.2 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.3 Transformations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.3.1 Initialization (re-)structuring transformations . . . . . . . . . . . . . 62
4.3.2 Mapping transformations . . . . . . . . . . . . . . . . . . . . . . . . 65
4.4 Implementation phase . . . . . . . . . . . . . . . . . . . . . . . . . . 71
4.4.1 Input to the implementation phase . . . . . . . . . . . . . . . . . . . 71
4.4.2 Automatic translation to implementation-level specification . . . . . 73
4.5 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

**5 Case studies 77**
5.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.2 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.3 Experimental context and setup . . . . . . . . . . . . . . . . . . . . . 79
5.3.1 Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.3.2 Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.4 Dedicated mapping of the two networks . . . . . . . . . . . . . . . . . 81
5.4.1 Handcrafted design . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.4.2 Semi-automated design . . . . . . . . . . . . . . . . . . . . . . . . 82
5.4.3 Integration of programmable IP components . . . . . . . . . . . . . 85
5.4.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.5 Interfacing of the two networks . . . . . . . . . . . . . . . . . . . . . 89
5.5.1 Standard interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.5.2 Merging the two architectures . . . . . . . . . . . . . . . . . . . . . 91
5.5.3 Requirements for future IP-based designs . . . . . . . . . . . . . . . 94

5.6 Related work . . . . . . . . . . 95
5.7 Conclusions . . . . . . . . . . 96

**6 Conclusion and future work 99**
6.1 Conclusion . . . . . . . . . . 99
6.2 Future work . . . . . . . . . . 101

**A An overview of specification-level models of computation 103**
A.1 Summary . . . . . . . . . . 103
A.2 Introduction . . . . . . . . . . 103
A.3 State-based and event-triggered models . . . . . . . . . . 104
A.3.1 Finite State Automata . . . . . . . . . . 104
A.3.2 StateCharts . . . . . . . . . . 104
A.3.3 Petri Nets . . . . . . . . . . 106
A.3.4 Process algebras . . . . . . . . . . 106
A.3.5 DE and DDE . . . . . . . . . . 107
A.4 Stream-based and dataflow models . . . . . . . . . . 107
A.4.1 KPN . . . . . . . . . . 107
A.4.2 Dataflow models . . . . . . . . . . 108
A.5 Heterogeneous models . . . . . . . . . . 110
A.5.1 Synchronous/Reactive language . . . . . . . . . . 111
A.5.2 *-charts and El Greco . . . . . . . . . . 111
A.5.3 SDL . . . . . . . . . . 111
A.5.4 CFSM . . . . . . . . . . 112
A.5.5 RPN . . . . . . . . . . 112
A.6 Related work . . . . . . . . . . 113
A.7 Conclusions . . . . . . . . . . 113

**Bibliography 114**

**Curriculum Vitae 123**

**Samenvatting 125**

# Acknowledgments

This dissertation is the result of work conducted at the Netherlands Foundation for Radio Astronomy (ASTRON) in Dwingeloo and the Leiden Institute of Advanced Computer Science (LIACS), in the context of the joint MASSIVE project.

At ASTRON, I would like to thank Arnold van Ardenne for giving me the opportunity to start the PhD on the MASSIVE project. I would like to thank my team-mates on the MASSIVE project and in the Digital Embedded Signal Processing Group.

My frequent visits at Leiden University allowed me to get to know Bart Kienhuis, Laurentiu Nicolae, Claudiu Zissulescu, Todor Stefanov and Hristo Nikolov with whom I had interesting discussions.

I would like to thank the friends who supported me during my stay in the Netherlands. They are too many to mention, but know how much their presence was appreciated.

Finally, I would like to thank my family for always encouraging me through the years.

Jérôme Lemaitre, Nantes, September 7, 2008

# Chapter 1

# Introduction

For large-scale signal processing systems such as radio telescopes, converting abstract *system-level specifications* (i.e., a specification in terms of *application*, *architecture*, and association between the two) to *implementation-level specifications* (i.e., a specification that commercially available tools should be able to convert automatically to a real implementation) is an extremely challenging task.

Making significant steps in the analysis of the content, structure and evolution of the universe requires increasing the sensitivity of the radio telescopes. There are several ways to increase the sensitivity of radio telescopes [1], such as increasing the integration time, the bandwidth, and in particular, the collecting area. For example, the next generation Square Kilometer Array radio telescope (*SKA* [2]) requires an increase of two orders of magnitude in sensitivity with respect to current radio telescopes at meter to centimeter wavelengths. To achieve this goal will require a telescope with one square kilometer of collecting area - one hundred times more collecting area than the Very Large Array radio telescope (VLA [3]).

However, relying on traditional parabolic dishes [4] with diameters larger than hundreds of meters would be too expensive due to mechanical-related issues. Instead, the trend is to rely on arrays of many small antennas [5], as in the Low Frequency Array radio telescope (*LOFAR* [6]), where tens of thousands of antennas are distributed in a collecting area with a diameter of a few hundreds of kilometers, and where signals are processed with digital electronics that is also distributed next to the antennas.

Given the large-scale and distributed nature of the radio telescopes that are considered in this thesis, a massive amount of signals has to be processed and transmitted. This massive amount has to be reduced before a final sky image is produced. This reduction requires adequately associating advanced signal processing algorithms together with advanced digital processing and communication technologies to improve the overall performance/cost of the system. Also, the distributed systems we consider have to be able to swap between several high-level signal processing functions. For example, the LOFAR radio telescope must be able to swap at run-time between operation modes that range from spectroscopy to pulsar obser-

vations, or to searches for transients in two frequency ranges. To integrate the most advanced algorithms and digital technologies, and to support several signal processing functions, a structured approach is required when deriving system-level specifications [1] and converting them to implementation-level specifications [2] (see Figure 1.1).

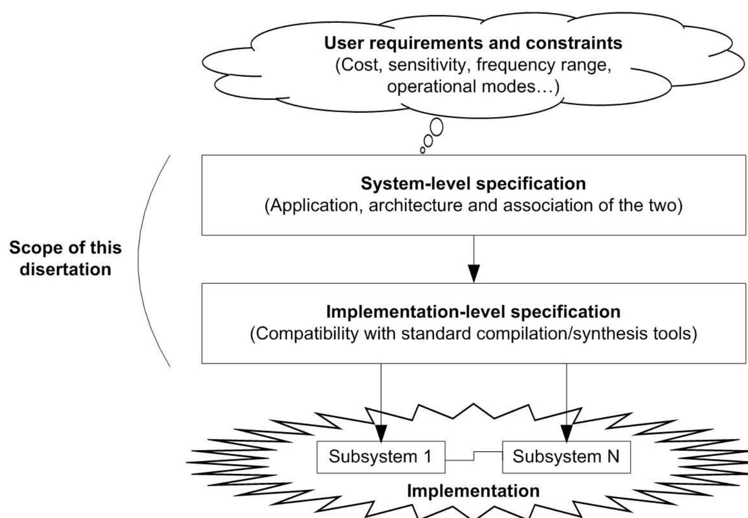


Figure 1.1: Radio telescope designers translate astronomer requirements and constraints to a system-level specification that is to be converted to an implementation-level specification, from where compilation and synthesis tools should take over to obtain automatically a real implementation.

In this thesis we present an approach to structure the design process from system-level specification to implementation-level specification for high-throughput, large-scale and distributed digital signal processing systems, and we focus on the case of phased array radio telescopes.

The remainder of this chapter is organized as follows. In section 1.1 we introduce the scale and hierarchy of the systems we consider, and the signal processing tasks and control/monitoring tasks that are executed in these systems. We present our problem statement in section 1.2 and our solution approach in section 1.3. Then, we summarize our contributions in section 1.4 and give related work in section 1.5. We give the outline of this thesis in section 1.6.

[1] A *system-level specification* consists of an application specification, an architecture specification, and the mapping of the former onto the latter.

[2] An *implementation-level specification* is an abstract specification that includes all the information that is required to be converted automatically to an actual implementation based on commercially available compilation and synthesis tools. Different parts of the system are implemented using different tools.

## 1.1 Large-scale and hierarchical signal processing systems

In this section we present the main properties of the systems we consider, with respect to their scale and hierarchy both from the application and the architecture point of view. We give an overview of the type of signal processing tasks and control and monitoring tasks that are executed in the digital parts of these systems.

### 1.1.1 Large scale and distributed system

Large scale phased array systems connect arrays of sensors to extract some information (e.g., position and velocity) about the source of a signal carried by propagating wave phenomena [7] [8]. To observe celestial sources with a sufficient sensitivity in the low frequency range from 30 MHz to 240 MHz, the LOFAR radio telescope requires approximately 100 arrays, which are distributed in an area that has a diameter of 350 km. Each array is a *station*. In Figure 1.2, stations are represented with circles.

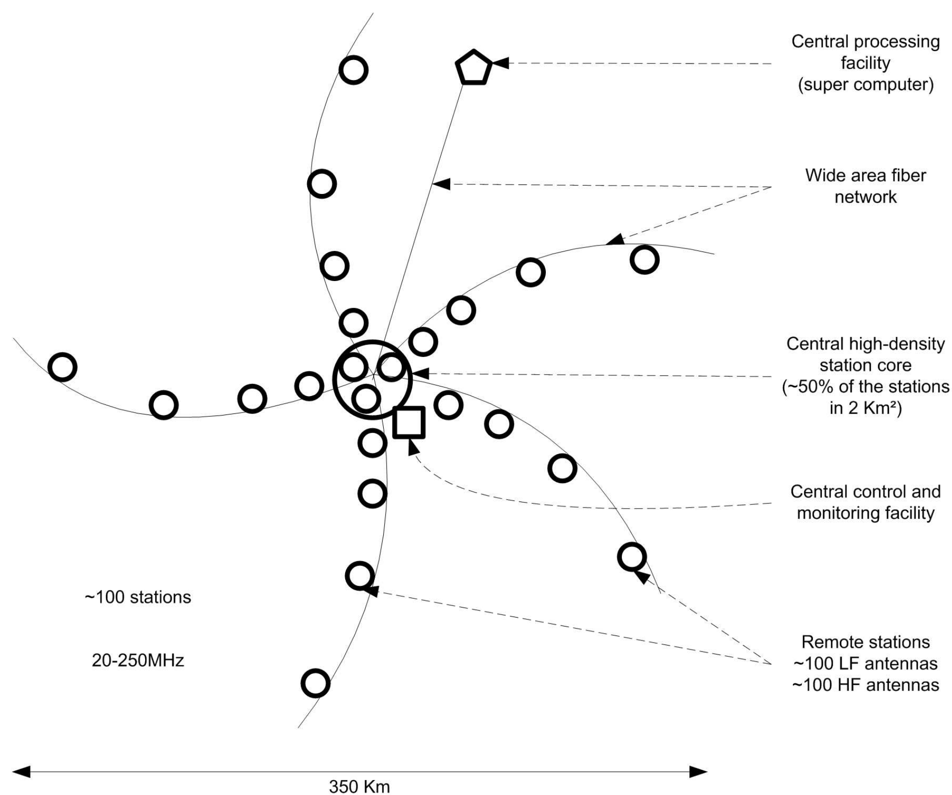


Figure 1.2: Distribution of stations in the large-scale LOFAR radio telescope. The data output by stations is monitored and sent to a central computing facility where a final image is formed.

Stations are clusters of beamforming antennas, and together also form an array. In the middle of the collecting area, about 50 stations are concentrated and together constitute a high density station core of about 2 km of diameter, which provides ultrahigh brightness sensitivity. The remaining 50 stations are placed on 5 spiral arcs and transmit data to the high density station core through a wide area fiber network.

Each station includes approximately 100 dual pole low frequency (LF) antennas and 100 tiles of 16 dual pole high frequency (HF) antennas that receive signals in the 30-80 MHz range and 120-240 MHz range, respectively [9]. The system can operate either in the low frequency range or in the high frequency range. The data output by the stations is sent over the wide area network to a central processing facility (a supercomputer with a processing power equivalent to 10,000 PCs [10], represented with a pentagon in Figure 1.2) where it is further processed in order to obtain a sky image. A central control and monitoring facility (represented with a square in Figure 1.2) monitors the stations output data and sends control messages to the stations so as to adapt the behavior of the system at run-time depending on the observed data [11]. Thus, the system can operate in modes that range from imaging and spectroscopy to pulsar observation or searches for transients, in two frequency ranges.

To focus, we do not cover the design of the front-end stage (i.e., the LF and HF antennas in the stations), and we do not cover the design of the back-end stage (i.e., the supercomputer). Instead, we focus on the design of the intermediate data-reduction stage, which essentially is the digital signal processing and control/monitoring in the stations that all have the same architecture.

### 1.1.2 Hierarchical architecture

The intermediate data reduction stage consists of about 100 stations. In each station, LF and HF antennas are distributed in a relatively small area, say of the size of a football field. Signals that are received by the antennas are amplified and sent to a digital signal processing cabinet in the center of the field. These signals are digitized at a sampling rate of 200 million samples per second, and the input data rate at station level is about 460 Gbps. In the digital signal processing cabinet, this high throughput data is reduced by processing and combining signals in components (equivalent to 100 FPGAs [12] [13]) so as to obtain an output data rate of 2 Gbps at station level. This output data is sent over the wide area network (WAN) to the high density station core, through a unique bidirectional access point, which is indicated with a black square in Figure 1.3. In the components, operations on signals are supported by specialized modules that constitute the lower level of the hierarchy in the architecture.

The control and monitoring facility at the central core is a root for the hierarchical architecture. It governs the behavior of the stations, by sending control messages that transport requests to reconfigure processing tasks that are executed locally in all stations (messages may be different for different stations). In a station, control messages are received through the single access point, processed in that single access point, and then sent down to the level of components to re-configure the signal processing tasks. For example, different beamforming weights may be sent to two stations and applied when processing data in components separately in these two stations.

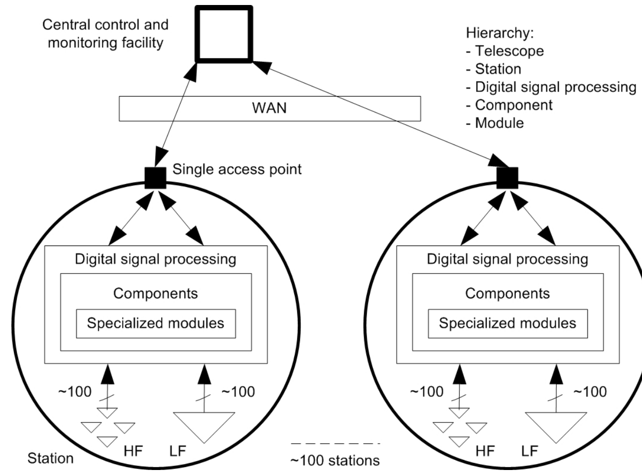


Figure 1.3: Each station processes high throughput data originating from HF and LF antennas, and communicates with the central control and monitoring facility through a single access point. Signals are processed in modules that are internal to components.

### 1.1.3 Hierarchical application

All stations operate in the same mode among a fixed and pre-defined number of modes, and can swap between these modes together at run-time.

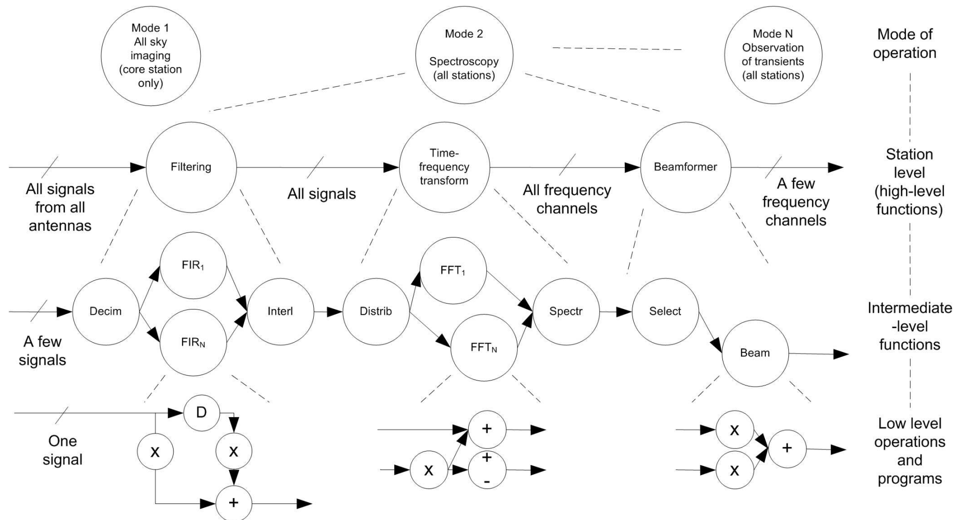


Figure 1.4: Example of levels of hierarchy for one mode of operation. High-level functions at station level consist of intermediate-level functions that are themselves compositions of low-level operations and program instructions.

In each operation mode, signals are processed in a chain of high-level functions such as filtering, time-frequency transformation and beamforming. See standard books for details (e.g., [14] [15]).

Figure 1.4 gives an example of a visual representation of the behavior of a station as a chain of high-level functions that operate on signals originating from all the antennas. High-level functions consist of a network of intermediate-level functions such as decimation and filtering functions, which operate on signals originating from a few antennas. Intermediate-level functions are themselves decomposed down to the level of basic operations and program instructions that operate on one signal. Signal processing functions are parameterized, and parameter values can be re-configured at run-time by sending messages in the control and monitoring network. For example, a filter may be parameterized in terms of number and values of coefficients, such that different filter characteristics (low-pass, high-pass, etc) can be applied by re-configuring parameter values.

## 1.2 Problem statement

In this section we first give a general problem context when aiming at structuring the design process from system-level specification to implementation-level specification for large scale and distributed digital signal processing systems. Then we focus on our specific problem statement, and we give a visual representation of the general problem and specific problem.

### 1.2.1 General problem context

Our general problem context is to convert system-level specifications in a structured way to an implementation-level specification, such that decisions taking is an unambiguous process [3]. This is a problem because 1) the systems we consider are large scale and distributed, and 2) an application has to be portable across several architectures so as to benefit from advanced technologies and improve the performance/cost of the system.

Scale is a problem since it is not clear a priori how to specify the application, the architecture and the association between the two, and how to scale designs starting from system-level specifications. In particular, the association between the application and the architecture can be simulated very accurately on lower levels of the hierarchy for local (sub-)systems, therefore leading to actual performance and cost numbers. However, when scaling the number of subsystems, simulating the complete system with the same accuracy as in lower hierarchical levels is not a realistic option anymore since it would be too demanding in terms of processing power, and too time-consuming. Thus, we need methods to master the scale and complexity of the systems we consider, and to take design decisions in a structured way on all hierarchical levels.

Implementing an application in several architectures is a problem because these architectures consist of heterogeneous components for processing, storage and communication on all levels

[3] We want to avoid taking decisions intuitively or based on experience of individuals.

of the hierarchy. Since we rely on commercially available compilation and synthesis tools to implement different parts of the system, our problem is to know how to structure the design process such that system-level specifications are converted to implementation-level specifications that are compatible with these tools.

### 1.2.2 Specific problem statement

Our particular problem is to interface and synchronize the signal processing part and the control and monitoring part when the two parts are first considered in isolation.

This is a problem because the two parts are differently structured and behave differently. In the systems we consider, the signal processing and control and monitoring parts have to be interfaced and synchronized such that 1) the functional behavior of the dominant signal processing part is not obstructed by the interfacing with the control and monitoring part, and 2) the system can be scaled without altering the performance/cost of the two parts.

When it comes to real implementations, some completion logic (or glue logic [4]) is necessary to interface subsystems that were developed separately for the two parts early in the development phase. Time-consuming and error-prone handcrafted ad-hoc glue logic development must be avoided so as to facilitate implementations.

#### Visual representation of the problem

We give a visual representation of the general problem (structuring the design process from system-level specifications to implementation-level specifications) and specific problem (separating the signal processing part and control and monitoring part before interfacing and synchronizing the two) in Figure 1.5 for a simple system.

- In this example, the application (functional behavior) consists of three signal processing nodes ($P_1$, $P_2$ and $P_3$). These processing nodes can operate in two modes that are specified in the control and monitoring nodes $C_1$ and $C_2$. The particular problem is to specify the behavior of the signal processing part separately from the behavior of the control and monitoring part, and to interface and synchronize the two parts without obstructing the behavior of the dominant signal processing part.
- The architecture (non-functional behavior) is a composition of processing units ($PU$) that communicate through communication, synchronization and storage infrastructures ($CSSI$). The particular problem is to specify the most appropriate composition of components separately for the signal processing part and for the control and monitoring part, and to interface the two compositions without significantly altering their individual performance/cost when scaling the system.
- The *mapping* of the application on the architecture consists of associating nodes with components. For example, $P_1$ may have to be mapped onto $PU_3$ (this is represented

[4] Glue logic deals with low-level communication protocols and signals between components.

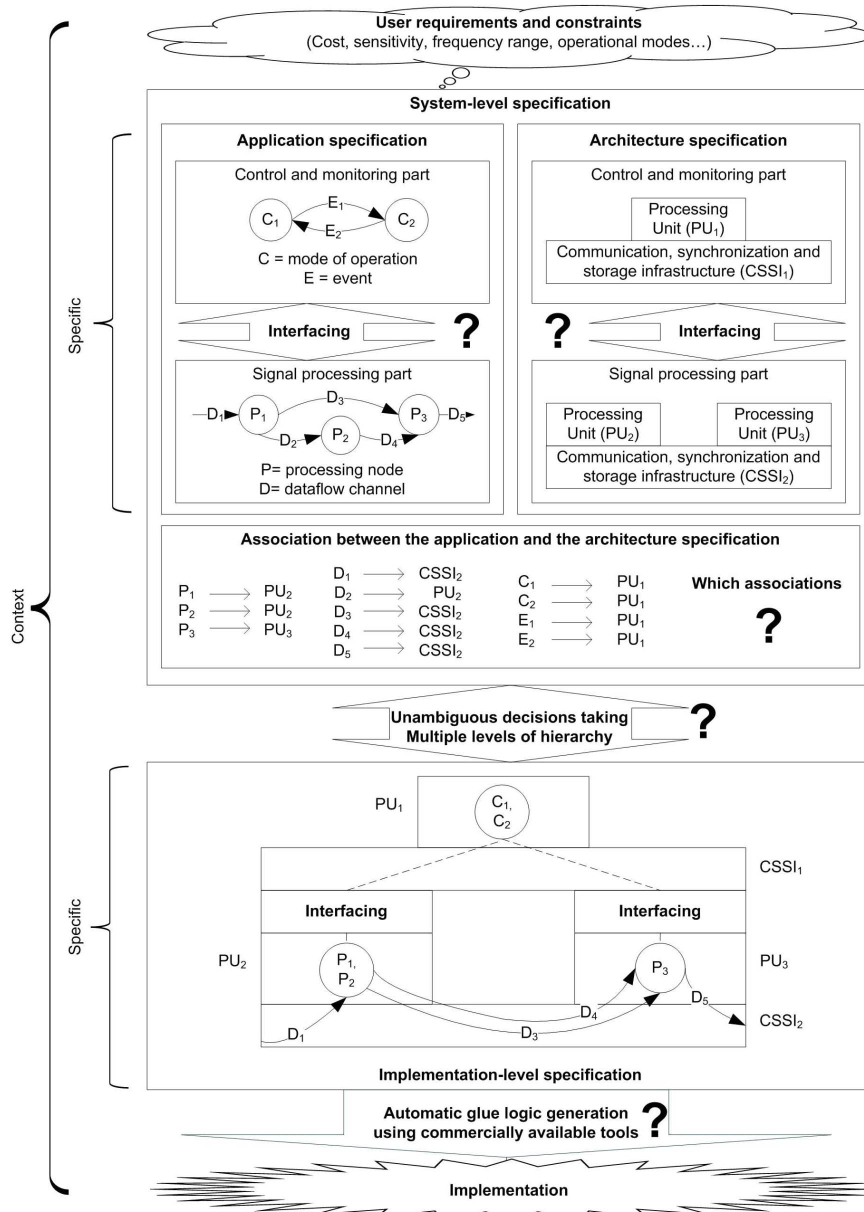


Figure 1.5: Problem context: how do we derive system-level specifications, and how do we convert them in a structured way to implementation-level specifications given that our systems are large scale and distributed? Specific problem: can we separate the signal processing and control/monitoring parts, which have different fundamental behaviors and structure, and interface them such that their individual (model) semantics are preserved?

with $P_1 \rightarrow PU_3$ in Figure 1.5). The general problem is to map the application onto the architecture on all levels of the hierarchy. The particular problem is to take decisions in a consistent way at all levels of the hierarchy when mapping the interfacing between the signal processing part and the control and monitoring part.

- The last general problem is to make sure that the initial system-level specification, which is independent of any implementation tool, can be converted to an implementation-level specification, which commercially available compilation and synthesis tools should be able to convert automatically to a real implementation. In particular, these tools should automatically generate glue logic and lead to the expected behavior and performance.

## 1.3 Solution approach

In this section, we first give the approach to the general problem of structuring the design process from system-level specification to implementation-level specification, which has been successfully applied to the signal processing part of the system [16]. Then we give our approach to the specific problem of separating signal processing and control/monitoring parts, and the property preserving interfacing and synchronization of the two parts.

### 1.3.1 Approach to the general problem

To master the complexity of the large scale and distributed systems we consider, we have to raise the level of abstraction. This implies that we need unambiguous models, such that we can take design decisions based on models rather than on intuition or ad-hoc details in the actual systems. Because we have to derive specifications at system-level, we propose to express our specifications based on models. We propose to do that in a particular way, by adhering to the separation of concerns principle [17].

At system-level, we separate the model-based specification of the application from the model-based specification of the architecture, and we provide means to associate these two specifications together. Although we separate the application model from the architecture model, they roughly match in the sense that both are specific to our application domain: both rely on ”parallel” models. Nevertheless, we can take decisions about the application and the architecture models separately.

The application is specified in terms of a so-called Model of Computation (*MoC* [18] [5]) whose semantics capture unambiguously the way data is processed in processing ”nodes” and communicated between processing ”nodes”.

The architecture is specified in terms of interconnected components that are taken from a unique library, and that support specialized services for processing, storage, communication, etc. The library includes information on the performance (e.g., processing delays, power consumption, etc) and cost of the components, and rules to obey when connecting components.

[5] There are many MoCs. For now, we leave the choice of a particular MoC open.

Basic components that are used at lower levels of the hierarchy are modeled as white boxes, whose internal modules are accessible for simulation so as to obtain actual performance and cost numbers. At higher levels of the hierarchy, components are modeled as black boxes, whose internal modules are hidden. Black boxes relate output quantities to input quantities based on simple equations, where parameter values are calibrated using information obtained from lower levels of the hierarchy.

The main difference between the two models is that the application model is transformative, i.e., deals with functional behavior, whilst the architecture model is reactive, i.e., deals with non-functional behavior (timing, resources).

The association of application and architecture models together - called mapping - is based on iterative transformations that close the matching gap between the two separately defined models. In this thesis we assume that mapping transformations are available in a library, and that designers can select which transformation to apply during which iteration. Mapping transformations improve the model matching in terms of resolution of detail (in the nodes and communication between nodes on the one hand, and in the components for processing, storage and communication on the other hand).

After each mapping transformation, functional and non-functional behaviors can be co-analyzed. When transformations are applied at lower levels of the hierarchy, this analysis phase relies on simulation of modules that are internal to white boxes, or on prototype implementations. When transformations are applied at higher levels of the hierarchy, performance/cost information is obtained based on simple equations that relate output quantities to input quantities in black boxes. The result of the analysis phase is sent back to the application and architecture specifications, which can be updated based on this feedback before possibly applying another mapping transformation.

Once the matching between the application and the architecture specifications is satisfactory and includes all the information that is necessary to go to an implementation, the corresponding specifications, which are still abstract, are converted to an implementation-level specification. During this translation, mapping transformations can still be applied interactively and locally to optimize the performance/cost of parts of the system, without modifying the input specification. The resulting implementation-level specification still includes rules to obey when composing components. These rules drive the generation of glue logic that should be automated by compilation and synthesis tools to integrate actual components, including HW/SW IP-components that are designed and owned by third parties.

With this approach, we strive for implementations that are correct by construction. This approach is neither purely top-down nor purely bottom-up. It is a meet in the middle design approach in the sense that the association of the model-based application and architecture specifications relies on information about the behavior, performance and cost that is obtained by simulating or prototyping actual components at lower levels of the hierarchy, and by using this information in formulae when moving up in the hierarchy.

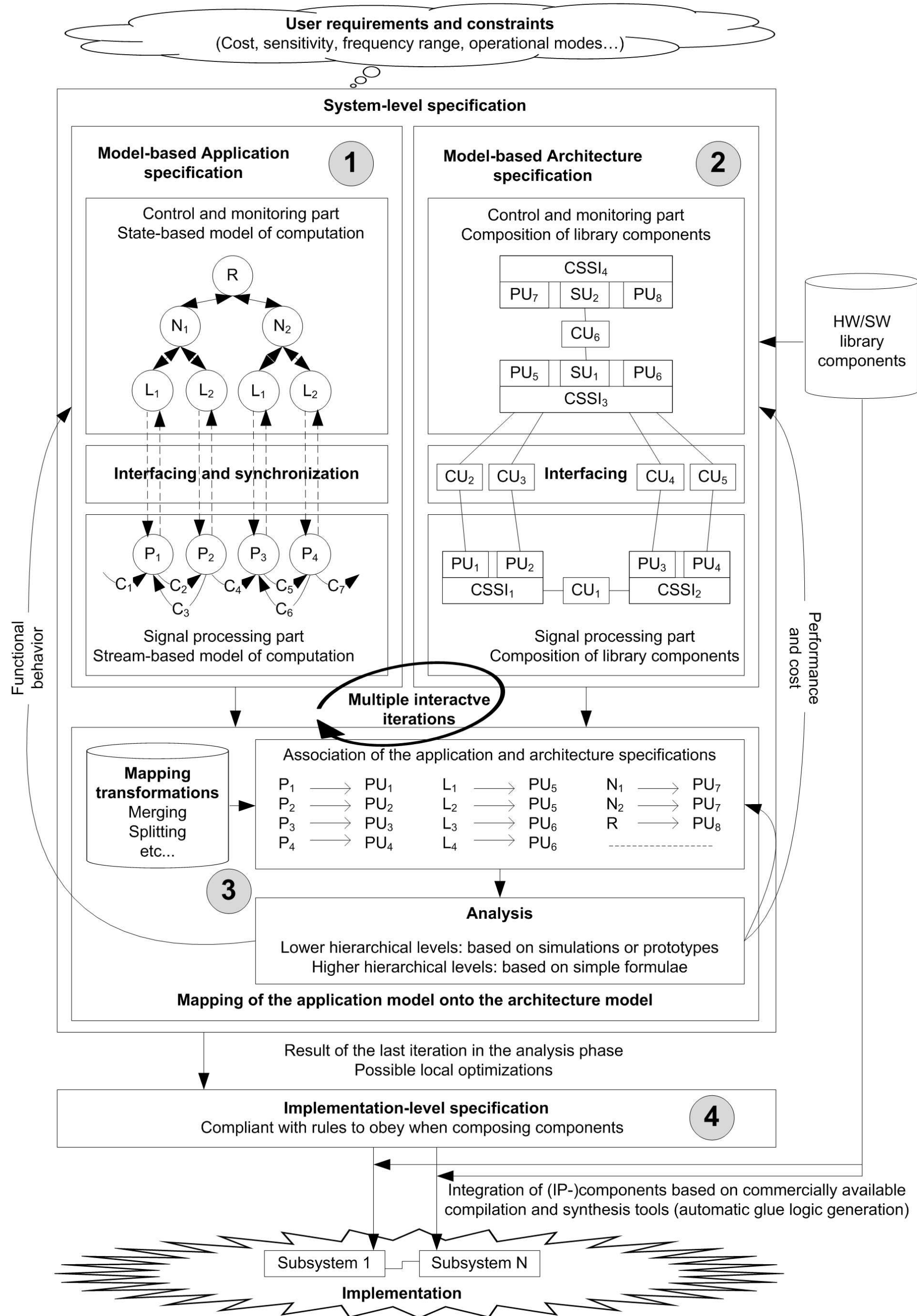


Figure 1.6: Model-based approach to structure the design process from system-level specification to implementation-level specification. The application is captured based on models of computation. The architecture is represented by composing hierarchical library components. The application is mapped onto the architecture based on iterative transformations. The analysis and back-annotation of the functional behavior and performance/cost of the system relies on information that is obtained by simulation of real components at lower levels of the hierarchy, after which analytical models are applied at higher levels of the hierarchy.

### 1.3.2 Approach to the specific problem

As shown in Figure 1.6, we separate the modeling of the signal processing and control/monitoring parts of the system, and we interface them in a way that the semantics of both models are preserved.

In the application (step 1 in Figure 1.6), the functional behavior of the signal processing part is specified based on a stream-based model of computation that is well suited to represent streaming applications that have a high degree of parallelism, and where tasks have a repetitive behavior. The functional behavior of the control and monitoring part is specified based on a state-based model of computation that is well suited to represent the execution of tasks in reaction to events. The synchronization problem is addressed by introducing a notion of time that is known only to the control and monitoring part, and by relating this notion of time to periodic intervals within which tasks are executed in the signal processing part.

In the architecture (step 2 in Figure 1.6), we compose library components separately in the signal processing part and in the control and monitoring part. The composition in the signal processing part sustains intensive computations on and transport of high throughput data with a high degree of parallelism. The composition in the control and monitoring part permits transferring sporadic messages and executing sequential tasks in reaction to these messages. The control and monitoring model has a tree-like structure, whose leave nodes are interfaced with the computational nodes in the signal processing model of computation.

In the mapping of the application onto the architecture (step 3 in Figure 1.6), mapping transformations that can be chosen by designers at each iteration are restricted by the interfacing between the signal processing part and the control and monitoring part. For example, a node in the signal processing part may be split only if there exists a complementary operation in the control and monitoring part, such that each node in the signal processing part is interfaced with a complementary leaf-node in the control and monitoring part, and such that the performance/cost prediction of the resulting system is satisfactory after the splitting transformation.

During the translation of the output of the analysis phase to an implementation-level specification (step 4 in Figure 1.6), mapping transformations (high-level compilation steps) can be applied locally and automatically to optimize the performance/cost of a part of the system, without modifying the input specification. When the specification is refined down to the level of (networks of) multiprocessor systems-on-chip, each part is implemented based on appropriate compilation and/or synthesis tools, which integrate (IP-)components and generate glue logic according to rules that are defined in the library in order to connect (IP-)components with each other.

## 1.4 Research contributions

Recall that we do not consider the specification and design of front-end and back-end of the system. Our focus is on the intermediate digital data-reduction part, i.e., the system's stations. The path from requirements and constraints to specification has already been addressed for the dominant signal processing part of such systems in [16]. However, the inherent con-

trol and monitoring part has been left out. In this thesis, we adopt the decision that has already been taken to specify application, architecture, and mapping separately at system level, and we start from an abstract (model-based) system-level specification, which has somehow been derived from requirements [6]. We address the path from system-level specification to implementation-level specification. We consider that commercially available compilation and synthesis tools can convert implementation-level specifications to real implementations [7]. We focus on the issue of modeling the signal processing and control/monitoring parts separately, and the interfacing of these models.

We bring together three parts of the design specification, namely the model-based application specification, the model-based architecture specification, and the mapping of the former onto the latter based on transformations. Our contribution is to relate these parts in the context of large-scale and distributed digital signal processing systems, and to separate the modeling of the signal processing part from the modeling of the control and monitoring part. In particular, we give restrictions for the interfacing and synchronization of the signal processing part and control and monitoring part.

- The synchronization problem is addressed in the modeling of the application by introducing a time model that is known only to the control and monitoring part, and by relating this time model to periodic intervals within which tasks are executed in the signal processing part.
- The interfacing problem is addressed in the modeling of the architecture by using dedicated library components to link processing units at the lowest hierarchical levels of the two parts based on a unique design pattern. With this domain-specific approach, the interfacing remains uniform when scaling the system.
- On all levels of the hierarchy, the mapping of the application onto the architecture is restricted by the interfacing between the signal processing part and the control and monitoring part.

## 1.5 Related work

The problem of designing (robust) control for large-scale systems has received significant attention over the past few decades [19]. For example, a mathematical approach has been proposed in [20] to produce decentralized control structures that are suitable for a large scale multiprocessor system. This approach is based on linear matrix inequalities (LMI) and allows to minimize inter-processor communication for a moderate amount of computation. Although such mathematical frameworks cover the control part of large-scale systems, they do not address the specific problem of the interfacing with a dominant signal processing part that is also a multiprocessor system as in the systems we consider.

[6] In this thesis, we do not deal with the path from user requirements and constraints to high-level system specifications.

[7] We evaluate the capacity to go from implementation-level specification to real multiprocessor system-on-chip implementation (for which tools do exist) in chapter 5.

The CoSMIC tool [21] supports the Model Driven Architecture [22] approach, which separates the application from the underlying technology before associating the two based on mapping transformations. CoSMIC can be used to configure and assemble component middleware required to deploy distributed and real-time embedded applications, and focuses on Quality of Service (QoS) issues. In our approach, we take the view that an implementation can be obtained in a straightforward way based on commercially available tools starting from an abstract implementation-level specification. Moreover, the systems we consider do not rely only on instruction set architecture (ISA) components, but also involve more specialized components that are customized to operate with a high degree of parallelism without any potentially overwhelming operating system.

In neutrino telescopes such as the deep see Antares [23] or the Antarctic ice shield IceCube [24], detectors have to cover a volume on the order of 1 $km^3$ or more to detect neutrinos with statistical significance. A methodology for designing and evaluating high-speed data acquisition (sub)systems in such telescopes is presented in [25]. This methodology unifies the specification of the data-driven application and the multi-processor architecture based on models of computation in the Ptolemy II [26] framework. However, this methodology does not study the control flow and the interfacing with the dataflow.

In the latest Large Hadron Collider in CERN, a network is required to transmit data from the detector front ends to the 1800 computer farm where trigger algorithms run. This network must be robust, have low latency, maximize data throughput, control data flow and minimize errors. In [27], two complementary approaches are used to simulate networks that are candidates and to evaluate their performance. The first approach consists of simulating the hardware performance of a network component at a level of detail where switch fabric and logic are complex or unavailable and where parametrization is impossible. The data obtained may be input to the second approach, which relies on software simulation based on parametrization of data queuing, packing and switching, and which allows to reason about the performance at full scale. However, this approach assumes that the mapping of the application that runs in the candidate networks is given.

In [16], an approach is presented to specify large scale array signal processing systems and to explore the performance and cost of these systems. This approach, which is based on the separate modeling of the application and architecture before mapping the application onto the architecture, has been applied to the specific case of large scale radio telescopes, and in particular during the preliminary design phase of the LOFAR radio telescope. However, the control and monitoring part has not been considered separately from the signal processing part. This separate modeling and interfacing of these two parts is one of our contributions.

The Berkeley Emulation Engine (BEE2 [28]) is a scalable FPGA-based computing platform with a software design methodology. This platform targets a wide range of high-performance applications, including real-time radio telescope signal processing [29] [30]. The architecture consists of four FPGAs that are connected in a ring topology and that are all interfaced with another FPGA that is dedicated to control. This regular architecture can be duplicated and interfaced based on standard protocols so as to rapidly scale systems to thousands of FPGAs. Customized hardware and software library components are available to implement control functionalities, signal processing functionalities, and their interfacing starting from specifications in the Matlab/simulink environment. However, these specifications are limited

to the SDF model, and the mapping is limited to a manual assignment of library (functional) components to FPGAs.

In Thales [31], radar and sonar applications are specified using nested loop algorithms and are mapped onto large scale array signal processing systems. The architecture model may have different levels of hierarchy. Loops are transformed so as to extract their cores, which are supposed available from libraries of functions, with each function having possibly several optimized implementations for different target processors. On the lowest level of abstraction, mapping is expected to define the role of each actor of the architecture so that appropriate compilers can produce executable code and glue components together automatically. Higher levels represent virtual components which are composite blocks including a local multi-components architecture. Mapping is human-driven. Commands are proposed to the user for application partitioning and allocation, insertion of communications, fusion of tasks and scheduling.

The Metropolis framework [32] offers syntactic and semantic mechanisms to store and communicate all the relevant design information, and it can be used to plug in the required algorithms for a given application domain or design flow. Its semantics can be shared across different models of computation and different layers of abstraction. Architectures are represented as computation and communication services to the functionality. Metropolis can analyze statically and dynamically functional designs with models that have no notion of physical quantities, and mapped designs where the association of functionality to architectural services allows to evaluate the characteristics (such as latency, throughput, power etc) of an implementation of a particular functionality with a particular platform instance. During the mapping, synchronization constraints are added to force the functional model to inherit the concurrency and latency defined by the architecture while forcing the architectural model to inherit the sequence of calls specified for each functional process [33]. In this manner, mapping eliminates some of the nondeterminism present in the functional and architectural models by intersecting their possible behaviors. After mapping, a mapped implementation is created. The Metropolis framework makes it possible to incorporate external tools that can take a mapped implementation as an input, and thus addresses the problem of design chain integration by providing a common semantic infrastructure. The next generation Metro-II framework will enhance three key features, namely heterogeneous IP import, orthogonalization of performance from behavior, and design space exploration [34]. These issues are also addressed in this thesis for the particular case of large scale and distributed signal processing systems.

## 1.6 Thesis outline

In Chapter 2, we present the approach to model the functional behavior of the systems we consider. We select two models of computation, namely communicating Kahn process networks (*KPN* [35]) and communicating finite state machines, to specify separately the functional behavior of the signal processing part and the functional behavior of the control and monitoring part, respectively. We give the approach to synchronize these two models based on relations between a notion of time that is known to the control and monitoring network only, and peri-

odic intervals within which signal processing tasks are executed. We illustrate the functional behavior of the interfacing with examples of synchronization and re-configuration.

In Chapter 3, we present the approach to specify the non-functional behavior and related performance and cost of the systems. We specify the signal processing architecture, the control and monitoring architecture, and the interfacing between the two, in terms of interconnected components from a unique library. At lower levels of the hierarchy, components have a white box appearance, and their performance/cost is obtained by simulating the dynamic behavior of their internal modules. At higher levels of the hierarchy, components have a black box appearance, and their performance/cost is obtained based on simple equations that relate output quantities to input quantities, and whose parameter values are calibrated with numbers obtained at lower levels. The signal processing architecture model supports intensive computations and transport of high throughput data. The control and monitoring architecture model supports the transport of control messages that trigger the execution of sequences of operations in reaction to sporadic events. At the lowest hierarchical levels of the two parts, processing units are interfaced based on a unique design pattern that uses dedicated library components.

In Chapter 4, we present the approach to associate the application and architecture specifications together. We present the mapping transformations we need to improve the matching between the application and the architecture both in terms of performance and cost, and in terms of granularity of items. We assume these transformations are available in a library, such that they can be called iteratively. After each mapping transformation, the functional behavior and performance/cost of the system are analyzed, and decisions can be taken based on the result of the analysis. Mapping transformations are constrained by the interfacing between the signal processing part and the control and monitoring part. From an implementation point of view, mapping transformations are compilation steps above implementation-level specifications, which is considered to be the level of abstraction from where implementation becomes well established based on standard compilation and synthesis tools.

When mapping large and high-throughput signal processing applications onto multi-processor architectures, parts of these applications are assigned to re-configurable components. Automating such mappings without delving deep into details implies the (re-)use of IP components both in the signal processing part and in the control and monitoring part. In Chapter 5 we present case studies around the integration and porting of IP-components starting from high-level specifications, and around the interfacing between components in the signal processing part and components in the control and monitoring part based on glue logic. These case studies reveal the weakness of otherwise highly desirable system-level design methods when evaluated with respect to fast, accurate, and systematic IP integration.

Chapter 2

# Application specification

## 2.1 Summary

In this chapter, we specify the functional behavior of applications that run in large-scale and distributed digital signal processing systems that maintain a permanent interaction with their environment, such as stations in phased array radio telescopes. These applications include a signal processing part, and a control and monitoring part. We specify the functional behavior of these two parts separately, based on models of computation, before we interface the two models. Model-based specifications are unambiguous and permit structuring the design process from system-level specifications [1] to implementation-level specifications [2]. We use the operational semantics of Kahn Process Networks (KPN) and communicating Finite State Machines to specify the way data is simultaneously computed and communicated in the signal processing part and in the control and monitoring part, respectively. The synchronization between the two parts is based on relations between a time model that is known only to the control and monitoring part, and periodic intervals within which tasks are executed in the signal processing part, such that the interfacing does not alter the behavior of the dominant signal processing part. We give examples of synchronization, (re)configuration and monitoring, and discuss limitations that result from the synchronization method.

[1] Remember that a system-level specification consists of an application specification (the scope of this chapter), an architecture specification, and the mapping of the former onto the latter.

[2] Remember that we consider implementation-level specifications as the level of abstraction from where commercially available compilation and synthesis tools should take over to obtain a real implementation.

## 2.2 Introduction

The functional behavior of an application is typically captured using specification-level models of computation [36], which represent computation in nodes that communicate in a well-defined way through channels in a network. The way data is simultaneously processed in nodes and communicated over channels in the network is specified using the formal semantics of a model. Modeling of applications is an appealing way to get unambiguous specifications to structure the design process from specification to implementation. Model-based specifications hide overwhelming details of the actual system and allow to predict the functional behavior of the system by reasoning about the models. Semantics can be denotational, operational or axiomatic, depending on the desired level of formalism. Denotational semantics is an approach to formalizing the semantics of a (computer) system by constructing mathematical objects (called denotations or meanings) which express the semantics of the system. Operational semantics is a way to give a meaning to computer programs in a mathematically rigorous way. It describes unambiguously how a valid program is interpreted as a sequence of computational steps. These sequences then become the meaning of the program. Axiomatic semantics is an approach based on mathematical logic to prove the correctness of computer programs.

Applications that will run in future large-scale and distributed embedded signal processing systems such as the SKA radio telescope [2] will include both a signal processing part, and a control and monitoring part. These digital systems are reactive in the sense that the two parts maintain a permanent interaction with their environment [37]: the control and monitoring part reacts to sporadic events, while the signal processing part reacts to input data streams by transforming them to output data streams. The behavior of the interfacing between the two parts must be judiciously and unambiguously specified so as to avoid obstructing the behavior of the dominant signal processing part. In theory this can be done based on the formal semantics of models of computation. From a separation of concerns viewpoint we are interested in specifying the functional behavior of the two parts using the most convenient models of computation.

In this chapter we use the operational semantics of two models of computation to specify the way data is simultaneously processed and communicated in both parts. A state-based model is used to specify the behavior of the control and monitoring part. A stream-based model is used to specify the behavior of the signal processing part. We reconcile the two parts by relating the notion of periodic execution cycles in the signal processing part to a notion of time that is superimposed to the control and monitoring part. This chapter is organized as follows. We first introduce some terminology to analyze operational semantics in section 2.3. Then we select a stream-based model to specify the behavior of the signal processing part in section 2.4, and a state-based model to specify the behavior of the control and monitoring part in section 2.5. We superimpose a notion of time to the control and monitoring part in section 2.6. Then, we give our contribution concerning the interfacing between the stream-based model and the state-based model in section 2.7 and we discuss limitations that result from the superimposed synchronization method. Finally we give our related work in section 2.8 and conclusions in section 2.9.

## 2.3 Terminology

In this section we introduce the terminology that is used in the remainder of the chapter to discuss the operational semantics of a few stream-based and state-based models of computation in terms of computation, communication, (a)synchrony, determinism, composition, etc. An overview of models of computation can be found in Appendix A.

#### Token, signal, clock

A *token* is an abstract aggregation of a value-tag pair [18]. The role of tags is to order tokens in a sequence of tokens called a *signal* (see Figure 2.1). This ordering relation may be total or partial. When the order of tokens is total, tags are also called *timestamps*.

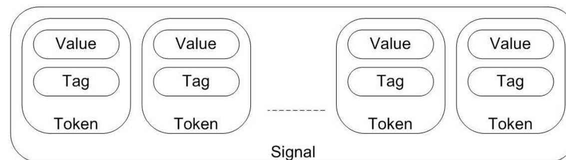


Figure 2.1: A signal is a set of tokens that are value-tag pairs. Tags order tokens.

#### Process

A *process* executes a sequential program, i.e., a sequence of reading actions, execution actions and writing actions. A process has at least one input port or one output port through which it exchanges tokens with other processes in a network. We say that a process reads (or consumes) tokens on input ports and writes (or produces) tokens on output ports. As shown in Figure 2.2, a process has a functional behavior and a token ordering behavior.

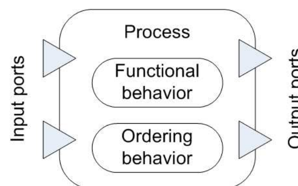


Figure 2.2: A process has input and output ports, a functional behavior and a token ordering behavior.

The functional behavior includes reading of tokens from input ports, executing one or more tokens mappings in a sequential order, and writing of tokens to output ports. The ordering behavior specifies the order in which input and output tokens are consumed and produced from and to input and output ports during the reading and writing phases, respectively, and assigns a tag to each token that is produced by the process.

The execution of a program in a process is synchronous when the sequential program waits for the actions to signal back end of execution. It is asynchronous when the sequential program continues its execution without waiting for the end-of-execution of the actions.

### Communication

Processes communicate in a network by transferring tokens through communication channels. This communication can be modeled as unicast, broadcast or multicast. A unicast (point-to-point) transfer involves a single producer and a single consumer. Broadcast and multicast transfers link a single producer to many receivers. A broadcast transfer sends a token to all processes in the network, whereas a multicast transfer delivers a token to a group of processes in a network as depicted in Figure 2.3.

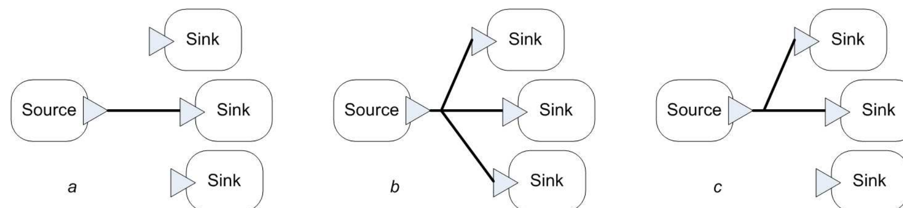


Figure 2.3: Communication between processes in a network. *a*: unicast, *b*: broadcast, *c*: multicast.

Processes may exchange tokens over channels in a network in a synchronous or asynchronous way. Synchronous communication occurs when all processes involved in a communication are present at the same time. There is no intermediate storage between the communicating processes.

The communication is asynchronous when at least one of the processes involved in a communication is not available for the communication, or when an arbitrary amount of time elapses between the desire of communication and the actual communication. There is an intermediate buffer such as a single-place buffer or an (un-)bounded FIFO, etc (see Appendix A). A producer process sends a message to a buffer, and the consumer process can take the message from there - now or later. This communication can be lossless or lossy. To guarantee a lossless communication, a form of 'synchronization' is needed, for example by means of blocking writes and reads. This additional synchronization does not mean that the model has become synchronous. It can become synchronous when it is guaranteed that the producer can send to the intermediate buffer without check, and that the consumer can receive from the buffer without check as in clocked synchronous circuits.

### Concurrency

Communication and concurrency are complementary notions: a process is either interacting with other processes by communicating with them, or it is processing independently of them.

Processing in a process may occur concurrently, simultaneously with the processing and communication in other processes [38]. Parallel and distributed are two examples of concurrent execution [36].

#### Determinism, causality

The behavior of a composition of communicating processes is *deterministic* if for a given input signal, the composition has exactly one behavior, independent of the chosen schedule. Causality relates input tokens (causes) chronologically to output tokens (effects). The tag in an input token can not be higher than the tag in the resulting output token.

#### Composition, abstraction, hierarchy

*Compositionality* is a desired property of any model of computation: it preserves the semantics when combining processes. However, not all models respect this property (see Appendix A).

Abstraction is inversely related to the resolution of detail. If there is much detail, or high resolution, the abstraction is said to be low. Levels of abstraction are hierarchically ordered. A specification at a given abstraction level is described in terms of a set of constituent items and their inter-relationships, which in turn can be de-composed into their constituent parts at lower levels of abstraction [39].

#### Network consistency

Consider a network that consists of a producer process $P$ and a consumer process $C$. $P$ sends tokens to $C$ through a communication channel $A$. Consistency means that the number of tokens written by $P$ in $A$ is equal to the number of tokens read by $C$ from $A$. Consistency may be violated when switching from one set of (valid) parameter values in the functional behavior (sequential program) of a process to another (valid) set at an arbitrary point in time. A pair of parameter $N$ and $M$ is a valid pair if 1) the values are within a predefined range of values $[Nmin, Nmax]$, $[Mmin, Mmax]$, and 2) they satisfy possible relation constraints (e.g., $M \geq N$). More information about parameters validity and network consistency issues can be found in [40].

## 2.4 Selection of a model to specify the behavior of the signal processing network

In this section we first give the main functional requirements and constraints in the signal processing part in the digital systems we consider. Then we select a stream-based model of computation that will be used to specify the behavior of this part of the system.

### 2.4.1 General requirements and constraints

The elementary function of a radio telescope is to collect signals of celestial sources and to transform these signals into images and spectra. The application is large, and consists of large pieces that have to be distributed, and that have to support a few main modes of operation ranging from all sky monitoring, to pulsar observation, to searches for transients. These large pieces have to communicate data streams, and they have to be so specified as to transform input data streams to output data streams [41]. Data streams must not transport control-related information. Instead, control information must come from the control and monitoring network, which imposes the mode of operation to the signal processing network.

Moreover, the applications we consider have to be deterministic, and the loss of data is not acceptable in any mode of operation. Also, it must be possible to refine the abstract large-scale application specification by decomposing it into smaller pieces at a lower level of abstraction (recall that levels of abstraction are hierarchically ordered).

The model used to specify the behavior of the signal processing part must be compositional, such that the properties of sub-systems are preserved when composing a large system. Due to the complexity of the systems we consider, processes have to run autonomously. Since the application has to operate on a continuous flow of data, each process must execute and repeat its main sequential program over and over again (non terminating process) on new input data. Moreover, it must be possible to refine the main sequential program that is executed in a process. Also, to preserve the deterministic behavior of the model, we do not allow interrupts. Communication channels have to be able to support the transferring of streaming data in a smooth way. Moreover, tokens must not be lost in the signal processing part.

### 2.4.2 Selection of the KPN model

Given our requirements, it seems natural that the KPN model should be chosen to specify the functional behavior of the signal processing part. Indeed, the Kahn Process Network (KPN [35]) model of computation specifies an application as a network of autonomous processes that run concurrently and execute sequentially. Kahn processes communicate through unbounded point-to-point FIFO channels. Kahn processes synchronize locally through a blocking and destructive read mechanism: a process that tries to read from a FIFO waits until a token is available on that FIFO. Since each FIFO is read and written by exactly two processes, the speed of the processes does not affect the sequence of tokens sent through the FIFOs. This property and the fact that each process is only affected by the input sequences show that Kahn Process Networks are deterministic [42]. The deterministic property of the KPN model is appealing to specify the signal processing part of our systems where each process in a KPN can be viewed as a composition of hierarchically lower processes that are governed by the same semantics.

The KPN model may be a bit too general to specify the behavior of our application. The model we use is closer to the *Dataflow Process Networks* model [43], which is a particular case of KPN. In Dataflow Process Networks, each process consists of repeated firings of a dataflow actor. By dividing processes into actor firings, which define how many tokens

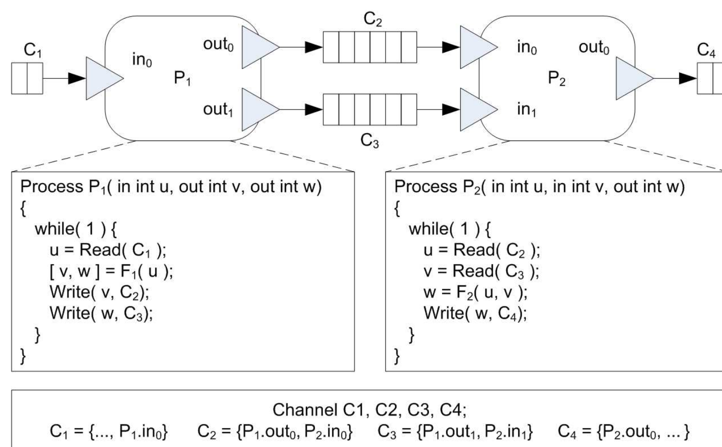


Figure 2.4: Example of a KPN with two processes, and corresponding sequential pseudo code using abstract instructions. Processes $P_1$ and $P_2$ execute the functions $F_1$ and $F_2$, respectively, repetitively. Communication channels are unbounded unidirectional FIFOs.

must be consumed and produced on their input and output ports, respectively, the potential overhead of context switching incurred in implementations of Kahn process networks is avoided. Smaller Dataflow Process Networks can be refined to the level of dataflow graphs that are even more specialized, where operations that are executed in actors are (mathematical) functions, and where actors are globally scheduled [43]. In particular, Dataflow Process Networks can be refined down to the level of the Synchronous Dataflow model (SDF [44], see Appendix A). The *SDF* model allows taking scheduling and buffer size decisions at compile-time that can generally not be taken when using the general KPN model.

Figure 2.4 shows a simple example of a KPN and the corresponding pseudo code for two processes that communicate through two channels. The bodies of the two processes consist of sequential abstract instructions. Communication is achieved with the $Read$ and $Write$ abstract instructions. The functions $F_1$ and $F_2$ are called after the $Read$ instruction and before the $Write$ instruction in the processes $P_1$ and $P_2$, respectively.

## 2.5 Selection of a model to specify the behavior of the control/monitoring network

The functional behavior of the control and monitoring network is fundamentally different from the functional behavior of the signal processing network. In this section we first give the main functional requirements and constraints in the control and monitoring network. Then we select a state-based model of computation to specify the behavior of this network.

### 2.5.1 General requirements and constraints

The control and monitoring application is also large and distributed. It must be possible to send events to any station from the central control/monitoring facility. This naturally suggests a tree-like structure in the control and monitoring network. Events have to be executed with strict timing constraints, in a deterministic way, and must not be lost in the control and monitoring network. In reaction to these events, parameter values have to be updated in nodes/actors in the signal processing network, without explicitly interrogating signals in the signal processing network. When interrogating signals, the control and monitoring part must do so independently on information contained in the signals themselves (the central control/monitoring facility is aware of signal characteristics in the signal processing part that operates in pre-defined modes).

Moreover, compositionality is required in the control and monitoring network. For example, the root node of the tree may be viewed as a leaf node that executes a single main sequential program to control and monitor the behavior of the complete signal processing network.

Procedures that are executed in control/monitoring nodes are not repetitive. Instead, the processing of events is time-triggered. Procedures have to terminate in a time period, and can start again on the occurrence of another event. Thus, interruptions are not required. Moreover, all nodes have to be synchronized at regular time intervals such that they can start operating in the same mode simultaneously.

When nodes communicate, events have to be transferred without postponing the transfer because the processing of events is time-triggered. Single-place buffers are required to avoid queuing events in communication channels. Also, protocols such as a handshake protocol must be available to avoid the loss of events. Events must be read in a destructive way since they have to be processed only once. Also, since the processing of events is terminating with strict timing constraints, the number of (sporadic) events that are communicated must be limited.

### 2.5.2 Selection of communicating state machines

The functional behavior of the control and monitoring network may be specified with a state-based event-triggered model. A detailed analysis of these models is given in Appendix A. *Statecharts* [45] reduce the visual complexity of traditional Finite State Machines (FSM) and support hierarchy, concurrency and abstraction. However, by allowing executing actions in both states and transitions, the effect of a transition may be contradictory to its cause. Moreover, this formalism is synchronous.

Process algebras (or process calculi) include Calculus of Communicating Systems (*CCS* [46]), Communicating Sequential Processes (*CSP* [47]) and Algebra of Communicating Processes (*ACP* [48]). CSP specifies applications as networks of sequential processes. In a communication, both a producer and a receiver are blocked until a token is transferred. A transfer takes place when the producer and the receiver are in the same state. This simple communication mechanism is also known as a rendezvous. In our case, neither the producer nor the consumer should be blocked during a transfer. Moreover, CSP has a non-deterministic behavior.

Co-design Finite State Machines (CFSMs [49]) are reactive finite state machines with data-paths communicating through single-place buffers. Communication in a *CFSM* network is asynchronous. A transmitter sends data without waiting for the receiver to be ready [42] and may overwrite a token in a single-place buffer. Moreover, the CFSM model supports interrupts, and is non-deterministic.

To specify the behavior of the control and monitoring network, we introduce timed-Communicating State Machines that communicate through single-place buffers, with a handshake mechanism to avoid loss of events. We use a a clock network [3] (called timing network) because the ultimate task is to make sure that processes in the signal processing network behave not only in the way that is required, but also at time that is required.

## 2.6 Superimposing a timing network for the synchronization

In this section we first specify the behavior of the *timing network* that is superimposed to the control and monitoring network. Then we give a representation of the processes in the control and monitoring network, and we specify the timed behavior of this network. We illustrate this behavior with a simple example and we discuss the limitations resulting from the superimposing of the timing network.

### 2.6.1 Superimposing pulse trains

We assume that the high-speed clock that synchronizes the data acquisition from the antennas in the stations has a period $T_S$, which we will call $time\ unit$ in the remainder of the chapter. The timing network sends the time unit $T_S$ to modulo counters in the nodes of the control and monitoring network as depicted in Figure 2.5. Modulo counters generate equidistant pulses. Every node in the control and monitoring network thus encompasses a dedicated pulse train. Each pulse increments the node's clock.

Recall that the control and monitoring network has a tree structure. The node that is the root for the tree is called $Root\text{-}Node$. Nodes in the lowest level are called $Leaf\text{-}\ Node$, and are the only nodes that are interfaced with the processes in the signal processing network. Nodes that are intermediate to the root node and the leave nodes are called $Intermediate\text{-}Node$.

Thus a leaf node $LN$ is interfaced with a process $P$ in the signal processing network. Pulses in a leaf node pulse train are $T_{LN}$ time units ($T_S$) apart, where $T_{LN}$ is an integer. We require that the processing of the $while(1)$ body in a process $P$ in the signal processing network falls well within the period $T_{LN}$ . The start of each interval $T_{LN}$ coincides with the occurrence of a pulse that is generated by the modulo counter in the leaf node.

After a synchronization procedure (which is detailed later on), all leave node clocks have a common starting point, $t$=0, which is a *global synchronization point*. Global synchronization

[3] A clock distribution is practically feasible.

points occur periodically with period $TG$. An example is shown in Figure 2.6 for the system modeled in Figure 2.5.

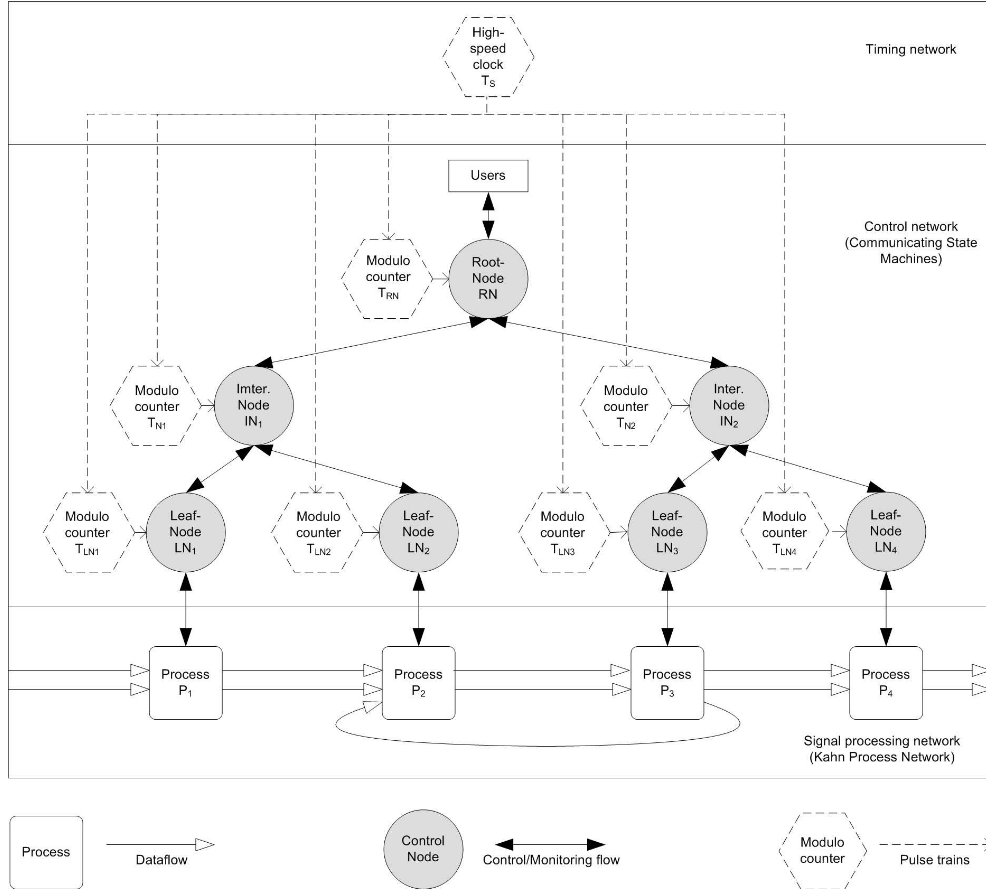


Figure 2.5: Superimposing of a timing network on top of a tree-like control and monitoring network. A high-speed clock $T_S$ (time unit) is distributed to modulo counters in the network nodes that increment their internal clock.

The *multiplicity* $M$ of a leaf node is the number of periods $T_{LN}$ that fall in between two adjacent global synchronization points. This interval between two global synchronization points is assigned a value $TG$ that is the least common multiple ($lcm$) of all multiplicities $M_i$ of all leave nodes as given in Equation 2.1:

$$TG = lcm(M_1, M_2, ..., M_n).T_S \tag{2.1}$$

The period $T_N$ of a pulse train that goes with an intermediate node or root node that is hierarchically higher than the leave nodes or intermediate nodes *j* to *k*, respectively, can be computed with the greatest common divider function ($gcd$) as given in Equation 2.2:

$$T_N = \frac{TG}{gcd(T_{N_j}, ..., T_{N_k})} \quad (2.2)$$

When a leaf node, intermediate node or root node receives a pulse, it increments a local notion of time with an increment corresponding to its own period. With this approach, all nodes have a local notion of time between global synchronization points, and all nodes are synchronized at global synchronization points. The notion of time is used to timestamp the execution of procedures in nodes in the control and monitoring network.

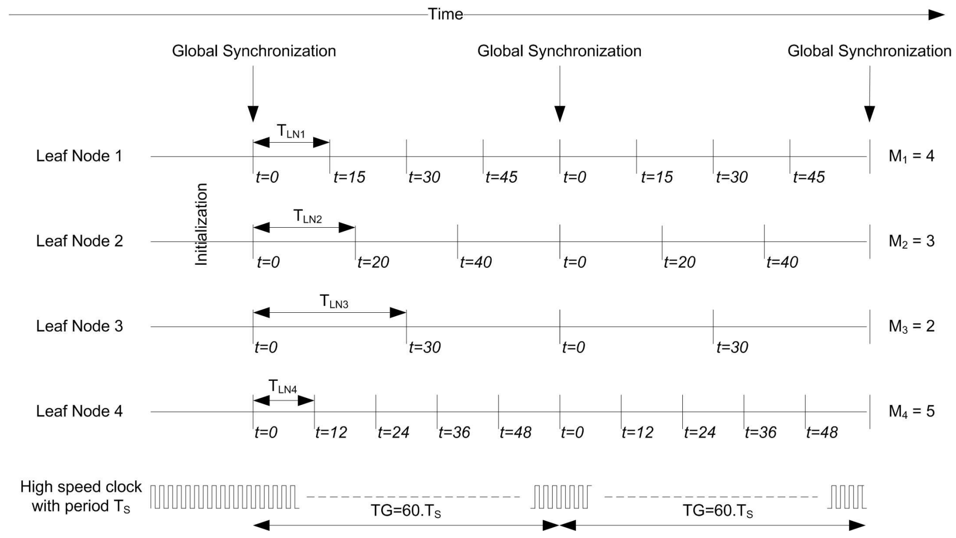


Figure 2.6: Relation between periodic pulse trains in leave nodes in the control and monitoring network shown in Figure 2.5. We assume that the execution cycle of a process $P$ in the signal processing network falls well within the pulse period $T_{LN}$ of the leaf node that is interfaced to $P$.

**Example 1**

Suppose that the system shown in Figure 2.5 has already been synchronized and that each leaf node is attached to a process such that $M_1$=4, $M_2$=3, $M_3$=2, and $M_4$=5 as depicted in Figure 2.6. This leads to:

$TG = lcm(M_1, M_2, M_3, M_4).T_S = lcm(4, 3, 2, 5).T_S = 60.T_S$.

Thus, by dividing the time interval that separates two global synchronization points into 60 segments, it is possible to access the beginning of any cycle in any control node. The periods of the leave nodes may be expressed in function of the time unit $T_S$ as follows:

$T_{LN_1} = \frac{TG}{M_1} = \frac{60.T_S}{4} = 15.T_S$ , $T_{LN_2} = 20.T_S$, $T_{LN_3} = 30.T_S$ and $T_{LN_4} = 12.T_S$

A pulse is produced in leaf node $LN_1$ every 15 time units ($T_S$). On the occurrence of such a pulse, $LN_1$ adds 15 to its local notion of time. Concurrently, all nodes in the control network increment their own counter with an offset that corresponds to their own period. The periods of the pulses that go with the two intermediate nodes $N_1$ and $N_2$ can be computed based on these numbers ($N_1$ is connected with $LN_1$ and $LN_2$, whereas $N_2$ is connected with $LN_3$ and $LN_4$):

$T_{N_1} = \frac{TG}{gcd(T_{LN_1},T_{LN_2})} = \frac{60.T_S}{gcd(15,20)} = \frac{60.T_S}{5} = 12.T_S$ and $T_{N_2} = 10.T_S$

Finally, since the root node is connected with the two intermediate nodes $N_1$ and $N_1$, the period of the pulses that go with the root node is obtained as follows:

$T_{RN} = \frac{TG}{gcd(T_{N_1},T_{N_2})} = \frac{60.T_S}{gcd(12,10)} = \frac{60.T_S}{2} = 30.T_S$

### 2.6.2 Utilization of the notion of time

In this subsection we specify the behavior of the timed control and monitoring network. We first give the structure of the packets (tokens) that are exchanged between nodes. Then we specify the functional behavior of the nodes. We detail the communication among nodes using timestamps, and we discuss the limitations that result from the superimposing of the timing network.

#### Control packets

In the control and monitoring network, all nodes have a unique identifier such that they can be addressed individually. The root node governs the execution of procedures in intermediate nodes and leave nodes. Procedures specify the behavior of the system in all possible operation modes. The root node may start the procedures by itself or upon request from users [50]. Intermediate nodes and leave nodes also send information back to nodes above in the control and monitoring network. The exchange of information between nodes is based on tokens that are called control packets. As shown in Figure 2.7, *control packets* consist of two parts: 1) a header, and 2) control data information.

The header contains four elements:

- A destination ($ID$), which identifies the destination node(s).
- A command ($CO$), which requests executing a control procedure.
- A timestamp ($TM$), which indicates the time at which the procedure needs to be executed with respect to the time unit $T_S$.
- A priority ($PR$), which indicates an execution priority order given a timestamp.

Control data information ($D_1, ..., D_N$) contains either parameters that come together with the command that is in the header of the same packet, or monitored data.

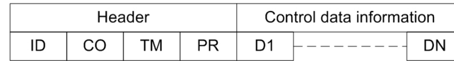


Figure 2.7: Visual representation of a control packet

### Intermediate nodes

To enforce the tree structure in the control and monitoring network, each *intermediate node* has a single port to its parent node. It is also interfaced with one or more child nodes. It reads and writes control packets from and to single-place buffers. Moreover, each intermediate node receives pulses from the timing network through a dedicated port. The pseudo-code corresponding to the execution of an intermediate node is given on the right-hand side in Figure 2.8. An intermediate node executes high-level abstract $Read$, $Execute$ and $Write$ instructions. The $Execute$ instruction is supported by three lower-level instructions: $Switch$, $Queue\text{-}and\text{-}Order$ and $Procedure\text{-}Repertoire$. We give a visual representation of an intermediate node and the corresponding pseudo code in Figure 2.8.

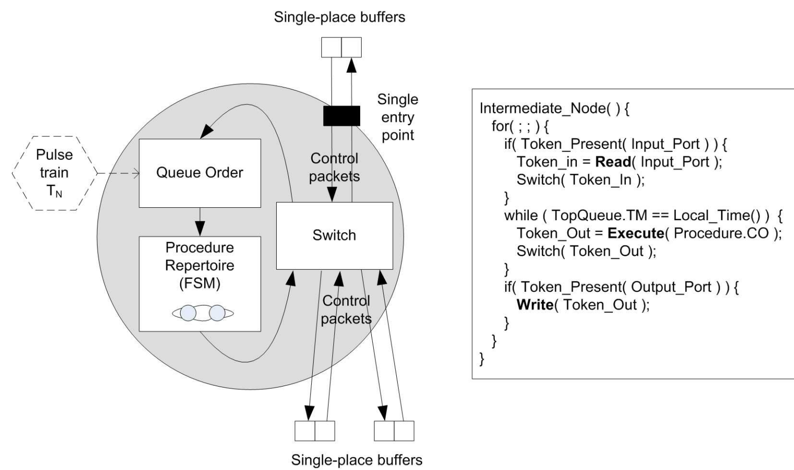


Figure 2.8: Visual representation and pseudo code of an intermediate node.

The abstract Read instruction reads a single-place buffer in a destructive way, with a handshake protocol: when a packet is read from a single-place buffer, the node sends an acknowledge signal back to the node (parent or child) that wrote the control packet in that buffer.

When a control packet is received in an intermediate node, it is first processed by the Switch instruction. If the ID in the packet does not match the identifier of the node, then the packet is forwarded to its destination, else it is processed as follows. On the occurrence of the next pulse, the node increments its local notion of time. Then the Queue-and-Order instruction orders packets with respect to the timestamps and priorities, and converts the commands obtained from all packets with identical timestamp into a single macro-command. Thus, a macro-command corresponds to a sequence of commands that have to be executed in the same period.

Before writing a token in a single-place buffer, the abstract Write instruction checks if that buffer is empty (i.e., it checks if it received an acknowledge signal from the node it communicates with through that buffer). With this simple *handshake* mechanism, packets can not be lost in the control and monitoring network since they can not be overwritten in single-place buffers.

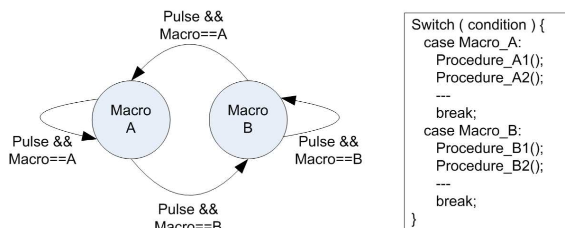


Figure 2.9: Example of FSM in a Procedure Repertoire. Each state corresponds to the execution of a macro-command (a sequence of commands).

The macro-command whose timestamp matches the local notion of time is processed by the Procedure-Repertoire instruction, which executes an FSM. Each state in the FSM corresponds to the execution of a particular macro-command. Procedures use parameters provided in the control data information, and generate packets that are sent to child or parent nodes in the control and monitoring network. Figure 2.9 shows an example of a FSM that consists of two states ($MacroA$ and $MacroB$) and the corresponding sequential pseudo-code. Transitions from one state to the other are triggered by the pulse and by the presence of a macro-command to be executed during the next period.

### Leave nodes

In contrast with an intermediate node, a *leaf node* has only two ports to below: a command/parameters output port and a monitoring-data input port. A leaf node is interfaced with a process in the signal processing network through two single-place buffers: a command/parameters buffer, and a monitoring-data buffer.

A leaf node executes abstract $Read$, $Execute$ and $Write$ instructions, which are supported by three lower-level $Switch$, $Queue\text{-}and\text{-}Order$ and $Procedure\text{-}Repertoire$ instructions. A visual representation of a leaf node is shown in Figure 2.10.

The processing of control packets is done as in an intermediate node. A procedure that is executed in the Procedure-Repertoire (FSM) sends a command/parameters token per period, possibly empty. On the occurrence of the next pulse, the leaf node writes this token to the command/parameters single-place buffer. This token does not include a header since it will be read and processed during the next cycle in the process that is attached to the leaf node.

A leaf node also reads a monitoring-data token per period, from the monitoring-data single-place buffer. This token is first assigned a header by the Switch instruction. This header includes the leaf node identifier, the local notion of time, and a monitoring command. Then, this packet is processed as any other packet. Monitoring procedures are specified in the

Procedure-Repertoire (FSM) and typically check the state of a process in the signal processing network. Depending on the outcome of this verification, a leaf node may send control packets to intermediate nodes above in the control and monitoring network. Such packets will request executing higher-level control procedures.

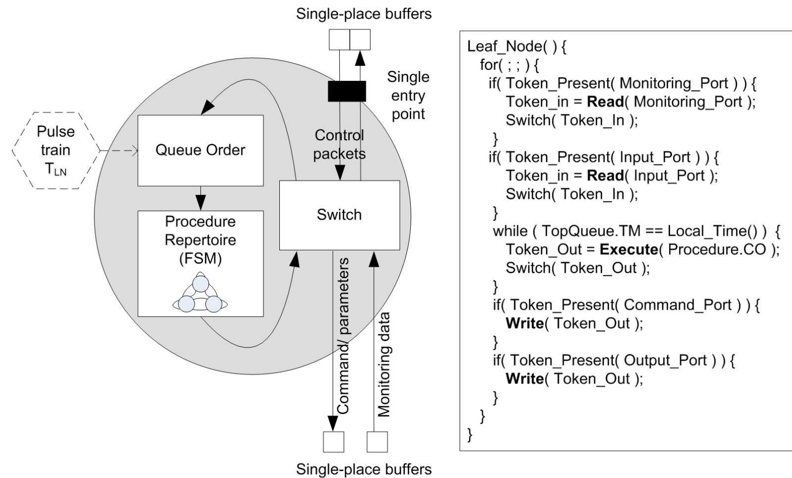


Figure 2.10: Visual representation and pseudo code of a leaf node.

### Communication in the control and monitoring network

Nodes in the control and monitoring network communicate among themselves by exchanging control packets. Each packet carries a command that is to be processed in a FSM in a node at a particular time.

When generating packets, nodes have to associate an ID and a timestamp together with a command in a header. As a consequence, all nodes have to be aware of the identifiers and local time increments of all nodes they may interact with, in all possible operation modes. Indeed, timestamps have to be integer multiple of the local time increment of the destination node, as shown in Figure 2.11 where nodes generate control packets concurrently. Since all nodes restart counting from $t = 0.T_S$ at each global synchronization point, all timestamps are bounded.

### Example 2

Consider the system given in Figure 2.5 and suppose that the current global notion of time is $t = 17.T_S$ during the first global period depicted in Figure 2.6. Suppose that a user sends parameters $u_1$ and $u_2$ to update a parameter $d_1$ in leaf node $LN_1$ and a parameter $d_2$ in leaf node $LN_2$. Also, suppose that the functions $f$ and $g$ such that $d_1 = f(u_1)$ and $d_2 = g(u_2)$ are defined locally in the Procedure-Repertoire instruction in intermediate node $N_1$.

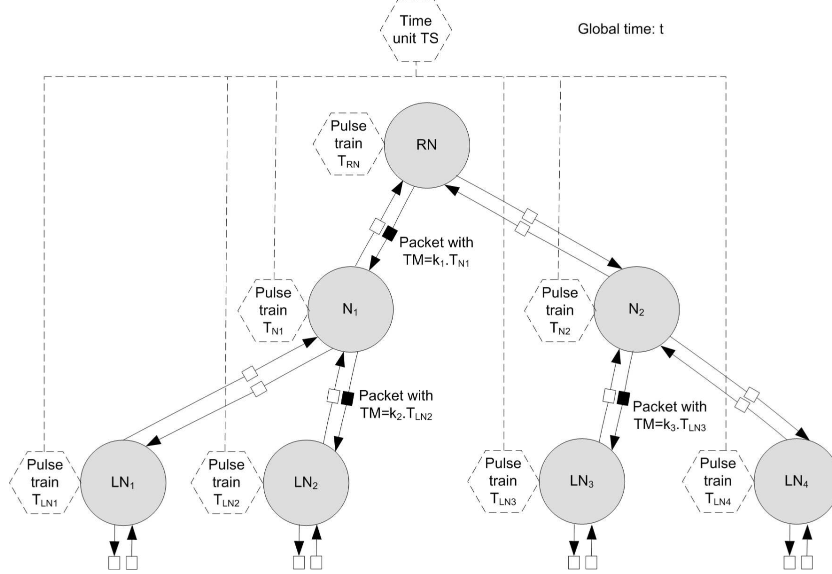


Figure 2.11: Concurrent execution of control packets in nodes of the hierarchical control network. All $k$ are integers. $TM$ stands for timestamp in a header in a control packet.

Root node is aware of its synchronization with $N_1$ ( $T_{N_1} = 12.T_S$, see example 1), and will convert the request to a control packet at time $t = 17.T_S$. Therefore, the timestamp must be greater than 17 and a multiple of 12, which is 24. To execute the pre-defined procedure that will process the functions $f$ and $g$ in $N_1$, the command $convert$ is expected. Thus, root node will generate the following packet ([header][control data information]):

$[ID = N_1; CO = convert, TM = 24; PR = 0; ][D_1 = u_1; D_2 = u_2; ]$

This packet will be received and queued in $N_1$ between time $t = 17.T_S$ and time $t = 24.T_S$. $N_1$ is aware of its synchronization with $LN_1$ and $LN_2$ ($T_{LN_1} = 15.T_S$ and $T_{LN_2} = 20.T_S$). At time $t = 24.T_S$, $N_1$ will receive a pulse from the timing network and update its local notion of time before executing the update command. This execution will process functions $f$ and $g$ sequentially to produce $d_1$ and $d_2$. The timestamps that are required to reach the beginning of the next periods in $LN_1$ and $LN_2$ are 30 and 40, respectively. If the command to update a parameter is $update$, then $N_1$ will generate the two following control packets:

[ID=$LN_1$; CO=$update$, TM=30; PR=0;][$D_1$=$d_1$]

[ID=$LN_2$; CO=$update$, TM=40; PR=0;][$D_1$=$d_2$]

Finally, $LN_1$ will update $d_1$ at time $t = 30.T_S$ and $LN_2$ will update $d_2$ at time $t = 40.T_S$, on the occurrence of their dedicated pulse.

**Implied restrictions**

Since the number of procedures that can be processed sequentially in a node during a period is limited, the number of packets that are sent to a node must also be limited. This is the case in our systems where events are sporadic.

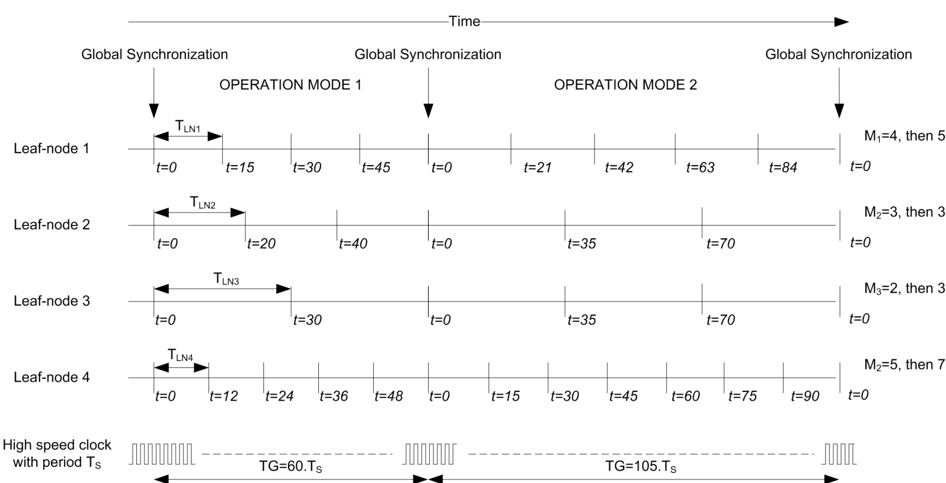


Figure 2.12: Periods may be changed only at global synchronization points in the control and monitoring network. Nodes may have different local time increments for each mode of operation.

Moreover, when swapping operation mode, periods and multiplicities may have to be modified to support different functions in the signal processing network. Such changes can only occur at global synchronization points, when all nodes are synchronized and restart counting from $t = 0.T_S$. Indeed, it is then possible to update the local time increments of all nodes and still re-synchronize all nodes after a fixed number of periods. An example is given in Figure 2.12 for the network that is shown in Figure 2.5. The multiplicity of the leave nodes $\{LN_1, LN_2, LN_3, LN_4\}$ changes from $\{M_1, M_2, M_3, M_4\} = \{4, 3, 2, 5\}$ in the first operational mode to $\{M_1, M_2, M_3, M_4\} = \{5, 3, 3, 7\}$ in the second operational mode. Equation 2.1 leads to $TG = 60.T_S$ for the first operational mode and to $TG = 105.T_S$ for the second operational mode. Equation 2.2 leads to different local time increments in the leave nodes for the two modes of operation. This means that all nodes must be aware of the local increments for each mode of operation, and for all nodes they interact with.

## 2.7 Modeling of the interfacing between the two networks

In this section we give our contribution concerning the modeling of the functional behavior of the interfacing between the control and monitoring network and the signal processing net-

work. As already shown in Figure 2.5 in the previous section, each process [4] $P$ in the signal processing network is interfaced with a leaf node $LN$ in the control and monitoring network. This interfacing relies on the fact that repetitive execution cycles of signal processing tasks fall well within periodic time intervals $T_{LN}$ in the control and monitoring network.

We first give a visual representation of a process in the signal processing network. Then we detail procedure that permits synchronizing the start of all execution cycles of all processes in the signal processing network. Then we detail the functional behavior of the interfacing between a leaf node and a process that executes cyclically.

### 2.7.1 Representation of a process in the signal processing network

A process has input ports and output ports through which it exchanges tokens with other processes in the signal processing network [5]. A visual representation of a process is given on the left-hand side in Figure 2.13, and the pseudo-code corresponding to the execution of a process is given on the right-hand side. This representation is similar to that of an object in the Stream Based Function (*SBF* [51]) model of computation. Nevertheless we add two ports for the interfacing with a leaf node in the control and monitoring network: a command/parameters input port and a monitoring-data output port.

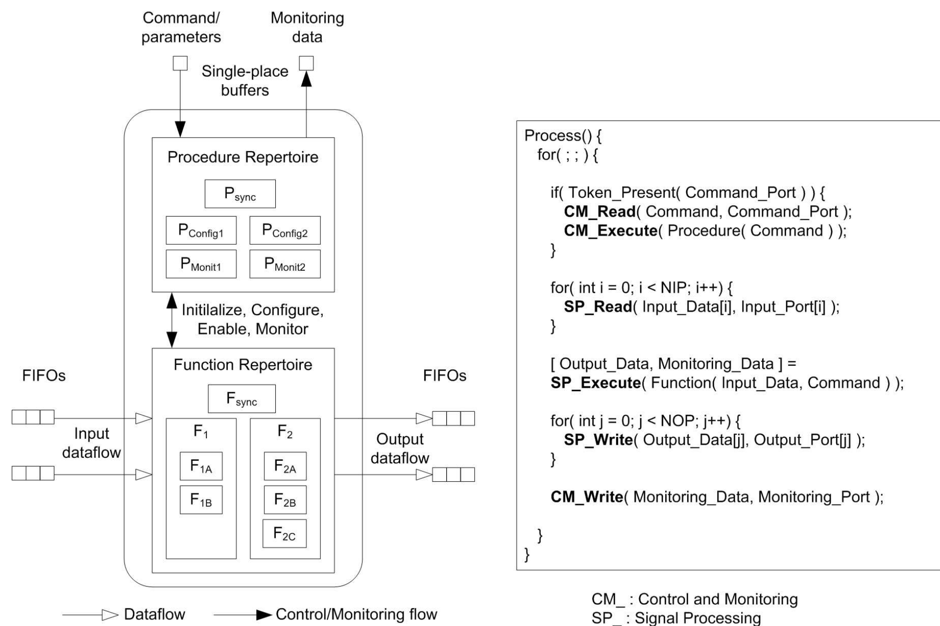


Figure 2.13: Representation of a process. A process is interfaced with a leaf node through 2 ports: a command/parameters input port and a monitoring-data output port.

[4]Recall that a process may be refined as an actor.

[5]In contrast to control packets, tokens in the signal processing network transport raw data only. Such tokens do not include a header and control-information

The process shown in Figure 2.13 can execute the following signal processing functions that are defined in the $Function\text{-}Repertoire$: the synchronization procedure $F_{sync}$, and $F_1$ and $F_2$, which are sequences of functions $\{F_{1A}, F_{1B}\}$ and $\{F_{2A}, F_{2B}, F_{2C}\}$, respectively. They can be re-configured and monitored by executing procedures ($P_{Config1}$, $P_{Monit1}$, etc) that are defined in the $Procedure\text{-}Repertoire$.

#### Complementarity between functions and procedures

A process executes different abstract $Read$, $Execute$ and $Write$ instructions to operate on control/monitoring tokens and signal processing tokens. The $CMRead$ instruction reads only command/parameters tokens from a single-place buffer in a blocking and destructive way (recall that a command/parameters token will be there, possibly empty). The $SPRead$ instruction reads only dataflow tokens from FIFOs in a blocking and destructive way as well. The $SPWrite$ instruction writes only dataflow tokens in FIFOs, while the $CMWrite$ instruction writes only monitoring-data tokens in the other single-place buffer.

Complementarity

| Command/ parameter token | (Sequence of) signal processing functions in the Function Repertoire | (Sequence of) control/ monitoring procedures in the Procedure Repertoire |
|---|---|---|
| Synchronization | $\{F_{sync}\}$ | $\{P_{sync}\}$ |
| $Configuration_1$ | $\{F_{1A}, F_{1B}\}$ | $\{P_{Config1}, P_{Monit1}\}$ |
| Empty | $\{F_{1A}, F_{1B}\}$ | $\{P_{Monit1}\}$ |
| $Configuration_2$ | $\{F_{2A}, F_{2B}, F_{2C}\}$ | $\{P_{Config2}, P_{Monit2}\}$ |
| Empty | $\{F_{2A}, F_{2B}, F_{2C}\}$ | $\{P_{Monit2}\}$ |

Figure 2.14: Example of possible complementary sequences of functions and procedures during an execution cycle of the process shown in Figure 2.13. The sequences that are executed are imposed by the command/parameters token sent by a leaf node.

The abstract $Execute$ instruction is interpreted as a pair of complementary $Function\text{-}Repertoire$ and $Procedure\text{-}Repertoire$ instructions. To each sequence of signal processing functions (such as $\{F_{1A}, F_{1B}\}$ in Figure 2.13) that is executed during a cycle corresponds a complementary sequence of procedures (such as $\{P_{Config1}, P_{Monit1}\}$ in Figure 2.13) that is executed during the same cycle. Figure 2.14 gives an example of possible complementary pairs during an execution cycle for the process shown in Figure 2.13. When the control/monitoring token is empty, the same sequence of signal processing functions is repeated, and the complementary procedure is a monitoring procedure.

### 2.7.2 Functional behavior of a process

In this subsection we give the functional behavior of the processes in the interfacing with leave nodes. We first give the synchronization procedure. Then we give the behavior of repetitive execution cycles and re-configuration cycles.

### Synchronization procedure

The purpose of the *synchronization procedure* is to synchronize the start of all execution cycles in all processes with a global synchronization point ($t = 0$) in the control and monitoring network. This procedure consists of 7 steps that are described below, and that are illustrated in Figure 2.15 with a simple example.

- Each process first executes a $P_{sync}$ procedure that waits for a $synchronization$ token on the command/parameters port.
- The root node sends a $synchronization$ packet to all leave nodes.
- When a leaf node receives a synchronization packet, it writes a synchronization token on its command/parameters port.
- When a process receives a synchronization token, the $P_{sync}$ function writes a $ready$ token on the monitoring-data port to indicate that it is ready to start processing, and waits for a $start$ token on its input command/parameters port. Complementarily, $F_{sync}$ may write tokens to FIFOs if required.
- When a leaf node receives the $ready$ token from the process it is attached to, it sends a $ready$ packet to the intermediate node above in the control and monitoring network. Once an intermediate node has received $ready$ packets from all the leave nodes that are interfaced with it, it sends an $ready$ packet the node above in the control and monitoring network.
- Once the root node has received $ready$ packets from all the intermediate nodes that are interfaced with it, all processes are in the same state, i.e., waiting for a start token. Then, the root node sends a $start$ packet to all leave nodes $LN_i$, to be executed at $t = TG\text{-}T_{LN_i}$, i.e., during the period that precedes the synchronization point.
- All leave nodes process the start packet at $t = TG\text{-}T_{LN_i}$, and write a start token on their command/parameters port on the occurrence of the next pulse, i.e., at $t = 0$. Then, all processes receive a start token on their command/parameters port simultaneously. Thus, the two networks are synchronized at $t = 0$, and all processes start executing repetitive execution cycles together.

### Repetitive execution cycles

At the end of the synchronization procedure, all processes start executing repetitive execution cycles as follows.

- 1) The $CMRead$ instruction reads a command/parameters token (from the corresponding single-place buffer). If the token is empty, the process jumps to step 2. Else the token is processed in the $Procedure\text{-}Repertoire$, to swap between signal processing functions or simply update parameters in a signal processing function in the $Function\text{-}Repertoire$, without executing that function.

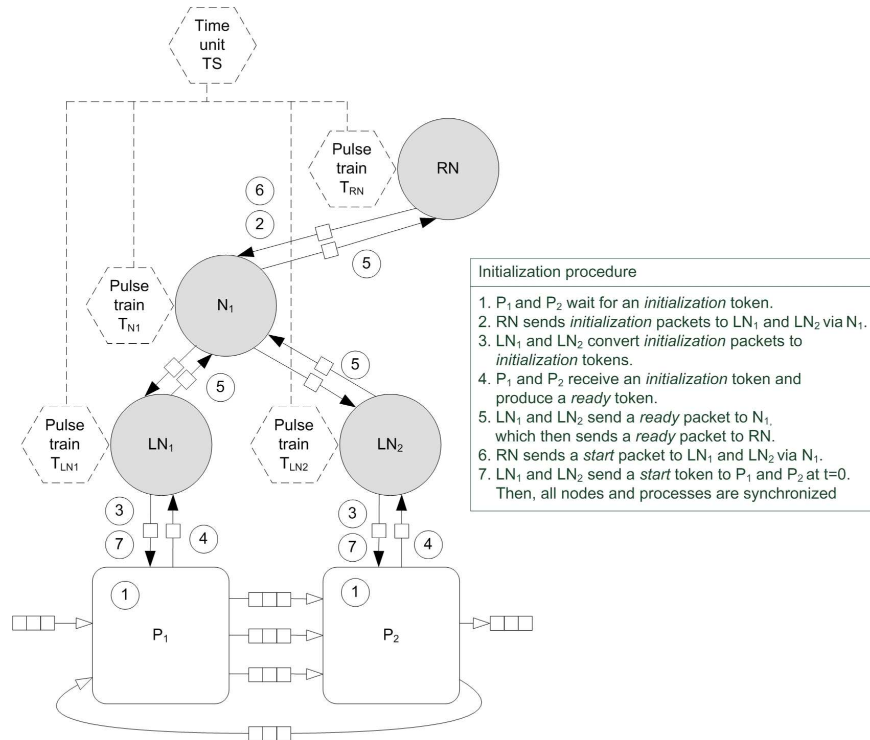


Figure 2.15: Synchronization procedure: example of a simple system. At the end of the procedure, all processes and all nodes are synchronized with a global synchronization point ($t = 0$).

- 2) The $SPRead$ instruction reads dataflow tokens (from FIFOs) in a blocking and destructive way and sends these tokens to the $Function\text{-}Repertoire$, where they are processed in a sequence of signal processing functions. This sequence of functions produces tokens that are written on the dataflow output ports by the $SPWrite$ instruction.

- 3) The $CMWrite$ instruction is executed, and writes a monitoring-data token to the corresponding single-place buffer. Then the process enters another execution cycle by executing a $CMRead$ instruction (step 1).

Recall that a leaf node $LN$ produces one command/parameters token (possibly empty) for a process $P$ per period $T_{LN}$, and that $LN$ expects one monitoring-data token from $P$ before the end of $T_{LN}$. This means that the execution of the sequence of signal processing functions and complementary control/monitoring procedures have to be finished before $LN$ receives the next pulse. As a consequence, $P$ can not wait indefinitely long for input dataflow tokens. However, rare unexpected delays may appear in dataflow communication channels. As a result, leave nodes may not receive monitoring-data tokens before the occurrence of the next pulse. If such a fault is detected, the signal processing network and the control and monitoring

network must be re-synchronized. Example 3 gives an example of such a procedure, which is similar to the synchronization procedure.

**Example 3**

Consider the system shown in Figure 2.5 and suppose that the Function-Repertoire in $P_1$ contains two signal processing functions that have the same execution time. Suppose that the procedure executed at $t = 30.T_S$ in $LN_1$ in example 2 requests swapping between these two functions. $LN_1$ will send a $swap$ command token to $P_1$ between $t = 30.T_S$ and $t = 45.T_S$ (because $T_{LN_1} = 15.T_S$, see Figure 2.6). At the beginning of a new execution cycle, the process will consume this command and swap function in the Function-Repertoire.

Now, suppose that $P_1$ did not generate a monitoring-data token in the time interval [0;15] after a global synchronization point. $LN_1$ generates a control packet to report this fault to the root node (the timestamp is $TM = 30$ since it is the closest multiple of $T_{RN} = 30.T_S$ after $t = 15.T_S$):

$[ID = RN; TM = 30; CO = delay; PR = 0][]$

When it receives this packet, the root node executes a re-synchronization procedure. It sends a first packet to all leave nodes such that all processes in the signal processing network enter a re-synchronization state immediately after the next global synchronization point ($t = 0.T_S$):

$[ID = leavenodes; TM = TG - T_{LN}; CO = synchronize; PR = 0][]$

Then it sends another packet to all leave nodes such that all processes in the signal processing network start processing immediately after the next global synchronization point ($t = 30.T_S$):

$[ID = leavenodes; TM = 30 - T_{LN}; CO = start; PR = 0][]$

At time $t = 30.T_S$, all processes start executing repetitive execution cycles in the signal processing network. Thus, it took one global synchronization period to re-synchronize the two networks. This is acceptable as long as the chance of encountering such synchronization errors remains low. This is the case in our systems where the dataflow originating from the antennas is not interrupted.

## 2.8 Related work

Complete end-to-end low-level simulations failed for the digital processing part of a station of a radio telescope. In practice, models are implicit or intuitive, and may lead to ambiguous decisions. We tried to make these models more explicit and structured to help decisions taking. Also, the current practice assumes from beginning that both the signal processing part and the control and monitoring part are timed and synchronous. This approach may be too restrictive in large-scale and distributed systems, where the two parts have to be autonomous. We specified the signal processing part based on an un-timed model and we synchronized it with the control/monitoring part that is specified based on a timed model.

In the Parameterized Synchronous Dataflow model (*PSDF* [52]), the dataflow specification

is separated from the control specification, with the objectives of staying within one model of computation in such a way that it remains possible to derive schedules. We are dealing with a large signal processing network and a large control and monitoring network that can not be interfaced as in [52] because the behavior of the control and monitoring network is sporadic.

In [53], applications are modeled using Process Networks and SBF with non-static parameters. Process can be re-configured only after a complete network cycle. In our systems, processes have to be re-configured at run-time. The Reactive Process Networks model (*RPN* [54]) extends the KPN model with reactive semantics. The RPN model identifies events with functions that transform the configuration of the RPN. However, re-configuring the behavior of the KPN processes requires interrupting the processing, such that the deterministic behavior of the processes is not preserved.

The SYRF (Synchronous Reactive Formalism [55]) research project investigates some long time research tracks ranging from the combination of declarative and imperative formalisms, integrating synchrony and asynchrony while preserving a clean formal semantics, connecting with hardware/software co-design, program verification, code distribution and multi-tasking.

## 2.9 Conclusions

In this chapter we presented our approach to specify the functional behavior of the applications that run in large-scale and distributed digital signal processing systems, based on models of computation. We separated the modeling of the signal processing part from the modeling of the control and monitoring part. The former has been modeled based on a KPN and the latter has been modeled based communicating finite state machines. The interfacing between the two models is based on relations between a notion of time that is superimposed on top of the control and monitoring network only, and periodic intervals within which repetitive tasks are executed in the signal processing network, such that the functional behavior of the untimed dominant signal processing network is not altered by the interfacing with the control and monitoring network that is timed.

We detailed the functional behavior of the two parts by giving a visual representation of nodes in both networks so as to conveniently specify the way they concurrently process and communicate data. Nevertheless, this representation is independent from any architecture. The modeling of the architecture is addressed in the next chapter.

Chapter 3

# Architecture specification

## 3.1 Summary

The large-scale and distributed digital systems we consider consist of a signal processing part and a control and monitoring part. In this chapter we concentrate on the model-based specification of the architecture of such systems [1]. We rely on models at multiple levels of abstraction and hierarchy to master complexity and hide irrelevant details when appropriate. Our objective is twofold. On the one hand we want to predict the non-functional behavior and related performance and cost of the architecture. On the other hand we want to assure that the architecture can be constructed starting from abstract specifications, and meet computation and communication performance requirements. We separate the modeling of the signal processing architecture from the modeling of the control and monitoring architecture so as to take appropriate decisions concerning their individual computation and communication properties. The signal processing architecture supports intensive computations on and transport of high-throughput and parallel data streams. The control and monitoring architecture supports the execution of sequences of procedures in reaction to sporadic events in a tree-like structure. The interfacing between the control and monitoring part and the signal processing part occurs at the lowest level of the hierarchy of the two parts, where signal processing units and control/monitoring tree leave nodes are interfaced through dedicated point-to-point links. This approach permits avoiding ad-hoc interfaces between the two parts when scaling the system, and leads to an architecture model onto which the application model can be mapped.

[1] Remember that the architecture specification is part of the system-level specification that also includes an application specification, and the mapping of the application onto the architecture.

## 3.2 Introduction

Recall that we strictly adhere to the separation of concerns principle [17] in the sense that we model the application (functional behavior) separately from the architecture (performance and cost), and make a distinction between computation and communication in both models. The only condition we impose on the relation between the two models is that they are specific to a particular domain of application, and that if the architecture supports parallelism, then the application should be specified in some parallel language.

In principle, an architecture can be modeled in terms of a mixture of models of computation that are available for application modeling as given in chapter 2. In general terms, an architecture consists of a composition of computation components, communication components, and storage components, together with data transfer and synchronization primitives and protocols, as well as software to operate the architecture. Components may be modeled as ”black boxes” or ”white boxes” or both. A black-box model is a relation between input quantities and output quantities expressing performance and cost in terms of simple service equations, which can be used for performance/cost analysis. A white-box model consists of a composition of executable modules. When specifying a large scale and distributed architecture, both kinds of models are used: white box models at the lower levels where simulation is feasible, and black-box models at higher levels where simulation is not feasible. Both black-box and white-box models are abstract, and one of the objectives is to convert them to an implementation-level of abstraction, from where implementation should become straightforward based on commercially available tools.

Recently, the so-called platform-based design approach has emerged from the need to re-use designs, reduce design costs and cope with time-to-market constraints [56] [57]. Intuitively, a platform defines a family of permissible architectures in terms of building blocks and compositions of building blocks. In general, a platform is derived from an analysis of an application domain. Such an application domain concerns the next generation large-scale and distributed radio-telescopes, such as LOFAR and SKA. These systems support signal processing as well as control and monitoring functionalities. From a separation of concerns (orthogonality) viewpoint, we virtually separate signal processing platforms from control/monitoring platforms. With this distinction, we aim at simplifying the system architecture modeling, and decision making concerning their respective (dataflow-driven or control-driven) computation and communication properties. Once the two platforms are specified, the superposition and interfacing of permissible architectures is constrained by the mapping of application specifications on the architecture specification (see chapter 4).

In this chapter we present our approach to model architectures of large-scale distributed and hierarchical systems. Section 3.3 gives definitions of platforms, architecture templates, and architectures. Section 3.4 presents domain-specific platform component models. Sections 3.5 and 3.6 present the modeling of the signal processing part and control/monitoring part, respectively, with a particular emphasis on their computation and communication performance. The superposition of the two parts is discussed in section 3.7. Finally we give related work in section 3.8 and draw conclusions in section 3.9.

## 3.3 Definitions

In this section we introduce platforms and platform derivatives in an abstract way as defined in the platform-based design paradigm, and we model domain-specific platform components in a way that is tailored to our purpose.

### Platform

A *platform* defines a domain-specific restricted family of admissible architectures. It consists of a library of parameterized components for computation, communication, and storage together with permissible component interconnection rules, and software to operate the architectures. This is illustrated in Figure 3.1. A platform includes means to evaluate, or information on, the performance and cost of the components in terms of relevant metrics.

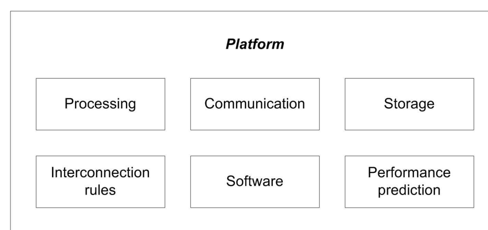


Figure 3.1: A platform includes processing, communication and storage units, interconnection rules, software to operate compositions of components, and information about performance and cost of the components.

*Components* in a platform can be either basic components or compositions of basic components. Components can be modeled at various levels of abstraction using models that are adequate on the given level of abstraction. A component may be hardware, software or a combination of the two. It can be a processing unit, a communication unit or a storage unit. These units can in turn be modeled as a composition of modules that model component-specific (dynamic) services ranging from instruction interpretation, to communication interfacing, to arbitration [58].

### Architecture template

An *architecture template* is a particular parameterized composition of components that is obtained from a platform. It consists of a set of computational units, and a communication, synchronization and storage infrastructure as depicted in Figure 3.2. The behavior of a template is non-functional, and the performance and cost are expressed in terms of relevant metrics such as throughput, delay, bandwidth, memory usage, power consumption, etc.

Architecture templates can become components in a platform intended to model large-scale systems as shown in Figure 3.3. It seems natural to call templates that are compositions of basic components first-order templates. First-order templates are themselves higher-order

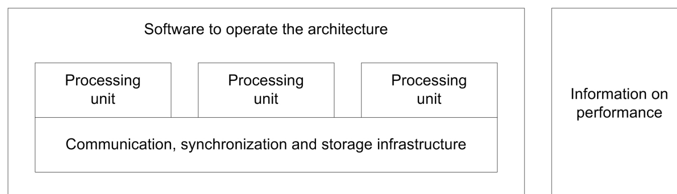


Figure 3.2: Example of architecture template. Processing units are interfaced to a communication, synchronization and storage infrastructure. The template includes software components to operate the particular composition and contains information on the performance of this composition. It is still parameterized.

components. Compositions of first-order templates are, then, higher-order templates. A composition also comes with software components that support operations such as data routing across architecture templates using specific primitives and protocols. Information on the performance of the composition is also available, possibly after simulating the behavior of the composition on the level of basic components and modules as in [58], or after interpolating/extrapolating from calibrated information based on formulae at higher levels of abstraction as in [16].

Composing with architecture templates can be repeated over and over again to model architectures of large-scale systems. Because we consider a specific application domain, and components are parameterized, the number of architecture templates is restricted. Indeed, the number and type of processing units in a template are parameterized.

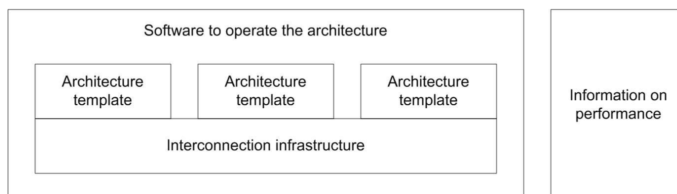


Figure 3.3: Composition of architecture templates to model large-scale systems. Architecture templates are possibly components in a platform intended to model large-scale systems.

### Architecture

An *architecture* is an instance of an architecture template, in which parameters have been given values. For example, an architecture may consist of two Digital Signal Processors (DSPs), a shared memory, and a bus connecting the DSPs and the memory.

We are aware of the fact that complex architectures may incur dependability problems [59] that affect security, availability and integrity constraints. Here we assume that hardware and software components encompass modules to deal with these aspects. Notice however, that a secure system is a system that behaves as intended, and that model-based specifications are

also intended to mitigate the problem by striving for correctness by construction, if possible.

## 3.4 Representation of platform components

In this section, we model basic platform components, which include processing units, communication units, and storage units. Because we consider that specifications are on a level of abstraction above implementation-level specifications (which can be converted to an actual implementation using commercially available compilation and synthesis tools), we need to rely on abstract models.

### 3.4.1 White-box and black-box models

Models of platform components should be tailored to our twofold objective. First, they should allow to predict the non functional behavior of the system and to give performance/cost numbers when moving up in the hierarchy. Second, they should include all the information that is necessary to go down to the implementation level. We model components with two appearances. The first one is a *white-box* appearance that is tailored to the performance/cost estimation objective at lower levels of the hierarchy. The second one is a *black-box* appearance that is tailored to the performance/cost prediction objective at higher levels of the hierarchy, and to the implementation objective.

A white-box model of a component consists of executable modules that model the dynamic internal behavior of the component. Modules are needed to obtain fine grained values of the parameters in terms of processing delays, communication latencies, etc, if not known. Thus, most parameter values in a white-box component model are typically obtained through calibration as in [16], whilst others need to be derived, possibly in a simulation-based approach as in [58], because they may be application/data dependent.

A black-box model of a component hides the internals of the component by providing service relations between input and output quantities in terms of performance and cost metrics (e.g., static estimated frequency of usage at run-time) as in [60], where rules provide a closed-form expression language that allows calculating the output from the input using common mathematical functions or operations, interpolation or table lookup. Black-box models are typically used at higher levels of the hierarchy where simulation is not an option. In general, the abstract specification of an architecture is given as a topology in which components have a black-box appearance.

Figure 3.4 shows an example of white-box specification and representation (on the left-hand side) and black-box specification and representation (on the right-hand side) for a component that has two input ports ($P_1$ and $P_2$) and two output ports ($P_3$ and $P_4$). In the white-box model, internal modules ($M_1, ..., M_4$) and signals ($s_1, ..., s_6$) are available and their dynamic behavior can be simulated to obtain parameter values. In the black-box model, internal signals and modules are hidden, and each service provides output quantities as (simple, usually linear) functions of input quantities ($q_3 = f_1(q_1, q_2)$ and $q_4 = f_2(q_1, q_2)$).

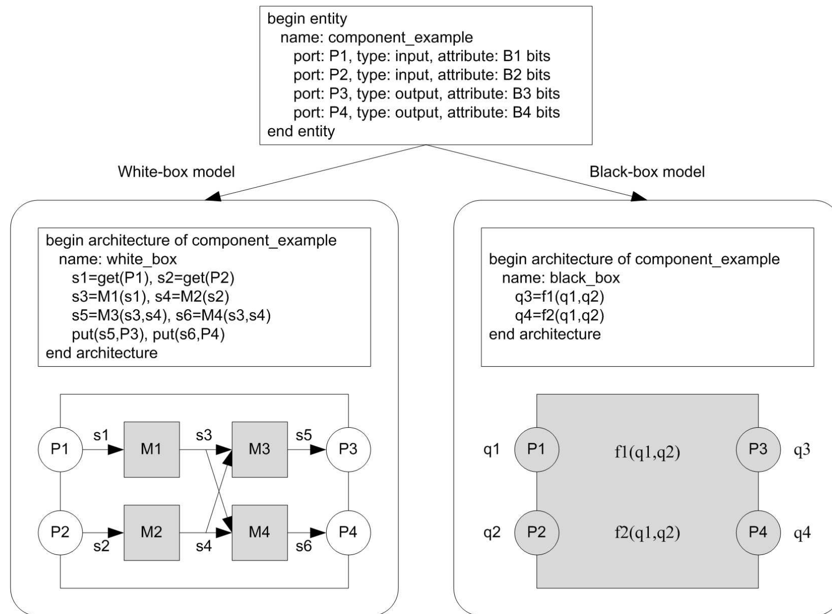


Figure 3.4: Example of white-box model (left-hand side) and black-box model (right-hand side) of a component; $f_1$ and $f_2$ are services.

When building a first-order architecture template, white-box models of basic components are replaced by black-box models. In white-box models, parameter values can be obtained by relying on calibration and simulation. Then, parameter values can be derived that have a meaning for the black-box model of the template as a whole. Deriving these values for the black-box model of the template reduces complexity: from here on, i.e., for higher-order templates, performance/cost analysis through simulation is no longer possible, because first-order template components have a black-box appearance and hide executable modules. Instead, performance/cost parameter values are predicted by propagating input-output quantities and evaluating their relations as specified in the body of the components. Black-box models of templates become platform components that are available for high-level architecture specifications, and that can in principle be converted to an implementation using commercially available compilation and synthesis tools.

As shown in Figure 3.5, we distinguish three types of components (processing, communication, and storage units). Components can be hardware and/or software components. Internally to components, one can find (software) services and/or (hardware) units. In the remainder of this section we focus on the white-box and black-box modeling and interfacing of these components. The modeling of first order and higher order architecture templates is dealt with in section 3.5 for the signal processing part, and in section 3.6 for the control/monitoring part.

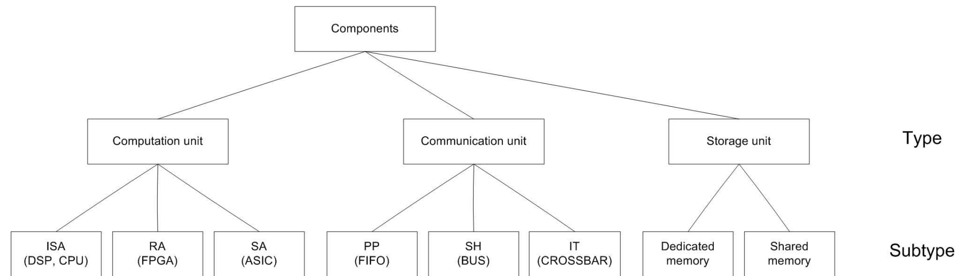


Figure 3.5: Classification of components.

### 3.4.2 Processing units

As shown in Figure 3.5, we distinguish three subtypes of processing units: Instruction Set Architecture (*ISA*), Re-configurable Architecture (*RA*), and Specific Architecture (*SA*). These processing units include modules and execute specialized services for processing, storage, communication, and synchronization [2]. White-box models of processing units are parameterized as detailed in Table 3.1. The average power consumption parameter value for ISA (e.g., DSPs and CPUs) is usually higher than in SA (e.g., ASICs) and RA (e.g., FPGAs). RA and SA processing units can be customized so as to obtain a lower processing delay parameter value than in ISA by reducing the sharing of internal modules and memories when processing data streams.

| Name | Information | Unit |
|---|---|---|
| NUM-IOP | # input/output ports | |
| PROG | programmability | |
| RECONFIG | re-configurability | |
| NUM-MOD | # processing modules | |
| NUM-MEM | # storage modules | |
| SIZE-MEM | size of storage modules | bit |
| FREQ | processing frequency | Hz |
| DELAY | processing delay | s |
| POWER | average power consumption | W |
| COST | cost | $ |

Table 3.1: Parameters of white-box models of processing units.

Parameters of black-box models of processing units are given in Table 3.2. In contrast to white-box models, services in black-box models are abstract (expressed based on functions) and do not necessarily have a one-to-one relation with internal modules. For example, the power consumption parameter can be expressed in function of input quantities.

[2] A priori, specialized (architecture) services do not math abstract $Read$, $Execute$ and $Write$ instructions in the application model. The matching is improved based on mapping transformations as detailed in chapter 4.

| Name | Information | Unit |
|---|---|---|
| NUM-IOP | # input/output ports | |
| NUM-SERV | # services | |
| PROC-FUNC | processing service cost function | |
| MEM-FUNC | memory service cost function | bit |
| FREQ-FUNC | frequency cost function | Hz |
| TRHOU-FUNC | throughput cost function | tokens/s |
| POWER-FUNC | power consumption cost function | W |
| COST-FUNC | cost function | $ |

Table 3.2: Parameters of black-box models of processing units.

### 3.4.3 Communication units

As shown in Figure 3.5, we distinguish three subtypes of communication units: Point-to-Point (*PP*), Shared (*SH*) and Intermediate (*IT*) communication units. These communication units include modules for storage and modules for synchronization. White-box models of communication units are parameterized as detailed in Table 3.3. A PP (e.g., a FIFO) has a unique input interface and a unique output interface, and transfers a single type of tokens. A SH (e.g., a bus) has multiple input interfaces and multiple output interfaces, and may transfer different types of tokens. A IT (e.g., a crossbar) has multiple input interfaces and multiple output interfaces, and transfers a singles type of tokens.

| Name | Information | Unit |
|---|---|---|
| NUM-INT | # interfaces | |
| NUM-MEM | # storage modules | |
| SIZE-MEM | size of storage modules | bit |
| ARBITER | availability of arbiter module | |
| LATENCY | latency | s |
| POWER | average power consumption | W |
| COST | cost | $ |

Table 3.3: Parameters of white-box models of communication units.

Figure 3.6 shows an example where the processing units $PU_1$ and $PU_2$ exchange tokens with the processing units $PU_3$ and $PU_4$ through the communication unit $CU$, which has two interfaces ($Int_1$ and $Int2$). $PU_1$ and $PU_2$ send/get tokens to/from $Int_1$, and $PU_3$ and $PU_4$ send/get tokens to/from $Int2$. The processing units and the communication unit include modules that control the distribution of tokens.

Black-box models of communication units are service-relation models. The performance in terms of memory usage, throughput, cost, etc (see Table 3.4) of each service is expressed based on simple functions.

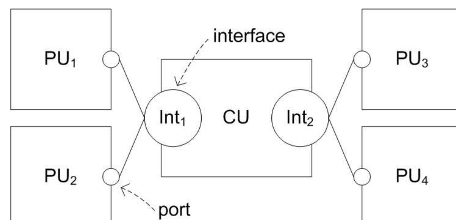


Figure 3.6: Example of communication between four processing units through a communication unit that has two interfaces.

| Name | Information | Unit |
|---|---|---|
| NUM-INT | # interfaces | |
| NUM-SERV | # services | |
| MEM-FUNC | memory cost function | bit |
| THROU-FUNC | throughput cost function | tokens/s |
| POWER-FUNC | power consumption cost function | W |
| COST-FUNC | cost function | $ |

Table 3.4: Parameters of black-box models of communication units.

### 3.4.4 Storage units

Storage units are present as modules in processing units and in communication units. A white-box model of a storage unit is a service-relation model. It has a fixed number of input and output ports, which have the same type. A white-box model of a storage unit is parameterized in terms of number and size of internal modules, and in terms of latency and power consumption. Caches are accepted, and controlled by the software to operate the architecture. In a black-box model of a storage unit, the performance/cost of a memory service is a simple function of, for example, an access cost and storage capacity.

## 3.5 Signal processing architecture model

Recall that the signal processing systems we consider consist of three stages: a front-end stage, a back-end stage and an intermediate data-reduction stage (see Figure 3.7). The front-end stage includes low frequency (LF) and high frequency (HF) antennas that acquire data in stations. This data goes through low-noise amplifiers before it is digitized locally next to the antennas at a high sampling rate. This high-throughput digitized data feeds the intermediate data-reduction stage, which is also distributed locally next to the antennas in the stations. The intermediate data-reduction stage consists of approximately 100 stations, which run concurrently and must sustain the high throughput of the data that is imposed by the front-end stage. The nature of computation and communication progressively changes according to the nature and rate of the data, from single tokens, to vectors of tokens, to large and lower-rate blocks

of tokens. These blocks are then sent through a wide area network to the back-end stage (a supercomputer).

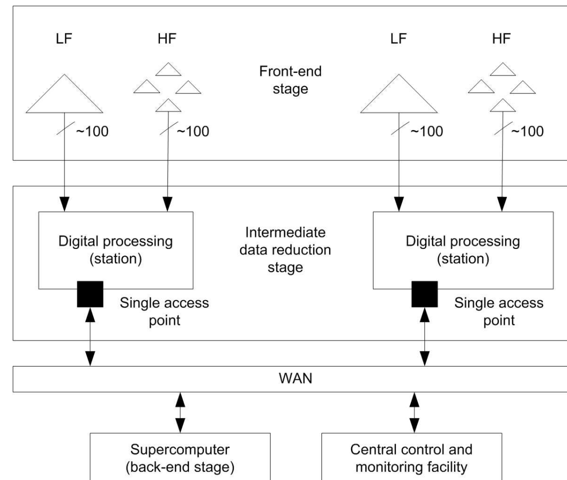


Figure 3.7: The intermediate data reduction stage processes high throughput data originating from the antennas before sending it to a supercomputer.

The modeling of the signal processing part of the systems we consider has already been addressed in [16]. In this section we give a representation of the intermediate data-reduction stage based on the models that have been introduced in section 3.4. We present two first-order architecture templates (composition of basic components). Then we present a higher-order architecture template (composition of first-order templates at station level).

### 3.5.1 First-order architecture templates

In the intermediate data reduction stage, we distinguish subsystems that are close to the front-end stage from those that are close to the back-end stage. We give first-order architecture templates for these two parts.

#### Front-end architecture templates

Front-end architecture templates ($FT$) operate on tokens originating from the front-end stage and are modeled as a composition of library components that are arranged in a Single-Instruction Multiple-Data topology (SIMD according to Flynn's taxonomy [61]), which supports the required degree of parallelism for that part of the system. Processing units are modeled as SA and RA (ASICs and FPGAs) since both subtypes can receive, operate on and produce multiple streams with a high degree of parallelism. Intermediate computations are stored in dedicated storage units that are attached locally to the processing units so as to maintain low latencies by avoiding sharing resources. Communication units are modeled as PP

(FIFOs [3]) that transport tokens with low latencies. IT communication units (crossbars) can be used as long as the required throughput is sustained. Wether this is feasible or not is determined by simulating the behavior of these particular components based on their white-box models as in [58].

The black-box model of a FT is parameterized in terms of number of input ports and output ports, and in terms of number of processing units. This allows to achieve the required degree of parallelism. The latency, throughput, power consumption and cost parameters of the black-box template can be obtained based on interpolation/extrapolation from calibrated information for each of the internal components. An example of a composition of components in a FT is given on the left-hand side in Figure 3.8. The Matrix Shuffling Processing (MSP) architecture, which has been used in the *THEA* system (Thousand Elements phase Array telescope [62]) is of that type. The MSP architecture provides processing nodes (processing-storage unit pairs) that are connected with point to point links in a mesh network.

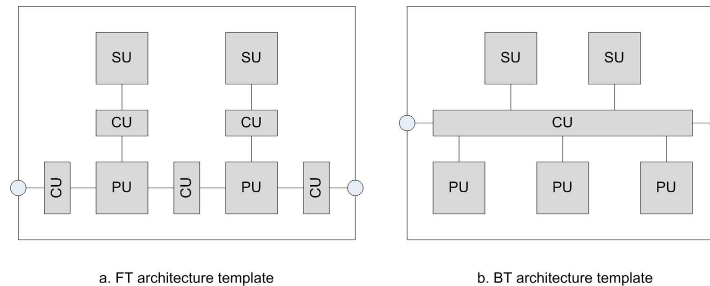


Figure 3.8: Examples of composition of library components in (a) FT template and (b) BT template. Processing Units (PU) are interfaced with Storage Units (SU) through Communication Units (CU).

### Back-end architecture templates

Back-end architecture templates (*BT*) operate on tokens issued from FT templates and send tokens to the back-end stage. They are modeled as compositions of library components that are arranged in a Multiple-Instruction Multiple-Data topology (MIMD [61]), such that the degree of parallelism is lower than in FT templates. Processing units are ISA (such as Digital Signal Processors) that operate on tokens sequentially. Intermediate computations are stored in storage units that can be accessed by services executed in different processing units. Communication services are executed by a communication unit that is modeled either as a IT (crossbar) or SH (bus). If a bus is chosen, then an arbiter module must be available as well. This decision is taken by simulating the data-dependent dynamic behavior of these components.

The black-box model of a BT has a fixed number of input and output ports. It is parameterized in terms of number of processing units and storage units. An example of composition of

[3]Asynchronous KPN communication channels in the application model may be implemented by means of synchronized (clocked) FIFOs at lower levels in the architecture model.

library components in a BT is given on the right-hand side in Figure 3.8. The Fast Pipeline Processing (FP) and Selection Cache Storage Processing (SCSP) architectures in [16] are of that type. The former is well suited to process blocks of data, and relies on a crossbar component for communication, whilst the latter is well suited for the storage and re-ordering of blocks of data, and relies on a bus for communication.

### 3.5.2 Higher-order architecture template

The signal processing part of a station is modeled as a higher-order architecture template, which is a composition of black-box models of first-order FTs and BTs. An example is shown in Figure 3.9. FTs and BTs communicate through library components that are modeled as PP communication units, which sustain a high-throughput. The High-Speed Link (HSL) architecture in [16] is of that type. It consists of optical to digital and digital to optical converters, and a switch, and includes means to multiplex, serialize and de-serialize tokens.

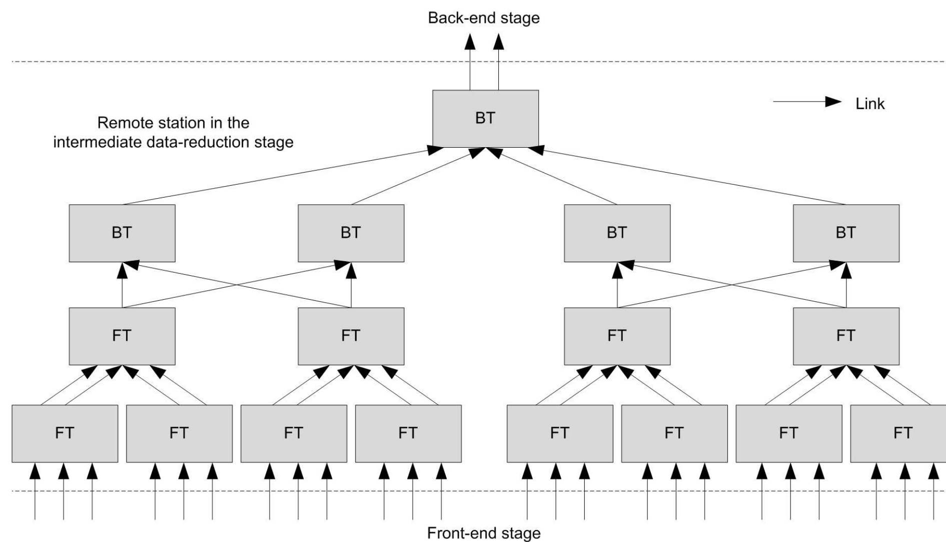


Figure 3.9: Example of a composition of black-box models of FTs and BTs to model the signal processing architecture of a station in the intermediate data reduction stage. FTs and BTs exchange communicate through high speed links.

The black-box model of the signal processing part at station level is a (higher-order) component that is parameterized in terms of number of input ports, and in terms of number of FTs and BTs. These parameters permit adapting the degree of parallelism to sustain the throughput of data that is imposed by the front-end stage. At this level of abstraction, parameter values can not be obtained by simulation anymore since modules are hidden in the FTs and BTs. Instead, they are obtained based on constraints propagation and simple equations that relate output quantities to input quantities in the FTs and BTs.

## 3.6 Control and monitoring architecture model

Recall that stations send monitoring tokens and receive control tokens to/from a central control and monitoring facility (see Figure 3.7). In this section we focus on the modeling of the control and monitoring part of the architecture that controls the intermediate data-reduction stage. We give the main computation and communication properties of a first-order control and monitoring architecture template. This leads to a higher-order architecture template to model the non-functional behavior and performance/cost of the control and monitoring part at station level.

### 3.6.1 First-order architecture template

A control and monitoring architecture template (*CT*) has a single bidirectional access point. Tokens that are exchanged through this single access point are stored in a storage unit that is internal to the CT. Control and monitoring procedures, which are not as computationally intensive as signal processing tasks, are executed sequentially in (re-programmable) processing units that are modeled as ISA. These processing units access specific instructions and topology-related information, and write and read intermediate computations to/from a local storage unit through a PP communication unit. The number of processing units is parameterized. This permits adapting the performance/cost of the template to the intensity of the control and monitoring flow.

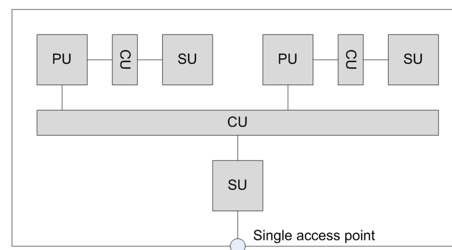


Figure 3.10: Example of CT template. Tokens that are exchanged with other CTs are stored in a unique memory that is shared between specialized services executed in processing units to read and write control/monitoring tokens from/to other CT templates.

All processing units share a global notion of time that is provided by a Real-Time Operating System (RTOS) module. This notion of time permits scheduling the transmission of tokens. Processing units communicate among themselves through a communication unit that is modeled as a black-box SH or IT (a bus or a crossbar). An example of CT is represented in Figure 3.10. The Acquisition Main Control (AMC) architecture in [16] is of that type. It includes a memory that can be accessed from an ISA processing unit through a crossbar.

### 3.6.2 Higher-order architecture templates

At station level, the control and monitoring architecture is modeled as a higher-order architecture template where components are CTs that are arranged in a tree topology. Each CT in a given hierarchical level communicates with other CTs in the tree through point-to-point bidirectional links that are modeled with two unidirectional PP (FIFOs), as shown in Figure 3.11. The software that operates the control and monitoring architecture at station level permits avoiding contentions in these interfaces.

At station level, the black-box model of the higher-order control and monitoring architecture has a single bidirectional access point, through which tokens are communicated to initiate the execution of control and monitoring procedures according to a specific operation mode, and to report monitoring information. This black box model is parameterized in term of number of CTs. This allows to adapt the workload of the control and monitoring network to the control/monitoring flow. The value of this parameter is obtained analytically since modules are hidden in CTs.

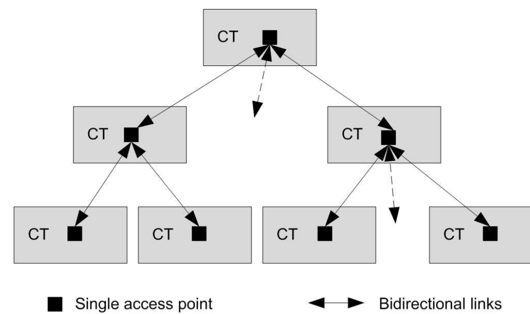


Figure 3.11: Example of control and monitoring architecture model at station level. CTs are modeled as black boxes and are arranged in a tree topology.

## 3.7 Interfacing of the two architectures

We separated the modeling of the signal processing architecture and the modeling of the control and monitoring architecture as their individual non-functional behavior, constraints, and performance are fundamentally different. In this section we deal with the interfacing of the two architectures. We first give the interfacing at the level of first-order FTs, BTs and CTs. Then we give the interfacing on the level of basic components that are internal to these first-order architecture templates.

### 3.7.1 Interfacing at station level

We model a station as a composition of first-order signal processing architecture templates (black-box FTs and BTs), and first-order control and monitoring architecture templates (black-

box CTs) that are arranged in a tree topology. Each template in the signal processing part (FT or BT) is interfaced with a unique template in the control and monitoring part (CT). This simple interfacing permits avoiding ad-hoc interfaces, and leads to an architecture that can serve as a starting point to map the application. The number of links in this one-to-one interfacing is a parameter that corresponds to the number of processing units that are internal to the FTs and BTs. Note that links that are dedicated to the communication of tokens in the signal processing network are kept separated from links that are dedicated to the communication of tokens in the control and monitoring network, so as to avoid obstructing the high throughput in the signal processing part with sporadic control/monitoring tokens.

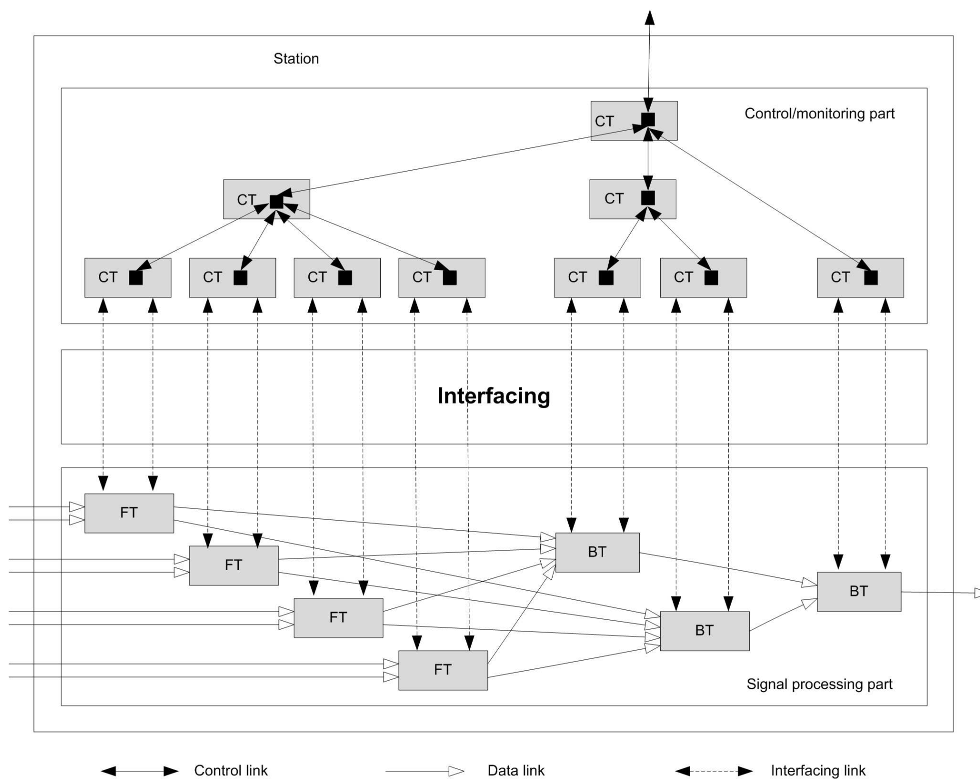


Figure 3.12: Modeling of the interfacing between control and monitoring architecture templates (CT) and signal processing architecture templates (FT and BT) at station level. The number of interfaces for a signal processing architecture template corresponds to the number of processing units that are internal to that template. The resulting architecture can serve as an input to map the application.

An example is shown in Figure 3.12, where all FTs and BTs include two processing units. The interfaces are represented with double-sided dotted arrows, and transfer command/parameter tokens from CTs to FTs (and from CTs to BTs), and monitoring-data tokens from FTs back to CTs (and from BTs back to CTs). The large number of CTs in such an initial architecture may be reduced when mapping the application onto the architecture as discussed in chapter 4. The

single entry point of the CT that is a root at station level becomes the station's single entry point, through which tokens are exchanged with the central control and monitoring facility.

When duplicating subsystems, the approach to interface the control and monitoring architecture model and the signal processing architecture model remains the same. An example is shown in Figure 3.13. Note that CTs may have to be added on top of the tree in the control and monitoring model in order to obtain a single entry point for the complete architecture. This potential insertion of CTs leads to a minor cost increase, but does not affect the interfacing between the two parts.

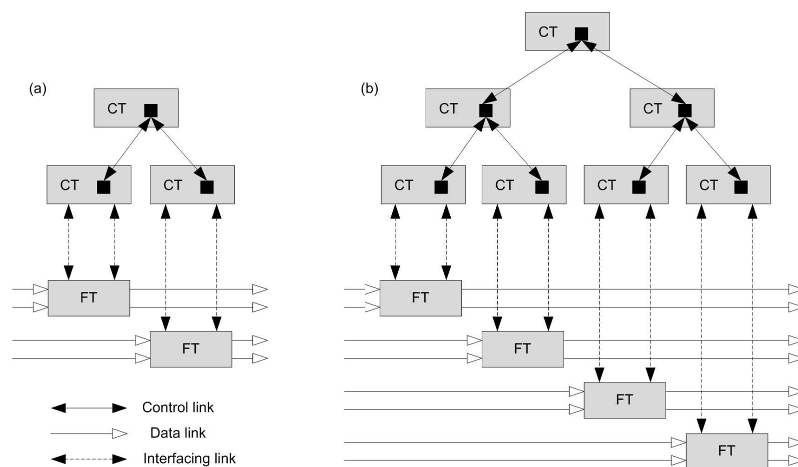


Figure 3.13: Scaling the number of FTs and BTs from system (a) to system (b) leads to a predictable scaling of the number of CTs. In this example, all signal processing architecture templates have two interfaces because they include two processing units.

### 3.7.2 First-order architecture templates interfacing

When interfacing first-order architecture templates of the two parts, the interfacing occurs only when strictly necessary. Each processing unit in a signal processing template (FT or BT) is interfaced with a processing unit in a CT through a storage unit that is separated from the two architectures, such that the two architectures can operate at their own speed. A port is added in each FT, BT and CT for the interfacing of each processing unit [4]. The number of ports and storage units is a parameter of the interfacing. An example is shown in Figure 3.14, where the components that constitute the interfacing between the two architectures are represented in dark grey. In this example, two ports are added to the templates because there are two processing units in the FT and BT. Notice that the interfacing is not ad-hoc, which facilitates design scaling.

In the CTs, ports that are dedicated to the interfacing are connected directly to the SH com-

[4]This interfacing is different from the interfacing in the application model in the sense that several interfaces between control/monitoring tasks and signal processing tasks may be mapped onto a single communication unit.

munication unit that is shared between the processing units. The processing units in the CTs execute services to write command/parameters tokens sequentially and to read monitoring-data tokens sequentially to/from the storage unit that is dedicated to the interfacing. In the FTs and BTs, ports that are dedicated to the interfacing are connected to the processing unit through a communication unit that reads command/parameter tokens and writes monitoring-data tokens in parallel from/to the storage unit that is dedicated to the interfacing. With this approach, communication latencies are not significantly altered by the interfacing in the two networks.

Simulating the dynamic behavior of the interfacing at lower levels of the hierarchy permits ensuring that requirements are met in the interfacing. Also, it allows to get the actual timing of the periods in the signal processing network, after which the periods of the pulses that are superimposed to the control and monitoring network can be computed as detailed in chapter 2.

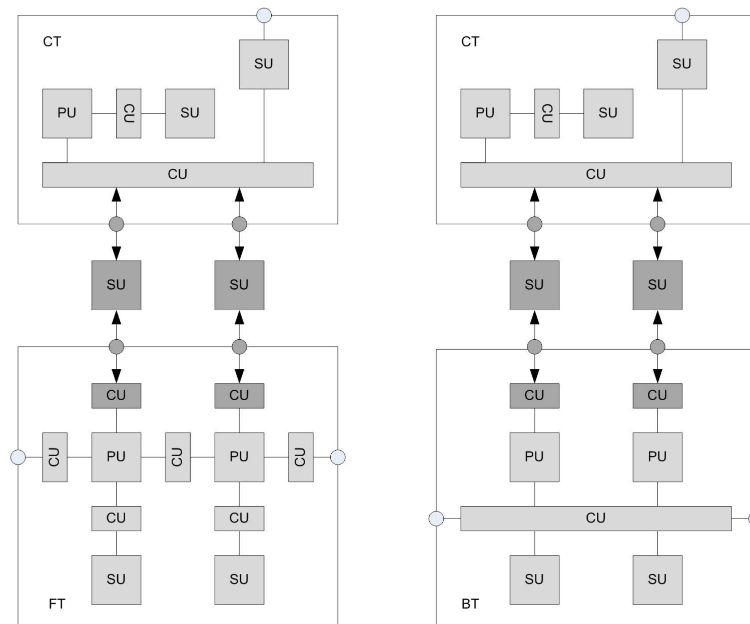


Figure 3.14: Modeling of the interfacing between signal processing architecture templates and control and monitoring architecture templates. The number of interfaces corresponds to the number of processing units that are internal to the signal processing architecture templates. Resources that constitute the interfacing are represented in dark grey.

## 3.8 Related work

The Software Communicating Architecture (SCA [63]) is an object-oriented framework that was established to enhance interoperability, upgradeability and scalability of communication among heterogeneous Joint Tactical Radio Systems (JTRS) while reducing development and

deployment costs. SCA separates applications from operating environments by defining common services to support device and application portability. We consider systems that are more homogeneous such that their non-functional behavior can be modeled and their performance predicted by composing with components from a unique library.

In [39], an internal/external component interface concept is presented. When viewed from outside, only the external structure and behavior are available. Implementation details are hidden inside the package. The internal design and external properties of a component can be viewed at different levels of abstraction. Also, the principle of a performance model is presented at the multiprocessor network level. The internal structure of the components is not usually described in this model, whose primary purpose is to determine sufficiency of the following selections in meeting the system processing throughput and latency requirements: the number and type of elements, the size of memories and buffers, the network topology, network bandwidths and protocols, application partitioning, mapping, scheduling of tasks onto processor elements.

The Berkeley Emulation Engine (BEE2 [29] and the LOFAR Remote Station Signal Processing platform (RSP3 [64]) are FPGA-based hardware architectures that can support radio-telescope signal processing applications. Both architectures consist of four FPGAs that are connected in a ring topology and that are interfaced with another FPGA that is dedicated to control. In RSP3, this FPGA belongs to the ring, whereas in BEE2 it is interfaced with the four FPGAs via point-to-point channels, such that control and signal processing links are separated.

## 3.9 Conclusions

In this chapter we presented our approach to model the architecture of the large-scale and distributed digital signal processing systems we consider. We modeled the signal processing architecture separately from the control and monitoring architecture. The former supports concurrent computations and transport of high-throughput parallel data streams. The latter supports sequential execution of instructions in reaction to sporadic events. The two architecture models and their interfacing rely on components that are taken from a unique platform. This platform permits predicting the performance/cost of the two parts based on simulation of components that are modeled as white boxes on lower levels of the hierarchy, and based on equations that relate output quantities to input quantities of components that are modeled as black boxes at higher levels of the hierarchy. The modeling of the interfacing between the two architectures is based on dedicated point-to-point links between processing units at the lowest hierarchical level of the two architectures, such that the two parts can still be interfaced in a structured way when scaling the system. This simple interfacing leads to an architecture model that can serve as an input to map the application model as discussed in the next chapter.

Chapter 4

# Mapping

## 4.1 Summary

A system-level specification consists of an application specification, an architecture specification, and the mapping of the former onto the latter. In this chapter we discuss the mapping of large-scale and distributed digital signal processing systems. When mapping, our objective is twofold: we want to implement the systems, and we want to analyze their performance and cost. Analysis concerns a model-based search for attractive points in the design space in terms of performance and cost metrics. At lower levels of the hierarchy, the values of these metrics are obtained from a calibration based on simulation or prototyping of actual components. At higher levels of the hierarchy, the values are obtained in an analytic way based on simple equations that relate output quantities to input quantities. Implementation, on the other hand, is a translating process that takes abstract system-level specifications to a lower level of abstraction, which we call implementation-level specification, from where implementation should become straightforward. Analysis and implementation should match in the sense that the output of the analysis phase should be a valid input for the implementation phase.

In this chapter we assume that mapping transformations are available in a library, and that they can be called iteratively and in any order (interactively) to associate items in the application model together with items in the architecture model on all levels of the hierarchy. These iterative transformations constitute high-level compilation steps above implementation-level specifications, which standard tools should be able to convert automatically to implementations in components. Mapping transformations are restricted by the interfacing between the signal processing part and the control and monitoring part. Once performance/cost predictions are satisfactory on all levels of the hierarchy and the last system-level specification includes all the information that is necessary to go to implementation, this specification is translated automatically to an implementation-level specification, from where we assume that different tools can take over to implement different parts of the system.

## 4.2 Introduction

Recall that a system consists of an application, an architecture and a (mapping) relation between the two. We specify the application and the architecture based on models, and separately under the assumption that the two specifications are specific to the same application domain, and that they match in the sense that if the architecture of a large scale and distributed system has a high degree of parallelism, then the application that is to be mapped onto it should be specified in some parallel language. Thus, we reason on the properties of the application (functional behavior) and the architecture (non-functional behavior) in isolation on levels of abstraction and hierarchy. [1]

In the application, concurrent processing and communication of tokens between nodes are specified unambiguously based on the operational semantics of (mathematical) models of computation. In the architecture, operations on tokens and transport of tokens are supported by admissible compositions of processing, communication and storage units whose performance/cost properties are known. However, the matching between the items in the application specification and the items in the architecture specification is not complete. The application and architecture specifications must be transformed so as to improve the matching. Such transformations are called *mapping* transformations. As shown in Figure 4.1, when mapping an application to an architecture, two directions can be taken: analysis, and implementation. Analysis concerns a model-based and iterative search for attractive points in the design space in terms of performance and cost metrics. This is called design space exploration [2]. Implementation, on the other hand, is a translating process that takes abstract system-level model-based specifications to a lower level of abstraction, which we call implementation-level specification, from where commercially available compilation and synthesis tools should be able to take over to obtain real implementations.

We would like to keep analysis and implementation concerns separated. However, the analysis phase is part of a design trajectory where the implementation phase is the last phase. In the analysis phase, parameter values can change in the functional model, non-functional model, and selected mapping transformations depending on the performance and cost numbers obtained in an exploration cycle (dotted arrows in Figure 4.1). In the implementation phase, parameter values are fixed, and the models and mapping specifications obtained after the last iteration of the analysis phase are translated automatically to an implementation-level specification, from where commercially available tools can take over. Thus, the final specifications output by the analysis phase have to be compatible with the input of the implementation phase.

We restrict ourselves to mapping transformations on abstract levels above the Register Transfer Level (*RTL*). On the level of basic components that are modeled as white boxes (in the sense that their internal modules are accessible, see chapter 3), mapping transformations are selected based on performance and cost numbers that are obtained by simulating [58] or prototyping the behavior of the modules. These numbers are used to calibrate formulae that relate output quantities to input quantities at higher levels of the hierarchy, where compo-

[1] In our model-based approach, levels of abstraction are hierarchically ordered.

[2] In this thesis, the term design-space exploration mainly refers to the performance and cost analysis that is part of it. Exploration approaches and methods are not considered in detail.

nents are modeled as black boxes (in the sense that their internal modules are hidden). Thus, the mapping approach is neither entirely top-down nor entirely bottom-up, but rather a meet in the middle approach.

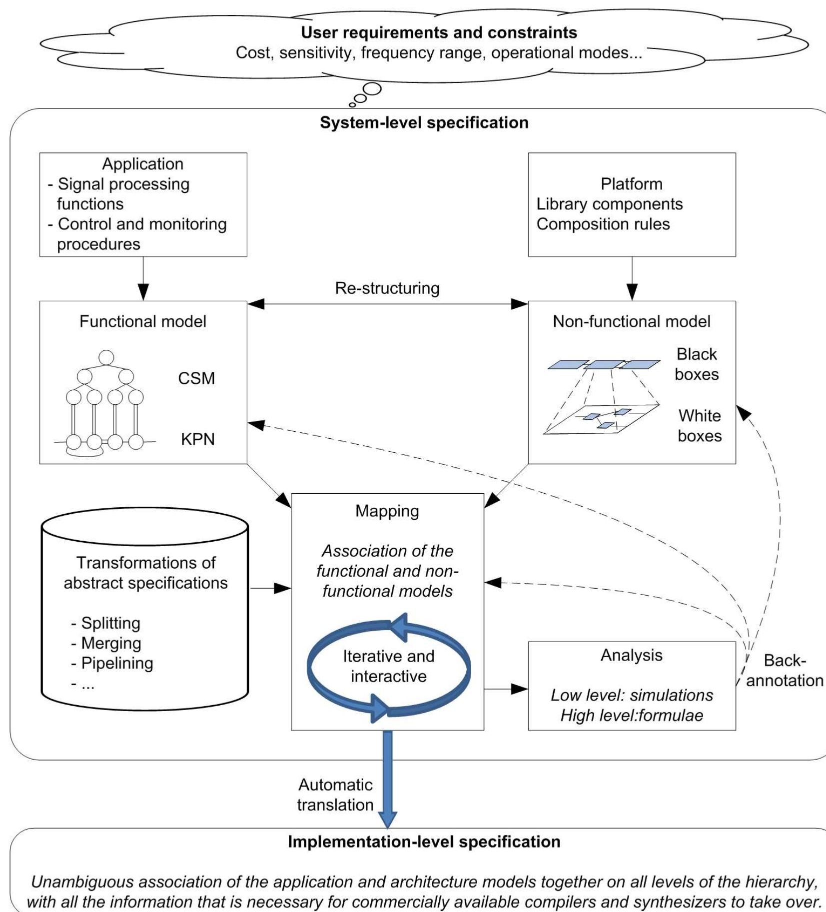


Figure 4.1: Mapping associates a functional model and a non-functional model together based on transformations that are taken from a library. From an implementation point of view, mapping transformations correspond to high-level compilation steps above the level of abstraction where standard compilers and synthesizers should be able to take over.

As we shall see throughout this chapter, the signal processing network and the control and monitoring network can only be partly separated when performing mapping, and this implies that some restrictions will apply that would not be necessary if the two parts were fully separated. The rest of the chapter is organized as follows. In section 4.3 we consider some transformations that permit improving the matching between the application an architecture models during the analysis phase. In section 4.4 we discuss the translation of the system-level specifications output by the analysis phase to implementation-level specifications during the implementation phase. We give our related work in section 4.5 and conclude in section 4.6.

## 4.3 Transformations

In this section we give the principle of some transformations that improve the matching between the application and the architecture specifications. Some of these transformations serve as high-level compilation steps above the level where standard compilers and synthesizers can take over to obtain the actual implementation. Recall that we assume that these transformations are available in a library and that they can be called in any order, on all levels of abstraction, which are hierarchically ordered as shown in Figure 4.2.

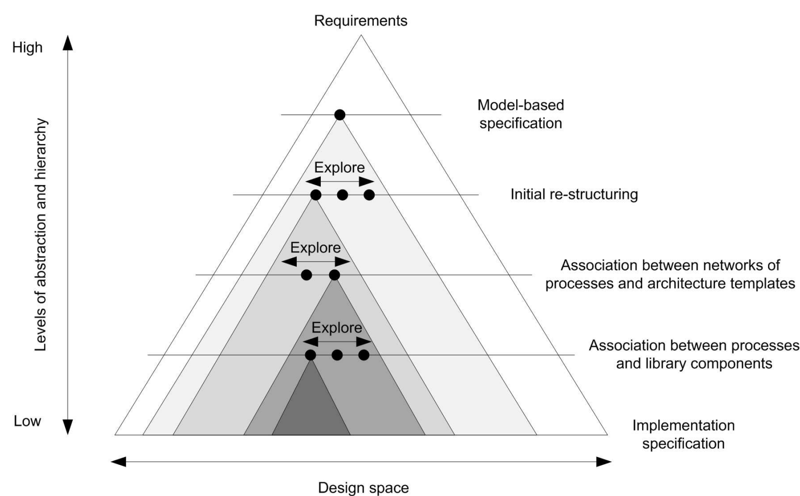


Figure 4.2: The association between the application and architecture is refined across levels of abstraction based on mapping transformations that are taken from a library.

We first give the principle of (re-)structuring transformations, which fix a few parameter values in the initial models based on coarse estimations. The output of the (re-)structuring transformation serves as an input to the analysis phase. This *analysis phase* progressively fixes remaining parameter values. Some mapping transformations operate on the abstract $Read$, $Execute$ and $Write$ instructions that are executed in nodes in the control and monitoring part and processes in the signal processing part, in order to improve their matching with specialized services that are supported by component models. Some of the mapping transformations are also known in standard compilers, and are used to optimize a system on a particular level of abstraction.

### 4.3.1 Initialization (re-)structuring transformations

The application and architecture models have to be manipulated with (re-)structuring transformations to obtain an initial configuration from which the analysis phase can start [65] [3]. These transformations improve the initial matching in terms of quantities related to perfor-

[3] In [65], (re-)structuring is called (re-)configuration.

mance and cost by changing the structure in the functional and non-functional models based on intuitive (experience-based) estimations of the system.

Note that (re-)structuring transformations can also be applied based on mapping transformations, where decision taking relies on feedback information that is obtained after exploration cycles, i.e., after simulating the behavior of the system at lower levels of the hierarchy, or after analyzing this behavior based on formulae at higher levels.

### Application (re-)structuring

The initial structure of the application model may be manipulated so as to improve the matching with the architecture as long as the functional behavior is preserved.

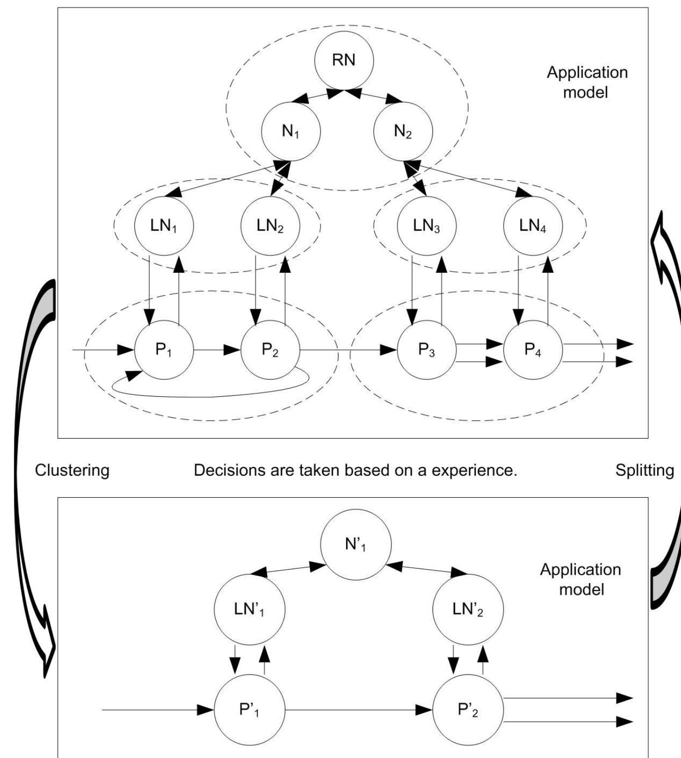


Figure 4.3: Example of initial application (re-)structuring.

*Clustering* is an example of such manipulation. It aims at reducing the concurrency in the application model based on coarse estimations so as to meet particular resource constraints and/or load balancing constraints from the architecture model. The converse manipulation, i.e., process-splitting, is another example of such manipulation. Theses types of transformations on a high level of abstraction are addressed in [66], where the transformations are applied on initial sequential specifications after which parallel specifications are automati-

cally derived. Figure 4.3 shows an example where the processes $P_1$ and $P_2$, and $P_3$ and $P_4$ in the signal processing part are clustered in $P_1'$ and $P_2'$, respectively. In the control and monitoring part, the leave nodes $LN_1$ and $LN_2$, and $LN_3$ and $LN_4$ are clustered in $LN_1'$ and $LN_2$, respectively, and the intermediate nodes $N_1$ and $N_2$ and the root node $RN$ are clustered in $N_1'$.

### Architecture (re)-structuring

The architecture is modeled as a composition of parameterized architecture templates, which are admissible compositions of black-box/white-box components for processing, storage and communication as introduced in chapter 3. Architecture (re-)structuring aims at choosing particular initial architecture templates by setting intuitively (based on experience) some parameter values to meet performance and cost requirements. For example, low latency requirements in terms of communication may lead to the architecture that is shown on top in Figure 4.4, whereas the architecture that is shown at the bottom may be preferred if the application that is to be mapped onto it is likely to exploit complex communication mechanisms.

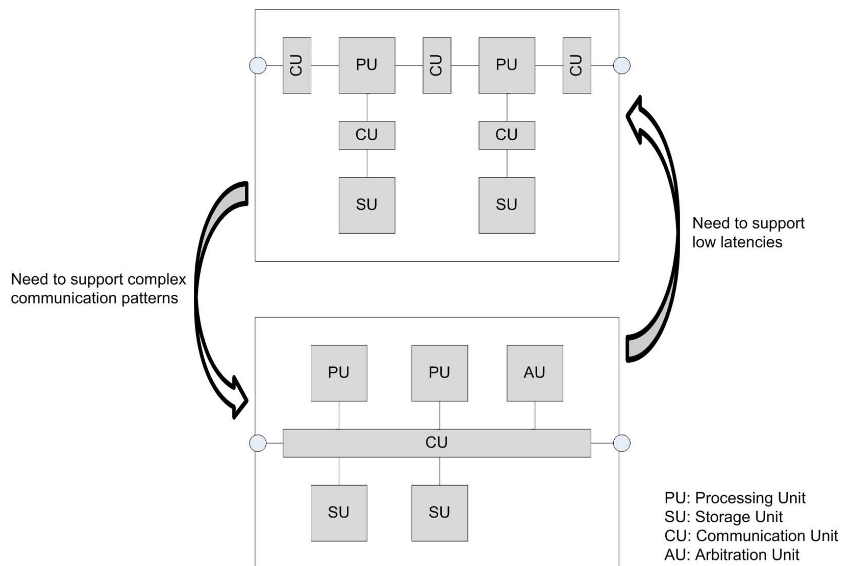


Figure 4.4: Example of initial architecture re-structuring.

Parameter values that are not fixed by the (re-)structuring transformations will be fixed iteratively with mapping transformations in the analysis phase, where decisions will be taken based on performance/cost information that is obtained analytically rather than based on estimations.

### 4.3.2 Mapping transformations

In the previous subsection we discussed the interest of (re-)structuring application and architecture specifications to obtain an estimated good match as a starting point to explore the design space, and to go to implementation-level specifications. In this subsection we give the principle of a few mapping transformations that may be used to improve the matching between the application and architecture, both in terms of abstraction of items and in terms of performance/cost. We assume that these mapping transformations are available in a library and that they can be called by designers in any order. With respect to our analysis objective, mapping transformations allow to set parameter values on all levels of the hierarchy based on predictions returned after each exploration cycle. With respect to our implementation objective, mapping transformations correspond to high-level compilation steps above the level of abstraction where we assume that standard compilers and synthesizers can take over to implement (parts of) the actual system.

We give the principle of four types of mapping transformations (assignment, process splitting, instruction refining and instruction re-ordering) and we pay attention to the interfacing between the signal processing part and the control and monitoring part, which plays a role in terms of restrictions.

#### Assignment

The *assignment* is the first mapping transformation that is applied in the analysis phase. This transformation permits designers assigning computation and communication components in the application model to computation and communication components in the architecture model. Designers assign (networks of) processes to (networks of) architecture templates. We use the symbol $\rightarrow$ to represent an assignment. (Networks of) Kahn processes (KPN [35]) are assigned to (networks of) signal processing architecture templates (ST), and (networks of) communicating state machine (CSM, as introduced in chapter 2) processes are assigned to (networks of) control and monitoring architecture templates (CT):

$KPN \rightarrow ST$, and $CSM \rightarrow CT$

The assignment is restricted by the interfacing between the signal processing part and the control and monitoring part. If CSM processes communicate with KPN processes in the application model, then these CSM processes must be mapped onto CTs that are interfaced with the STs the KPN processes are mapped onto. This simple restriction is checked by the assignment transformation. Also, a consequence of the assignment transformation is a binding in terms of abstract $Read$, $Execute$ and $Write$ instructions. This binding is modeled by means of traces, which are derived automatically from the assignment. Traces can be manipulated by other mapping transformations to further improve the matching.

In the example shown in Figure 4.5, two KPNs are assigned to two STs ($KPN_1 \rightarrow ST_1$ and $KPN_2 \rightarrow ST_2$), and the two complementary sets of leave nodes are assigned to the two complementary CTs ($CSM_1 \rightarrow CT_1$ and $CSM_2 \rightarrow CT_2$). In this example, the assignment of the nodes above in the control and monitoring network ($CSM_3 \rightarrow CT_3$) is constrained by the fact that $CSM_3$ is a root for $CSM_1$ and $CSM_2$ and must be interfaced with these two

sets through single access points (which are represented with black squares in Figure 4.5).

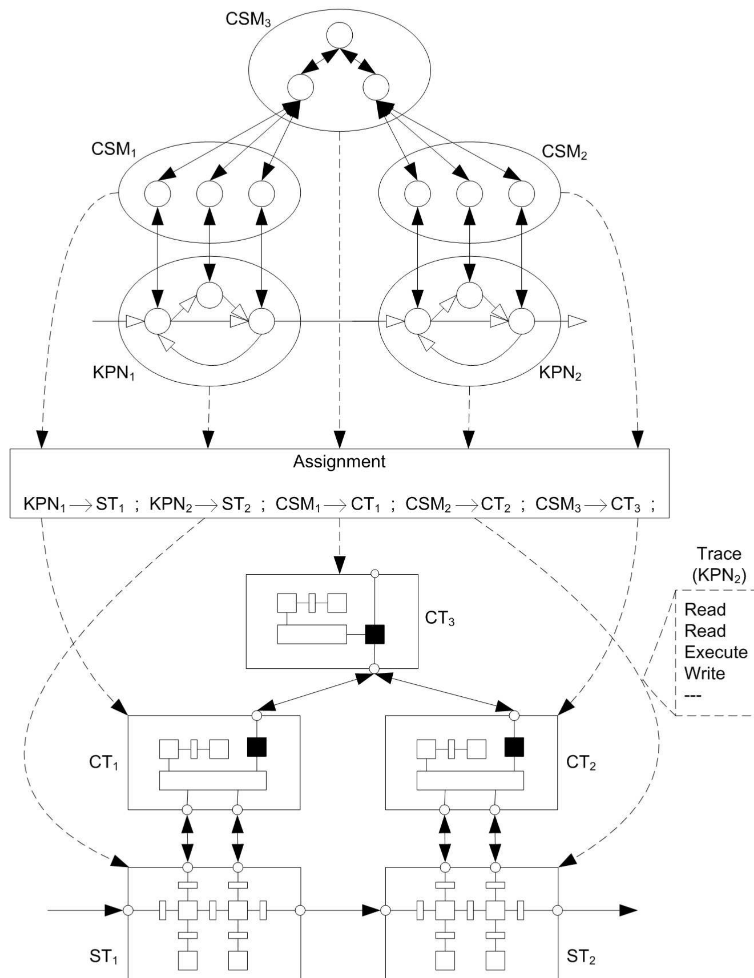


Figure 4.5: Example of assignment of networks of processes and sets of nodes to architecture templates. The assignment is restricted by the interfacing between the signal processing part and the control and monitoring part. A consequence of the assignment is a binding in terms of abstract instructions. This binding is modeled by means of traces.

### Process splitting

Process splitting is a mapping transformation that increases the degree of parallelism in a (network of) process(es). This transformation can be applied when disclosing more details in hierarchical components at lower levels, or to optimize the performance/cost of a system at a particular level by exploiting the parallelism that is available in the architecture.

Details of this transformation can be found in [67], where any process in the signal processing network can be split in a parameterized number of processes, in three steps. The first step duplicates a process according to a splitting factor that is given by a designer, and re-structures the code that is executed in each duplicate process, by introducing modulo conditional expressions. The second step adapts the number of ports in processes that are connected to the original process and that are affected by the first step. The third step refines the second step by removing some ports and channels where data will never be communicated.

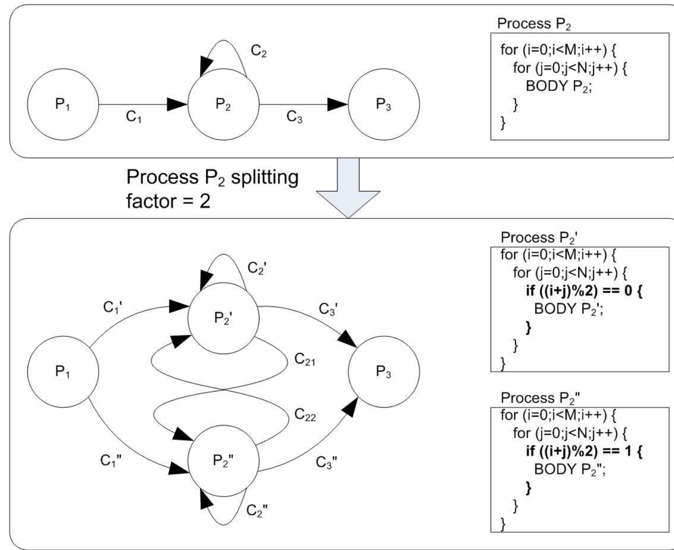


Figure 4.6: Example of process splitting transformation in the signal processing network.

Figure 4.6 gives an example where process $P_2$ is split in two processes $P_2'$ and $P_2''$. Modulo conditional expressions (represented with the symbol %) are added around the body in the code that is executed in $P_2'$ and $P_2''$. Communication channels $C_1$ and $C_3$ are duplicated in $\{C_1', C_1''\}$ and $\{C_3', C_3''\}$, respectively. The feedback channel $C_2$ is duplicated in $\{C_2', C_2''\}$ that are local to $P_2'$ and $P_2''$, and $\{C_{21}, C_{22}''\}$ that connect $P_2'$ and $P_2''$.

Splitting a process $P$ with a factor $k$ in the signal processing network implies a complementary splitting transformation of the leaf node $LN$ that is attached to $P$, such that each duplicate process is attached to a dedicated duplicate leaf node in the control and monitoring network. This splitting transformation also has an impact on the intermediate node above the duplicate leave nodes, in the sense that procedures that were $LN$-specific have to be split into $k$ specific procedures, and that each duplicate leaf node has to be interfaced with that intermediate node through dedicated communication channels. Note that the periods $T_k$ of the duplicate leave nodes may be different from the period of the original leaf node. In such a case, the relation between the periods in the control and monitoring network have to be updated after a splitting transformation, based on the equations given in chapter 2.

Figure 4.7 shows an example of leaf node splitting for a factor $k = 2$, to complement the splitting transformation shown in Figure 4.6. For the sake of clarity, we do not show the

communication channels in the signal processing network, and we do not show the timing network that is superimposed to the control and monitoring network. In this example, the intermediate node $N_1$ needs to send a control packet to $LN_2'$ and $LN_2''$ after the transformation, instead of a single packet to $LN_2$ before the transformation, and both $LN_2'$ and $LN_2''$ execute a procedure ($ProcLN_2'$ and $ProcLN_2''$, respectively) instead of a single procedure ($ProcLN_2$) in $LN_2$ before the transformation.

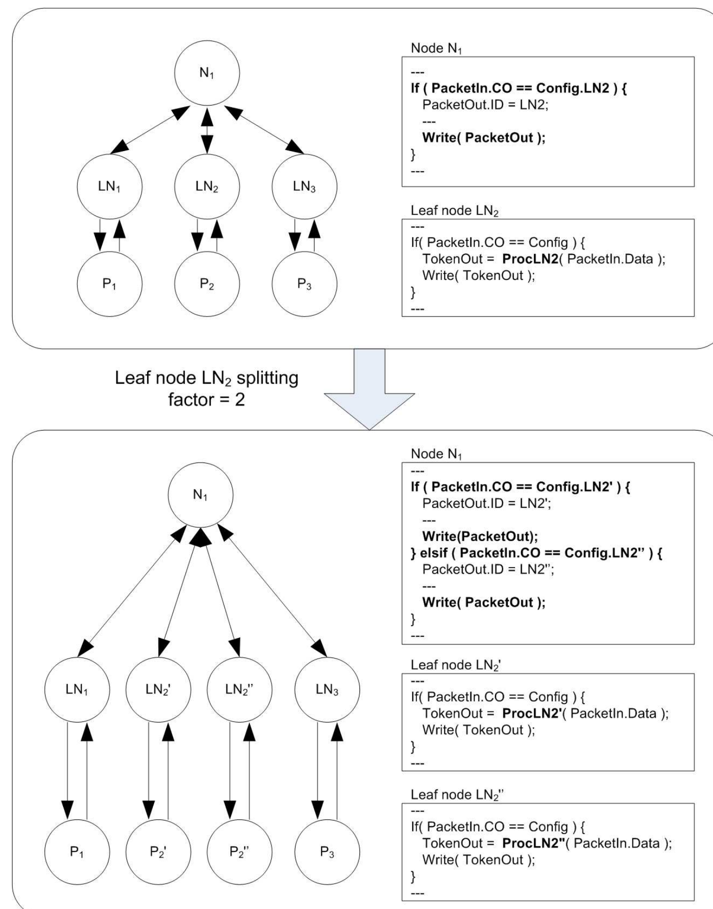


Figure 4.7: Example of complementary splitting transformation in the control and monitoring network after a splitting transformation in the signal processing network.

### Instruction refining

Recall that we do not permit mapping a process or node in the application model to more than one component in the architecture model. However, abstract instructions that are executed by a (network of) process(es) or (set of) node(s) in the application may not match the specialized

services of a unique component in the architecture. Abstract instructions that are executed in a process may have to be refined so as to disclose more details about that process, such that each refined instruction can be mapped onto a processing unit, storage unit or communication unit in the architecture model.

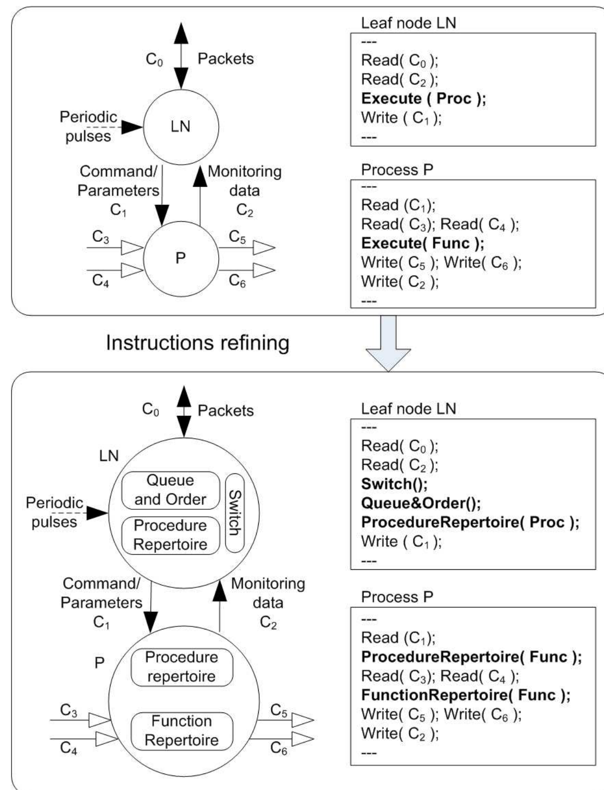


Figure 4.8: The $Execute$ instruction in a leaf node or process in the application model can be refined as a sequence of instructions to improve the matching with the architecture model.

As already discussed in chapter 2, an abstract $Execute$ instruction may be refined as a sequence of $Switch$, $Queue\text{-}and\text{-}Order$ and $Procedure\text{-}Repertoire$ instructions in nodes in the control and monitoring network, and to complementary $Function\text{-}Repertoire$ and $Procedure\text{-}Repertoire$ instructions in processes in the signal processing network. As a consequence, the bindings in terms of abstract instructions (traces) are modified by this transformation. These new traces can be manipulated by other mapping transformations to further improve the matching. An example of instruction refining transformation is given in Figure 4.8 for a leaf node $LN$ that executes a control procedure $Proc$ and for the attached process $P$ that executes a signal processing function $Func$.

### Instruction re-ordering

In contrast to the instruction refining transformation, the *instruction re-ordering* transformation does not involve disclosing more details about the application or architecture models, but changes the ordering of instructions at a particular level of abstraction to improve the performance of a system (load balancing). Instructions are re-ordered by designers based on experience and assumptions. At lower levels of abstraction, the re-ordering is accepted (or not) after simulating the behavior of the system or by monitoring the dynamic behavior of a prototype implementation that provides actual performance/cost numbers. At higher levels, the re-ordering is accepted (or not) after analyzing the performance/cost numbers that are returned by simple equations that relate output quantities to input quantities.

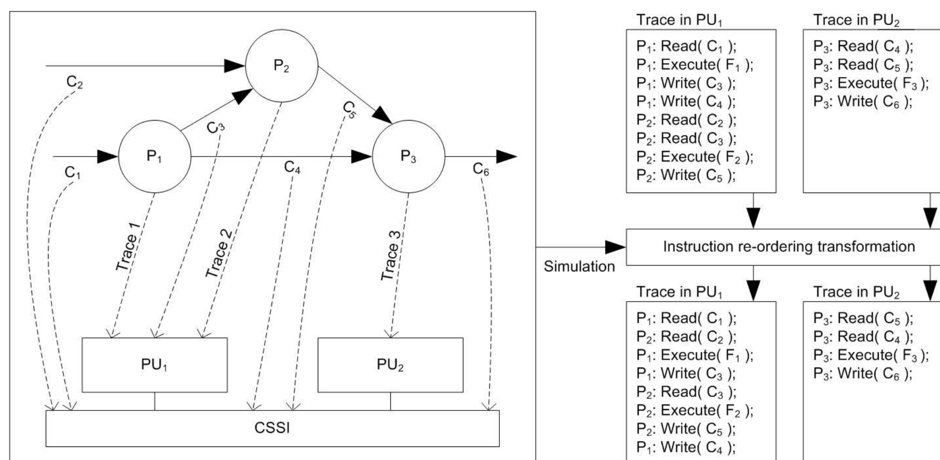


Figure 4.9: Example of instruction re-ordering transformation, which allows to optimize the performance/cost of the (sub)system. The re-ordering is accepted (or rejected) after simulation (at lower levels of the hierarchy) or formulae-based analysis (at higher levels).

In the example shown in Figure 4.9, a network of three processes ($P_1$, $P_2$ and $P_3$) is mapped onto a simple architecture that consists of the processing units $PU_1$ and $PU_2$ that are interfaced through the communication, synchronization and storage infrastructure $CSSI$ as follows: $\{P_1, P_2\} \rightarrow PU_1$, and $P_3 \rightarrow PU_2$ (see dotted arrows in Figure 4.9). Before applying the re-ordering transformation, all processes are executed sequentially in all processing units as shown on top in Figure 4.9, and the performance/cost of the corresponding system is known. In this example we change the order among abstract $Read$, $Execute$ and $Write$ instructions as shown at the bottom in Figure 4.9. The performance/cost of the corresponding transformation is obtained through simulation or prototyping (at lower levels of the hierarchy) or based on simple equations (at higher levels). If performance/cost results are more satisfactory than with the previous ordering, the transformation is accepted.

## 4.4 Implementation phase

The role of the *implementation phase* is to translate specifications output by the analysis phase to implementation-level specifications, i.e., to specifications that commercially available compilers and synthesizers should be able to convert to an actual implementation. The analysis phase and implementation phase are related in the sense that simulating the behavior of implementations at lower levels of the hierarchy leads to actual performance and cost numbers, which are used to calibrate the analytical models at higher levels of the hierarchy. Thus, the two phases belong to a meet in the middle design approach.

In this section we first give the type of input to the implementation phase. Then we give the principle to translate such an input (specification) to an implementation-level specification, from where implementation should become straightforward based on standard tools. We evaluate the capacity to go from abstract implementation-level specifications to real implementation in chapter 5.

### 4.4.1 Input to the implementation phase

At the input of the implementation phase, the application and architecture models are different from the initial application and architecture models at the input of the analysis phase. Indeed, the initial specification had to be transformed so as to improve the matching. In the system-level specification output by the analysis phase, the application and architecture models match, and predictions in terms of performance/cost are satisfactory. This unambiguous system-level specification serves as an input to the implementation phase, and consists of three parts:

- A final abstract application specification in terms of processes that execute sequences of instructions and that communicate through channels in a network.
- A final abstract architecture specification in terms of interconnected library components that have fixed parameter values and that expose specialized services.
- A mapping specification that corresponds to the assignment of each process and channel in the final application specification to a unique component in the final architecture specification.

This system-level specification is complete in the sense that it includes all the information that is necessary to be translated automatically to an implementation-level specification, without modifying the input specification. This is a major difference with the analysis phase, where specifications are modified iteratively after each mapping transformation.

In the example shown in Figure 4.10, the application is specified as a network of three processes ($P_1, P_2, P_3$) that execute sequences of abstract $Read$, $Execute$ and $Write$ instructions. The architecture specification consists of the processing units $PU_1$ (FPGA) and $PU_2$ (DSP) that communicate through a communication, synchronization and storage infrastructure ($CSSI$), which includes a bus, an arbitration unit $AU$ and a storage unit $SU$ (RAM).

Each process is assigned to a processing unit, and each communication channel is assigned to a storage unit in $CSSI$, except channel $C_2$ that is assigned to $PU_1$'s internal memory since it connects $P_1$ and $P_2$ that are assigned to $PU_1$.

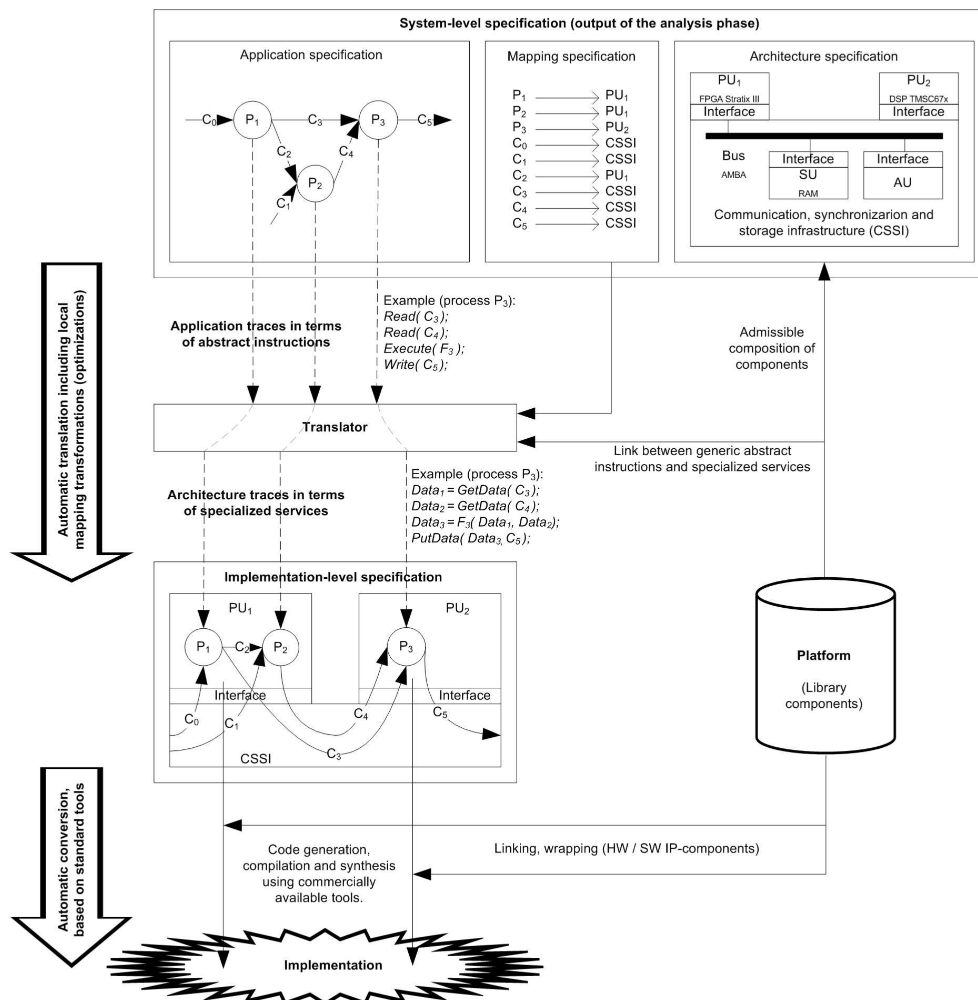


Figure 4.10: During the automatic translation of system-level specifications output by the analysis phase to implementation-level specification, optimizations are applied locally to some parts of the system. We assume that implementation-level specifications can be converted to actual implementation based on commercially available compilers and synthesizers that integrate HW/SW IP-components and generate glue logic between these components.

### 4.4.2 Automatic translation to implementation-level specification

A translator converts application traces, which are sequences of abstract $Read$, $Execute$ and $Write$ instructions for each process in the application specification, to architecture traces, which are sequences of specialized services for processing, communication and storage in the architecture specification. To do so, the translator needs a list of assignments of processes to components, and a list of specialized services that are supported by each component in the architecture model. These lists are provided by the mapping specification and component library, respectively, as shown in Figure 4.10.

Specialized services are implemented using HW/SW library (IP-)components, which are designed and owned by third parties. A specialized service may have different versions since it may be implemented with different (IP-)components that lead to different performance/cost tradeoffs. This performance/cost information is included in the library. During the translation, the selection of a particular service version leads to the selection of the corresponding (IP-)component in the implementation phase.

The input specification includes all the information that is necessary to be translated automatically to implementation-level specification. Nevertheless, mapping transformations can still be applied locally during the translation, e.g., to optimize local performance (reduce latencies, resource usage, etc) by selecting a particular IP-component to implement a service. Note that such a mapping transformation remains local and does not modify the input specification.

In the example shown in Figure 4.10, traces of abstract $Read$ and $Write$ instructions that are executed in the process $P_3$ are translated to specialized communication services ($GetData$, $PutData$) that are supported by the processing unit $PU_2$, and the abstract $Execute$ instruction is translated to a specialized processing service ($F_3$) that is supported by a SW (IP-)component. We assume that all IP-components that are used in the resulting implementation-level specification can be linked by commercially available tools to obtain an actual implementation.

#### Implementation

The implementation-level specification output by the translator is to be converted to an actual implementation. This final step is not done at once for the complete systems we consider because there is no tool that can deal with the large number of heterogeneous components. Instead, different parts of the system are implemented separately with the most adequate tools. Moreover, the implementation is done only at lower levels of the hierarchy (such as multiprocessor systems-on-chips), where standard tools do exist that can take over and that lead to close to optimal performance.

The implementation may involve code generation (programming) and compilation when targeting (embedded) processors [68]. It may also involve hardware synthesis. During this compilation/synthesis step, tools link (or wrap) HW/SW (IP-)components that implement the required services. For example, SW libraries (DLLs) may be linked when targeting the DSP that is part of the architecture in Figure 4.10, while RTL IP-components may be linked when synthesizing for the FPGA. Moreover, this final step inserts glue logic (or completion

logic) between (IP-)components whose interfaces are not compatible with each other so as to glue (IP-)components together. At lower levels of the hierarchy, the insertion of glue logic is done systematically by some tools that include their own (local) mapping transformations. This glue logic handles signals such as start/stop/reset/enable, read/write addresses, etc, that are different for different IPs. The systematic integration of IP-components and porting of IP-based designs across architectures will be investigated in case studies in chapter 5.

## 4.5 Related work

A complete framework is presented in [69] to design and program embedded multi-processors systems. This framework includes a task-level interface that permits developing parallel application models (with several communication mechanisms) and that can serve to implement applications on multi-processors architectures. However, this environment does not standardize the description of tasks at a high-level of abstraction.

In the Artemis framework [70], an application is specified as a sequential program, which is converted to a functionally equivalent KPN specification using Compaan [71]. The mapping layer supports transformations to refine the application specification between levels of abstraction so as to match the level of granularity of the underlying architecture model, and to perform quantitative performance analysis on levels of abstraction. The final specification is converted to a system on a chip implementation using Laura [72]. Nevertheless, the problem of implementing applications in large-scale architectures that require different specialized tools is not faced.

In Thales [31], signal processing applications are specified using nested loops and are mapped onto large-scale array signal processing systems. Loops are transformed so as to extract their cores, which correspond to functions that are available in a library and that can be implemented in different components. Mapping is human-driven: commands are proposed to the user for application partitioning and allocation, insertion of communications, fusion of tasks and scheduling.

The Model-Driven Architecture framework (MDA [22]) permits linking object models together when building (possibly large-scale and distributed [73]) systems. The MDA design process starts from a UML-specification [74] of a Platform Independent Model (PIM) that is marked so as to obtain Platform Specific Models (PSMs) for mapping onto different software platforms. MDA incorporates automatic mapping transformations into the early stage of software development. The systems Modeling Language (SysML [75]) is another UML-based modeling language for specifying, analyzing, designing, and verifying complex systems that include hardware and software. SysML provides a semantic foundation for modeling system requirements, behavior, structure, and integration. Constraints on performance can be captured and serve as a means to integrate the specification and design models with engineering analysis models.

## 4.6 Conclusions

In this chapter we discussed the mapping of the application model onto the architecture model for the large-scale and distributed digital signal processing systems we consider, with the objective of analyzing performance/cost and implementing the systems. We presented mapping transformations that can be taken from a library to iteratively improve the matching between the two models on all levels of the hierarchy during the analysis phase. The association between the application and architecture models is restricted by the interfacing between the signal processing part and the control and monitoring part. The output of the analysis phase serves as an input to the implementation phase.

In the implementation phase, abstract instructions that are executed in processes in the final application specification are translated automatically to specialized services that are supported by library components in the final architecture specification. During the translation, mapping transformations (high-level compilation steps) can be applied for local performance/cost optimization, without modifying the input specification. The output of the translation is an implementation-level specification, which still has to be converted to an actual implementation based on commercially available tools. These tools implement specialized services based on (IP-)components. Case studies on the integration and porting of (IP-)components are reported in the next chapter.

Chapter 5

# Case studies

## 5.1 Summary

As far as the dominant signal processing part of the system is concerned, a down-scaled version of the digital data-reduction subsystem in a station has been modeled, analyzed and implemented as described in the previous chapters. Details can be found in [16]. We did not model and implement the control and monitoring part for the same subsystem because there was no option to construct the integrated system. Instead, we targeted multiprocessor systems-on-chip architectures for which implementation tools do exist. We used these tools and assessed their ability to convert implementation-level specifications to actual implementations.

We have conducted a number of experiments that are focusing on problems related to 1) the systematic integration and re-use of IP components, which are designed and owned by third parties, 2) portability of designs across components, and 3) the interfacing between leave nodes in the control and monitoring part with processes in the signal processing part. The first issue is important because integrating IP components in a design causes almost always difficulties: ad-hoc glue logic is often required to deal with low-level signals and protocols due to the lack of standardization. The second issue comes from the fact that components in a library may come in types and versions that may evolve over time. The third issue is important because the interfacing is one of the main aspects in this thesis.

## 5.2 Introduction

When a digital signal processing system is implemented starting from model-based specifications of application and architecture, it is the case that constituent components in both specifications are pre-defined library components that are to be integrated in the ultimate implementation. For example, an application process may execute a function $f$ that is taken from an (application) function library $L_{AP}$ as $f_{AP}$. The corresponding architecture processor will take from the (architecture) routine library $L_{AR}$ the routine $f_{AR}$ to execute the function $f$. Both $f_{AP}$ and $f_{AR}$ are typically intellectual property (*IP*) components, which are designed and owned by third parties.

This IP integration concept is appealing as it should free the designer from detailed low-level design tasks that can be taken care of by IP design experts. However, this approach implies that IP components come with interfaces that comply to some standard, and that there is a unique functional relation between corresponding members in the two libraries $L_{AP}$ and $L_{AR}$. Neither of the two assumptions are practically satisfied. Two IPs that are functionally equivalent may come with different interfaces, which require IP component specific integration solutions. Similarly, there may be no $f_{AR}$ in the library $L_{AR}$ that is functionally equivalent to an $f_{AP}$ in the library $L_{AP}$. For example, $f_{AP}$ may be operating on vector-valued tokens, while $f_{AR}$ operates on scalar-valued tokens without including a vector to scalar conversion. Moreover, both mismatch issues hinder possible porting of implementations from an architecture (component) to another architecture (component) that may come from the same or different manufacturers.

One of the objectives in the path from system-level specification to implementation is to structure the design process as much as possible. A systematic design approach should allow to automate the integration of IP components, and should permit obtaining actual performance/cost numbers to calibrate the exploration of large-scale systems based on analytical performance/cost models. However, it may be clear now that there is a tension between the automation objective and the IP integration problem.

In this chapter we investigate the possibility to go from implementation-level specifications to real implementation based on standards tools. We evaluate the capacity of different commercial and academic tools to deal with the systematic integration of IP components, porting of designs across architectures (components), and interfacing between control and monitoring leave nodes and signal processing processes, based on case studies. The remainder of this chapter is organized as follows. In section 5.3 we present the experimental context and the setup we adopted to conduct our case studies. In section 5.4 we present two case studies on the re-use of IP components and development of glue logic in the dominant signal processing network only, and one case study on the re-use of IP components in the control and monitoring network only. In section 5.5 we present two case studies on the interfacing between the signal processing network and the control and monitoring network based on IP components. Finally we give related work in section 5.6 and conclusions in section 5.7.

## 5.3 Experimental context and setup

In this section, we give our implementation objectives in terms of performance and cost. Then we present the experimental setup we adopted to conduct our case studies.

### 5.3.1 Objectives

An application is to be mapped onto an architecture that is modeled as a composition of library components. The mapping needs to satisfy a number of performance and cost objectives that are summarized in Table 5.1. To reduce the architecture cost, we want to maximizing the throughput in the signal processing network, while minimizing resource usage in the signal processing network, in the control and monitoring network, and in their interfacing. Shortening the development time and supporting design scaling reduces the development cost.

| Maximize | Minimize |
|---|---|
| Throughput | Resource usage |
| Scalability | Development time |
| Re-usability | Manufacturer dependence |

Table 5.1: Performance and cost related objectives that must be satisfied when implementing large scale signal processing systems.

In principle, in the systems we consider, IP integration issues and porting issues are encountered on all levels of the hierarchy, in the signal processing part, in the control and monitoring part, and in their interfacing. In this chapter we conduct case studies to investigate different methods that deal with these issues.

### 5.3.2 Setup

To focus on the critical interfacing between leave nodes in the control and monitoring network, and processes in the signal processing network, we adopt the experimental setup that is shown in Figure 5.1. We consider two architectures, onto which we map the two networks separately. This setup permits facing the IP integration problem separately for the two networks, as well as for their interfacing. The mapping can be addressed on levels of the hierarchy ranging from a composition of components to the level of basic modules that are internal to the components. Also, the two architectures may be merged, such that parts of the application may have to be ported from one architecture to another. This permits facing the porting issue. For example, leave nodes may be ported to the architecture the signal processing network is mapped onto, such that the interfacing between the two networks also becomes internal to that architecture.

To avoid too much abstraction, we map each network onto separate architectures that are based on FPGAs for which IP components and automation tools do exist. We use these tools

and we assess their ability to convert implementation-level specifications to actual implementations. In section 5.4, we focus on the separate mapping of the two networks. In a first case study, we act as a skilled hardware IP designer and we develop our own glue logic to handcraft the porting of a design in the signal processing network only. In a second case study, we address the re-use, scaling and development time issues by relying on tools to semi-automate the mapping of signal processing applications, still based on hardware IP components. In a third case study, we focus on the re-use and porting of programmable IP components to map the control and monitoring network only. In section 5.5, we present two case studies around the main issue of the interfacing of the two networks. First, we evaluate the interest of an emerging standard to wrap and interface IP components similarly in the two networks. Then we quantify the effects of a domain specific interfacing between programmable IP components in the control and monitoring network, and hardware IP components in the signal processing network, and we focus on design scaling and performance in terms of throughput and resource usage.

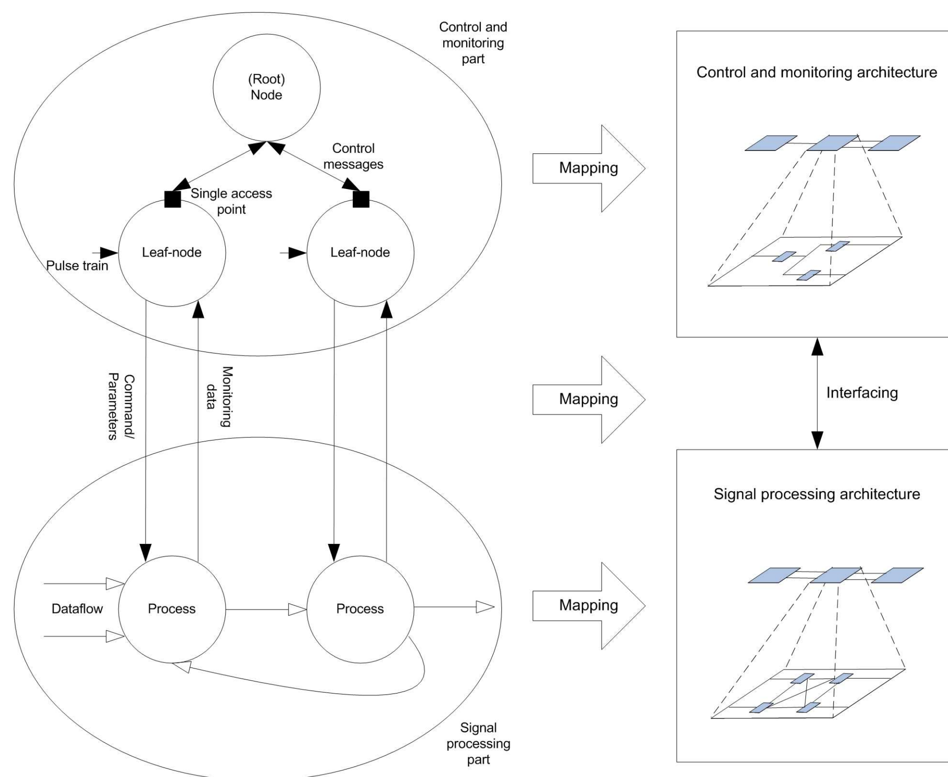


Figure 5.1: Experimental setup to map a partition of the application. The partition includes the interfacing between the signal processing part and the control and monitoring part.

## 5.4 Dedicated mapping of the two networks

In this section we present two case studies on the mapping of the signal processing network only, on a dedicated architecture, and a case study on the mapping of the control and monitoring network only, on a dedicated architecture as well. Too avoid too much abstraction, we consider architectures that are based on an FPGA for which IP components and implementation tools exist. In the first case study we handcraft the integration and porting of IP components. In the second case study we focus on the (re-)use, scaling and development time issue by semi-automating an implementation. The third case study focuses on the integration, porting and performance of programmable components. Finally we give requirements for better mapping strategies in our application domain.

### 5.4.1 Handcrafted design

As an example of a high throughput signal-processing algorithm we implemented a polyphase filterbank (FB). Roughly speaking, a polyphase filterbank consists of a number of finite impulse response (FIR) filters whose inputs are derived from a decimated stream of signal samples, and whose outputs are discrete Fourier transformed (DFT [1]) as shown in Figure 5.2.

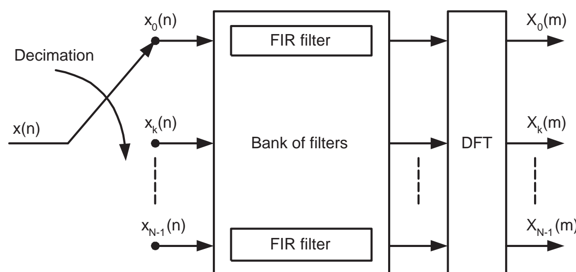


Figure 5.2: Functional diagram of a polyphase filterbank.

The implementation must be parallel in order to sustain a high throughput. Therefore, choices must be made concerning the degree of parallelism. These choices determine the partitioning of the application. In the FB application, the functions (decimation, filters, and DFT) are separated in blocks as is shown in Figure 5.2. Inside the filter block, filters are mapped on a set of N multiply-accumulate modules and memories. The DFT is a higher grain function that can be integrated as an IP module in the implementation. We propose to map the FB on an FPGA architecture from a first manufacturer (Stratix from Altera [12]) and to port it on another FPGA architecture from another manufacturer (Virtex-II Pro from Xilinx [13]). These two architectures offer distributed local memories and embedded multipliers that can be seen as low grain IPs.

The interface and granularity of the DFT IP that is used for the FB implementation are different for the two architectures. One implementation relies on a pipeline of 4 DFT IPs that

[1]The DFT is implemented as a Fast Fourier Transform (FFT), see [14]

access vectors stored in a buffer, whereas the other implementation relies on a single DFT IP that receives samples as tokens. Moreover, the input and output data formats are not equivalent for the two IPs. The interfacing with the filters, then, must be dedicated in order to match this specificity. In the filters, low grains IP (embedded multipliers, digital signal processing blocks or specific memory blocks) are also different for the two architectures. Thus, some glue logic (also called completion logic) is required to glue these IP components. The glue logic deals with local specific signals (start/stop/reset/enable, read/write addresses) to match IP specific communication and operation protocols.

| Target | IP cores | LE(k) | EM | Mem(kbits) | Speed(MS/s) |
|---|---|---|---|---|---|
| Stratix | 4, pipe, float | 10.5 | 32x9 bits | 200 | 80 |
| Virtex-II Pro | 1, fix | 7.5 | 12x18 bits | 200 | 80 |

Table 5.2: Polyphase filterbank benchmark on Stratix and Virtex-II Pro.

This local glue logic was handcrafted and had to be modified when porting the IP-based design from one FPGA to another. The handcrafted design and porting of the FB took approximately 4 months. About half of the time was spent on validations (including IP verification). The rest of the time was mainly spent on (handcrafted) glue logic development. The results concerning the porting of the FB application from a Stratix EP1S20 to a Virtex-II Pro XC2VP20 are given in Table 5.2. These results are given in terms of resource usage (logic elements: LE, embedded multipliers: EM , memory: Mem) and throughput (Million Samples per second) for 256 filters of 16 taps each, followed by a 256-point FFT. Given the same throughput requirements, the implementations satisfy the same objectives in terms of size, memory usage, etc., as defined in Table 5.2. The variations concerning the logic elements (cells) come from the fact that the 4 pipelined IPs (Stratix) need more cells than the single IP (Virtex-II Pro). Although handcrafted integration of IP components can lead to optimized performance in terms of throughput and resource usage, developing IP-specific glue logic is a time consuming and error prone activity.

### 5.4.2 Semi-automated design

In the previous subsection we have considered a hand-crafted FPGA implementation of a high-throughput signal processing application. We paid in particular attention to problems related to glue logic and portability when it comes to the integration of IP components, and to their effect on implementation cost in terms of effort and development time. In the last decade or so, several research groups in academia and industry have been proposing and prototyping methods and tools for the (semi-)automated mapping of applications into FPGAs. We want to assess if these methods can convert implementation-level specifications to implementation. We distinguish approaches that start from a schematic specification from those that start from a language-based specification. We tried one approach in each category. With the schematic approach, we again looked at the FB implementation and focused on the development time and design scaling issues. With the language-based approach, we focused on the use of mapping transformations (high-level compilation techniques, see chapter 4) to implement a

cross correlation function with complex interconnects. We detail these case studies in the remainder of this subsection.

### Schematic approach

One concept followed by the Electronic Design Automation (EDA) industry [76] [77] is a graphical entry of the functions in a schematic. We experienced the implementation of the FB on an FPGA starting from a graphical specification in an extension of the Matlab/Simulink [78] environment. This extension is tailored to implement signal processing applications in an FPGA from a unique manufacturer (DSP Builder from Altera). The approach consists of specifying a digital signal processing application graphically as a composition of library components in a single user-friendly development environment that encompasses simulation, verification and synthesis tools.

The library of IP components can be extended by importing handcrafted IP components such as the filter function that is part of the FB, and the associated glue logic. An implementation is specified by drawing point to point connections between ports of these representations. Note that a similar tool is available for Xilinx FPGAs (e.g., System Generator for DSP [13]), and that there are also tools that target either Altera or Xilinx from a unique specification in a similar environment (e.g., Synplify DSP from Simplicity [79]), therefore addressing the porting issue. We draw here some conclusions concerning this schematic approach.

Positive aspects:

- It allows to combine handcrafted portable glue logic and architecture-dependent IPs.
- It facilitates the integration of manufacturer IPs by providing a user friendly simulation, verification, synthesis and debugging environment.

Limitations:

- Simulation is bit and cycle accurate on all levels of the hierarchy. This makes it hard to abstract from low level details when moving up in the hierarchy.
- The functions are connected and parameterized graphically. This is a restriction when the system must be scaled because new connections must be inserted by hand.

These tools have been used successfully to fast-prototype the FB in a few days. This development time is more satisfactory than the handcrafted implementation, and enables obtaining actual performance and cost numbers rapidly. The performance in terms of throughput and resource usage is slightly lower than in the handcrafted implementation, because the single environment hides some compilation and synthesis parameters that were accessible in the handcrafted implementation. Moreover, these tools are not efficient to specify and verify applications with asynchronous communication schemes. Some language-based approaches address this limitation.

### Language-based approach

In language-based approaches such as Compaan [80], dataflow applications are specified in a language such as Matlab or C. Applications that are specified in Compaan can be converted to GALS (Globally Synchronous Locally Synchronous) implementations. We used the Compaan tool to extract the parallelism of a cross-correlation application, which requires more complex communication mechanisms than in the previous case studies. Compaan starts from a specification in the form of a nested loop algorithm in a subset of Matlab, and compiles specifications written in this language into a Compaan Process Network specification (CPN, a special case of KPN).

During the compilation step, designers can apply mapping transformations such as process splitting and channel merging. This allows to obtain different CPN specifications with particular degrees of parallelism starting from a unique specification. The CPN specifications can be simulated and analyzed in the Ptolemy II framework [26]. This approach is convenient to explore the mapping of large applications since the verification is done based on operational semantics rather than based on overwhelming implementation details. Figure 5.3 shows two possible process networks for our cross correlation application. The configuration on the left-hand side uses three correlator IPs in parallel. The data distribution is relatively simple for this network. The configuration on the right-hand side needs only one correlator IP to perform the cross-correlation of the signals X and Y. In this case, the data distribution is more complex and requires re-ordering of tokens. This operation is done automatically in Compaan.

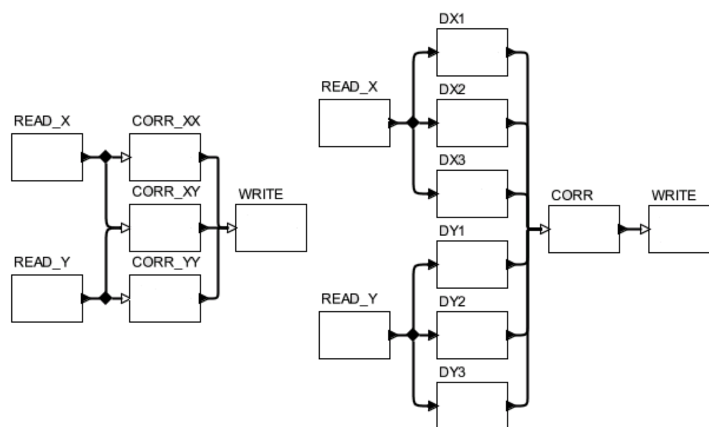


Figure 5.3: Re-use of correlator IP for cross-correlation.

The CPN representation output by Compaan can be converted to an FPGA implementation using Laura [72] or ESPAM [81], where part of the conversion of high-level specifications to implementation specification is automated. Laura generates the skeleton of the partitioned application and the glue logic corresponding to the data distribution automatically. Hardware IP components such as the correlator IP are taken from a library and are integrated by the designers in nodes where a local controller, which is generated automatically, governs their

behavior and their interaction with other nodes via FIFOs, which are also generated automatically. Thus, IP components can be upgraded or scaled independently of each other. The development time, including the analysis phase, is comparable to that of an implementation that starts from graphical specifications.

Positive:

- Compaan provides means to transform sequential specifications to a parallel representation in a correct by construction manner, and shortens the time to analyze the partitioning.
- Laura generates the skeleton of the network to encapsulate and separate IPs, and local controllers to handle asynchronous communication between IPs.

Limitations:

- The local control logic that governs the IPs induces a 35% increase in resource usage compared to the standalone IPs, and reduces the operating frequency of the IPs to 60% of their maximum frequency in this case study.
- Inserting IPs in nodes must be repeated (by hand) when scaling the system.

In particular, if the requirements are changed, the controller and FIFOs can be generated automatically again, but the IPs need to be re-inserted and the integrated design re-verified. Nevertheless, this approach helps to scale designs in an efficient way by modifying only the high-level specifications. Indeed, it was possible to scale the correlator IP to build a cross-correlator simply by modifying the specifications of the Matlab code (entry of Compaan). Even if the Compaan-Laura tool chain is currently FPGA dependent, porting of applications across FPGAs from different manufacturers could be supported by this approach in the future.

### 5.4.3 Integration of programmable IP components

In the previous case studies, we investigated different methods to integrate and port hardware IP components when mapping the signal processing network. There are also programable IP components, such as the nios II soft-core from Altera, the microblaze softcore from Xilinx and the portable soft-core Leon2 [82]. In the remainder of this subsection we first introduce these soft-cores, and we present our case study on the mapping of the control and monitoring network based on these IP components.

#### Soft cores

A *soft-core* is a general-purpose Reduced Instruction Set Computer (RISC) processor. It is reconfigurable in the sense that some features can be added or removed on a system-by-system basis to meet performance and cost requirement and constraints. For example, the size of the embedded memory that stores instructions can be configured, the instruction set can be customized, and specialized arithmetic and logic units can be instantiated at compile time.

We first considered the LEON2 [82] software processor, which is a fully synthesizable and parameterizable VHDL model of a SPARC V8 architecture. This software processor uses a standard on-chip bus from ARM (AMBA [83]). We ported a LEON2 soft-core across the two FPGAs we considered during our handcrafted implementation in the first case study. However, the mapping of a LEON2 soft-core uses 5 times more resources and can hardly sustain half the throughput compared to a typical implementation of a soft-core that is manufacturer dependent. When designing with manufacturer-dependent soft-cores such as nios II or microblaze, the application is specified in a user friendly environment (SOPC and IDE for Altera; XPS and EDK for Xilinx), which explicitly separates the hardware specification from the software specification. Many peripherals are available as library (IP-)components in these environments, and can be connected to a manufacturer dependent bus (AVALON [12] or OPB/PLB [13]). We experienced these two design flows to lead to short development times, and mean performance in terms of resource usage and speed (about 1,000 logic elements and up to 150MHz per soft-core).

This performance is hardly half the performance that can be achieved with hardware IPs. To sustain the same throughput as hardware IPs, soft cores must be implemented in parallel (multiprocessor system on chip - MPSoC). This results in an increase in resource usage and system cost. Moreover, soft cores execute instructions in a sequential order. However, our signal processing applications require a high degree of parallelism. Thus, in contrast to [81], we do not map the signal processing network onto soft cores. Nevertheless, we mapped the control and monitoring network, where tasks are executed at a lower rate than in the signal processing network, onto a hierarchical network of soft cores.

### Hierarchical network of soft cores

As introduced in chapter 2, processes in the control and monitoring network (root node, intermediate nodes and leave nodes) are arranged in a tree topology. In our approach to map this part of the system, root-node is mapped on a PC, and the rest of the system is mapped on an FPGA as shown in Figure 5.4, where the intermediate-node becomes the single access point for control and monitoring.

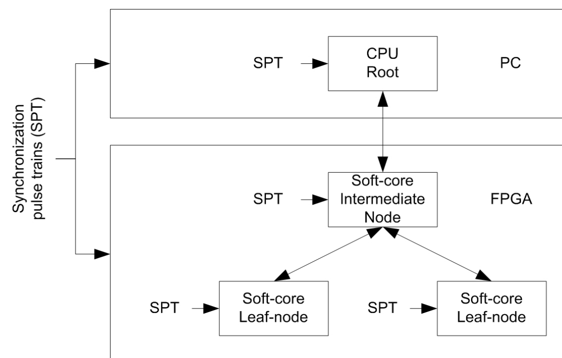


Figure 5.4: Mapping a control and monitoring network onto a network of soft cores.

Intermediate nodes and leave nodes are mapped onto soft-cores embedded in an FPGA. The code that is executed in the soft cores clearly separates generic high-level primitives, which implement the functional behavior of the nodes in the control and monitoring network, from specialized services that are supported by soft cores. This is illustrated in Figure 5.5 for the mapping of an intermediate-node as a simple state machine. The default state is represented in grey (*INIT*, lines 11-15) and corresponds to the initialization of the communication channels in the control and monitoring network. These channels are first checked for the presence of packets (*READ&CHECK*). When a packet is received, there are two possibilities (*SWITCH*): it is either sent to the appropriate destination node in the control tree (*ROUTE*), or it is inserted in a priority queue (*QUEUE&ORDER*, lines 16-22). Finally, there are again two alternatives (*CHECK*): the command that is in the packet on top of the queue must be executed on the occurrence of the next synchronization pulse (*EXECUTE*), else the communication channels are checked again until a new packet enters the node or a new command must be executed.

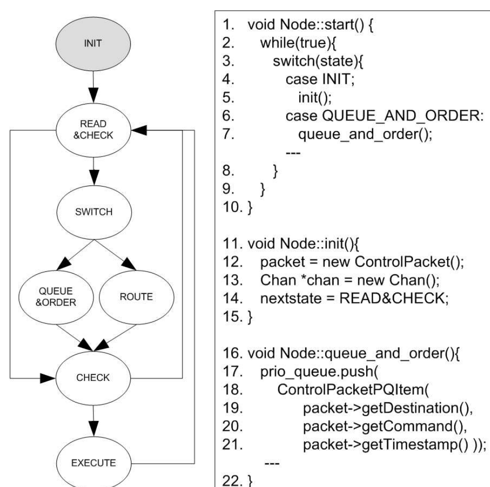


Figure 5.5: Mapping a node as a FSM onto a soft-core: state diagram and corresponding architecture-independent pseudo code.

The two leave nodes and the (intermediate) node above in the hierarchical control and monitoring network are mapped onto nios II soft-cores in a Stratix II FPGA [12], and support a portable real-time kernel (MicroC/OS-II [84]) to handle synchronization pulses as interrupts. The (intermediate) node also supports a portable light-weight TCP/IP stack [85] to communicate with the root-node that is mapped on a CPU in a PC. The intermediate node and leave nodes communicate through communication channels that are mapped onto on-chip dual port memory as detailed in [86]. We used a middleware library to govern the exchange of control packets among soft-cores such that this exchange relies on the same instructions at higher levels of the hierarchy in the control and monitoring network. This middleware required approximately 1.3MB of external SDRAM for each soft-core. This is acceptable in the systems we consider since a few GB of memory are typically available to store large snapshots of data next to the FPGAs.

With this approach, all soft-cores operate independently of each other and communicate in a unique manner. This allows to scale the system without inserting new types of interfaces, and without necessarily modifying the code that is executed in the soft cores. It took approximately 4 months to first prototype and validate the complete system. Note that the validation was done by probing the signals at run time rather than by simulating the MPSoC implementation, since the simulation speed was too slow. We draw here some conclusions concerning this design approach.

Positive aspects:

- The application software is based on function calls that abstract the underlying hardware architecture. This facilitates porting of applications across different architectures.
- The architecture can be configured at compile-time to match specific performance and cost requirements.

Limitations:

- The verification of the multiprocessor system-on-chip is difficult and time consuming.
- Implementing communication channels between soft-cores is not automated.

### 5.4.4 Conclusions

The comparisons between the approaches evaluated in this section with respect to the objectives given in Table 5.1 are summarized in Table 5.3. Depending on the chosen approach to specify the system, different results are obtained. Nevertheless, we were not entirely satisfied with any of the approaches. Results in terms of throughput and resource usage were satisfactory only for handcrafted implementations in our case. Scalability and reusability were considered to be limiting factors when a modification of high-level parameters implied handcrafted interfacing of IP-components. It was satisfactory when a skeleton was automatically generated for the entire application as in Compaan-Laura. We experienced the development time to be satisfactory when it took less than a week, and limiting when it lasted longer than a month (as in handcrafted implementations). Concerning the porting objective, we were not satisfied with any of the design approaches. Indeed we had to use tools that did not support identical library components, and that slowed down the verification procedures since they were hardly compatible with each other for the manipulation of test vectors.

For an automated mapping to be feasible, the IP interfaces must be described and characterized in a standard way. This is true on all levels of the hierarchy. Moreover, large scale systems can not be validated by simulating low level details on all levels of the hierarchy. To facilitate the comparison of multiple mapping options at higher levels of abstraction, models of the IP components, and models of their interfacing must be available and compliant to some standard. In this way, iterative refinement methods will become more transparent and larger designs could be handled. This is true in the signal processing network, in the control and monitoring network, and, as we shall see in the next section, in their interfacing.

| | Handcrafted | Simulink | Compaan-Laura | Nios II |
|---|---|---|---|---|
| Throughput | 1 | 2 | 3 | 2 |
| Resource usage | 1 | 1 | 3 | 2 |
| Scalability | 2 | 3 | 1 | 2 |
| Re-usability | 2 | 3 | 2 | 2 |
| Development time | 3 | 1 | 2 | 1 |
| Dependence | 2 | 3 | 2 | 3 |

Table 5.3: Design approaches and observations with respect to the criteria given in Table 5.1 (1: satisfactory; 2: can be improved; 3: limitation).

## 5.5 Interfacing of the two networks

In the previous section we investigated different approaches to integrate IP components and port applications across architectures when mapping the signal processing network and the control and monitoring network on separated architectures. A down-scaled version of the digital data-reduction subsystem in a station has been modeled [16]. We did not model and implement the control and monitoring part for the same subsystem because there was no option to construct the integrated system. Instead, we conducted two case studies around the interfacing between the signal processing network, and the control and monitoring network. This interfacing is one of the main aspects of this thesis.

In the first case study, we evaluate an emerging standard to interface IP components in a unified way in the two networks. Then we quantify the effects of a domain specific interfacing between programmable IP components in the control and monitoring network, and hardware IP components in the signal processing network, and we focus on design scaling and performance in terms of throughput and resource usage. Finally, we give our views on the interfacing of IPs.

### 5.5.1 Standard interfaces

We tested the OCP-IP standard [87], which connects IP components based on point to point communication channels, and which follows a master-slave protocol. For a component to be both a master and a slave, it must be assigned two communication channels. The OCP-IP standard is fully compliant with our objective in term of portability. Indeed, it is bus independent (and therefore manufacturer independent). Moreover, the OCP standard supports many interfacing mechanisms, ranging from simple reads/writes to burst, pipeline or concurrent transfers. On the one hand, this configurability is convenient since it gives freedom to specify our application from a high-level without being dependent on a unique communication protocol on all levels of the hierarchy. On the other hand, as mentioned in [88], the OCP standard does not support mechanisms for arbitration or address decoding. This can be a limitation when several components must share a resource such as a memory.

We evaluated OCP using its SystemC [89] libraries. There are two levels of abstractions

in these libraries: a first one (TL1) corresponding to a functional level and a second one (TL2) is closer to the hardware implementations. These two levels are compatible. This allows for progressive mapping refinement when designing from high-level. Therefore we used TL1 to specify a signal processing application consisting of two concurrent IPs (FIR filters) as shown in Figure 5.6. Each filter is connected to a dedicated leaf node. In this case, the two leave nodes are identical and are governed by a single intermediate node. The intermediate node and the two leave nodes receive the same synchronization pulse, indicating when specific actions such as the re-configuration of the filter coefficients or the monitoring of the data can take place. These actions are based on commands (contained in packets) received asynchronously by the intermediate node.

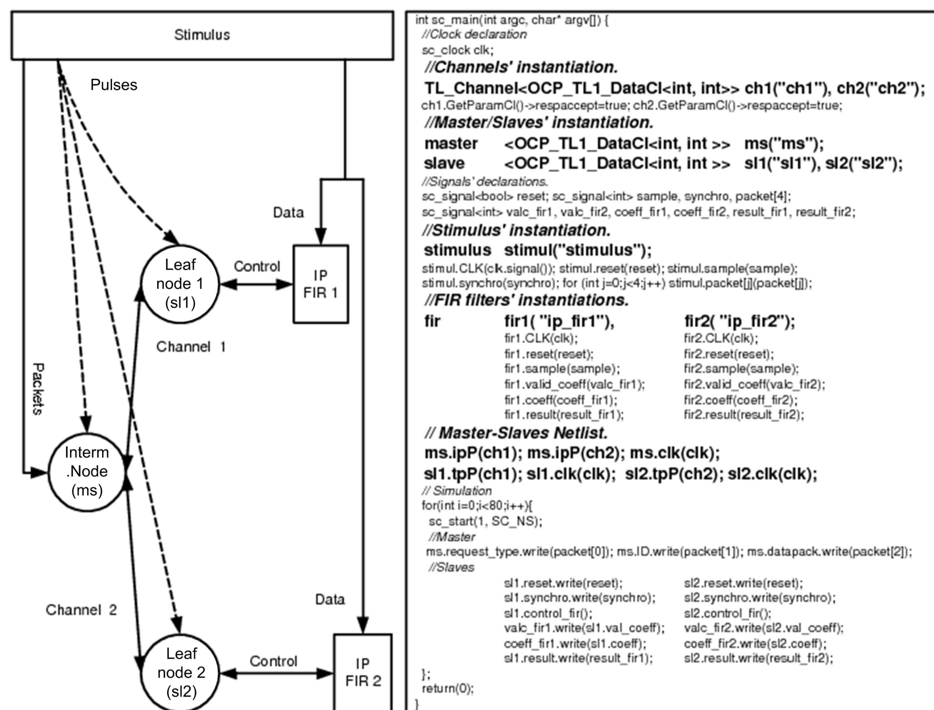


Figure 5.6: High-level specification of (OCP) interface in SystemC.

As in [90], we co-simulated SystemC and VHDL. Thanks to a particular function provided by OCP-IP, each OCP communication channel could be monitored separately during this co-simulation, without altering the behavior of the complete design. Moreover, scaling the design from one leaf node and filter to two leave nodes and filters was a very simple step. Indeed, we only had to instantiate one more FIR filter IP, one more node and one more OCP channel between this node and the intermediate node. Therefore the combination of SystemC and OCP for the specification of our signal processing applications from a high-level is efficient with respect to our objectives in term of scalability. However the co-synthesis of SystemC and VHDL remains an issue when aiming at both high throughput and low resource usage. We can derive the following conclusions concerning this approach:

Positive:

- It facilitates design-scaling by supporting several interfacing schemes and their monitoring during simulation.
- It facilitates portability thanks to manufacturer independence.
- It supports refinement across levels of abstraction.

Limitation:

- For this emerging standard, co-synthesis is not mature enough yet to compete with handcrafted implementations in terms of performance.

This co-synthesis and performance limitation motivated us to investigate a more pragmatic way of integrating IP components, by interfacing programmable IP components (soft cores) in the control and monitoring network with hardware IP components in the signal processing network.

### 5.5.2 Merging the two architectures

In this case study, we merge the FPGA onto which the control and monitoring network is mapped using programmable IP components, with the FPGA onto which the signal processing network is mapped using hardware IP components. Thus, the interfacing between the two networks becomes internal to that FPGA. To anticipate design scaling and design re-use issues, all hardware IP components are wrapped and connected with the soft cores in a uniform way. The two types of IPs are implemented in separated clock domains so as to avoid obstructing the high speed execution of the hardware IPs when interfacing them with low speed programmable IPs.

#### Hardware IP wrapping

Each leaf-node controls and monitors the behavior of a (KPN) process that is mapped on a wrapped IP component. As detailed in [91], the hardware realization of a wrapper is made of four components that are shown in Figure 5.7: a *Dataflow Read Unit* that gets tokens from dataflow input channels, multiplexes and transmits them to an *Execute Unit*, which consumes these tokens, performs computation using an IP in a *Function Repertoire* and produces output tokens towards a *Dataflow Write Unit*. This unit includes a private memory. It demultiplexes the output tokens and sends them to output dataflow channels. The fourth component is the *Controller* that governs the execution of the three other units.

In the interface between a leaf-node and a process, the additional command/parameter port is connected to a *Control Read Unit*. The Dataflow Read Unit, Dataflow Write Unit and Function Repertoire get their own configuration parameters under the supervision of the Controller. The additional monitoring-data port is connected to a *Control Write Unit*, which

probes the dataflow in the Dataflow Write Unit, as well as the state of the node in the Controller.

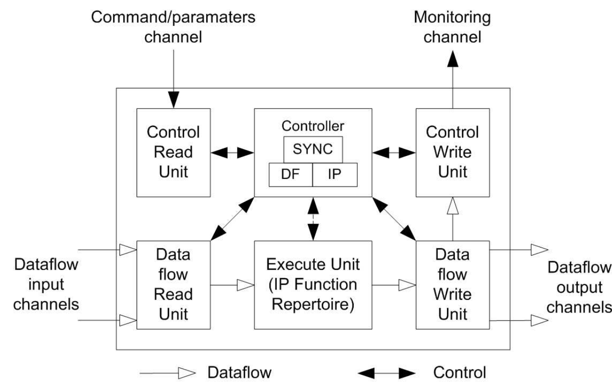


Figure 5.7: Hardware implementation of a process, including everything that is needed for the interfacing with a leaf node.

The Controller keeps the dataflow processing, control and monitoring processing and synchronization mechanism separated by means of three concurrent FSMs. A first *SYNC* FSM synchronizes the execution of commands issued from a leaf-node with the (periodic) execution of the signal processing function. A second *DF* FSM controls the dataflow Read and Write ports in a Stream Based Functions (SBF [51]) dataflow model of computation. A third *IP* FSM controls the IP *Execute* function repertoire in the SBF dataflow model of computation. Portable and synthesizable HDL code is generated automatically for these three FSMs from a high-level architecture-independent graphical specification.

#### Prototype implementation and results

In our prototype implementation, the signal processing network is limited to a single KPN with two processes as shown in Figure 5.8, and each IP function repertoire contains a unique re-configurable IP-component. The first process wraps an IP that generates periodic dataflow patterns (e.g. impulses, ramps or sinewaves) that can be re-configured (e.g. amplitude, frequency) at run-time by the first leaf node. The second process wraps a FIR filter IP whose taps can be re-configured (e.g. low-pass, band-pass, high-pass characteristics depending on the operational mode) at run-time from the other leaf node[2]. These two processes are implemented in a first clock domain.

The two leave nodes and the intermediate node above in the control and monitoring network are executed in nios II soft cores as discussed in section 5.4. The three soft cores run at the same frequency and are implemented in a clock domain that is independent from the clock domain of the signal processing network.

[2]In this case study, we restrict ourselves to run-time re-configurations that do not change the length of the dataflow patterns and the number of taps in the filter. We modify the length of the patterns and the number of taps at compile-time.

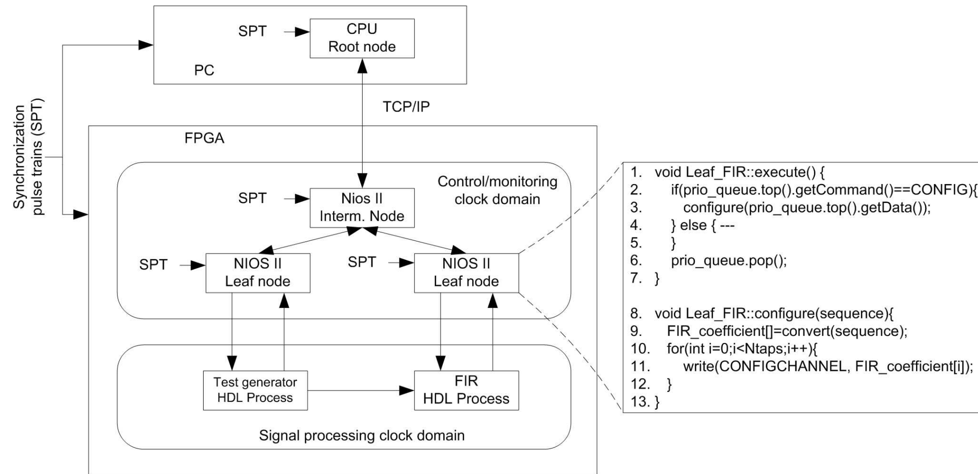


Figure 5.8: FPGA implementation of a hierarchical control network in soft cores to control and monitor a test generator IP and a FIR filter IP.

Re-configuring the FIR filter IP coefficients requires converting the new coefficients to the IP-specific format because coefficients are stored in partial order and distributed in embedded memory segments. This conversion is done in the $Execute$ state of the leaf-node that controls the IP as shown in the pseudo-code in Figure 5.8. On the occurrence of a synchronization pulse in the leaf node, a control packet is processed. If a packet requests re-configuring the filter, then a program is called (lines 2-4) that converts the configuration data to the IP-specific sequence. This sequence is sent to the re-configuration channel (lines 10-12) and the control packet is removed from the queue after its execution (line 6). The leaf node may then send a command to activate the re-configuration of the process at the beginning of the next execution cycle.

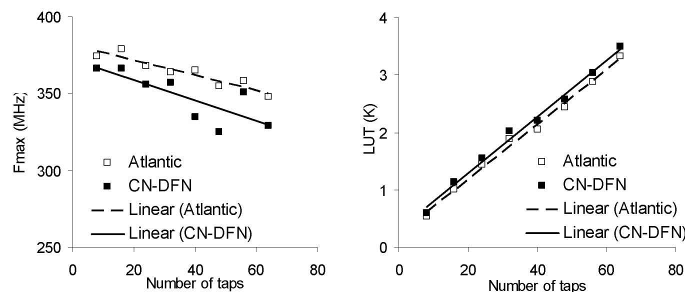


Figure 5.9: Impact of design-scaling on throughput and resource usage (designs are scaled at compile-time).

We scaled the number of taps in the FIR IP, i.e., the period of this process, at compile time, without altering the behavior of the interfacing with the control and monitoring network.

Figure 5.9 shows the impact of the FIR IP function-scaling on throughput (maximum frequency sustained by the dataflow in the process after synthesis, on the left-hand side) and resource usage (LUT, on the right-hand side). Results are given for the specific interfacing between the two networks (control and dataflow; CN-DFN) presented in this section and for a manufacturer-dependent dataflow only interface (Atlantic [12]). Our interface needs a few more resources since it includes both dataflow and control. The loss in the throughput is less than 10% with respect to the manufacturer-dependent interface. Thus, the performance of the FIR IP is not significantly altered by the interfacing with the leaf-node.

Portable HDL has been generated from high-level graphical specifications in StateCAD [13] for the three FSMs that are executed in the IP-wrappers. Thus, we avoided time consuming and error prone HDL handcrafted glue logic development. The interfaces we prototyped permit de-coupling low-speed clock domain(s) in the control and monitoring network (soft-cores hardly run faster than 150MHz) from the high-speed clock domain(s) in the signal processing network. We can derive the following remarks concerning the interfacing approach that we used in this prototype implementation:

Positive:

- The control and monitoring network can be mapped onto soft-cores and interfaced with IP-wrappers without significantly obstructing the performance of the dominant signal processing network.
- Dedicated links between soft cores and hardware IPs allow to scale the the IPs in the dominant signal processing network independently of each other and still get close to optimal performance.

Limitations:

- The approach is restricted to the interfacing with large hardware IP components, which use more resources than soft cores, such that the signal processing network remains dominant.
- The current implementation does not allow to change the periods at run-time in the signal processing network.

### 5.5.3 Requirements for future IP-based designs

The comparisons between the approaches evaluated in this section with respect to the criteria given in Table 5.1 are summarized in Table 5.4. In contrast to the approaches that have been investigated in section 5.4 for the mapping of the two networks in isolation, none of the two approaches we evaluated in this section is considered to be limited regarding any of the objectives.

The standard interface approach relies on high-level synthesis that exploits hardware heterogeneity at the cost of gross performances. The interfacing between soft cores and hardware IPs leads to better performance but is not fully automated yet. It may be interesting to include

| | SystemC-OCP | soft cores & wrapped IPs |
|---|---|---|
| Throughput | 2 | 1 |
| Resource usage | 2 | 2 |
| Scalability | 1 | 1 |
| Re-usability | 2 | 2 |
| Development time | 2 | 2 |
| Dependence | 1 | 2 |

Table 5.4: Design approaches and observations with respect to the criteria given in Table 5.1. (1) satisfactory; (2) can be improved; (3) limitation.

models of our specific interfacing approach in a framework. This framework should support standardization on multiple levels of abstraction, and transformations to semi automate the mapping of high level application specifications down to the level at which component specific compilers and synthesizers can implement the system.

## 5.6 Related work

F. Wagner et al. [88] identified strategies and gave research directions for the integration of IP components in systems-on-chip altogether with the generation of an OS. The challenges of co-simulation and co-synthesis of IP-based designs are covered. In our approach we are interested in mapping our applications onto networks of re-configurable architectures. Automated task-level mapping of video-audio application on FPGA with systematic data-flow IP interfacing is covered in [92] at a level of details, which is closer to our work. However the problem of mapping onto multiple architectures is not addressed.

Organizations such as Virtual Socket Interface Alliance [93] or the Object Management Group [94] are specifying open interfaces standards for components to be integrated into sockets. Others, such as the System Design Industrial Council-Telecom (SYDIC-Telecom [95]), analyze also the design flows in relation with specification languages and formalism analysis to address the issues of design re-use starting from a system level conceptual level. Voros [96] contributed to this research by identifying the common basis in all these approaches in particular the parametrization issue [97] in various aspects of granularity and the encapsulation of IPs in an object oriented system design. Objects provide for a structural representation of the information through all abstraction levels.

The objective of El Greco [98] is to provide a path to implementation as embedded software, synthesizable hardware or both. Applications are specified in the form of dataflow graphs with Kahn or CSDF-like semantics [99], hierarchical synchronous finite state machines with Esterel-like imperative semantics [100], or a mixture, arbitrarily nested. In [53], applications are modeled using Process Networks and Stream-Based Functions (SBF [51]) with non-static parameters. These applications are also mapped onto an FPGA and get configuration data from outside. However, the re-configuration is only possible after a complete network cycle. We want to be able to re-configure each individual periodic dataflow process during

any period at run-time, without stopping the entire signal processing network since the high throughput dataflow originating from the antennas is permanent in our systems.

Library components are available to implement control functionalities, signal processing functionalities, and their interfacing in the Berkeley Emulation Engine (BEE2 [29]. This is also the case for the LOFAR Remote Station Signal Processing platform (RSP3 [64]). Although the mapping of applications onto these FPGA-based architectures is manual and starts on a level that is partly dependent on these architectures, a station can be obtained by duplicating the resulting subsystems.

An approach to dynamically reconfiguring a streaming application in a hierarchical SoC with a multiprocessor subsystem is presented in [101]. Processing tasks can be reconfigured through inserting reconfiguration tokens in the data streams. We avoid such insertions by physically separating dataflow and control paths in our implementations. Nevertheless, combining the generic services offered by the shell described in [101] with a standard task-level specification as in [69] would lead to optimized SoC implementations and re-usable IP-components.

## 5.7 Conclusions

Five case studies have been conducted to focus on systematic integration and re-use of IP components, portability of designs across components, and interfacing between leave nodes in the control/monitoring part with processes in the signal processing part. We first implemented a high throughput signal processing application across different FPGA fabrics. The specified implementation matched our requirements. However, handcrafting completion logic to glue IP components hindered design scaling, porting of designs, and was too time consuming and error-prone. We then used two approaches to automate the mapping in the signal processing network only, starting from abstract graphical and language-based specifications, respectively. Then, we evaluated the use of programmable IPs to implement the control and monitoring network only. These three approaches helped to shorten the overall implementation specification time but we encountered many disparities between the tools we manipulated, especially concerning IP interfacing and (re-)use of glue logic. We derived requirements to improve the mapping of our applications with respect to these issues based on standard interfaces.

Then we evaluated two approaches to interface the two networks. The first approach is a bus-independent emerging standard that enables wrapping all IPs in a uniform and systematic way from high-level specifications. This facilitated the duplication of IP components in our applications, but did not lead to satisfactory performance. To address this limitation, we evaluated a more pragmatic approach in a prototype implementation. In this approach, some glue logic is generated automatically from graphical representations of FSMs. Control and monitoring tasks have been implemented in programmable IPs in a first clock domain, and signal processing tasks have been implemented in hardware IPs in another clock domain. The interfacing between the two domains relies on point-to-point interfaces between leave nodes and processes, which keep the clock domains separated, and which permit adding IP components without significantly obstructing the performance of the dominant signal processing

network. However, the digital systems we consider consist of many parts, which require different implementation tools. These tools imply different restrictions in terms of compilation steps and glue logic generation, such that it is currently hard to separate data from control when targeting a large system from a single implementation-level specification. Bringing a regular and more homogeneous structure, both at multiprocessor system-on-chip level and at higher levels of the hierarchy in future tools, would simplify this implementation task.

Chapter 6

# Conclusion and future work

## 6.1 Conclusion

In this thesis we gave an approach to structure the path from abstract system-level specifications (in terms of application, architecture, and their association together) to implementation-level specifications (the level of abstraction from where compilation and synthesis tools should take over to obtain a real implementation). We focused on large-scale and distributed embedded signal processing systems and considered the particular case of the digital processing part in stations in next generation phased array radio telescopes such as SKA [2]. The approach consists of expressing system-level specifications in terms of models that provide a good foundation to take design decisions in a consistent way, therefore avoiding intuitive decisions. We brought together different approaches that have been proposed by others for the separate modeling of the application, architecture, and mapping of the former onto the later for the signal processing part only [16]. We focused on the control and monitoring part of the system that we separated from the dominant signal processing part. This raised the specific problem of the interfacing and synchronization of these two parts. We addressed this problem separately in the application, architecture, and mapping.

In the application specification, we expressed the functional behavior of the system based on the operational semantics of a model of computation that describes unambiguously the way data is simultaneously processed in computation nodes and communicated between these nodes in a network. We specified the behavior of the signal processing part with a transformative stream-based model of computation, and the behavior of the control and monitoring part with a reactive state-based model of computation. To synchronize the two models, we related a timing model that is known only to the control and monitoring part, with the repetitive behavior of processes in dataflow process networks.

In the architecture specification, we expressed the non-functional behavior of components

and their interconnection interfaces that are taken from a library of components, including permissible interconnection rules. At lower levels of the hierarchy, components are modeled as white boxes whose internal modules provide simulation means to obtain actual performance/cost numbers. At higher levels of the hierarchy, components are modeled as black boxes whose performance/cost behavior is modeled in terms of analytical relations between input quantities and output quantities. In these relations, parameter values are calibrated with information obtained from lower-level white-box model simulations. The composition in the signal processing part sustains intensive computations on and transport of high throughput data streams with a high degree of parallelism. The composition in the control and monitoring part supports the exchange of sporadic messages and the execution of sequential procedures in reaction to these messages. The two architectures are interfaced through point-to-point links between processing units at the lowest hierarchical level. This simple interfacing leads to an architecture model onto which the application model can be mapped.

The transformation-based mapping associates the application model together with the architecture model, and improves their matching in terms of level of detail and in terms of performance/cost on all levels of the hierarchy. We assumed that mapping transformations are available in a library, and that they can be called iteratively and interactively in any order. After each mapping transformation, functional and non-functional behaviors of the system are analyzed. Such analysis is the core function of the design space exploration method that should converge to (pareto)optimal system specifications. The abstract specifications that are suggested by the last iteration in the exploration process are finally converted to implementation-level specifications. During that conversion, additional transformations (high-level compilation steps) can still be applied automatically to optimize the performance of parts of the system. We assumed that different parts in the implementation-level specification can be converted to real implementations based on different commercially available tools.

To assess the capacity to obtain real implementations while dealing with Intellectual Property (IP) components integration constraints, we conducted a few experiments using different model-based approaches supported by commercially available and academic tools for multiprocessor systems on chip. The specification of the signal processing part and control/monitoring part is unified in the application and/or architecture in most approaches. This does not permit taking decisions separately about the two parts. Moreover, the performance/cost of the candidate mappings is evaluated based on simulation with the same level of detail on all levels of the hierarchy in most approaches. This is a limitation for large-scale and distributed systems where analytical models are required to fasten the design-space exploration process at higher levels of the hierarchy. Mapping transformations are hidden in compilation steps, and their order is fixed in most approaches. This restricts the number of mapping alternatives but facilitates the systematic integration of IP components by generating automatically glue logic between IP components.

A major difference between large-scale embedded systems and relatively small-scale embedded systems (such as a multiprocessor system-on-chip, which can also be extremely complex) is that the former include more levels of hierarchy than the latter. When composing sub-systems in large-scale systems, assumptions are different from assumptions in SoC (for example, the signal processing part and control and monitoring part are mixed in a SoC, whereas they can be separated in large-scale systems). However, when moving up in the

hierarchy, and to scale the system, it is necessary to bring a regular structure in both types of systems. At SoC-level, this is achieved by organizing the architecture in a regular, structured, and homogeneous way at the highest level of abstraction [102] [103]. In the large-scale systems we consider, this is achieved based on networks of stations.

## 6.2 Future work

The design approach we presented in this thesis is neither purely top-down nor purely bottom-up. It is a meet in the middle design approach since the analysis that is part of the mapping process is calibrated based on simulation or prototyping of actual components. In this approach, we assume that all components that are involved in a system are available as white-box/back-box models in a unique library. In practise, of course, new components progressively appear when constructing and scaling a system, and corresponding component models must be added to the library. This insertion may seem to be a burden during the development phase. We argue that facilitating this insertion in existing design space exploration frameworks for large-scale and distributed embedded signal processing systems such as the Massive Exploration Tool [104], has a substantial reward in terms of design re-use, design scaling, and design porting. Means should be provided to automate the mapping of abstract instructions (which are executed in nodes in the application model) onto specialized services (which are supported by new components) in such frameworks.

Since we decided to separate the modeling of the signal processing part from the modeling of the control and monitoring part, we naturally gave two separate architectures, which we interfaced only at lower hierarchical levels through point-to-point links. This may be viewed as a restriction if both the signal processing part and the control and monitoring part have to be mapped on a single architecture. We argue that the approach we presented is powerful enough to address the mapping of the two parts on a single architecture, by going through additional cycles in the iterative and interactive analysis phase. We gave a pragmatic approach to address this situation in a case study. To address this situation automatically in the future, additional mapping transformations should be provided, and glue logic should be generated according to (preferably domain-specific) standards.

Given the scale and complexity of future large scale and distributed signal processing systems such as SKA, identifying hot spots for power consumption and minimizing their impact will become a major issue. It could be done by raising the level of abstraction (for example, by associating Globally Asynchronous Locally Synchronous models in the application together with white-box/black-box models in the architecture) while applying domain-specific design patterns as advocated in this thesis. Moreover, these systems are likely to be developed by teams that will be spread all over the globe. Adopting the model-based approach presented in this thesis will help to structure the design process and to interface these subsystems so as to reach the next order of magnitude of scale.

Appendix A

# An overview of specification-level models of computation

## A.1 Summary

In this Appendix we analyze specification-level models of computation in a way that is accessible to a broad community. The analysis is not formal but rigorous enough to understand and compare some fundamental properties of state-based and dataflow-based models of computation in terms of synchrony, concurrency, communication, determinism and composition. We use operational semantics so as to conveniently analyze computation and communication details for each individual model.

## A.2 Introduction

Models of computation have formal semantics that allow specifying applications unambiguously. These semantics are denotational, operational or axiomatic [42]. In this Appendix we use operational semantics to analyze the way data is simultaneously computed and communicated in a large set of models of computation. In Section A.3 we analyze state-based models and event-triggered models that may be used to specify the behavior of the control part of the large-scale systems we are concerned with. In Section A.4 we analyze stream-based models and dataflow models that may be used to specify the behavior of the dominant signal processing part in these systems. Models that mix state-based and stream-based semantics are analyzed in Section A.5. We give our related work in Section A.6 and draw conclusions in Section A.7.

## A.3 State-based and event-triggered models

In this Section we analyze the semantics of models of computation that naturally capture changes in the behavior of control-dominated systems in reaction to events that are produced by their environment.

### A.3.1 Finite State Automata

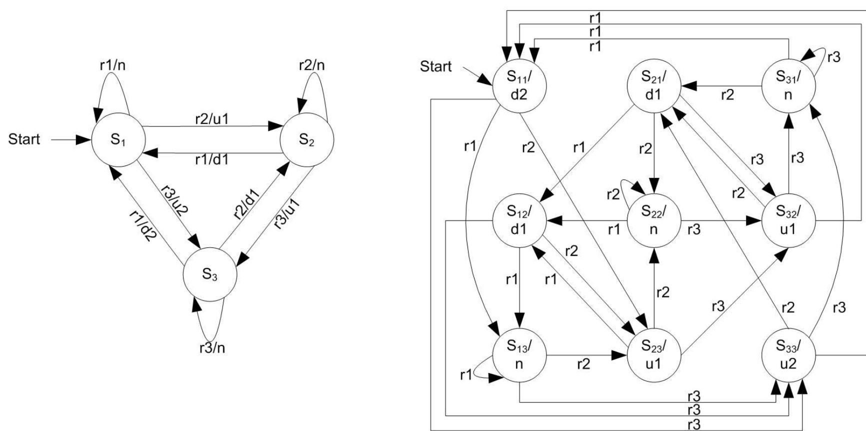


Figure A.1: Traditional FSM model - elevator controller example. Left: Mealy; Right: Moore.

Moore and Mealy Finite State Machines (*FSM*) consist of a set of states, a set of input, a set of output that are given in function of states and input, and a next-state function. Mealy FSMs modify output on transitions between states without delay, whereas Moore FSMs modify output within a state after a delay as depicted on the left-hand-side and right-hand side in Figure A.1, respectively, for an elevator controller example. These FSMs are intended for systems with relatively low complexity. They permit modeling deterministic sequential behaviors and can very well be analyzed. Code can easily be generated from these models.

However, the description and visualization of a system may become untractable with traditional flat FSM models. Moreover, modeling concurrency is impossible with traditional FSMs semantics.

### A.3.2 StateCharts

Harel [45] proposed the StateCharts, whose semantics rely on events that trigger transitions, conditions that guard these transitions, and resulting actions. StateCharts allow reducing the visual complexity of traditional FSMs by supporting three mechanisms (Figure A.2 illustrates the first two mechanisms).

- *Depth (hierarchy)*: being in a state *a* means that the machine is in one of the states of another machine enclosed by *a*. The states of *a* are called *or* states.

- *Orthogonality (Concurrency)*: a system can be in one state of several parallel state machines simultaneously. Such parallel states are called *and* states.

- *Non-determinism/abstraction*: states can be abstracted (not fully determined) so as to allow their future description.

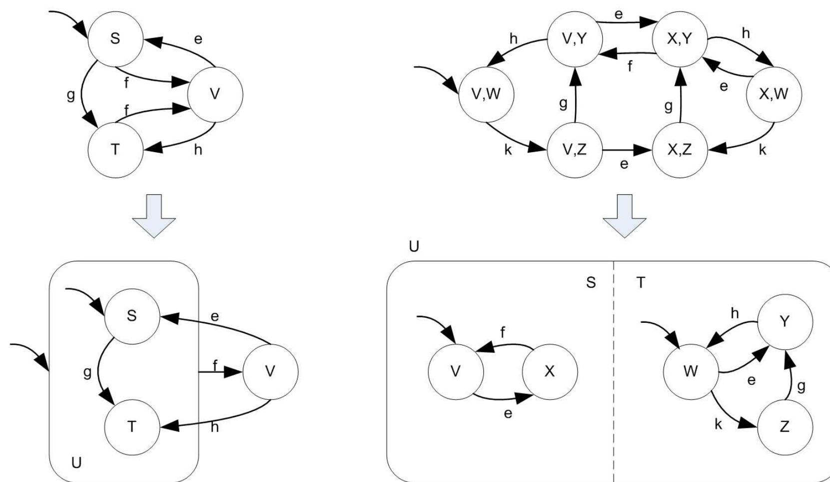


Figure A.2: OR-decomposition (left) and AND-decomposition (right) in StateCharts. OR: when it is in state U, the machine is either in state S or T; AND: when it is in state U, the machine is in a state of S and in a state of T. Concurrency is represented with a dotted line.

**FSMD**

A FSM with Datapath (FSMD [105]) is a combination of two (locally) synchronous FSMs arranged on top of each other: a datapath FSM and a controller FSM. The datapath FSM contains a set of functions and internal signals. It implements the functional behavior of the FSMD. The controller FSM selects a function in the repertoire in the datapath FSM depending on the values of the signals in the datapath FSM, but does not manipulate input and output signals. A FSMD has a deterministic behavior. Communication is synchronous in a network of FSMD, and a single output port may be connected to several input ports. A hierarchical composition of FSMD preserves determinism if the following four rules are respected: no variable should be assigned multiple values, no operand should be undefined, there should not be combinational loops, and all output signals should always be defined.

### A.3.3 Petri Nets

A Petri net [106] is a state-transition model [36] that represents discrete systems graphically with place nodes, transition nodes, and directed arcs connecting places with transitions as depicted in Figure A.3. Events within signals need not be ordered. The state of a Petri net is given by the marking of its places, that is a non-negative integer corresponding to the number of tokens assigned to each place. The execution of a Petri net is given by the firing of the transitions. A transition can fire only if each of its input places is marked (has at least one token). When it fires, a transition decrements the marking of the input places and increments the marking of the output places.

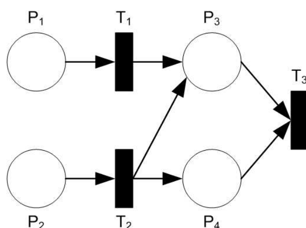


Figure A.3: Petri Net example. Places are connected through transitions in a bipartite graph.

Petri nets are well suited for the representation of causality (places linked by transitions are said to be in a predecessor/successor relationship) and concurrency. Colored Petri nets extend Petri nets semantics to accept multiple predecessors and successors in a place. This leads to a more compact representation. Time has also been introduced in timed Petri nets by associating timestamps to events. Details of these extensions can be found in [107].

### A.3.4 Process algebras

Process algebras (or process calculi) include Calculus of Communicating Systems (CCS [46]), Communicating Sequential Processes (CSP [47]) and the Algebra of Communicating Processes (ACP [48]). It is also a state-transition model [36] that aims at modeling concurrent systems as a network of independent processes using a small collection of indivisible (or atomic) primitives. These primitives are combined in operators, for which algebraic laws can be defined so as to manipulate expressions using equations.

A detailed comparison between CCS and CSP with respect to (non-)determinism, communication, recursion, abstraction and deadlock is given in [108]. In CCS, a process is considered to go through a number of states, which are determined by the actions the process is ready to engage in. Any action a process can perform is regarded as a state transition. In CSP, a process is considered to run in an environment that can veto atomic actions during the execution of this process. In CCS, communication and interleaving are specified with a unique operator, while CSP has different operators for both. In CCS, communication between two processes occurs if one of the processes offers an action and the other offers the complementary action. In CSP, communication between two processes occurs if both of them offer the same action.

Nevertheless these two communication types are often called rendezvous communication in the literature in the sense that they wait for each other to engage in a particular action.

### A.3.5 DE and DDE

Discrete-event (DE) is usually a simulation model. Nevertheless it can also be considered as a specification-level model [36]. Entities execute concurrently and share a global notion of time. They communicate by exchanging events that are time-stamped and totally ordered. Thus, the DE model is timed. After processing an event, an entity produces one or more events. It assigns future or same timestamp as the one in the processed event so as to enforce causality. However, events that have the same timestamp may not be ordered in an entity. Thus, an input may lead to different output. As a consequence, DE is non-deterministic. To avoid this limitation, an instant may be split into a potentially infinite number of totally ordered delta-steps [109].

Simulators that support this model typically sort events by timestamps in a global queue. The experimental Distributed Discrete Event (DDE in Ptolemy II [26]) model extends DE semantics to order events locally in distributed queues rather than in a global queue. As in DE, time progresses in the form of timestamps that are associated with events, and successive events emitted from any entity's output port must have a timestamp that is greater than or equal to that of the previous event. The main differences with DE are that 1) DDE implements a notion of time that is distributed and localized in each entity, and 2) communication is done through unidirectional bounded FIFO channels.

## A.4 Stream-based and dataflow models

In Section A.3 we analyzed state-based models and event-triggered models that are well suited to specify control-dominated reactive systems. In this Section we analyze models that are more appropriate to specify the behavior of signal processing-dominated systems that transform streams into stream. In the signal processing community, a signal is a special stream in the sense that it is an unbounded sequence of tokens that are separated by regular intervals.

### A.4.1 KPN

In a Kahn Process Network [35], processes are autonomous and execute concurrently. They communicate point-to-point through unidirectional and unbounded FIFO channels. This FIFO property implies that the set of tags is totally ordered in each individual signal. However, a specificity of the *KPN* model is that the set of all tags is partially ordered (there is no global scheduler). Thus, a KPN model is globally asynchronous. Each process synchronizes locally with a blocking read primitive (ordering behavior), where the reading on an empty port (that has no token) results in a suspension of the process until a token arrives. As a consequence, a process is not allowed to test an input port for the presence of tokens and then

branch to a point where it will read another port. Thus, a KPN has a deterministic behavior, i.e, the outcome is independent of the chosen schedule. A composition of processes in a KPN preserves the semantics of the model.

A dataflow process network [43] is a special case of a KPN [18]. Processes, which are called actors, have the following firing rules: they consume and produce tokens according to partial ordering rules using a blocking read primitive and a blocking write primitive, respectively. All input tokens are evaluated simultaneously. A dataflow process network is a GALS network of such concurrent actors, which are scheduled statically or dynamically.

### A.4.2 Dataflow models

In a dataflow graph, computation is represented with nodes (actors) that exchange data values through edges (arcs) that represent FIFO queues. Edges thus represent causality between computations. Actors consume data from their input ports, perform computations (fire) and produce data on their output ports. Thus, dataflow graphs relate to KPN and dataflow process networks.

Several models have been proposed that are based on dataflow with restricted semantics in order to simplify implementations in hardware or software. The computation graph [110] model is one of the earliest and provided some of the foundations for the SDF model, which is probably the main dataflow model.

#### SDF and CSDF

The main objective of the synchronous dataflow model (*SDF* [44]) is to schedule the firings statically in order to derive a direct implementation that avoids potential overhead of run-time multi-tasking (context-switching). A first restriction in SDF is to limit the number of functions to a single one per actor. A second restriction is to fix at compile-time the number of tokens that are produced and consumed by each actor, such that the rates of firing of actors are fixed relative to one another. Multi-rate processing is modeled with uninterrupted sequences or reads and writes of single tokens. A third restriction is to use a global scheduler. The SDF model offers many opportunities such as software synthesis and code generation in DSP systems [111].

In the cyclo-static dataflow model (CSDF [99]), an actor contains more than one function. Token production and consumption can change from one function to another as long as the variation is a periodic (or cyclic) pattern. This model allows to derive cyclic schedules at compile time. This can lead to several benefits over SDF, including decreased buffer sizes.

#### MD-SDF, SSDF

Extensions of the SDF model have been proposed to adapt to a broader range of applications while preserving compile-time scheduling properties. In the multi-dimensional dataflow model (MD-SDF [112]), communication channels carry tokens that may have multiple di-

mensions (as in image processing). This allows to expose parallelism explicitly. In a scalable synchronous dataflow model (SSDF [113]), each actor can process any integer multiple of the SDF token production (consumption) on its output (input) ports. This reduces context switching between actors and may improve implementation performance.

### BDF and DDF

The Boolean dataflow model (BDF [114]) adds a port, called conditional port, to the actors. The number of tokens produced or consumed on a dataflow port is either fixed, or is a two-valued function (boolean) of a token present on the conditional port. In some cases, a sequence of executions can be scheduled that returns a network of BDF actors to its original state (complete cycle). This leads to compile-time schedules, where each firing is annotated with the run-time conditions under which the firing should occur, as depicted in Figure A.4 for two actors: switch and select. Thus, potentially costly run-time scheduling techniques are avoided in implementations that rely on BDF.

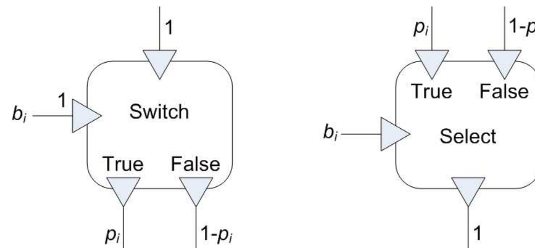


Figure A.4: Switch (left-hand side) and Select (right-hand side) actors annotated with the average rate of tokens produced/consumed per firing, as a function of $p_i$, the proportion of tokens from the boolean stream $b_i$ that are True.

Dynamic Dataflow networks (DDF [115]) are Boolean Dataflow networks with one additional variation: the conditional ports mentioned in the BDF model can read multiple token values and the actors can fire conditionally based on the conditional ports read, i.e. a control token may be a vector of integers that indicate how many dataflow tokens must be consumed and produced.

### PSDF and PCSDF

A parameterized synchronous dataflow model (PSDF [52]) allows dynamically determining the number of tokens that are consumed and produced on the input and output ports of each actor. As depicted in Figure A.5, a PSDF subsystem consists of an init graph, a subinit graph and a body graph. The main dataflow (functional) behavior of the actor is implemented in the body graph, which may accept and produce dataflow tokens. The init and subinit graphs control the (ordering) behavior of the body graph by configuring its parameters. The init graph does not participate in the dataflow. It can configure the parameters of the subinit

and body graphs. The subinit graph can only accept dataflow tokens on its input ports, and configures the parameters of the body graph.

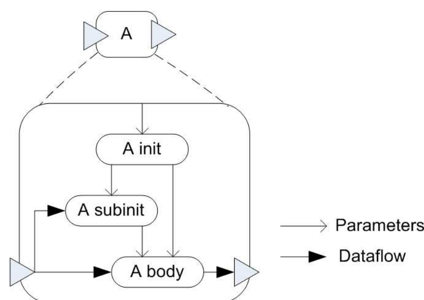


Figure A.5: Structure of a PSDF subsystem.

Intuitively, the execution phase of every PSDF subsystem is preceded by a configuration phase. The init graph is invoked once at the beginning of each execution of the parent graph the subsystem belongs to. The parameters of the init graph are set by the init and subinit graphs of hierarchically higher-level subsystems. The subinit graph is invoked at the beginning of each execution of a subsystem, and the body graph is finally invoked after the subinit graph. Thus, the locally synchronous semantics of PSDF forbid dataflow to depend on internal or external subsystem parameters. As a consequence, each PSDF graph always behaves like a SDF graph in each of its invocation. It is then possible to generate a quasi-static schedule.

The abstract mathematical model presented in [116] is an attempt to formalize PSDF. Parameters may be re-configured only at specific execution points called quiescent points, which are approximations of the set of parameters change context that are derived after analyzing all possible executions of a model.

The semantics of the Parameterized Cyclo-Static Dataflow model (PCSDF) are similar to that of PSDF, except that neither the functionality (internal behavior) nor the dataflow (ordering) behavior are parameterized directly. They vary cyclically, however, and it is precisely this cyclic pattern that is parameterized in PCSDF [117]. The period of a cycle and the corresponding production/consumption rates can vary before the execution of this cycle, and must remain constant throughout the execution of the cycle. Clustering PCSDF subsystems is achieved by extracting a CSDF graph for each individual cycle, and then by clustering this CSDF graph.

## A.5 Heterogeneous models

A specification of a reactive signal processing system should capture both the reaction to sporadic control events and the reaction to data streams. In this Section we analyze heterogeneous models of computation, which combine state-based and stream-based semantics.

### A.5.1 Synchronous/Reactive language

The Synchronous/Reactive model of computation (SR) specifies systems as concurrent and synchronized processes (called modules). Modules communicate by exchanging signals that carry possibly empty events at each clock tick [109] through non-buffered channels. The synchronous languages [37] implement this model. Esterel [100] is an imperative language that uses sequential and concurrent statements to describe hierarchically arranged modules. Lustre [118] and Signal [119] are dataflow-oriented languages. Most of these languages are static in the sense that they do not allocate new resources at run-time. Such synchronous programs can be verified formally at compile-time and compiled into hardware [120].

The SACRES project has been exploring the combination of stream-based semantics in Lustre and state-based semantics in StateCharts to relate the synchronous reaction to the data flow processing and the asynchronous reaction to events. The SYRF project has resulted in the development of cross-compilation tools for Lustre, Signal and Esterel (loose integration), an environment for multi-paradigm modeling (tight integration), and code distribution for embedded systems [36].

### A.5.2 *-charts and El Greco

*-charts [121] (pronounce ”starcharts”) shows how to embed hierarchical FSM within a variety of concurrency models in Ptolemy [26] such as DE, SR and a subset of dataflow called Heterochronous Dataflow - HDF, which extends the SDF model by allowing changes in tokens production/consumption rates only between iterations. As an example, in a FSM-HDF combination, each state corresponds to an iteration with fixed production/consumption rates, i.e. to a static schedule. Intuitively, *-charts semantics are so specified that the hierarchy can be arbitrarily deep and that compositions of concurrency models and FSM can be nested anywhere within the model. A module that belongs to a specific state in hierarchical FSM is active if and only if the FSM is in that state. Thus, *-charts do not define a model of concurrency [122].

The objective of El Greco [98] is to provide a path to implementation as embedded software, synthesizable hardware or both. Applications are specified in the form of dataflow graphs with Kahn or CSDF-like semantics, hierarchical synchronous finite state machines with Esterel-like semantics, or a mixture, arbitrarily nested. When dataflow graphs are placed in a control context, the graph execution is fully controllable.

### A.5.3 SDL

SDL (Specification and Description Language [123]) is a formal language intended for the specification, simulation and design of complex interactive applications involving many concurrent activities, such as telecommunication protocols [124]). Each process in the SDL model is implemented as a two-level hierarchical (extended) Finite State Machine. The ordering behavior of a process is modeled by a first FSM that receives and sends tokens whose tags are partially ordered. The functional behavior is modeled by a second FSM that ex-

ecutes (possibly recursive) procedures whose input and output tokens have totally ordered tags. Thus, in contrast to FSMD where communication is synchronous, the SDL model is a GALS model.

Communication in SDL can be done with two basic primitives, by exchanging tokens asynchronously via a unique unbounded FIFO queue per process, or via remote procedures calls. The network topology and the size of the queues may change at run-time. Implementing such a behavior may be done by instantiating all processes and pre-sizing queues at compile-time, or by using a real-time operating system to support dynamic memory allocation and task allocation [109].

### A.5.4 CFSM

The Co-Design FSM model (CFSM) is the backbone of POLIS [49], which is a system developed for function-architecture co-design with particular emphasis on control-dominated applications and software development [109]. CFSMs are reactive FSMD that run concurrently and communicate through unidirectional single-place (1-deep) buffers. In contrast to SDL, communication channels are bounded. Events are timestamped and totally ordered with respect to a global notion of time. Their (local) processing in the functional behavior of a CFSM is synchronous. Communication in a CFSM network is asynchronous and has undefined delays. Thus, CFSM is a GALS model. Each input port stores the most recently received event in a single place buffer and is read exactly once per transition. However, a CFSM has a non-deterministic behavior because reading and writing events may occur in nearly any order.

Note that the communication primitive is different from a rendezvous since many processes can produce and consume asynchronous events concurrently. CFSMs may loose events because they run independently and communication channels are only 1-deep. Nothing prevents a producer overwriting an event that has not been consumed.

### A.5.5 RPN

The Reactive Process Networks model (RPN [54]) aims at integrating event-triggered aspects of modeling with dataflow aspects of modeling so as to allow re-configuration of dataflow. It starts from a KPN model and allows re-configuration only at specific execution points, between execution cycles of the processes. RPN explicitly separates dataflow ports from parameters ports and distinguishes dataflow FIFO channels from events FIFO channels. The structure of a RPN may be modified dynamically by identifying events with functions that transform the configuration of the RPN. Although it is possible to link functions to events in object-oriented languages and thus support dynamic network reconfiguration, it is unlikely that functions are being communicated in an implementation.

## A.6 Related work

It would be wrong not to acknowledge the influence of the FunState (Functions driven by State machines [122]) model, which explicitly separates control and data, and supports state-based and stream-based models. A comparison of specification-level models of computation is given in [18] and [109], which rely on the tagged signal (meta-)model. Our separation of state-based and stream-based models is inspired by the classifications given in the Artist roadmap [36] and in [42]. Our contribution is an extension of the analysis and classification to other models of computation.

## A.7 Conclusions

In this Appendix we analyzed the semantics of state-based and stream-based models of computation in a way that is accessible to a broad community. We discussed the interest of their individual semantics in terms of synchrony, concurrency, communication, determinism and composition. Our analysis and classification may be biassed since our objective is the modeling of large-scale and distributed embedded signal processing systems such as next generation radio telescopes.

# Bibliography


[1] J. D. Bregman. System optimization of multi-beam aperture synthesis arrays for survey performance. *The SKA: an engineering perspective*, P.Hall - Springer, 2005.

[2] Square kilometer array radio telescope, www.skatelescope.org.

[3] Very large array, http://www.vla.nrao.edu/.

[4] A. R. Thomson, J. M. Moran, and G. W. Swensson Jr. . *Interferometry and Synthesis in Radio Astronomy*. John Wiley & Sons Pub., 1986.

[5] H.R. Butcher. On the horizon: a new generation of radio telescopes. *SPIE Astronomical Telescopes and Instrumentation 2000*, 4015 Radio Telescopes, 2000.

[6] Lofar radio telescope, www.lofar.org.

[7] J.H. Justice, N.L. Owsley, J.L. Yen and A.C. Kak. *Array Signal Processing*. Prentice-Hall, 1985.

[8] D.E. Dudgeon. Fundamentals of digital array processing. *Proceedings IEEE*, 65:898–904, June 1977.

[9] A. Gunst. Lofar architectural design document. *LOFAR-ASTRON-ADD-006*, February 2007.

[10] IBM BlueGene/L Team. An overview of the bluegene/l supercomputer. 2002.

[11] A. Huijgen. Lofar remote station subsystem requirement specification. *Astron internal document LOFAR-ASTRON-SRS012*, 2005.

[12] Altera. *website: http://www.altera.com*.

[13] Xilinx. *website: http://www.xilinx.com*.

[14] R. Crochiere and L. Rabiner. *Fundamentals of Multirate Signal Processing*. Prentice-Hall, 1983.

[15] P. Vaidyanathan. *Multirate Systems and Filter Banks*. NJ: Prentice-Hall, 1993.

[16] S. Alliot. Architecture exploration for large scale array signal processing systems. *Ph.D dissertation, Leiden University, The Netherlands*, December 2003.

[17] I. Somerville. *Software Engineering, 6th Edition*. Addison Wesley, 2000.

[18] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits ans Systems*, 17(12), December 1998.

[19] D.D. Siljak and A.I Zecevic. Large-scale and decentralized systems. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999.

[20] A.I. Zecevic and D.D. Siljak. Control of large-scale systems in a multiprocessor environment. *Applied Mathematics and Computation, Elsevier Pub.*, (164), 2005.

[21] D. Schmidt et al. Cosmic: An mda generative tool for distributed real-time and embedded component middleware and applications. *OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, November 2002.

[22] S.J. Mellor, K. Scott, A. Uhl and D. Weise. *MDA Distilled - Principles of Model-Driven Architecture*. Addison-Wesley Publishers, 2004.

[23] The ANTARES Collaboration. Antares - a deep sea telescope for high energy neutrinos. *Technical Proposal 99-01*, 1999.

[24] The IceCube Collaboration. Icecube:a kilometer-scale neutrino observatory. *Proposal to theNational Science Foundation*, 1999.

[25] S. Neuendorffer J. Ludvig, J. McCarty and S. R. Sachs. Reprogrammable platforms for high-speed data acquisition. *Design Automation for Embedded Systems*, 7(4):341–364, November 2002.

[26] E. A. Lee J. Buck, S. Ha and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994.

[27] R. McNulty et al. Simulations and prototyping of the lhcb l1 and hlt triggers. *Computing in High Energy Physics and Nuclear Physics 2004*, 2004.

[28] Berkeley Emulation Engine's website. http://bee2.eecs.berkeley.edu/. 2006.

[29] J. Wawrzynek R. Brodersen, C. Cheng and D. Werthimer. Bee2: A multi-purpose computing platform for radio telescope signal processing applications. *International SKA meeting*, July 2004.

[30] A. Parsons et al. Petaop/second fpga signal processing for seti and radio astronomy. *14th Asilomar Conference on Signals, Sysyems and Computers*, pages 2031–2035, October 2006.

[31] E. Lenormand and G. Edelin. An industrial perspective : Pragmatic high end signal processing design environment at thales. *Proceedings of the 3rd International Samos Workshop on Synthesis, Architectures, Modeling, and Simulation*, 2003.

[32] H. Hsieh L. Lavagno C. Passerone F. Balarin, Y. Watanabe and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer Magazine*, pages 45–52, April 2003.

[33] A. Sangiovanni-Vincentelli. Quo vadis sld: Reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[34] A. Davare et al. A next-generation design framework for platform-based design. *Conf. on Using Hardware Design and Verification Languages (DVCon)*, February 2007.

[35] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, August 1974.

[36] B. Bouyssounouse and J. Sifakis (Eds.). Embedded systems design - the artist roadmap for research and development. *Springer Pub.*, 2005.

[37] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proc. of the IEEE*, 79(9):1270–1282, September 1991.

[38] R. Miller. *Communication and Concurrency*. Prentice Hall, 1989.

[39] VSI Alliance. Vsia system level design model taxonomy document. July 2001.

[40] H. Nikolov and E. Deprettere. Parameterized stream-based functions dataflow model of computation. *Proc. of the 6th Workshop on Optimizations for DSP and Embedded Systems (ODES2008)*, April 2008.

[41] W. Lubberhuizen. Epa-rsp firmware functional specification. *Internal report LOFAR-ASTRON-SDD-001*, September 2005.

[42] S.A. Edwards. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, 1997.

[43] E.A. Lee and T.M. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–799, May 1995.

[44] E.A. Lee and B. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, September 1987.

[45] D. Harel. Statecharts, a visual formalism for complex systems. *Science of Computer Programming*, (8):231–274, 1987.

[46] R. Milner. A calculus of communicating systems. *Springer-Verlag*, 1980.

[47] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[48] J. Bergstra and J. Klop. Acp: A universal axiom system for process specification. *CWI Quarterly*, (15):3–23, 1987.

[49] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-Design of Embedded Systems – The POLIS Approach.* Kluwer Academic Publishers, 1997.

[50] E. Lawerman and T. Muller. Mac-cep use cases. *Internal report LOFAR-ASTRON-UCD-005*, December 2004.

[51] B. Kienhuis and E. Deprettere. Modelling stream-based applications using the sbf model of computation. *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 34(3):291–300, 2003.

[52] B. Bhattacharya and S. Bhattacharyya. Parameterized dataflow modeling of dsp systems. *Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, June 2000.

[53] T. Stefanov H. Nikolov and E. Deprettere. Modeling and fpga implementation of applications using parameterized process networks with non-static parameters. *Proc. of the Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, April 2005.

[54] M. Geilen and T. Basten. Reactive process networks. *Proc. of EMSOFT04*, September 2004.

[55] Synchronous reactive formalism project (syrf), http://www.verimag.fr/synchrone/syrf.

[56] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign*, March 2002.

[57] F. De Bernardinis A. Sangiovanni-Vincentelli, L. Carloni and M. Sgroi. Benefits and challenges for platform-based design. *Proc. of the Intl. Design Automation Conference (DAC 2004)*, June 2004.

[58] V. Dobrosav Zivkovic. Architecture exploration... *Ph.D dissertation, Leiden University, The Netherlands*, to be published.

[59] M. van Veelen. Considerations on modeling for early detection of abnormailities in locally autonomous distributed systems. *Ph.D dissertation, Groningen University, The Netherlands*, March 2007.

[60] A. Caldwell et al. Gtx: The marco gsrc technology extrapolation system. *DAC*, pages 693–698, 2000.

[61] M.Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, C-21:948, 1972.

[62] S. Alliot, E. Deprettere, and A. Kokkeler. A modular approach for a large scalable embedded signal processing system. *Proceedings Workshop on Embedded Systems, Progress 2000 second edition*, October 2000. http://www.astron.nl/∼alliot.

[63] A. Roman V. Bhanot, D. Paniscotti and B. Trask. Using domain-specific modeling to develop software defined radio components and applications. *OOPSLA Workshop on Domain-Specific Modeling*, October 2005.

[64] E. Kooistra. Fpgas for lofar remote station signal processing. *FPGAs in Radio Astronomy Workshop*, February 2007.

[65] Q. Zhu A. Davare and A.L. Sangiovanni-Vincentelli. A platform-based design flow for kahn process networks. Technical Report UCB/EECS-2006-30, EECS Department, University of California, Berkeley, March 28 2006.

[66] Bart Kienhuis Todor Stefanov and Ed Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. *International symposium on Hardware/software codesign*, 2002.

[67] A. Turjan. Compiling nested loops programs to process networks. *Ph.D dissertation, Leiden University, The Netherlands*, March 2007.

[68] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[69] P. van der Wolf et al. Design and programming of embedded multiprocessors: An interface-centric approach. *CODES+ISS'04*, September 2004.

[70] C. Erbas A. D. Pimentel and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. on Computers*, 55(2), February 2006.

[71] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from Matlab for embedded signal processing architectures. *8th International Workshop on Hardware/Software Co-Design (CODES'2000)*, May 2000.

[72] C. Zissulescu, T. Stefanov, B. Kienhuis, and Ed Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proceedings 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, September 2003.

[73] A. Nechypurenko et al. Applying mda and component middleware to large-scale distributed systems: A case study. *IST 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, March 2004.

[74] Unified modeling language (uml). *http://www.uml.org/*, 2006.

[75] Systems modeling language (sysml). *http://www.sysml.org/specs.htm*, 2006.

[76] Mentor Graphics. *website: http://www.mentor.com*.

[77] Synopsys. *website: http://www.synopys.com*.

[78] The Mathworks and Altera. Dsp builder user guide. 2002.

[79] Simplicity. *website: http://www.simplicity.com*.

[80] B. Kienhuis T. Harris, R. Walke and E. Deprettere. Compilation from Matlab to process networks realized in FPGA. *35th Asilomar Conf. on Signals, Systems and Computers*, November 2001.

[81] T. Stefanov H. Nikolov and E. Deprettere. Multi-processor system design with espam. *Int. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS'06)*, October 2006.

[82] LEON2. *website: http://www.gaisler.com*, 2004.

[83] ARM AMBA. *website: http://www.arm.com*.

[84] J. Labrosse. *MicroC/OS-II, The Real-Time Kernel, 2nd Edition*. CMP Books, 2002.

[85] Opencores. www.opencores.org.

[86] J. Bol and M. Lammertink. Astron technical report: Middleware-based communication mechanism for a network of software processors in an fpga. Technical report, Astron, Dwingeloo, The Netherlands, June 2005.

[87] Open Core Protocol International Partnership OCP-IP. *website: http://www.ocpip.org*, 2004.

[88] F. Wagner et al. Strategies for the integration of harware and software ip components in embedded systems-on-chip. *Integation, the VLSI Journal*, 37:223–252, 2004.

[89] StstemC. *website: http://www.systemc.org*.

[90] M. Bombana and F. Bruschi. Systemc-vhdl co-simulation and synthesis in the hw domain. *in Proc. of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, 2003.

[91] C. Zissulescu B. Kienhuis S. Derrien, A. Turjan and E. Deprettere. Deriving efficient control in process networks with compaan/laura. *International Journal of Embedded Systems*, 1(7), 2005.

[92] A. Fraboulet and T. Risset. Efficient on-chip communications for data-flow ips. *in Intl. Conference on Application-specific Systems, Architectures and Processors (ASAP'04)*, 2004.

[93] Virtual Socket Interface Alliance VSI. *website: http://www.vsi.org*.

[94] The object management group (omg). *http://www.omg.org/*, 2006.

[95] System Design Industrial Council of European Telecom Industries CSYDIC-Telecom. *website: http://www.sydic.vitamib.com*.

[96] N.S. Voros. *System Design Reuse, Chapter 3*. Kluwer Academic Publishers, 2003.

[97] F. Vahid T. Givaris and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized soc. *IEEE/ACM ICCCAD*, 2001.

[98] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in el greco. *Proc. of the Intl. Workshop on Hardware/Software Codesign (CODES)*, May 2000.

[99] R. Lauwereins G. Bilsen, M. Engels and J.A. Peperstraete. Cyclo-static data flow. *Proc. ICASSP*, 1995.

[100] F. Boussinot and R. De Simone. The esterel language. *Proc. of the IEEE*, 79(9), 1991.

[101] J. van Eijndhoven K. Walters M. Rutten, E-J. Pol and G. Essink. Dynamic reconfiguration of streaming graphs on a heterogeneous multiprocessor architecture. *SPIE Electronic Imaging: Embedded processors for Multimedia and Communications II*, 5683, January 2005.

[102] P. Stravers and J. Hoogerbrugge. Single-chip multiprocessing for consumer electronics. *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation (SAMOS 2004)*, pages 215–234, 2004.

[103] A. Radulescu and K. Goossens. Communication services or networks on chip. *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation (SAMOS 2004)*, pages 193–214, 2004.

[104] M. van Veelen S. Alliot, L. Nicolae and J. Lemaitre. An exploration tool for the large scale signal processing systems. *Progress Symposium*, 2003.

[105] S. Shukla P. Schaumont and I. Verbauwhede. Design with race-free hardware semantics. *Proc. of the Intl. Conf. on Design and Testin Europe (DATE'06)*, March 2006.

[106] C.A. Petri. Communication with automata. *phD thesis, Darmstadt Inst. of Technology*, 1962.

[107] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[108] R.J. Glabbeek. Notes on the methodology of ccs and csp. *Proc. of the Intl. Workshop on Algebra on communicating processes*, 1997.

[109] A. Sangiovanni-Vincentelli L. Lavagno and E. Sentovich. Models of computation for embedded systems design. September 1998.

[110] R. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *Appl. Math.*, 14(6):1390–1411, 1966.

[111] P. K. Murhpy S. Bhattacharyya and E.A. Lee. Software synthesis from dataflow graphs. *Kluwer Academic Publishers*, 1996.

[112] E.A. Lee. Representing and exploiting data parallelism using multidimensional dataflow diagrams. *Proc. ICASSP*, 1993.

[113] M. Pankert S. Ritz and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. *Proc. Int. Conf. on Application-Specific Array Processors*, 1993.

[114] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs using the token flow model. *Proc. of the Int'l Conf. on Acoustics, Speech, and Signal Processing*, April 1993.

[115] J.T. Buck. A dynamic dataflow model suitable for efficient mixed hardware and software implementations of dsp applications. *Proc. of the IEEE*, 1994.

[116] S. Neuendorfer and E.A. Lee. Hierarchical reconfiguration of dataflow models. *Conf. on Formal Methods and Models for Codesign (MEMOCODE'04)*, June 2004.

[117] S. Puthenpurayil S. Saha and S. Bhattacharyya. Dataflow transformations in high-level dsp system design. *Proc. of the Intl. Symposium on System-on-Chip*, November 2006.

[118] P. Raymond N. Halbwachs, P. Caspi and D. Pilaud. The synchronous data flow programming language lustre. *Proc. of the IEEE*, 79(9):1305–1319, May 1991.

[119] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Trans. on Automatic Control*, 35(5):525–546, May 1990.

[120] G. Berry. A hardware implementation of pure esterel. *Proc. of the Intl. Workshop on Formal Methods in VLSI Design*, January 1991.

[121] B. Lee A. Girault and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(6), June 1999.

[122] D. Ziegenbein R. Ernst L. Thiele, K. Strehl and J. Teich. Funstate - an internal design representation for codesign. *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD99)*, 1999.

[123] W. Takach and A. Wolf. An automaton model for scheduling constraints in synchronous machines. *IEEE Tr. on Computers*, 44(1):1–12, January 1995.

[124] CCITT (International Telecommunication Union). Functional specification and description language (sdl), recommandations z.101-z.104. VI, Fasc. VI.7, 1981.

# Curriculum Vitae

Jérôme Lemaitre was born on March 22nd, 1979 in Compiègne, France. In 1997 he received his Baccalaureate in the scientific branch after which he joined a preparatory class in mathematics and physics for one year. This was followed by studies in electronics and optics at the advanced engineering school, Ecole Supérieure des Procédés Electroniques et Optiques (ESPEO) in Orléans, France. He completed a post-graduate degree in 2002 at the ESPEO, majoring in embedded systems. The same year, he joined the Netherlands Foundation for Radio Astronomy (ASTRON) in Dwingeloo, the Netherlands, where he was first involved in the implementation of digital filters for the Low Frequency Array radio telescope (LOFAR) project. After a year, he worked on the specification and implementation of control interfaces for signal processing applications onto re-configurable platforms in the context of the MASSIVE project with Leiden University, which culminated in this thesis.

# Samenvatting

**Model-based Specification and Design of Large-Scale Embedded Signal Processing Systems**.

Fasegestuurde radiotelescopen zoals de Low Frequency Array (LOFAR) en de Square Kilometer Array (SKA) zijn grootschalige en gespreide signaalverwerkingssystemen. In deze systemen worden signalen verkregen via duizenden antennes voordat zij bij een hoge snelheid worden gedigitaliseerd en in toenemende mate beperkt door het verwerken van taken in clusters (stations genoemd). Het gedrag van het signaal dat taken verwerkt die in stations worden uitgevoerd, wordt bestuurd en gecontroleerd in runtime. Besturingsinformatie wordt gezonden van een centrale locatie en steeds verder gedecentraliseerd naar het niveau van gespecialiseerde componenten in stations. Deze componenten voeren signaalverwerkingstaken uit en zenden controle-informatie terug naar de centrale locatie.

In dit proefschrift concentreren we ons op de specificatie en het ontwerp van dergelijke stations, wat in feite het digitale signaalverwerkingsgedeelte, het besturings- en controlegedeelte en de synchronisatie en koppeling van de twee gedeeltes inhoudt. Wij nemen aan dat systeemniveau-specificaties worden uitgedrukt in termen van applicatie, architectuur en het in kaart brengen van de eerste in de laatste. Wij zijn van mening dat er een niveau van abstractie is, implementatieniveau-specificatie genaamd, vanwaar verschillende delen van het systeem kunnen worden omgezet in een echte implementatie gebaseerd op commercieel beschikbare tools. Onze algemene probleemcontext is de systeemniveau-specificatie op een gestructureerde manier om te zetten naar een implementatieniveau-specificatie. Het bijzondere probleem dat in deze these wordt behandeld is de koppeling en synchronisatie van het signaalverwerkingsgedeelte en het besturings- en controlegedeelte, terwijl de twee gedeeltes eerst gesoleerd worden beschouwd. Dit is een probleem omdat de twee gedeeltes verschillend zijn gestructureerd en zich verschillend gedragen. Deze twee gedeeltes moeten gekoppeld en gesynchroniseerd worden zonder het functionele gedrag van het dominante signaalverwerkingsgedeelte te wijzigen en zonder de prestatie/kosten van de twee gedeeltes significant te wijzigen tijdens het schalen van het systeem.

Om de complexiteit van de systemen te beheersen gaan wij ervan uit dat we het abstractieniveau moeten verhogen. Dit houdt in dat we nduidige modellen nodig hebben, zodat we ontwerpbeslissingen kunnen nemen gebaseerd op modellen eerder dan op intutie of echte implementatiedetails. Omdat wij specificaties op systeemniveau moeten afleiden, drukken we onze specificaties uit gebaseerd op modellen. Wij maken de modelgebaseerde specificatie van de applicatie los van de modelgebaseerde specificatie

van de architectuur en verschaffen middelen om deze twee specificaties met elkaar te combineren.

In Hoofdstuk 2 concentreren wij ons op de specificatie van de applicatie. Wij gebruiken modellen waarvan de semantiek de manier waarop gegevens in knooppunten worden verwerkt en gecommuniceerd tussen knooppunten in een netwerk nduidige vastleggen. Het functionele gedrag van het signaalverwerkingsgedeelte is gebaseerd op een stroomgebaseerd model van berekening dat uitstekend geschikt is de streamingapplicaties te vertegenwoordigen die een hoog niveau van parallellisme hebben. Het functionele gedrag van het besturings- en controlegedeelte wordt gespecificeerd op basis van een state-based model van berekening dat geschikt is de uitvoering van de taken in reactie op gebeurtenissen te vertegenwoordigen. De twee modellen worden gesynchroniseerd door de introductie van een begrip van tijd dat alleen bekend is in het besturings- en controlegedeelte en door dit begrip van tijd in verband te brengen met periodieke intervallen waarbinnen taken worden uitgevoerd in het signaalverwerkingsgedeelte.

In Hoofdstuk 3 concentreren wij ons op de specificatie van de architectuur, in verband met onderling verbonden componenten die worden genomen van een unieke bibliotheek. Deze componenten ondersteunen gespecialiseerde diensten voor verwerking, opslag, communicatie, etc. De bibliotheek omvat informatie over de prestatie en kosten van de componenten en regels waaraan moet worden vastgehouden bij het verbinden van de componenten. Componenten die worden gebruikt op lagere niveaus van de hirarchie worden aangegeven als witte vakjes: hun interne modules zijn toegankelijk voor simulatie. Op hogere niveaus van de hirarchie, worden componenten weergegeven als zwarte vakjes: hun interne modules zijn verborgen. Zwarte vakjes geven het verband aan tussen output-hoeveelheden en inputhoeveelheden gebaseerd op eenvoudige vergelijkingen, waarbij parameterwaarden worden gekalibreerd door gebruik te maken van informatie verkregen van lagere niveaus van de hirarchie. In het signaalverwerkingsgedeelte is de structuur in hoge mate parallel en ondersteunt de samenstelling intensieve berekeningen over en transport van hoge doorvoergegevens. De samenstelling van het besturings- en controlegedeelte maakt de overdracht mogelijk van sporadische berichten en het uitvoeren van opvolgende taken in reactie op deze berichten. Het besturings- en controlemodel heeft een boomstructuur, waarvan de knooppunten gekoppeld zijn met de berekeningsknooppunten in het signaalverwerkingsmodel van berekening.

In Hoofdstuk 4 concentreren we ons op het in kaart brengen van de applicatie in de architectuur, gebaseerd op iteratieve transformaties. We nemen aan dat deze transformaties beschikbaar zijn in een bibliotheek. Ontwerpers kunnen kiezen welke transformatie ze willen toepassen op elke iteratiecyclus. Transformaties worden beperkt door de koppeling tussen signaalverwerkingsgedeelte en het besturings- en controlegedeelte. Vanuit de implementatie gezien zijn mappingtransformaties hoog niveau compilatiestappen die lokaal en automatisch kunnen worden toegepast om de prestatie/kosten van een deel van het systeem te optimaliseren. Een implementatieniveau-specificatie is een specificatie die verfijnd is naar het niveau van (netwerken van) multiprocessor systems-on-chip (MPSoC). Van dit niveau nemen we aan dat elk gedeelte wordt gemplementeerd op basis van een geschikte compilatie en/of synthesetools die (IP-)componenten integreren en plakken volgens regels die in de bibliotheek zijn vastgesteld.

In Hoofdstuk 5 presenteren wij casestudies rond de integratie en porting van IP-componenten beginnend van hoog niveau specificaties en rond het koppelen tussen componenten in het signaalverwerkingsgedeelte en componenten in het besturings- en controlegedeelte. Deze casestudies laten de zwakte zien van in andere opzichten zeer gewenste systeemniveau ontwerpmethodes bij het evalueren met betrekking tot snelle, nauwgezette en systematische IP integratie.

ISBN 978-90-9023497-7