KTH

Royal Institute of Technology

School of Information and Communication Technology
Electronic Systems

Mapping to a Time-predictable Multiprocessor System-on-Chip

Master of Science Thesis in System-on-Chip Design
Stockholm, November 2012
TRITA-ICT-EX-2012:297

Author: Christian Amstutz
Examiner: Assoc. Prof. Ingo Sander, KTH, Sweden
Supervisors: Docent Johnny Öberg, KTH, Sweden
Francesco Robino, KTH, Sweden
Abstract

Traditional design methods could not cope with the recent development of multiprocessor systems-on-chip (MPSoC). Especially, hard real-time systems that require time-predictability are cumbersome to develop. What is needed, is an efficient, automatic process that abstracts away all the implementation details. ForSyDe, a design methodology developed at KTH, allows this on the system modelling side. The NoC System Generator, another project at KTH, has the ability to create automatically complex systems-on-chip based on a network-on-chip on an FPGA. Both of them support the synchronous model of computation to ensure time-predictability. In this thesis, these two projects are analysed and modelled. Considering the characteristics of the projects and exploiting the properties of the synchronous model of computation, a mapping process to map processes to the processors at the different network nodes of the generated system-on-chip was developed. The mapping process is split into three steps: (1) Binding processes to processors, (2) Placement of the processors on net network nodes, and (3) scheduling of the processes on the nodes. An implementation of the mapping process is described and some synthetic examples were mapped to show the feasibility of algorithms.

Keywords: Mapping, Synchronous Systems, Multiprocessor System-on-Chip, Design Methodology, Time-predictability
Acknowledgment

Firstly, I wish to thank the whole research group around Ingo Sander and Johnny Öberg for giving me the possibility to conduct my Master thesis at their department. I always had the feeling to get the best possible support when needed. I wish to thank Francesco Robino for his supervision and the guidance, that helped me to keep the work on track. I wish to thank Johnny Öberg for the help with the NoC System Generator. I wish to thank Seyed Hosein Attarzadeh Niaki, that he always took the time to answer my questions regarding ForSyDe. Last but not least, I wish to thank Ingo Sander for his inputs, that went beyond of what can be expected from an examiner. I will also remember the discussion we had about research in general.

Then, I wish to thank everybody that inspired and supported me on the path to my master studies in Stockholm. This starts with ABB that gave me the possibility to get to know Sweden in an exchange during my apprenticeship. I wish to thank also the professors at Fachhochschule Nordwestschweiz for the excellent education that they provided me. I also wish to thank Christoph Holliger for promoting exchange and master studies at Fachhochschule Nordwestschweiz. Then I wish to thank Deniz Akkaya for being my lab partner in many courses here at KTH. I will keep in my memories not only the times we had to work hard, but mainly the lunch breaks and the discussions we had.

Last but not least, I wish to thank my family and friends in Switzerland. My family that always supported me on my educational path and always makes feel home when I visit them. Finally, I wish to thank Christian Jenni and Biagio Mancina for their great friendship and their understanding, that I moved to Stockholm.

Christian Amstutz
Stockholm, November 2012
# Contents

List of Figures ............................................................ ix  
List of Tables .............................................................. xi  
List of Listings .............................................................. xii  
Abbreviations & Symbols ................................................ xiii  

1. **Introduction** .......................................................... 1  
   1.1. Background ......................................................... 1  
   1.2. Problem .......................................................... 2  
   1.3. Method .......................................................... 2  
   1.4. Outline .......................................................... 3  

2. **Synchronous System Design with ForSyDe** ......................... 5  
   2.1. Models of Computation ........................................... 5  
   2.2. Synchronous Model of Computation ............................ 6  
   2.3. Synchronous Hardware .......................................... 7  
   2.4. Synchronous Languages ........................................ 8  
   2.5. ForSyDe: A Design Methodology ............................... 9  
   2.6. The ForSyDe System Model ..................................... 11  
   2.7. ForSyDe Process Constructors of the Synchronous MoC .......... 12  
       2.7.1. Process Constructor: Combinational .................. 13  
       2.7.2. Process Constructor: Delay ........................... 13  
       2.7.3. Process Constructor: Zip / Unzip .................... 14  
       2.7.4. Process Constructor: Mealy, Moore .................. 14  
       2.7.5. Process Constructor: Source ......................... 15  
       2.7.6. Process Constructor: Sink ............................ 15  
   2.8. Model description files (XML & C++) .......................... 16  
       2.8.1. Structural Description (XML) ......................... 16  
       2.8.2. Process Code (C++) .................................... 19  
   2.9. Conclusion ..................................................... 20  

3. **The Network-on-Chip System Generator** ......................... 21  
   3.1. Networks-on-Chip ............................................... 21  
   3.2. Topology and Addressing ...................................... 22  
   3.3. Flow control .................................................. 23
## Contents

3.3.1. Deadlock & Livelock ............................................ 24  
3.4. Routing ............................................................... 25  
3.4.1. Routing algorithm classes ........................................ 25  
3.4.2. Dimension-Order-Routing ......................................... 25  
3.4.3. Deflection routing ................................................ 26  
3.4.4. Discussion on Deflection Routing ............................... 27  
3.5. Message Decomposition ............................................. 29  
3.5.1. Packet Structure ................................................ 29  
3.5.2. Flit Structure .................................................... 30  
3.6. Network Node ....................................................... 31  
3.6.1. The Network Switch .............................................. 31  
3.6.2. The Resource Network Interface (RNI) .......................... 32  
3.6.3. The Network Resource ........................................... 33  
3.7. Synchronizing the Execution: Heartbeat Principle ............ 34  
3.8. The System Designer’s View ....................................... 34  
3.8.1. The Network Transfer Process ................................. 34  
3.8.2. NoC Platform Generation Process .............................. 36  
3.8.3. Device Driver .................................................... 37  
3.8.4. Target Description Header (software_configuration.h) .... 38  
3.9. System Description Files ......................................... 38  
3.9.1. Target Description File (XML) ................................. 39  
3.9.2. Target code (C++) ............................................... 40  
3.10. Conclusion .......................................................... 42

4. From ForSyDe to the NoC platform ................................. 43  
4.1. The Automatic System Generation Process ....................... 43  
4.2. System Modelling with ForSyDe ................................... 45  
4.3. Code Generation (Pre-Mapping) .................................. 45  
4.4. Process Analysis ................................................... 46  
4.5. Mapping ............................................................. 47  
4.6. Code Generation (Post-Mapping) .................................. 48  
4.7. NoC Platform Generation .......................................... 48  
4.8. Conclusion .......................................................... 48

5. System Model and Mapping ........................................... 49  
5.1. Modelling of the Application .................................... 49  
5.1.1. Process Model .................................................. 49  
5.1.2. Signal Model .................................................... 51  
5.1.3. Time Model of a Process ....................................... 51  
5.2. Modelling of the Architecture ................................... 53  
5.2.1. Execution Unit Model ........................................... 54  
5.2.1.1. Hyper Process ............................................... 54  
5.2.2. Communication Link Model ..................................... 56  
5.2.2.1. Communication via the Network-on-Chip (Type I) ....... 57
## List of Figures

2.1. Examples of valid and invalid synchronous signals .................................. 7
2.2. Two different execution schemes used for synchronous languages ............... 9
2.3. The design process of ForSyDe ................................................................. 11
2.4. Symbols of combinational process constructors ......................................... 13
2.5. Symbols of delay process constructors .................................................... 13
2.6. Symbol of the zip and unzip process constructors ....................................... 14
2.7. Symbols and decomposition of the FSM process constructors ....................... 14
2.8. Symbol of the source process constructor ............................................... 15
2.9. Symbols of the sink process constructors ............................................... 15
2.10. Example ForSyDe system: multiply-accumulator ........................................ 17
3.1. Different network topologies used in Networks-on-Chip ............................. 22
3.2. Addressing of nodes in a 3x3x3 3D-mesh network .................................... 23
3.3. Examples for dimension-order-routing .................................................... 26
3.4. Deflection routing example ........................................................................ 27
3.5. Packet format in the generated NoC ......................................................... 30
3.6. Flit format in the generated NoC ............................................................... 30
3.7. Structure of a network node in a 2D-mesh network ..................................... 32
3.8. Structure of the RNI ................................................................................. 33
3.9. The Platform Generation Process of the NoC System Generator ................... 36
4.1. Automatic System Generation Process ....................................................... 44
5.1. Problem graph of a multiply-accumulator ................................................ 50
5.2. Three different time models of processes .................................................. 52
5.3. Architecture Graph of a 2x2 2D-mesh NoC .............................................. 54
5.4. Time model of a hyper process .................................................................. 56
5.5. Architecture graph of the NoC platform .................................................... 59
5.6. Example of a specifiaction graph ................................................................ 61
5.7. Mapping example to four execution units ................................................. 63
5.8. Example of a case in which congestion is allowed ..................................... 67
5.9. Examples of different optimal schedules ................................................... 70
5.10. Hyper process example that can be scheduled in different ways ................. 73
5.11. Heartbeat evaluation example .................................................................. 74
6.1. Syncmapper System Description File example ........................................... 78
6.2. Class structure of the syncmapper program ............................................. 81
List of Figures

7.1. Symbolic problem graphs of the synthetic examples . . . . . . . . . . 90
7.2. Speedup of the structure examples . . . . . . . . . . . . . . . . . . . 94
7.3. Efficiency of the structure examples . . . . . . . . . . . . . . . . . . 94
7.4. Speedup of the size examples . . . . . . . . . . . . . . . . . . . . . . . 97
7.5. Efficiency of the size examples . . . . . . . . . . . . . . . . . . . . . . 97
7.6. Speedup of the mpeg4 examples . . . . . . . . . . . . . . . . . . . . 98
7.7. Efficiency of the mpeg4 examples . . . . . . . . . . . . . . . . . . . 98

A.1. Problem graph of the chain10 example . . . . . . . . . . . . . . . . 115
A.2. Problem graph of the chain20 example . . . . . . . . . . . . . . . . 117
A.3. Problem graph of the chain40 example . . . . . . . . . . . . . . . . . 119
A.4. Problem graph of the parallel5 example . . . . . . . . . . . . . . . 121
A.5. Problem graph of the tree example . . . . . . . . . . . . . . . . . . . 123
A.6. Problem graph of the mpeg4 example . . . . . . . . . . . . . . . . . 125
A.7. Problem graph of the mpeg4_loop example . . . . . . . . . . . . . 127
List of Tables

3.1. Naming conventions for the mesh topology on the NoC platform . . 24
3.2. Network transfer process in the generated NoC . . . . . . . . . . . 35
5.1. Comparison of the three communication types on the NoC platform 57
7.1. Timing model parameters for the result generation . . . . . . . . . . 92
7.2. Heartbeat periods of the structure example mappings . . . . . . . 93
7.3. Heartbeat periods of the size example mappings . . . . . . . . . . 96
7.4. Heartbeat periods of the MPEG-4 decoder example mappings . . . 96
7.5. Processes per processor necessary to achieve a certain efficiency . . 99
A.1. Process and Signal properties of the chain10 example . . . . . . . 115
A.2. Process and Signal properties of the chain20 example . . . . . . . 116
A.3. Process and Signal properties of the chain40 example . . . . . . . 118
A.4. Process and Signal properties of the parallel5 example . . . . . . . 120
A.5. Process and Signal properties of the tree example . . . . . . . . . 122
A.6. Process and Signal properties of the mpeg4 example . . . . . . . . 124
A.7. Process and Signal properties of the mpeg4-loop example . . . . . . 126
List of Listings

2.1. ForSyDe XML description of a multiplier-accumulator . . . . . . . . 17
2.2. Example of a composite process in a ForSyDe XML . . . . . . . . . . . 18
2.3. Example of a leaf process in a ForSyDe XML . . . . . . . . . . . . . 19
2.4. Process code of an adder in ForSyDe-SystemC . . . . . . . . . . . . . 19

3.1. Structure of the Target Description File of the NoC generator . . . 39
3.2. Process Declaration within the Target Description File . . . . . . . 40
3.3. Structure of the process code provided to the network generator . . 41

6.1. Output file format (.map) of the syncmapper . . . . . . . . . . . . . 79
## Abbreviations & Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>Allocation</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Binding</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Execution Time</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Schedule</td>
</tr>
<tr>
<td>$\bar{s}$</td>
<td>Signal</td>
</tr>
<tr>
<td>BCET</td>
<td>Best-case Execution Time</td>
</tr>
<tr>
<td>$c$</td>
<td>Communication Time</td>
</tr>
<tr>
<td>$c_{OH}$</td>
<td>Communication Overhead</td>
</tr>
<tr>
<td>$E$</td>
<td>Execution Unit</td>
</tr>
<tr>
<td>$e$</td>
<td>Event</td>
</tr>
<tr>
<td>$G_A$</td>
<td>Architecture Graph</td>
</tr>
<tr>
<td>$G_P$</td>
<td>Problem Graph</td>
</tr>
<tr>
<td>$G_S$</td>
<td>Specification Graph</td>
</tr>
<tr>
<td>$L$</td>
<td>Communication Link</td>
</tr>
<tr>
<td>$P$</td>
<td>Process</td>
</tr>
<tr>
<td>$R$</td>
<td>Run Time of Execution Unit</td>
</tr>
<tr>
<td>$r$</td>
<td>Run Time</td>
</tr>
<tr>
<td>$r_{HP}$</td>
<td>Run Time of Hyper Process</td>
</tr>
<tr>
<td>$S$</td>
<td>Signal</td>
</tr>
<tr>
<td>$t_w$</td>
<td>Waiting Time</td>
</tr>
<tr>
<td>$t_{HB,min}$</td>
<td>Lower Bound of Heartbeat Period</td>
</tr>
<tr>
<td>$t_{HB}$</td>
<td>Heartbeat Period</td>
</tr>
<tr>
<td>$v$</td>
<td>Data Volume</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-case Execution Time</td>
</tr>
<tr>
<td>EU</td>
<td>Execution Unit</td>
</tr>
<tr>
<td>ForSyDe</td>
<td>Formal System Design</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
</tr>
<tr>
<td>PID</td>
<td>Process Identifier</td>
</tr>
<tr>
<td>PIO</td>
<td>Parallel Input/Output</td>
</tr>
<tr>
<td>RNI</td>
<td>Resource Network Interface</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
1 Introduction

1.1. Background

From the beginnings of computers, the clock speed of the processors was constantly increasing which allowed to design more and more powerful computer systems. In the mid 2000s, this trend was stopped as the higher clock rates started to cause problems with the power dissipation. The demand of more computational power was unbroken, and thus multi-core processors were introduced. As this trend will go on, we are heading towards what is called the “Sea-of-Cores era”, i.e. hundreds or thousands of cores will be integrated on a single chip. This will bring up new challenges. The main challenge of multi-core processors is the software development. The traditional methods, such as threads, are of limited suitability. New methods are needed that possibly hide the details of the platform to the system designer. The trend of multiprocessor chips moved also to the world of embedded systems where platforms with different processor types on one chip are usual. For embedded systems time-predictability is also an issue, especially for safety critical applications. The problem with multiprocessor systems is, that they are highly unpredictable and techniques to ensure the time-predictability must be developed.

ForSyDe [41] is a design methodology developed at KTH. A system designer using ForSyDe to model a system does not need to consider the implementation, and therefore it is a solution to the first mentioned challenge of multiprocessor platforms. ForSyDe is based on the concept of model of computation. If a system is modelled according to the properties of a model of computation, reasoning about some of the system’s properties become possible and the system can be verified by formal methods. For example, within the supported synchronous model of computation it is possible to reason about the timing behaviour of a system.

As more and more components were integrated on one chip, the communication bandwidth demand on a chip also increased. Traditional shared buses, as they were used could not scale with the development anymore and it was proposed to use packet-switched networks for the interconnections within the chips [14]. Out of the research conducted on these Networks-on-Chip (NoC) at KTH, Johnny Öberg
developed a NoC generator [49] for FPGAs. Together with the back-end tools of the FPGA vendors, the NoC System Generator is capable to generate automatically complex multiprocessor systems-on-chip. This generator can be used by researchers and system designer to fast prototype hardware platforms. The generated NoC has a special feature called heartbeat. It is similar to a hardware clock and is used to synchronize the different components within the network. Therefore, the heartbeat can ensure time-predictable execution.

1.2. Problem

Systems designed with ForSyDe conforming to the synchronous model of computation and the platforms generated by the NoC System Generator, ensure both time-predictability. The main vision behind this thesis is to combine these two projects to an automatic system design flow. This would allow to generate an entire time-predictable embedded system from a system model and some platform specifications.

A similar project carried out at several European universities is T-CREST [5], which has the goal to create a time-predictable multi-core architecture for embedded systems. The approach of the T-CREST is different from the one at KTH. Whereas the T-CREST project researched the different hardware components and software design methodologies separately for time-predictability, whereas the project around this thesis strongly based on the synchronous model of computation to ensure time-predictability.

The goal of this thesis is to develop a process which takes processes of a system designed by ForSyDe and maps them to a multiprocessor platform generated by the NoC System Generator. The need of an automatic mapping of processes to the NoC platform was already mentioned in [49]. The term mapping as used throughout consists of two parts. (1) The binding of the processes to the available processors, and (2) the specification of a running order of the processes on a processor — the schedule. The mapping should be optimized to reduce the heartbeat period in the NoC, as a consequence the throughput of the system is increased. Further goals are to implement the derived algorithm and apply it to an example to evaluate it.

1.3. Method

To achieve these goals a literature study on ForSyDe, the NoC System Generator, mapping theory, mapping algorithms was conducted. Then both projects, ForSyDe and the NoC generator were studied more deeply and their inputs, outputs and properties were analysed. Out of that the factors affecting the mapping were
collected and out of them models of the system were built. Based on these models a mapping process was developed. Finally, an implementation of the mapping process was programmed and checked for its feasibility by mapping synthetic examples to different networks-on-chip.

1.4. Outline

Following the introduction, Chapter 2 and 3 describe the existing parts on which this thesis builds up. Both chapters are conclusions of previous work and a reader familiar with the covered topics does not need to read all the details.

Chapter 2 describes first the basics of models of computation and more in detail the synchronous model of computation on which the thesis is based. In the second part the necessary parts of ForSyDe-SystemC relevant for the thesis are described.

The NoC System Generator is explained in detail in Chapter 3. The basics — such as topology, flow control, and routing of networks-on-chip — are briefly reviewed; and the theory is directly connected to the implementation in the NoC System Generator. Then the message structure is presented and a short overview of the hardware is given. The last chapter’s focus is on the user side of the platform. It is explained how the network is configured for generation and how the network is accessed by software. This chapter is quite extensive, as is also documents the actual state of the NoC System Generator.

Chapter 4 describes our suggested system design process, from modelling a system by ForSyDe-SystemC down to the generation of the final system with the NoC System Generator. Firstly, an overview of the whole process is given and then the different steps are described more detailed. The mapping process, developed by this thesis is positioned by this chapter.

The theoretical contribution of this thesis is presented in Chapter 5. It starts with modelling of the application as it is comes from ForSyDe. Then a model for the target architecture is described. The third part describes the mapping process itself, which is a three step process of assigning processes to logical processors, placing the logical processors in the network and schedule the processes on the different processors. Finally, the possible execution speed of the system is evaluated.

A first implementation of the mapping process is described in Chapter 6. First there is a short description of the usage is given. The class structure of the program written in C++ and details about the implemented algorithms are also part of this chapter.
1. Introduction

In Chapter 7 some results of the mapper are presented. The mapped systems are synthetic examples, which should reflect possible real systems. These examples are first described and then the results of the mapping are shown.

Chapter 8 concludes the thesis with the presentation of the achieved results. The thesis ends with a section with ideas for future work and proposed changes and possible improvements of ForSyDe and the NoC System Generator.
ForSyDe (Formal System Design) is a design methodology for system modelling of embedded systems and systems-on-chip on a high abstraction level. It was developed by Ingo Sander et al. [41], [42] at the Royal Institute of Technology, Stockholm. The synchronous model of computation, which is used throughout the thesis, is one among others supported by ForSyDe. This chapter starts with the explanation of the synchronous model of computation, this mainly based on material in [8] and [9]. Then, an overview of the modelling process and its underlying model of ForSyDe is given. The focus is put to principles and parts relevant for the thesis. It does not give a complete understanding that allows to model systems by ForSyDe. More detailed information, status of the research, tutorials, and the source code could be found on the ForSyDe web page [3].

2.1. Models of Computation

Today’s software could be rather complex to develop and the programs are not longer manageable by humans on the lowest level (e.g. assembly code), especially when it comes to heterogeneous, concurrent systems. With the help of a model of computation (MoC) lower levels can be abstracted, and an interface is given to the programmer on a higher level of abstraction. This leads to code that is easier to verify and programs that run more stable.

In [26] Lee and Seshia define three sets of rules which define a MoC for systems of concurrent components: the first set defines how a component is formed, the second specifies the concurrency mechanisms, and the third defines the communication mechanics. By creating a well-defined model using these rules, mathematical methods can be applied to analyse and verify the system. A MoC hides also implementation details from the designer. These and the exact program structure will be added by a compiler or a machine. An algorithm described in a MoC is therefore platform
independent and a MoC can also be seen as theoretical machine, on which an algorithm is executed.

Even tough, they are closely related, it is important to distinguish between a programming language and a MoC. According to Skillicron and Talia [44], every programming language could be seen as a MoC, since it provides a simplified view of the underlying level. But several programming languages could base upon the same MoC, and one language could even combine different MoCs. Therefore, if a new MoC is introduced new programming paradigms or languages arise, as stated in Fernández’s book [17]. The work of this thesis is based on the synchronous model of computation which is the foundation of the synchronous languages described in the next section. Whereas the ForSyDe methodology — described in Section 2.5 — combines different MoCs within one framework.

### 2.2. Synchronous Model of Computation

Lee and Sangiovanni-Vincentelli introduce in [25] a framework to describe different models of computation. The framework is based upon concurrent processes which are connected by signals. The signals consists of events \( e \) which are composed of a value \( v \) and a tag \( t \). \( V \) and \( T \) denotes the sets of all values and all tags.

\[ e = (t, v) \]

The tag is used to add properties to the event, which is used to model the MoC. Properties could be for example time, precedence relationships, or synchronization points. Basically the tags give some notion of order to the events.

A signal \( s \) is then a sequence of events

\[ s = \langle e_1, e_2, e_3, ... \rangle \]

The synchronous MoC needs the possibility to represent an event that does not have any value at a certain time instance. This \textit{absent} value is represented by the symbol \( \bot \) and is also called “bottom”. The absent value has to be seen like a usual value \( \bot \in V \). An example of a signal with absent values for events with tag 5 and 6 looks like:

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\( t \) & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\( v \) & 4 & 3 & 2 & 1 & \( \bot \) & \( \bot \) & 1 \\
\hline
\end{tabular}

Two signals are synchronous, if there exist for each event of one signal, exactly one event in the other signal with the same tag. If all the signals in a system are synchronous with each other, then the whole system is synchronous.
necessary that the event tags of a signal are *totally ordered*, which means that each tag is followed by one other and none of them is the same as another. Figure 2.1 shows a valid and two invalid signals within the synchronous MoC.

\[
\begin{array}{cccccccc}
\text{t} & 0 & 2 & 3 & 4 & 7 & 8 & 9 \\
\text{v} & 1 & 1 & 2 & 2 & 3 & 3 & 4 \\
\end{array}
\begin{array}{cccccccc}
\text{t} & 1 & 2 & 3 & 4 & 4 & 4 & 5 \\
\text{v} & 1 & 1 & 2 & 2 & 3 & 3 & 4 \\
\end{array}
\begin{array}{cccccccc}
\text{t} & 1 & 2 & 3 & 5 & 4 & 6 & 7 \\
\text{v} & 1 & 1 & 2 & 2 & 3 & 3 & 4 \\
\end{array}
\]

\begin{array}{c}
a) \text{valid} \\
b) \text{invalid} \\
c) \text{invalid} \\
\end{array}

**Figure 2.1:** Examples of signals: a) is a valid signal in a synchronous MoC, although events with tags ‘1’, ‘5’ and ‘6’ are missing, the events are totally ordered. b) is invalid because multiple events with tag ‘4’ exist. In c) the event with tag ‘5’ is preceding the event with tag ‘6’.

In their paper [8], Benveniste and Berry describe their synchronous hypothesis the following way:

“The ideal system produces the outputs synchronously with its inputs, the reaction of the outputs take no observable time.”

At the same time an ideal system can be split into other ideal systems with the same behaviour. Consequently, all the internal signals in the system change instantaneously with the input. In other words, neither the communications nor the computations take time. Indeed, this is not possible in real systems. We will present ways to implement this in Sections 2.4 and 3.7.

It is important not to mix up real time and tags. Tags just define an order of the events, but they do not indicate time instance they occur. For example, an event with tag ‘3’ must not necessarily happen the same time span after the event with tag ‘2’, as the time span is between the event with tag ‘1’ and the event with tag ‘2’.

One issue within this mathematical model are loops. Instantaneous loops cause non-deterministic behaviour which is unwanted and needs some special treatment. In many cases; this is solved by the constraint, that each loop must contain at least one delay element.

### 2.3. Synchronous Hardware

As hardware is inherently parallel, and the production of integrated circuits is extremely expensive; a method which allows reliable and time-predictable designs is
necessary. Therefore, synchronous circuits were used since the beginnings of digital circuits.

The processes of the synchronous model of computation correspond to combinatorial circuits in hardware. These combinatorial circuits are synchronized by flip-flops which are basically delay elements, which means that the calculated value of the first combinatorial circuit is available for the following combinatorial circuit after the next occurrence of the clock.

The maximum speed is given by the propagation the longest propagation between two flip-flops in the system. This time is calculated by CAD tools and gives the maximum clock rate with which system can run. The clock rate is fixed while the system is running, not considering certain power-saving techniques.

2.4. Synchronous Languages

Benveniste and Berry in [8], and Halbwachs et al. in [20] presented in 1991 their results on the research on synchronous languages. Later, Beneviste et al. summarized in [9] the research conducted in this area during the 1990’s. Especially, reactive systems — systems interacting with the physical environment — face some design challenges whereof three should be mentioned here. These systems are usually distributed and they can often be seen as blocks acting in parallel. This gives the first challenge: parallelism. The second challenge is, that reactive systems often must keep strict timing constraints under any circumstance. But, it is difficult to reason about time in traditional programming languages. The third challenge is dependability. Reactive systems are often used for safety-critical applications where verification of the correct behaviour of the system is essential, but cumbersome with traditional software design methods.

Traditionally, parallelism was realized by real-time operating systems or concurrent programming languages like ADA. The mechanisms provided by them are asynchronous and non-deterministic. In addition, real-time operating systems causes a significant execution time overhead. Hence, only the first mentioned challenge is mastered. The initial idea of the synchronous languages was to transfer the advantages of the hardware world to the software world, and thereby allowing reliable validation and time-predictable execution of software.

The synchronous languages are based on the synchronous MoC as described in Section 2.2. As this assumes to hold the synchronous assumption; all computations and communications should happen instantaneously, and the outputs are assigned synchronously with the inputs. This is unrealistic; since real systems are typically asynchronous systems, and execution of a process always takes some time. But it is possible to find asynchronous execution schemes, so that the systems still behaves
2.5. ForSyDe: A Design Methodology

in a synchronous way. Two different approaches were presented in [9] and are here shown in Figure 2.2.

<table>
<thead>
<tr>
<th>Initialize memory</th>
<th>Initialize memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each input event do</td>
<td>wait clock tick</td>
</tr>
<tr>
<td>Compute outputs</td>
<td>Read Inputs</td>
</tr>
<tr>
<td>Update memory</td>
<td>Compute Outputs</td>
</tr>
<tr>
<td>end</td>
<td>Update Memory</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
</tbody>
</table>

**Figure 2.2:** Two different execution schemes used by synchronous languages. The event-driven scheme on the left hand side has high processor occupation but lacks time-predictability. The scheme on the right hand side implements a software clock and ensures therefore time-predictability. Source: [9].

Both approaches create a kind of a software clock by waiting for an event to occur. The left of the two execution schemes shows an event-driven approach. The system waits for an event occurring at the input and executes then its code. With this scheme the events can be stored in a queue and the processor can be kept busy all the time. As the the exact waiting time of an event is unknown, the timing behaviour of such a system may be unpredictable. The execution scheme on the right hand side is closer to the way hardware works. It uses a clock signal that triggers the execution. The inputs are read right after the execution phase has started. Changes occurring at the inputs during the execution phase are therefore ignored. To ensure the correct behaviour the clock tick must not occur before the execution has finished. This makes on the one hand the timing predictable, but can lead to unwanted waiting time if the clock period is chosen unnecessarily long.

Several synchronous programming languages are commercially distributed, for example Esterel, Lustre, and SIGNAL as they are mentioned in [9]. Two areas were the synchronous languages were successfully used are flight control systems and nuclear power plants. Both areas require hard real-time systems where missing a deadline would be disastrous. The certification process, which is a required from the authorities for such systems, is also simplified by using languages based on the synchronous MoC.

2.5. ForSyDe: A Design Methodology

ForSyDe, as described in [28], [42] and [41], aims to allow the designer to describe a system-on-a-chip on a high abstraction level without the need to consider the implementation details. The modelling process becomes independent of the target
platform, and during the modelling it is not irrelevant if the final system will be implemented by hardware, software, or a combination of both. The first implementation of ForSyDe was realized in the functional language Haskell. Haskell has some language constructs that support well the concepts of ForSyDe. Recently, ForSyDe has been also ported to SystemC, this so called ForSyDe-SystemC will be used throughout this thesis.

In ForSyDe, the starting point of a design is called specification model. It is purely functional and deterministic, and can make use of ideal data types such as infinite lists. The modelling of the system must follow the guidelines given by the ForSyDe design methodology. Therewith, it is ensured that the model follows the underlying mathematical model and the further steps of the ForSyDe methodology can be applied. By applying formal methods to the specification model or simulating it, the functionality can be verified.

The other model described by the ForSyDe methodology is the implementation model. It contains all necessary details, in order that back-end tools can map the model to a real system. For example, infinite buffers must be exchanged by fixed-size buffer in the implementation model as they cannot be realized in a real system.

The process to get from the specification model to the implementation model is called refinement. One of the principles of the ForSyDe methodology is that the refinement takes place in the functional domain and not as usual in the implementation domain (synthesis). The Figure 2.3 shows the border between the functional and implementation domain. The original specification model is refined until an optimal implementation model for the target platform is found. As the refinement is performed on such a high abstraction level; different design alternatives could easily be evaluated and verified by the same formal methods as the specification model. The refinement is carried out by applying transformations with well-defined rules repeatedly to the system model. These transformations can be grouped into two classes, which were described in [42]:

**Semantic Preserving Transformations** do not change the meaning of the model and are mainly used to optimize the model for synthesis. An example is to move a delay on a chain of processes without loops. By using only semantic preserving transformations for refinement, the correct behaviour is still ensured.

**Design Decisions** change the meaning of the model. A typical example is to assign a fixed size to a buffer that was infinite in the specification model. While a design decision changes the semantics of the model, the transformed model may, but not necessarily need to, still behave in the same as the original model.

Figure 2.3 shows the complete ForSyDe design process. The graph is based on the process described in [28], but the new functionality of creating XML system descrip-
tion files is added. The design process starts by modelling the system specifications which results in the specification model. This can then be verified either by formal verification methods or simulation. Then, the model is refined step-by-step until all necessary implementation details are added. The result is the implementation model that can be verified by the same methods as the specification model. From the implementation model the system description files are created. In the case of ForSyDe-SystemC, these are a set of XML and C files, which will be described in detail in Section 2.8. Back-end tools create out of these files, together with templates from the design library, source code and hardware description files which are used to generate the system on a target platform.

2.6. The ForSyDe System Model

A ForSyDe model is a hierarchical network which basically consists of two different elements: concurrent processes and signals. The processes perform the computations and communicate with each other by the signals. The signals are modelled by the tagged-signal model, as described in Section 2.2. The processes $P$ map an input signal $\vec{i}$ to an output signal $\vec{o}$.

$$P(\vec{i}) = \vec{o}$$
In ForSyDe processes are created by process constructors. A process constructor is a higher-order function\(^1\) that takes one or more combinatorial functions as an argument and returns a process. The combinatorial functions describe the desired functionality of the process. The process constructor realizes the communication and synchronization with the other processes according to the specified MoC. Thereby, the process constructors separate computation from communication. For a possible implementation of the ForSyDe model every process constructor has assigned a possible software and hardware implementation.

A third element — the domain interface — can be part of a ForSyDe model. The domain interfaces are used in two cases: (1) Two areas of different MoCs are connected. (2) Two areas of the same MoC but with different properties are connected, e.g. two areas with the synchronous MoC with different clock rates. The latter is usually added during the design refinement and is part of the implementation model.

Initially, ForSyDe was developed for the synchronous MoC. Over the years other MoCs — e.g. continuous time, discrete event, or SDF\(^2\) — were added to the ForSyDe methodology. As the communication between the processes in ForSyDe bases upon the tagged-signal model as presented in Section 2.2, it was possible to include them to the existing system model. The hardware platform used in this thesis includes support for the synchronous MoC. Thus, for the rest of the thesis we discuss only the synchronous MoC of ForSyDe.

### 2.7. ForSyDe Process Constructors of the Synchronous MoC

This section describes the most common process constructors available for the synchronous MoC in ForSyDe-SystemC. For all the process constructors there is firstly an explanation of the general process constructor, and secondly the concrete available process constructors are presented. The “SY” in the end of the constructors’ names denote that these constructors belong to the synchronous MoC.

---

1. A higher-order function is a function that takes either another function as an argument, returns a function as result or both of them.
2. Synchronous Data Flow
2.7. ForSyDe Process Constructors of the Synchronous MoC

2.7.1. Process Constructor: Combinational

The $\text{combSY}$ process constructor is used to create combinational processes, i.e. a process that does not contain any state. The input to the constructor is a function.

The following versions are implemented in the actual version of ForSyDe-SystemC:

- $\text{comb}$: one input and one output
- $\text{comb2}$: two inputs and one output
- $\text{comb3}$: three inputs and one output
- $\text{comb4}$: four inputs and one output

2.7.2. Process Constructor: Delay

The $\text{delaySY}$ process creates a process that implements a delay element with one input and one output without any additional functionality. Two different process constructors exist: $\text{delay}$ and $\text{delayN}$. The former realizes a delay of one cycle, whereas the latter creates a delay of $n$ cycles. Both process constructors take an initial value as an argument which is output in the beginning of the execution.
2. Synchronous System Design with ForSyDe

2.7.3. Process Constructor: Zip / Unzip

The zipSY process constructor creates a process that takes two input signals and merge them to a tuple. The tuple is then sent to the output port. The tuples can be split again by the unzipSY process constructor. These process constructors were introduced, because a process in ForSyDe is seen as a high-order function, and has can only have one output per definition.

For both, two different implementations exist: zip and unzip which zip/unzip two signals, and zipN and unzipNSY which work with a variable number of signals.

2.7.4. Process Constructor: Mealy, Moore

Two separate process constructors for FSMs (finite state machine) exist. The first one describes a Moore machine mooreSY and its output signal only depend on the actual
state. The second one describes a Mealy machine \textit{mealySY}, whose output also depend on the actual input signal. Both process constructors can be built of \textit{combsY} and \textit{delaySY} blocks which is shown in the lower part of Figure 2.7.

Both process constructors take two functions and an initial state \(s_0\) as arguments. The first function \(f\) maps the input to the next state and the second one \(g\) generates the output. In the implementation the process constructors are called \textit{moore} and \textit{mealy}.

### 2.7.5. Process Constructor: Source

![sourceSY(f) \rightarrow \delta]

\textbf{Figure 2.8.:} \textit{Symbol of the general source process constructor.}

The \textit{sourceSY} process constructor creates a process that generates a signal and has therefore only one output, but no inputs. The main usage of this process constructor are test benches. But, it is also used to represent inputs from external hardware to a real system. Whereby, the code used for the implementation usually differs from the version used for modelling.

The actual implementation of ForSyDe-SystemC includes three source processes:

- \textit{source} takes the current state of the process constructors and applies a function to calculate the next value. The initial state and the function must be passed as an parameter.
- \textit{vsource} takes a vector as a parameter and outputs the values of the vector one by one.
- \textit{constant} outputs a constant value which is passed as a parameter.

### 2.7.6. Process Constructor: Sink

![\tilde{i} \rightarrow sinkSY(f)]

\textbf{Figure 2.9.:} \textit{Symbol of the general sink process constructor.}

The actual implementation of ForSyDe-SystemC includes three source processes:
The function of the \texttt{sinkSY} process constructor is to output the result of a system. They have only an input and like the \texttt{sourceSY} process constructors their main purpose are test benches, but also the modelling of the outputs in the implementation model.

The following two \texttt{sinkSY} process constructors are implemented:

- \texttt{sink} applies a function to each input sample
- \texttt{printSigs} prints the input value to the standard output

### 2.8. Model description files (XML & C++)

The ForSyDe-SystemC implementation has the possibility of creating XML files which describe the structure of the model\footnote{The XML description created by the Haskell implementation uses a different XML format and are not compatible.}. Together with the C++ files describing the functionality of the combinatorial processes; a format for describing the complete ForSyDe models exist. These files are intended to be used as input or output to ForSyDe-SystemC. The usage of the widely-used markup language XML for the model description leads to a format which is easily readable by humans and processable by software. Therefore, external tools could be used on the XML files to perform different tasks, such as visualization, refinement, verification, or synthesis of the model.

The code snippets in this sections are based on the multiply-accumulate example used in the \textit{ForSyDe-SystemC Tutorial for the Synchronous MoC} on the ForSyDe web page \cite{tutorial}. This example uses the synchronous MoC, but all the explanations in this section apply also to the other MoCs.

#### 2.8.1. Structural Description (XML)

The process structure of a ForSyDe model, i.e. the way the processes are connected with each other, is described by a set of XML files. To make the design more structured and to allow the reuse of components, the model description could be hierarchical. A separate XML file is then used to describe each level of hierarchy.
Figure 2.10: Example ForSyDe system: multiply-accumulator. Source: ForSyDe web page [3].

Listing 2.1: ForSyDe XML description of a multiplier-accumulator

```xml
<?xml version="1.0" ?>
<!-- Automatically generated by ForSyDe -->
<!DOCTYPE process_network SYSTEM "forsyde.dtd" >
<process_network name="mulacc">
  <port name="port_0" type="int" direction="in" bound_process="mul1" bound_port="iport1"/>
  <port name="port_1" type="int" direction="in" bound_process="mul1" bound_port="iport2"/>
  <port name="port_2" type="int" direction="out" bound_process="add1" bound_port="oport1"/>

  <signal name="fifo_0" moc="sy" type="int" source="mul1" source_port="oport_1" target="add1" target_port="iport_1"/>
  <signal name="fifo_1" moc="sy" type="int" source="accum" source_port="oport_1" target="add1" target_port="iport_2"/>
  <signal name="fifo_2" moc="sy" type="int" source="add1" source_port="oport_1" target="accum" target_port="iport_1"/>

  <leaf_process name="mul1">
    ...
  </leaf_process>

  <leaf_process name="add1">
    ...
  </leaf_process>

  <leaf_process name="accum">
    ...
  </leaf_process>
</process_network>
```
Listing 2.1 shows the XML description of the `mulacc` model shown in Figure 2.10. The file starts with a short header defining the used XML format. The model is described within the `<process_network>` element. The name of the model is given by the `name` attribute. This name is also used to reference to this description, when it is used as a component on a higher hierarchical level. The process network itself consists of declarations of ports, signals, and processes. Whereas the port declaration is omitted if the XML file describes the top level of the system.

The points where the signals are connected with the processes of a model are called ports. The ports provided by a process network to the higher level are declared by the `<port>` elements. Each port is described by a name (`name` attribute), its data type (`type` attribute), and a direction (`direction` attribute) that could be either "in" or "out". The data type is a C++ type as it is used by the process code.

The processes of the system are connected by signals. The signals are declared separately from the processes by `<signal>` elements. A name (`name` attribute), the data type (`type` attribute), and the associated MoC (`moc` attribute) are attached to each signal. To remind the reader, ForSyDe allows to use only one MoC within one design domain. Therefore, it is not a restriction that the MoC of a signal is fixed by the declaration. The other attributes of the `<signal>` element define the source and target where the signal is connected to. The points to which a signal is connected are defined by the name of the processes (`source` and `target` attributes) and their ports (`source_port` and `target_port` attributes).

Two type of processes exist in the structural description of a model: composite processes which are processes with an underlying hierarchy, and leaf processes declaring processes that perform computation.

The composite processes are defined by the `<composite_process>` element which takes two attributes: `name` and `component_name`. A name (`name` attribute) which identifies it within the actual file. The `component_name` attribute defines which process description is used for this process. The structures of the different composite processes are described in separate XML files. The Listing 2.2 shows how the above described process network (Listing 2.1) can be used on the higher hierarchical level.

```
Listing 2.2: Example of a composite process in a ForSyDe XML

<composite_process name="mulacc1" component_name="mulacc">
  <port name="iport_1" type="int" direction="in"/>
  <port name="iport_2" type="int" direction="in"/>
  <port name="oport_1" type="int" direction="out"/>
</composite_process>
```

Processes that contain the computation are called leaf processes and are declared by the `<leaf_process>` element. This element takes the attribute `name` which gives the process a name for identification – analogous to the composite processes. The
listing 2.3 shows the complete definition of the add1 process used in Listing 2.1. After the port declaration the leaf processes contains a process_constructor element that specifies the process constructor used to generate the process. The process constructor is defined by the MoC (moc attribute) and the process constructor type (name attribute) that has to be one of the available process constructors of the model of computation. A process_constructor element contains argument elements configuring the process constructor’s properties. An argument is indicated by the name attribute and its value is given by the value attribute. The only argument of the comb2 process, shown in Listing 2.3, is the function to be executed.

Listing 2.3: Example of a leaf process in a ForSyDe XML

```xml
<leaf process name="add1">
    <port name="iport_1" type="int" direction="in"/>
    <port name="iport_2" type="int" direction="in"/>
    <port name="oport_1" type="int" direction="out"/>
    <process_constructor name="comb2" moc="sy">
       <argument name="_func" value="add_func"/>
    </process_constructor>
</leaf_process>
```

2.8.2. Process Code (C++)

For each process constructor that takes a function as an argument, there is a SystemC file with the ending .hpp that describes its functionality. Listing 2.4 shows the code for an adder, as it can be used in the multiply-accumulator from the previous section. The parameters of the function and the first lines implement the connection with the ports of the process. The actual code of the process is surrounded by a #pragma pre-processor directive. If the model should be implemented on a target platform, only C++ language constructs supported by the compiler of that platform must be used in the process description.

Listing 2.4: Process code of an adder in ForSyDe-SystemC

```c++
void add_func(abst_ext<int>& out1, const abst_ext<int>& a, const abst_ext<int>& b) {
    int inp1 = from_abst_ext(a, 0);
    int inp2 = from_abst_ext(b, 0);

    #pragma ForSyDe begin add_func
    out1 = inp1 + inp2;
    #pragma ForSyDe end
}
```

The ports, as they are defined for a leaf process, are mapped in the order of appearance in the function definition. In our example, this means that iport1 of the adder is
mapped to the variable a and import2 to the variable b. They are extended with an absent value in the first two lines of the function body. The output of the process is mapped to the out1 variable.

2.9. Conclusion

The here presented synchronous model of computation allows the design of time-predictable systems. This is extensively used in digital hardware design where it also showed its capabilities. As in many applications the computational power is more important than predictability, the synchronous approach for software is only used in a niche, mainly for systems with high requirements for time-predictability. With the arise of multi-core platforms, it became harder to ensure time-predictability and thus many designer of embedded systems cannot yet make use of the full computational power of multiple cores. The application of the synchronous model of computation could help to overcome this problem.

The design methodology ForSyDe, described in this chapter, and the NoC System Generator, described in the next chapter, could create together a system design process that is based completely on the synchronous model of computation. As we will see, the synchronous model of computation brings also some advantages for the mapping of processes. Furthermore, as the software works in a similar way as digital hardware it will probably be possible to apply techniques used for digital hardware design also for the software design.
The Network-on-Chip System Generator

Based on the research on networks-on-chip (NoC) carried out at the Royal Institute of Technology, Stockholm; Johnny Öberg et al. developed the *NoC System Generator*. The goal of this project is to create a tool that automatically creates the interconnections, the network resources and the processes running on the nodes on a chosen target platform. Additionally, the platform implements a so called heartbeat that allows time-predictable execution of programs. The work has been documented in different papers [48], [49] and the thesis [32]. These do not reflect the current state of the project, therefore this chapter is based additionally on discussions with Johnny Öberg and the current code of the generator. The theoretical part are based mainly on Dally’s book “Principles and Practices of Interconnection Networks” [15]. This chapter gives a complete overview and also document the actual state of the NoC System Generator project, nevertheless not all the technical details are necessary to understand the rest of the thesis.

3.1. Networks-on-Chip

Around the year 2000, the increasing demand of bandwidth for on-chip communication brought the traditional shared-bus systems to their limits. To overcome their limitations, Dally and Towles [14] proposed in 2001 to use interconnection networks to connect the different components on a chip. The most common type of networks are packet-switched networks where the messages are split into packets and these are routed within the NoC.

A NoC consist of network nodes which are connected with each other by network links. The nodes are usually organized in regular structures as they are described in the next section. Every node consists of a network switch and a resource. The resources are the functional components of the network, e.g. processors, memories, IO devices. The switches realize the communication between the nodes. They implement the flow control (Section 3.3) the routing of the packets (Section 3.4).
3. The Network-on-Chip System Generator

3.2. Topology and Addressing

The topology of a network describes the way how the network nodes are connected with each other. Different, usually geometrically regular, patterns are used. The typical ones for NoCs are ring, 2D-mesh, 3D-mesh and torus which are shown in Figure 3.1. The topologies differ from each other by the available bandwidth, the latency of communication and the area used on the chip. In the current version of the network generator 2D- and 3D-mesh networks can be created. Other topologies like rings, tori, and even arbitrary networks will be implemented in the future. The largest network that can be realized on an Altera DE3-150 prototyping board is a 3x3x3 3D-mesh, which results in a network with 27 processing nodes consisting of single Altera Nios II-e cores with 8 kBytes of on-chip memory.

![Figure 3.1: Different network topologies used in networks-on-chip. From left to right: ring, 2D-mesh, torus.](image)

Similar to houses in a city every network node gets a unique address. In the case of the NoC System Generator these addresses are continuous integers. As shown in Figure 3.2, the addressing in a 3D-mesh network starts with node ’1’ which is in the left-front corner of the lowest layer, and increases first along the X-dimension (columns). When it reaches the last node in X-dimension, it continues with the next row in Y-dimension (rows). After all nodes of one layer are addressed, this is continued in Z-dimension (layers). The layers have an additional meaning: if a network is split up to different chips or boards, the network is cut at a layer border. This naming convention is summarized in Table 3.1. For a 2D-mesh network as used mostly throughout this thesis, the addressing works basically the same, just the layers are omitted. This is the internal naming convention of the hardware and it is hidden to the system developer. But, the mapper implementation described in Chapter 6 uses the same addressing scheme.

An important parameter for the evaluation of the communication time between two nodes is the distance. The distance between two nodes of a NoC is the minimum number of hops, that are necessary to reach the destination node from the source.
3.3. Flow control

Flow control allocates the resources of the network, such as bandwidth, buffers, and control state, to the packets within the network. In this way, flow control resolves conflicts also between different packets that want to access the same resource of the network. The simplest type of flow control is bufferless flow control, which is also used by the NoC System Generator. With bufferless flow control the switches do not have a buffer for the incoming or outgoing packets. Therefore, the nodes does not occupy much area on the chip.

As the flits cannot be stored they must be handled in the same cycle as they arrive. Two possibilities exist: either to drop a package or to misroute it. The latter one
3. The Network-on-Chip System Generator

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Name</th>
<th>Direction</th>
<th>max nr. of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Column</td>
<td>West East</td>
<td>8</td>
</tr>
<tr>
<td>Y</td>
<td>Row</td>
<td>South North</td>
<td>8</td>
</tr>
<tr>
<td>Z</td>
<td>Layer</td>
<td>Down Up</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.1: Compilation of the naming convention for the dimension and their maximum sizes in mesh networks of the NoC System Generator platform.

is the used by our platform, and will be further described in in the section about Deflection Routing 3.4.3. If a packet was dropped, this must be signalled to the source node to trigger re-transmission. In the second case the packet will not follow the designated path which can increase the length of the packet’s path. Thus, both methods lead to an increasing need of channel bandwidth. The network traffic also becomes unpredictable as the network contention causing dropping or misrouting can not be predicted.

3.3.1. Deadlock & Livelock

*Deadlock* occurs in a network if an agent (e.g. packet) holds a resource another agent wants to acquire. As a deadlock blocks parts of a network completely, it has fatal consequences for the functionality of a network. Two possibilities exist to handle deadlocks: deadlock avoidance and deadlock recovery. Deadlock avoidance is realized by choosing algorithms that are deadlock-free. For deadlock recovery the network must be monitored for deadlocks, and if one is detected countermeasures has to be taken.

*Livelock* is the case, if a packet keeps moving in the network without reaching its destination. This can be a problem for networks in which non-minimal routing is allowed. Livelock can be avoided in two ways. Firstly, probabilistic avoidance is based on the assumption that a packet will eventually reach its destination, this is ensured if the probability to be moved towards the destination is larger than 0. The problem is that it still can take a long time for a packet to reach its destination. Secondly, deterministic avoidance adds a mechanism, which adds a state to the packets that moves the packet towards the destination node. For example, a hop counter could be included in the packet and packets with a higher number of hops are given priority to the channels towards their destination.
3.4. Routing

Routing describes the way of finding a path through a network from the source node to the destination node. There are two main goals of a routing algorithm. Firstly, it tries to balance the traffic within the network by using path diversity. Secondly, it tries to reduce the latency of the data by assigning a short path. If the algorithm always uses a path with the minimal distance between the source and the destination, it is called *minimal routing*.

3.4.1. Routing algorithm classes

Dally describes in [15] three different classes of routing algorithms:

**Oblivious routing** does not take the network state into account. The load of the network could be balanced by not assigning always the same path to a packet.

**Deterministic routing** is a subset of oblivious routing. All packets from a specific source node routed by a deterministic routing algorithm will always take the same path to the destination node. In general, deterministic routing is minimal. It is easy to implement, but as it does not take path diversity into account, it does not avoid local congestions.

**Adaptive routing** algorithms consider the actual or preceding status of the network. Usually, the nodes use only local knowledge about the network status for the routing decisions. This could be realized by sensing the output channels for contention. Contention of a node is propagated back in the network as the buffers of the nodes get filled; this is called backpressure. Another advantage of the adaptive routing class is fault tolerance.

3.4.2. Dimension-Order-Routing

Dimension-order-routing is a deterministic and minimal routing algorithm for cube networks (tori and meshes). A packet routed by dimension-order follows first one dimension until it reaches the coordinate of its destination. Then it follows along the next dimension and so on, until it reaches the destination node. The problem with dimension-order-routing is the bad traffic balancing due to the deterministic paths. Nevertheless, it is often used because of the small size of the routers. In networks with a torus topology, both directions of one dimension could have the same distance; then the router distributes the traffic equally to both directions to balance the load. Dimension-order-routing is also deadlock-free, which means that deadlock can not occur.

Figure 3.3 shows a 3x3 mesh network with 3 packets travelling in it. The first packet A is sent from node '8' to node '3'. It is routed first along the X-dimension.
via node '7' to node '6', then along Y-dimension to its destination node '3'. The packet $B$ is sent similar from node '1' to node '5'. Packet $C$ is already at the correct X-coordinate, hence it is only routed along the Y-dimension.

![Network diagram](image)

**Figure 3.3.** Three examples of packet paths in a network with dimension-order-routing (X before Y).

### 3.4.3. Deflection routing

Deflection routing is a routing technique for networks with bufferless flow control. It is also called hot-potato routing and was first described by Baran in [7]. Due to the absence of buffers a received packet has to be processed immediately by the router and has to be sent to an output channel in the next cycle. So, the packets move all the time in the network. Therefore, if two packets want to access the same output channel, one of them must be misrouted. As data buffering is difficult to realize for optical signals, deflection routing is widely used in optical networks. It is also used in Networks-on-Chip as the absence of buffers allows the design of small and fast routers. Deflection routing is defined by two policies: The routing policy which defines on which path a packet moves to the destination, and the deflection policy which defines in which case and how a packet is deflected.

The generated NoC platform uses deflection routing to keep the size of the router small, but it realizes the deflection on flits\(^1\), which are fractions of a packet and described in Section 3.5. The used routing policy is dimension-order-routing as described in Section 3.4.2. The order of the dimensions is: X first, then Y, and Z last. In the optimal case, when no congestion occurs in the network, a packet follows the minimal path. In the case of deflection, the router tries to keep the misrouted flit on a minimal path if possible. An example is shown in Figure 3.4. Only if all of a node’s outgoing links, which move the flit closer to the destination, are occupied, the flit is misrouted away from the destination. The flit is preferably misrouted

\(^1\)flow control digit
in X- and Y-dimension, because the Z-dimension is used to distinguish between stacked hardware boards, and the connections between them are usually slower. The decision of which flit gets a link assigned and which one is deflected bases on fixed priorities, depending on the direction from where a flit arrives at the node. The order is: North-South-East-West-Up-Down-Resource.

![Deflection routing example](image)

**Figure 3.4:** Deflection routing example: There are two flits in the network; flit A travelling from node 4 to 5 and flit B travelling from node 3 to 8. Both wants to access the link between 4 and 5 at the same time, thus flit B is deflected to 7 as this still brings it closer to its destination.

Deadlock cannot occur in a network with deflection routing, as all the packets must be sent in the next cycle. But livelock can occur and must be avoided. A hop counter is added to the header of the flits in the generated NoC to solve this problem. This counter is increased every time a flit passes a node. By including the hop counter into the deflection decision, older packets can be given priority. Unfortunately, the deflection decision does not consider the hop counter is not implemented yet. However as long as there is no violation of the heartbeat delay (Section 3.7), no new packets will be injected before the end of a heartbeat period and livelock cannot occur.

As different packets (flits) travel on different paths with different lengths, they may not arrive in the same order as they were sent. To reconstruct the right order at the destination, all flits in the generated NoC include an ID.

### 3.4.4. Discussion on Deflection Routing

Beside the already mentioned possibility of constructing small and fast routers, deflection routing has some other advantages. As it is an adaptive routing method, it can handle local network congestion by routing the traffic to unoccupied links. The adaptivity adds also some fault-tolerance, if some network link becomes broken for any reason, the traffic will be moved around it.
The adaptivity makes the path on which a flit travels unpredictable, thus makes the arrival time of a flit unpredictable. Brassil and Cruz [12] extend the work of Hajek [19], and present some bounds for the arrival time of packets in a network with deflection routing. To apply these bounds to a network with continuous admission of packets, the routing policies must ensure that the oldest packet at a certain distance to its destination is assigned to the shortest path. However, they showed that the calculated bound is approximately two times larger than the highest observed packet arrival time in the simulation.

Lu et al. present their results on their research on deflection routing in NoCs in [29]. They simulated NoCs with different topologies, routing algorithm, and deflection policies to analyse their properties. The outcome is, that if the injection rate is kept low enough (<40%), it can be ensured that all packets will be delivered. They also state that prioritizing packets reduces the average-case, worst-case behaviour and resolves livelock.

Another approach could be the detailed analysis of the NoC by applying network calculus. This was done by Qian et al. in [36], but their work was limited to deterministic routing.

The network created by the NoC System Generator, although it is foreseen, has no age or priority attached to the packets. Therefore, a packet can be deflected theoretically as long as new packets are injected at some nodes of the network. Nothing ensures that the older packet moves towards the destination instead of the newly injected. Even if a priority is given to the older packets and the bound presented by Brassil and Cruz [12] can be applied, this bound would be not very tight which lead to a unnecessarily long heartbeat period (Section 3.7). The heartbeat could also give the network some time to recover, i.e. no new flits are injected after the execution of the last process, then all the flits in the network have time to arrive at their destination until the end of the heartbeat period.

The injection rate of every network node cannot exceed 25%, which ensures according to Lu [29], that the packet will eventually arrive, but no bound for the arrival time can be calculated, which would be essential for the heartbeat evaluation.

We conclude that the deflection routing used by the NoC in the actual implementation lacks of time-predictability. There are two ways that could be taken to ensure a time-predictable synchronous system. Firstly, congestion on links can be avoided and the paths in the network are deterministic as dimension-order routing is used. This is the approach applied in this thesis. The second approach is to make the upper bound of the arrival time calculable and as tight as possible.
3.5. Message Decomposition

An application programmer usually wants to neglect the details of a network transfer. He or she is only interested of getting an amount of data from one node to another. This one piece of data is called a message. The length of a message can be arbitrary long. To facilitate an efficient network design, a message is split into packets of limited size. Still, the size of a packet can be variable or fixed. Each packet contains information about routing and sequencing.

A packet may be split into several flits (flow control digit). This is the smallest unit which contains information about flow control. In general flits do not contain information about routing and all flits of the same packet have to follow the same path in the network. In the case of the generated NoC, this is not correct as the deflection routing requires the possibility to misroute every flit independently. Therefore, every flit must contain information about its destination and its position within the packet. The typical size of a flit is 16-512 bits.

On the lowest abstraction level of the network are the phits (physical digit). A phit represents the size of a physical channel in the network. Depending on the size of a flit, it is sent by one or several phits. The usual size of phits is 1-64 bits. In the generated NoC, the width of the physical channels is generated to fit the width of a flit. Therewith a phit is equal to a flit.

The following section explains in details how the decomposition into packets and flits is realized in the NoC generated by the NoC System Generator.

3.5.1. Packet Structure

A packet in the network is split into flits with the size of up to 64 bits, depending on the configuration of the NoC. A packet can consist of up to 128 flits. It starts with a Setup Flit followed by a Global Clock Flit. Following them, a packet can have a variable number of data flits. This structure is shown in Figure 3.5.

The first flit of each packet is a setup flit that contains the necessary information for the receiver to process the packet properly. The payload of this flit consists of the message length (including the Global Clock Flit) and the upper 9 bits of the global clock. It is followed by a Global Clock Flit that contains the lower 32 bits of the global clock. The global clock can be used to re-order the messages and for debugging purposes. The clock is a 41 bit wide field, that contains the number of passed heartbeat cycles passed since the system start.

Following the two header flits, up 126 data flits can be sent by a packet. Every flit carries a payload of 32 bits which means that up to 4032 bits can be sent by a single packet. As the current implementation just allows to send one packet between two
processes within one heartbeat cycle, this is also the maximum amount of data that can be transferred between two processes.

3.5.2. Flit Structure

Unlike the theory presented in Section 3.5 the flits contain routing and sequencing information. This allows that each flit can be deflected separately at each node and does not have to follow its preceding flits of the same packet.

The size of a flit is maximal 64 bits, whereof 32 bits are used as payload. The structure and size of the header fields can vary depending on the configuration of the network. This is generated automatically by the NoC System Generator and the user usually does not have to deal with it. The detailed structure of a flit is shown in Figure 3.6 and explained in the following list.

<table>
<thead>
<tr>
<th>Type</th>
<th>Flit ID</th>
<th>Source PID</th>
<th>Hop Count</th>
<th>Address</th>
<th>Payload</th>
</tr>
</thead>
</table>

**Figure 3.6:** The flit format in the NoC generated by the NoC System Generator. The indicated positions of flits are according to the maximum size of the fields.

**Type:** Indicates the type of the flit. The first bit indicates whether the flit carries valid data or not. If it is ‘0’ the flit is empty, if it is ‘1’ it contains data. The
second bit is '1' for a Setup Flit and '0' for a Data Flit. Size: 2 bits.

**Flit_ID:** The flit ID numbers the flits of a packet and allows re-ordering of the flits at the destination. Size: 7 bits.

**Source_PID:** The source PID indicates the process which has sent the flit. Maximum size: 7 bits.

**Hop_Count:** The hop counter is increased at every node the flit passes, therewith the age of the flit can be determined. This could be used to give older packets a higher priority but is not implemented yet. Size: 8 bits.

**Address:** The address field contains the destination address. Depending on the network topology, it is split into different sub-fields, e.g. for a 2D-mesh network the address field is split into a North-South and an East-West sub-field. Maximum size: 8 bits.

**Payload:** Every flit carries 32 bits of payload.

### 3.6. Network Node

Each node consists of a switch and a resource, which are connected by a *Resource Network Interface (RNI)*. Figure 3.7 shows a schematic overview of a node in the center of a 2D-mesh network. The switch is connected with the switches of the neighbouring nodes by buses with the width of a flit. The routing of the flits is realized in the switch.

The Resource Network Interface (RNI) connects the resource with the network. It prepares the packets for sending, and receives the packets from the network. The resource is connected to the RNI by a local bus. To the switch the RNI is connected in exactly the same manner as the switches of the neighbouring nodes.

#### 3.6.1. The Network Switch

The switch is the central unit of the network itself. The switch transfers the flits arriving at one input to one of the outputs by implementing the routing and flow control policies. It also puts the flits injected by the resource to the network. If the switch is part to the flit’s destination node, it routes the flit to the resource. In the concrete case of the generated NoC, the deflection routing is realized as a 4 cycle finite-state machine (FSM). Therefore the flits are only transferred every 4th clock cycle in the network.
3.6.2. The Resource Network Interface (RNI)

As already described, the RNI connects the resource with the network. On the network side it is connected by a port as any other network node. By the resource the RNI is accessed as a memory-mapped device on the local bus. The RNI consists of send channels, receive channels, a set of control registers and some control logic. Figure 3.8 shows the schematic structure of the RNI.

Every connection between two processes — regardless of whether they run on the same node or different nodes — occupies one send channel belonging sending process and one receive channel for the receiving process. These channels are assigned fix to the sending and receiving processes during the platform generation. The destination node of a packet is addressed by the process identifier (PID), and therefore the system designer does not need to know on which node a process is running. This detail is abstracted away by the NoC System Generator. The drawbacks of this addressing method are: The hardware must be generated again if the process structure of the system has changed, and due to the restricted size of the PID field in the flit the maximum number of processes is 128.
3.6. Network Node

The channels are basically buffers which can hold one packet. They are read and written by memory accesses from the resource. The size of them is configured in the hardware description file (Section 3.9.1) before the system generation. Actually, a node can have maximal 32 send and 32 receive channels.

The control logic takes care of the preparation of the packets from the send channels for the network, and decode the receiving packets and store them in the correct receive channel. The sending process is realized as a 16 cycle FSM, which means that one flit is sent every 16\textsuperscript{th} cycle. Given that the network switch only transfers a flit every 4 cycles it would be possible to send by a 4 cycle FSM. To reduce the possible congestion in the network the injection rate of the RNI is reduced to 25%. The control logic is steered and monitored by the control register set, e.g. there exist registers to invoke the transmission or check if a packet was received. These registers are accessible for the system designer by the device driver described in Section 3.8.3.

3.6.3. The Network Resource

All the functionality of the system is realized in the network resources of the NoC platform. The goal is that these network resources can be any piece of hardware that is able to communicate with the RNI. The built system can be a very heterogeneous system where one node is a *Altera Nios II* system, another node a system based on *Gaisler Leon 3* processor, and a third node is a hardware accelerator. It could be even possible that a network resource consists of multiple processors and some attached hardware blocks.

Currently, the NoC System Generator supports only the *Altera SoPC Builder*. Therefore, it must be possible to connect the used hardware via the *Altera Avalon*
bus with the RNI. For this thesis, we assume the network resource to be a single Nios II core with some local scratchpad memory. It is possible that some additional hardware is connected to a core, but as the mapping does not consider such hardware it is only possible to access this hardware by a single process.

3.7. Synchronizing the Execution: Heartbeat Principle

To enable time-predictable execution of synchronous systems modelled in ForSyDe, a so called heartbeat was introduced to the NoC System Generator platform. The heartbeat divides the execution into cycles of equal length, similar to a clock in synchronous hardware. Therefore, we will use throughout the thesis the terms frequency and period analogous to their usage in the context of a hardware clock. The term heartbeat delay is used to describe that something is delayed until the next occurrence of the heartbeat. To keep the system behaviour predictable it is necessary that the execution of all processes and all the transfers in the network have finished before the heartbeat occurs. If this is not the case, we call it a heartbeat violation.

All the data sent via the RNI is delayed by one heartbeat. For example, a packet that was sent by the sending process in heartbeat cycle 1 is available in the receiving channel of the destination process during heartbeat cycle 2 and only in that cycle. This also limits the number of packets that can be sent between two processes to one per heartbeat cycle. The system designer itself does not need to take care of the heartbeat, as long as he or she follows the design flow of the NoC System Generator. The synchronisation with the heartbeat is done within the main() function in the file synchronous_MoC_main.cpp which is automatically generated for every network node with an Altera Nios II processor.

3.8. The System Designer’s View

This section gives an overview of the usage of the NoC System Generator and points that a system designer has to consider. Firstly, the processes within the network and its generation are explained and then the details of the configuration files.

3.8.1. The Network Transfer Process

To understand how to write software properly considering the heartbeat cycles, and to analyse the timing behaviour, a good understanding of what is happening during
3.8. The System Designer’s View

A data transfer in the network is necessary. This applies also to messages which are sent between processes on the same node, as they are also sent via the network switch in the actual implementation. Table 3.2 shows all the steps of a message transfer from the source process to the destination process. It also shows, in the second column, the command which is used for this step. The third column shows to where the time of this step belongs to. This will be used to build up a timing model of the system in Chapter 5.

<table>
<thead>
<tr>
<th>action</th>
<th>command</th>
<th>associated time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Write data to send channel</td>
<td>IO write</td>
<td>WCET&quot; of process</td>
</tr>
<tr>
<td>2) Check if RNI is ready</td>
<td>NOC_RNI_STATUS</td>
<td>WCET + waiting time</td>
</tr>
<tr>
<td>3) Invoke sending of packet</td>
<td>NOC_RNI_SEND</td>
<td>WCET</td>
</tr>
<tr>
<td>4) Waiting for the network</td>
<td>—</td>
<td>transfer time</td>
</tr>
<tr>
<td>5) Transfer through network, including deflection</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>6) Reconstruction of data and write to receive channel</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>7) Check if packet was received</td>
<td>NOC_RNI_CHK_MSG</td>
<td>WCET</td>
</tr>
<tr>
<td>8) Read data from receive channel</td>
<td>IO read</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.2.:** Network transfer process in the generated NoC: The first column describes the different steps, the second column shows the associated commands in the software, and the third column contains the part of the timing analysis it is added to. The communication is shown as one process, thus step 7) and 8) happening within the next heartbeat period.

The first three steps are a part of the source process’ program. Firstly, the message is written to the send channel by a write accesses to the according memory addresses. The RNI is capable of executing only one network transfer at any one time, hence the RNI must be checked if it has finished the previous send command. If the RNI is ready, the packet can be sent. The time these steps take are covered by the worst-case execution time analysis. But for the second step, some additional waiting time can occur.

From step 4 on, the message is within the network. If all the network links of the node are assigned to other flits; the RNI, as it has the lowest priority among all the links, must wait to inject the flit. This is a rather rare case, but it must be considered for worst-case calculations of the transfer time. After a flit is injected...
to the network it follows its path through the network, but can be deflected which can prolong the transfer time. At the destination node the data is reconstructed and written to the receive channel. Steps 4-6 can be summed up to the network transfer time.

After the occurrence of the heartbeat, the receiving process can check if some data has arrived. If not, this can be interpreted as the absent value of the synchronous model of computation. If a message is available it can be read from the receive channel. The time of these two last steps is covered by the worst-case execution time analysis of the receiving process.

3.8.2. NoC Platform Generation Process

Figure 3.9: The Platform Generation Process of the NoC System Generator and the related files.

Figure 3.9 gives an overview of the platform generation process. The NoC System Generator stands in the center of the process. The generator reads the hardware specification and the processes structure from the Target Description File. Out
of this information it creates the **Hardware Description Files** and the **Software Projects**.

The **Hardware Description Files** consist of system descriptions in the format of targeted FPGA vendor and additional VHDL files that describe the Network-on-Chip and additional IP blocks. In the case of Altera FPGAs, the system description is a .sopc file that is processed by the Altera SOPC Builder.

Beside the hardware description, the NoC System Generator creates a software project for every processor in the NoC. It also creates a **Target Description Header** and a source file that schedules the processes. The **Target Description Header** (software_configuration.h), which will be explained in detail in Section 3.8.4, is node specific and contains all the necessary information about the system that a program need. The **Scheduler** (synchronous_MoC_main.c) contains the main() function of the node’s software. It has two simple functions: Synchronization with the heartbeat and executing the processes in the correct order. Additionally to the generated files, the NoC System Generator copies the driver (kth_avalon_noc_rni_regs.h) and the target code files belonging to the node to the software project folder.

Both the **Hardware Description Files** and the **Software Projects** are used by the back-end tools provided by the FPGA vendor to create a running system.

### 3.8.3. Device Driver

The network operations are abstracted by a device driver. In the actual version for the Altera Nios II processor the driver is a single .h-file called kth_avalon_noc_rni_regs.h. Before including this file to the source code the base address (NOC_RNI_BASE) of the RNI must be set. This section describes shortly the macros provided for the user. The parameter base – used by every command – is the base address of the RNI.

**NOC_RNI_SEND(base,priority,spid,dpid,buf,msg_size)** invokes the RNI to send the message stored in the channel buffer (buf) to its destination process defined by its PID (dpid). The PID of the sending process is indicated by the spid parameter. The message size is given by the msg_size parameter and a priority is assigned by the priority parameter.

**NOC_RNI_STATUS(base)** checks the status of a the preceding send command. Returns ’1’ as long the RNI is busy.

**NOC_RNI_CHK_MSG(base, channel)** checks for a received message channel. Returns ’1’ if there is a new message in the buffer.

**NOC_RNI_CLEAR(base, channel_id)** clears the ’message received’ flag of the channel (channel_id). Should always be called after the buffer was read.
3. The Network-on-Chip System Generator

NOC_RNI_MSG_LENGTH(base, src) returns the message length of the received
message in the channel indicated by src.

NOC_RNI_DEST_PID(base, src) returns the destination PID of a packet in the
receive buffer src.

NOC_RNI_NODE_NR(base) returns the number of the node on which a process
is running.

NOC_RNI_SRC_PID(base, src) returns the source PID of a packet in the receive buffer src.

NOC_RNI_READ_SYNCHRONIZER_FLAG(base) reads the synchronization flag
that indicates the occurrence of a synchronization event on the network.

NOC_RNI_CLEAR_SYNCHRONIZER_FLAG(base) clears the synchronization
flag. This must be done before a new heartbeat occurrence can be registered.

The driver works on a relatively low level. Therefore the programmer must be aware
of the sending and receiving processes within the RNI, as described in the previous
Section 3.8.1.

3.8.4. Target Description Header

(software_configuration.h)

The file software_configuration.h is generated by the network generator. One
of these system descriptions exist for each node. It is a C header file that specifies
the system specific parameters relevant to the software running on the node. To use
the network functionality by a processes, this file must be included in the code. The
device driver is included by the system description header, and it is not necessary to
include it separately. It contains all the PIDs of the processes running in the system,
and names are assigned to the receive and send channel numbers to make their
usage readable, e.g. recv_channel_p0_from_p1 for the channel on which process
0 receives data from process 1. The hardware address of the RNI is mapped to
NOC_RNI_BASE, which is used to access the network with the commands of the device
driver. The names of the nodes’ hardware resources are also mapped to names that
are easier to read, as they contain the node name.

3.9. System Description Files

In order to generate the NoC platform automatically; a set of configuration files
are passed to the NoC generator. The hardware specification and the process
mapping is described in a single XML file. The functionality of the processes is
described by a set of C++ files. The following section describes the format of these files.

### 3.9.1. Target Description File (XML)

The structure of the target description file is shown in Listing 3.1. The first section, just after the leading `<system>` tag, specifies the system parameters and the target platform. These parameters define the working path, the system name, the target manufacturer, the board type, etc. The generator uses these parameters to create the system configuration files for the chosen target platform.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<system name="NoC_2x2">

<!-- Platform specification -->
<parameter name="param1" value="val_1" />
<parameter name="param2" value="val_2" />
...

<hardware>

<noc>
    <parameter name="nocType" value="Mesh" />
    <parameter name="nocKind" value="2DNoC" />
    <parameter name="nrofCols" value="3" />
    <parameter name="nrofRows" value="3" />
    <parameter name="framesize" value="64" />
    <parameter name="GlobalSync" value="1/uni2423Hz" />
    ...
</noc>

<node nr="0" cpu="{nios,tiny}" ...
<node nr="1" cpu="{nios,tiny}" ...
...
</hardware>

<software>
...
</software>
</system>
```

The configuration of the hardware is embedded into the `hardware` element. The first part (within the `<noc>` element) specifies the parameter of the underlying network; such as topology, size, and version of the NoC. The `framesize` parameter defines the
size of the input and output buffers of the system and the GlobalSync parameter sets the heartbeat frequency.

The second part specifies the details of the different nodes. Each node has its own node element, which describes the hardware that should be created. The main parameter here is the processor of this node. Further parameters specify additional hardware connected to the processor, such as memory, PIO, JTAG, IP blocks, and so on.

Additional IP blocks can also be specified inside the hardware tag which is not further described here.

Listing 3.2: Process Declaration within the Target Description File

```xml
<software>
  <parameter name="Repository" value="D:/NoC/SW" />
  <process name="p0" moc="Synchronous" node="0"
        sources="\{p3\}" targets="\{p1\}"
        files="\{process_0.c\}" />
  <process name="p1" moc="Synchronous" node="0"
        sources="\{p0\}" targets="\{p2\}"
        files="\{process_1.c\}" />
  <process name="p2" moc="Synchronous" node="1"
        sources="\{p1,p3\}" targets="\{p3\}"
        files="\{process_2.c\}" />
  <process name="p3" moc="Synchronous" node="2"
        sources="\{p2\}" targets="\{p0,p2\}"
        files="\{process_3.c\}" />
</software>
```

The last section in the target description file — embedded within the <software> element — describes the mapping of the processes to the network nodes. An example mapping of four processes to three nodes is shown in Listing 3.2. The directory where the process code could be found is specified by the Repository parameter. Then every process is specified by a process element. A process is configured by the process name (name), its model of computation (moc), the node on which it runs (node), and the C++ file containing the process code. The processes from which the process receives data and the processes it sends data to are specified by the source and target parameters. This information is used to assign the input and output buffers to the associated process. The order in which the processes are specified here, is also the order of execution on the nodes.

3.9.2. Target code (C++)

The code of the processes has to be written in a certain way to work on the generated network according to the synchronous model of computation. The basic structure of a process description is shown in Listing 3.3. This is a sample code and many details
are left out. For a detailed example, the examples of the network generator can be consulted.

Listing 3.3: Structure of the process code provided to the network generator

```
#include "software_configuration.h"

void p0_init(void)
{
    // Perform initializations

    // Write initial message to RNI
    while (NOC_RNI_STATUS(NOC_RNI_BASE)!="0"
          NOC_RNI_SEND(...);
}

void p0_main(void)
{
    int recv_value = NOC_RNI_CHK_MSG(...);
    if (recv_value > 0)
    {
        // Read message from RNI
        // Process message
        // Write message to RNI

        while (NOC_RNI_STATUS(NOC_RNI_BASE)!="0"
               NOC_RNI_SEND(...);
               NOC_RNI_CLEAR(...);
    }
    else
    {
        // Absent value
    }
}
```

Firstly, the `software_configuration.h` header is included. As described in Section 3.8.4; this file loads the network driver and describes the processes in the system.

The code itself consists of two functions, which are called by the main code of the process which is created by the network generator as described in Section 3.8.2. The first function `xxx_init(void)` is called once when the process starts executing. The second one `xxx_main(void)` is executed in every heartbeat period.

The `init` function should perform the necessary initialization to guarantee a proper start of the system. As the initialization can be seen as the first cycle of the synchronous execution the initial values of delay elements also have to be sent to the network. This done by the three steps: (1) writing the message to the send buffer of
the RNI, (2) wait until the RNI is ready to send, and (3) initiate transmission to the network. It is important that the `init` that execution of the `init` function does not take longer than the heartbeat period.

The structure of the `main` procedure is slightly more complex. Firstly, the RNI is checked if a message was received. This is done by calling the driver macro `NOC_RNI_CHK_MSG`. If there was no message received, this is interpreted as: the `absent` value of the synchronous model of computation or a heartbeat violation occurred. In these cases the `else` branch is executed. In the case of a message was received, it is read from the receive buffer of the RNI, processed and written back to the send buffer of the RNI. As soon as the RNI is not busy anymore with sending the previous value (`NOC_RNI_STATUS`), the transmission can be started. Finally the flag signalling a received message must be reset by `NOC_RNI_CLEAR`.

### 3.10. Conclusion

The NoC System Generator gives the possibility to create complex systems-on-chip based on a network-on-chip automatically. The current version does not support all desired hardware vendors yet, but it shows already its potential. The functionality of the heartbeat enables time-predictable program execution within the synchronous model of computation. Therefore, the NoC System Generator can be used as the back-end for the mapping process developed in this thesis.

As already discussed in this chapter, the limited predictability of the deflection routing is a problem to ensure the desired predictability of the whole system. In the following chapters, the limitations that arise through will be shown several times. Apart from this, it is important for the thesis that the reader understand the heartbeat principle, the sending process, and the general process of the system generation, which will be connected with ForSyDe in the next chapter.
From ForSyDe to the NoC platform

The previous chapters outlined the ForSyDe design methodology and the NoC System Generator, which together provide the basis of the thesis. In this chapter these chapters are concluded and an outlook to the following chapters and to some future work is given. We propose a system creation process and give a brief overview how to get from a synchronous system design by ForSyDe down to a final system created by the NoC System Generator. In doing so, also the work of this thesis is positioned within this process.

4.1. The Automatic System Generation Process

The basic vision behind the thesis is a fully automated system generation process. This means that a system designer only has to model a system on a relatively high level of abstraction, give some target specifications, and performance goals. The remaining steps of the hardware and software creation will be conducted by the design tools.

The process we propose in this thesis is shown in Figure 4.1. The existing parts ForSyDe and NoC System Generator build the base on the front-end and back-end side. There are basically two different paths: On the left hand side is the path of the mapper which fits the process structure to the architecture, and on the right hand side the code generation which transforms the code of the model to code that can run on the target. This thesis focuses on the mapper which is drawn by a dark shaded box. A first implementation of the mapper will be presented in Chapter 6. All the functions drawn by white boxes does not exist yet, but are essential for a completely automated design process.

The whole process could be combined to a stand-alone program, or the different functions could be implemented separately and linked by a script.
4. From ForSyDe to the NoC platform

Figure 4.1: Automatic System Generation Process: The lightly shaded boxes show the existing parts and the dark shaded box is the mapper which is the core of the thesis.
4.2. System Modelling with ForSyDe

input:  
- Application specification

output:  
- Process structure of System (XML)
- Process Code (C++)

ForSyDe is used as the front-end of the automatic system generation process. A system designer models the system specifications by the ForSyDe design methodology, as briefly described in Section 2.5, i.e. the designer describes the system by processes constructors connected by signals. As only the synchronous model of computation is supported, only those process constructors are allowed to be used. Guidelines for the system designer should be developed, which helps him or her to model the system in a way that allows an efficient implementation. For example, a trivial guideline is to split the system into as many processes as possible. For the combinatorial and FSM process constructors, code is written in C++. The result is the specification model, whose functionality can be validated by compiling and running the SystemC code.

The next step is the design refinement of ForSyDe. By this step, the specification model is optimized for an implementation. We see three different areas where refinements can be applied: (1) Optimizations that improve the flexibility for the mapper, e.g. splitting processes. (2) Optimizing the delays between the processes. Because the delays are essential to allow the efficient mapping of a synchronous system (Section 5.3.2), as many delays as possible should be placed if possible. But, there are also limitations of delay placement in parallel paths and loops, as the system’s functionality changes when only one signal of two arrives later to some process. All these delay considerations can be implemented as design refinements, work in this area was presented by Raudvere in [37]. (3) The hierarchy of the system can be flattened which is a prerequisite for the mapper.

The result of the refinement process is the implementation model. This can be validated again by the same methods as the specification model. Finally, the system structure is exported to an XML file. Together with the process code, they are the input to the other steps of the system generation process.

4.3. Code Generation (Pre-Mapping)

input:  
- Process Code (C++)
- Process structure of System (XML)
- Process Constructor Templates

output:  
- Intermediate Code (C++)
4. From ForSyDe to the NoC platform

A part of the ForSyDe design methodology is the Design Library, which contains implementations for the different process constructors, i.e. for every process constructor type must exist an implementation for every target platform. In our system creation process we call them process constructor templates (PC Templates). The code generator creates code for every process in the structure description a code file by selecting the according PC template from the library.

The code generation is split into two parts pre-mapping and post-mapping. The reason for this is that the result of the mapper can have some impact on the code. The intermediate code generated by the pre-mapping code generator should be as close as possible to the final code, as the process analysis is run on this code. All changes on the code caused by the mapper must be considered within the mapping process and heartbeat calculation.

A special case are the delay process constructors. As it will be described in the section on the signal modelling (Section 5.1.2), the delays are modelled within the signals and not as processes. Therefore, the initial value of a delay process must be sent by the init() function of the target code (Section 3.9.2) of the preceding process.

For some process constructors — e.g. combinatorial, mealy, moore — additional code is necessary, which describes the function. This code is found in the process code files. The code in these files follow the conventions of ForSyDe and cannot used directly for the NoC System Generator. The input and output values of the function must be analysed and implemented as network communication. The code itself is marked in the process code file (Section 2.8.2) by the following pragma notation:

```
#pragma ForSyDe begin
// Code of function
#pragma ForSyDe end
```

As code accessing the target hardware cannot be run within ForSyDe-SystemC, this also has to be added by the code generator. This can be the case for source and sink processes. We propose to add special process constructors with a dedicated PC template for each hardware device that can be part of the system. This does probably not follow exactly the philosophy of ForSyDe, but allows the automatic generation of the system.

4.4. Process Analysis

- Intermediate Code (C++)
- Process Parameters
The mapper needs the properties of the processes to determine a mapping. The properties comprise timing properties such as worst-case execution time (WCET) and best-case execution time (BCET), and memory demand. As discussed by Wilhelm et al. [47], the evaluation of WCET and BCET is a hard problem. Bounds can be evaluated by timing analysis which can be either done by execution of the code on the target architecture or analysis of the program code. One tool that performs static timing analysis by analysing the code is the SWEET project [18] from Mälardalen University.

To analyse every process code file separately is not enough. Other factors also affect the execution time. For example the time the scheduler that invokes the execution of the processes must also be considered. This can be done separately, but the problem is that effects of the different processes to the execution of the scheduler are not covered. This problem is not easy to solve and so far we propose to add a large enough margin to the scheduler’s execution time. Also accesses to hardware can cause unexpected waiting time.

In summary, the process analysis is an important part to allow the mapper to find accurate solutions and must be carried out carefully.

4.5. Mapping

**input:**
- Process Structure of System (XML)
- Platform Specifications
- Process Parameters

**output:**
- Target Description File (XML)

The mapping is the central part of this thesis. On the one hand, it takes as an input the process structure of the ForSyDe model and the properties of the processes from the process analyser. On the other hand it takes the platform specifications, which describe the properties of the available platform. This could also include some performance requirements which must be met by the calculated mapping. The mapper itself assigns the processes to the available hardware resources and determines an execution order of the processes. The details will be explained in Section 5.3 et seq. The output of the mapper is the target description file that is used by the NoC System Generator to generate the system.

For the mapping of a synchronous system, as described by this thesis, also the heartbeat period calculation is done by the mapper and written to the target description file.
4. From ForSyDe to the NoC platform

4.6. Code Generation (Post-Mapping)

input:  
- Intermediate Code (C++)
- Platform Description (XML)

output:  
- Target Code (C++)

Some mapping decisions — e.g. the hyper processes creation which will be introduced in Section 5.2.1.1 — need changes in the target code. The post-mapping code generator takes the intermediate code and applies these changes to create the final code which will run on the target platform.

4.7. NoC Platform Generation

input:  
- Target Description File (XML)
- Target Code (C++)

output:  
- Hardware Description Files (.sopc & VHDL)
- Software Project Folders
- Target Description Header
- Scheduler

The generation of the hardware description files and the software projects for the designated target architecture is finally conducted by the NoC System Generator. This process is described in Section 3.8.2. The generated files are used to synthesize the system by the target architecture vendor’s tools.

4.8. Conclusion

In this chapter, we proposed a design flow for the complete system creation process. In this design flow, a system designer must solely model the system by the ForSyDe design methodology and specify the properties of the hardware platform. The rest of the system creation runs automatically. The mapper, which is the content of the rest of this thesis, is the central part for the hardware and process structure configuration. But it is not the only action necessary between ForSyDe and the NoC System Generator. The code for the processes must also be generated, so that it can run on the target platform. For reasons presented in the next chapter, it is essential for the mapper that it has accurate properties of the processes. Therefore, enough attention must be given to the selection of the process analyser.
5 System Model and Mapping

The mapping problem is modelled in this chapter by applying a bottom-up approach. In the first part, models for the application and the target architecture are established. Based on those, the developed mapping process is described in the second part, whereas the effect of the heartbeat is an important factor to consider. The modelling framework is based on directed graphs and is similar to the ones found in [10] and [45]. This chapter discusses the theoretical background and an implementation will be presented in Chapter 6.

5.1. Modelling of the Application

The problem graph represents the application as it is described by ForSyDe-SystemC. It is modelled as a dependence graph that is denoted as $G_P = (V_P, E_P)$. The vertices $V_P$ are either functional tasks or communication tasks. Figure 5.1 shows how a problem graph is constructed from a multiply-accumulator ForSyDe model. In general the processes $P$ of a ForSyDe model are represented by functional tasks and the signals $S$ by communication tasks. The exception are the delay process constructors which are included to the communication tasks. The edges connect the processes with the corresponding signals: incoming edges are signals received and outgoing edges signals sent.

5.1.1. Process Model

Computational and control parts are represented by processes $P$. According to the synchronous assumption described in Section 2.2, they must produce their output synchronously with their inputs, which means in other words: The output is produced immediately after an input was assigned. In reality, a process is a piece of sequential code whose execution takes some time. This time is called execution time $\epsilon$ and can vary from one run to another. Thus, bounds for the execution time are specified.
5. System Model and Mapping

Figure 5.1: Problem graph of a multiply-accumulator: The process constructors are represented by process nodes (white) in the problem graph, and the signals between them as signal nodes (grey). The exception is the delaySY process constructor which is included to signal $S_4$.

The worst-case execution time (WCET) represents the maximum time, the execution of a process can take. The best-case execution time (BCET) is the time, the fastest possible execution takes. The model developed in this thesis only uses the WCET, but if the time of occurrence of a packet in the network has to be determined the BCET is also important.

Other hardware related properties could be assigned to processes which should be considered for the mapping. One of them is the maximum memory utilization which is also a result of the process analysis. The memory utilization, as well as the WCET and BCET depend much on the processor, they are running on. Hence sets of these properties exist if the target architecture is heterogeneous, e.g. different processor types are assigned to different network nodes. Into the same area goes special hardware (IO devices, accelerators, etc.) to which a process needs to have access. All these hardware related issues are not further covered by this thesis.
5.1.2. Signal Model

Signals $S$ model the communication between two processes. Whereas the term 'signal' is lent from the ForSyDe terminology. Every signal is defined by two parameters: the data volume and the heartbeat delay.

In ForSyDe, data types assigned to the signal are fix. Therefore, the amount of data that is sent every cycle has also a fixed size. This is modelled by the data volume $v$. The time a communication takes, depends on the one hand on the data volume, and on the other hand on the mapping of the associated processes. The mapping defines the used communication link and if there is some network congestion that needs to be considered.

For two reasons, the heartbeat delays are also represented by the signal model and not by the process model, as this is the case in ForSyDe. Firstly, the delay is added between many of the processes to decouple them. It can be seen as a communication in time, in contrast to the communication between two different nodes which is communication in place. Secondly, the heartbeat delay support is integrated to the generated network, which is a facility for communication. The main parameter of the heartbeat delay is the number of delays that have to be added by the communication link.

Different factors affect the placement of delays in the system. To define exact rules, that specify under which conditions a delay can/must be placed for a certain signal, is rather complex. By giving the mapper some possibility to add or omit delays, it could gain flexibility for finding an optimal solution. Thus, a delay state property belongs to the signal model. In the current version of the model three different states for delay placement are defined: (1) A delay must be added for this signal, e.g as it is necessary in loops. (2) A delay must not be added at this signal, all the other signals in a loop need this setting to maintain the functionality. (3) For this signal a delay can be included arbitrarily. This is the case for paths where no dependencies to other paths exist.

5.1.3. Time Model of a Process

It is essential for the mapping process to know the timing of the processes and their associated signals. Here, we define some important time properties of processes.

The time model is based on the input-process-output scheme. A process can run as soon as the data from the incoming signals is available, processes this data and finishes the execution by sending the generated data. The output data is usually one output signal, or no signal if the process is built by a sink process constructor. Figure 5.2a shows this simple process model. Firstly, the process executes for the execution time $\epsilon$ and creates the data which are usually ready by the end of the
5. System Model and Mapping

execution. Thus the communication time is put after the computation time in the model.

While the communication is carried out another process can already start running. The communication time is therefore drawn shifted to the right. The time the data takes to be transmitted is called communication time $c$. The communication time is only the time the transmission takes on a communication link, but not the execution time of the commands that initiate the communication. The communication time is the time it takes until the output data is completely arrived at the destination.

![Diagram showing time models of processes](image)

**Figure 5.2:** Time models of processes: A simple process is shown in a), b) shows the case when a process sends two signals, and in c) $P_1$ must wait with sending the signal until the signal of $P_0$ is sent.

In two cases, it is possible that a process sends several signals; (1) the same signal is sent to several processes or (2) a process is created by a *unzip* process constructor. But for the model of the process itself and its timing properties, these signals can just be summed up, as they are always sent at the end of the process execution. An example of a process $P_1$ sending two signals $S_1$, $S_2$ is shown in Figure 5.2b.

As given by the RNI design (Section 3.6.2) only one packet can be sending at any time instant. If the transmission of the previous process’ packet is not finished, the process has to wait until the new send command can be initiated. Consequently,
the process occupies the hardware unit for a longer time. We will call the whole time a process occupies an execution unit \textit{run time} $r$, which is the larger one of the execution time of the actual process $\epsilon_i$ and the communication time of the previous process $c_{i-1}$ on the same processor.

$$r_i = \max[\epsilon_i, c_{i-1}]$$ \hspace{1cm} (5.1)

Figure 5.2c shows an example where a previous transmission $S_0$ has not finished before the computation of the process $P_1$. The process is stalled for some waiting time $t_w$ which is the difference of the running time $r$ and the execution time $\epsilon$.

$$t_w = r - \epsilon$$ \hspace{1cm} (5.2)

The waiting time over all processes can be used as a metric for the efficiency of a mapping.

To ensure a predictable behaviour, the worst-case scenario is the most interesting. The execution time of a process is therefore considered to be equal to the WCET. Depending on the mapping and implementation, some additional code will be necessary to realize the scheduling and mapping. The execution time of this code must also be added to the execution time of the processes.

5.2. Modelling of the Architecture

The target architecture is represented by the \textit{architecture graph} $G_A = (V_A, E_A)$ which is similar to the problem graph. The vertices are either functional resources or communication resources. Functional resources are units that processes the data in a certain way, and we call them \textit{execution unit} $E$. The communication resources represent the facilities by which the execution units can exchange data, they are called \textit{communication link} $L$. The directed \textit{edges} of the architecture graph shows the access that execution units have to the communication links. It is possible to use architecture graphs of different abstraction levels during different phases of the mapping.

An example of a 2x2 2D-mesh Network-on-Chip and the corresponding architecture graph is shown in Figure 5.3. Even though the Network-on-Chip consists of eight network links between the nodes, it is modelled as a single communication link $L_{NOC}$. This model of the Network-on-Chip was chosen, since a fixed assignment of signals to network links is only possible for deterministic routing policies, but not for non-deterministic routing.
5. System Model and Mapping

5.2.1. Execution Unit Model

An execution unit (EU) is any piece of hardware, or in the special case of the hyper process (described in the next section) a software construct, that is capable to execute processes. EUs can be for example different processors or hardware accelerators. All execution unit types have properties like execution speed and available memory. Those are not parameters of the EU itself, instead they are represented within a process by different WCETs.

Other hardware components can be connected to a EU as well, e.g. a display, switches or a timer. It is essential that the mapper has the information which hardware units are available, so processes using these hardware can be mapped to the same EU. The number of send and receive channels in the RNI of a network node is also limited and their number must not be exceeded by the processes running on an EU. As already mentioned, these hardware issues are not covered by the presented mapping algorithms.

5.2.1.1. Hyper Process

Communication between two processes by the network facilities always imply that the data is available to the receiving process in the next heartbeat period. There are two cases where this delay is unwanted. The first case is, if an additional delay would change the system’s functionality; for example in loops. The second case is the optimization of timing parameters. To avoid the heartbeat delay between two processes, their communication must not involve the RNI. The principle of the hyper process is introduced to cope this problem.
A hyper process is a software construct that groups processes together, and allows processes to communicate without a heartbeat delay. Depending on the situation, a hyper process is handled like an execution unit or like a process.

As the processes within a hyper process are not decoupled by the heartbeat, they cannot be executed independently anymore, as it is the case for other processes. The processes within a hyper process must execute in the correct order to fulfill their data dependencies. As we will see this is a drawback, as it reduces the flexibility of the mapping algorithms. To simplify the hyper process handling, in this thesis a hyper process groups the processes closely together and no other process can be executed between these processes.

In the mapping phase when processes are assigned to the EUs, it can be beneficial if a hyper process is modelled as an internal EU of a physical processor. Thereby, the differences between the communications within a hyper process and the communication between hyper processes can be part of the optimization. In this case, a hyper process is a sequential sub-processor that runs the assigned processes sequentially.

In other cases, a hyper process can be seen as a single process. The main difference is that the signals are not only sent at the end of the execution. Therefore not the complete communication time should be considered in the mapping algorithms, but only the communication of the last process, called communication overhead $c_{OH}$. The run time of the hyper process $r_{HP}$ is the sum of the run time $r_i$ of all assigned processors $N$.

$$r_{HP} = \sum_{i=0}^{N} r_i$$  \hspace{1cm} (5.3)

For the scheduling also the time of the first signal transmission is important. This is equal to the execution time of the first process of the hyper process and is denoted by $\epsilon_1$. Figure 5.4 shows this parameters in an example.

For the implementation in the NoC System Generator, a hyper process is the same as a single process. Therefore the processes must be combined together in one source code file during the design process, before the NoC system is generated. The communication within a hyper process must be realized by the local memory of the node.

Creating hyper processes has to be done carefully, as there are some design issues that can occur. The communication within a hyper process saves time, but more memory in the execution unit is necessary, since the data produced by one process must be stored locally in order to be used by the following processes.
5. System Model and Mapping

Figure 5.4.: Time model of a hyper process: The hyper process consist of processes $P_1$, $P_2$, $P_3$. The execution time of the first process $\epsilon_1$ and the communication overhead are the interesting parameters of a hyper process.

5.2.2. Communication Link Model

The channels for data exchange between the execution units are modelled by the communication links. The properties of a communication link, together with the data volume of the signal determine the communication time of a process.

There exist three different communication link types. The first one is the communication by the Network-on-Chip of the platform. The second one is the communication of two processes on the same network node via the Resource Network Interface\(^1\) (RNI). And the last one is the communication within a hyper process, which is done by combining the two communicating processes within the same hyper process.

All three have different characteristics of which the main ones should be mentioned: (1) The two communicating processes can be located on the same node or on different network nodes. (2) A heartbeat delay is necessary or desired between the two processes. (3) The communication time must be exactly predictable or not. (4) The time, it takes to send a signal from one process to the other. Table 5.1 gives an overview of the properties of the different communication types, before they are explained in the following sections.

\(^1\)The Resource Network Interface is the connection between the network resource and the network itself. More explanation in Section 3.6.2.
5.2. Modelling of the Architecture

<table>
<thead>
<tr>
<th></th>
<th>Process Location</th>
<th>Heartbeat-delay</th>
<th>predictability</th>
<th>communication time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td>different node</td>
<td>yes</td>
<td>low</td>
<td>medium-long</td>
</tr>
<tr>
<td>RNI</td>
<td>same node</td>
<td>yes</td>
<td>medium</td>
<td>short</td>
</tr>
<tr>
<td>Hyperprocess</td>
<td>same node</td>
<td>no</td>
<td>high</td>
<td>very short</td>
</tr>
</tbody>
</table>

*a*Given by the usage of deflection routing in the Network-on-Chip

Table 5.1.: Comparison of the three communication types on the generated NoC platform. The combination of process location and heartbeat delay restrict the choice between them.

The choice of one communication type is often limited by the system’s functionality and the mapping of the processes. For example, the communication by the network must be chosen if the two processes are located on different nodes. If no heartbeat delay is allowed between two processes, they must be combined in a hyper process. On the other hand the properties of the communication types can be used by the mapper to optimize the final implementation.

5.2.2.1. Communication via the Network-on-Chip (Type I)

If two processes are located on different nodes in the network, this is the only type of communication that can be used. But, the communication on the network is the slowest among all three. To evaluate the exact communication time, many factors are necessary to consider: the distance of the nodes, bandwidth of the network links, and congestion in the network. Basically, the complete traffic within the network must be known to get an accurate estimation of the communication time. Especially, if there is congestion in the network this can be very difficult, and if not enough information is available a high bound must be used to ensure that the system does not miss the heartbeat.

The usage of this communication type follows the descriptions about the network access as it is described in Section 3.8.

5.2.2.2. Node-internal Communication involving RNI (Type II)

The communication involving the RNI is used for the communication between processes or hyper processes running on the same network node. The processes are decoupled by the channels of the RNI and the data is accessible by the receiving process in the next heartbeat period.
In the actual implementation of the NoC System Generator platform, the data is transferred from the send channel to the receive channel via the network-on-chip. Thus, the flits are deflected to the network if another packet is received at the same time. A high value for the communication time must be chosen if it there is no guarantee that no other packet is arriving at the same time.

The transmission on this link type works the same as the one between two different network node, for details about the usage see Section 3.8.

5.2.2.3. Communication within Hyper process (Type III)

This communication differs from the others as it does not involve any hardware communication facilities. Thus, the communication within a hyper process is the fastest among the three communication types. The communication time itself is 0, but some additional execution time for the scheduling and memory access is necessary. Whereas, the difference in communication time to the node-internal communication with RNI comes mainly from actual implementation of the RNI that even internal communication is sent to the switch where congestion on the resource channel can occur. If no heartbeat-delay is allowed between two processes this communication type must be used. It can also be used to optimize the heartbeat period by reducing the communication time between two processes.

As previously described in Section 3.8, the processes in the network are addressed by their PIDs and since a hyper process is handled as a single process by the NoC System Generator; it is not possible that two processes within a hyper process send data through the network to two processes combined in another hyper process.

5.2.3. Architecture Graph of the NoC System Generator

Putting the models of the execution units and communication links together, the architecture graph of the NoC System Generator platform is modelled as shown in Figure 5.5. The shown example platform consists of four network nodes with single processors $E_1$, $E_2$, $E_3$, $E_4$ and the network-on-chip that connects them. Each processor contains three hyper processes to which the processes can be mapped. Within this model, processes cannot be mapped directly to the processors. Also single processes must be mapped to a hyper process.

The type I communication that is transferred in the network is represented by the $L_{NOC}$ vertex. The NoC can be accessed by any hyper process. The communication within a node including a heartbeat delay (type II) is represented by the $L_{RNI,1}$, $L_{RNI,2}$, $L_{RNI,3}$, $L_{RNI,4}$ vertices. All hyper processes within one node have access
to them. The type III communication is not modelled separately as it happens internally of a hyper process.

![Architecture graph of the NoC platform. The shown architecture is a NoC with four nodes (EU_P) and every node can take up to three hyper processes (EU_HP).](image)

**Figure 5.5.:** Architecture graph of the NoC platform. The shown architecture is a NoC with four nodes (EU_P) and every node can take up to three hyper processes (EU_HP).

### 5.3. The Mapping Process

The presented models for the application and the architecture will be connected now to derive a mapping for the system. The mapping process presented in the next sections creates a static mapping, i.e. the mapping is derived prior to the system generation and processes cannot move to another execution unit during the execution. This is feasible as the intended use is the generation of embedded-systems and simplifies the reasoning about time-predictability. Another advantage is that complex algorithms can be used to derive the mappings.
The first step is to create a specification graph. The specification graph \( G_S = (V_S, E_S) \) combines a problem graph with an architecture graph. The mapping edges \( E_M \) from the problem graph to the architecture graph show to which resources a task can be mapped. The specification graph shows all feasible mappings of the computation and communication tasks. It is possible that communication tasks are mapped to functional resources. An activation is defined as a function that assigns to each mapping edge \( E_M \) and each node \( V_S \) either the value '0' (not activated) or '1' (activated). The feasible mappings can be found in the set of all possible activations.

Figure 5.6 shows a specification graph of the previously presented problem graph of the multiply-accumulator and the architecture graph of the 2x2 2D-mesh network. The grey arrows are the mapping edges. The high number of mapping edges, even in this small example shows already that search space for real examples can be huge.

### 5.3.1. Definition of System Synthesis

System Synthesis, as defined in [10], describes the process of searching for an optimal activation in the specification graph. Three steps are defined to find this activation:

**Allocation (\( \alpha \))** An allocation is the selection of resources which will be available for the processes. Both type and number of resources can be chosen. The allocation can be given by the user, determined by the synthesis process or both.

**Binding (\( \beta \))** A binding is the assignment of a task to a resource. The resource must be one that is capable to execute the task.

**Schedule (\( \tau \))** A schedule defines the execution order of the tasks on one execution unit.

These steps can be executed in any order and can even be combined. The triple of a feasible allocation, a feasible binding, and a schedule is called *implementation*. In this thesis, the allocation is given by the user through the platform specifications. Only the binding and the schedule must be determined by the system synthesis. This combination of binding and scheduling is also called *mapping*.

### 5.3.2. Considerations about the Mapping to the Synchronous Platform

Mapping a synchronous system to a target platform with support of synchronous execution brings one big benefit:
5.3. The Mapping Process

The processes can be placed independently from each other, without affecting the functionality, as long as they are decoupled by delay elements.

Placed means here two things: Firstly, it is irrelevant to which execution unit and in which groups the processes are bound. Secondly, the order of the processes in the schedule of the execution unit is also not important for the functionality. Despite of the high flexibility of the mapping of a synchronous system, it is not trivial to achieve an optimal mapping. The overall goal of the mapping process is to find the mapping with the shortest heartbeat period. The result is the system with the highest possible throughput.

Given that the application is mapped to a homogeneous NoC, i.e. every network node consist of the same processor, the total execution time of all processes is fixed and
cannot be optimized. This is not completely true, as some additional code depending
on the mapping will be necessary, which will add some execution time. But, as this
amount is small compared to the execution time of the processes, it is neglected here.
Given the total worst-case execution time $WCET$ and the number of nodes $m$, to
which the application should be mapped, a lower bound for the heartbeat period
$t_{HB, \text{min}}$ can be calculated by:

$$t_{HB, \text{min}} = \frac{\sum_{i=1}^{m} WCET_i}{m}$$

(5.4)

Additionally to the computation, most of the processes send a signal. The signal
causes some communication time, whose exact value depends on the communication
type to which the signal is mapped. Therefore, the communication times of the
processes can be optimized.

For the given network with deflection routing the most important optimization goal
for the communication must be:

*Avoidance of congestion on network links.*

Congestion makes the communication time unpredictable for the actual version of
the network as discussed in Section 3.4.4. Even if a bound can be given for the
arrival time of packets, the communication would be much higher in the case of
congestion. If congestion can be avoided, short communication times allow the
mapper to schedule the communication parallel to some computation, which can be
formulated as another goal for the mapper:

*The communication should whenever possible take place parallel to some
computation.*

This has two effects: Firstly, the computation does not need to wait to initiate its
data transmission, due to the communication of a previous process that has not
finished. Secondly, the communication overhead, caused by the communication of
the last process in a schedule before the end of a heartbeat period, will be removed
or at least shortened.

These can be seen in Figure 5.7. This figure shows an example mapping of 11
processes to four execution units. Most of the processes does not have to wait for
the communication of the preceding process to finish. Only process $P_8$ has to wait
because the sending of signal $S_7$ cannot run completely parallel to any other process
on this execution unit. The second effect, communication overhead is kept at the
minimum in the shown example. The execution units $E2$, $E3$, and $E4$ have no
communication overhead, as the processes which send no signals $P_6$, $P_9$, and $P_{11}$ are
the last ones in the schedule. On the first execution unit $E1$, the communication
overhead is kept small by scheduling the process with the shortest communication
time $P_3$ last.
5.3. The Mapping Process

In the shown example, all the execution units run for the same amount of time. This helps to get closer to the lower bound of the heartbeat period, as it is given in Equation 5.4. Out of it, we formulate a last optimization goal:

*The run time should be evenly distributed over all execution units.*

Based on the presented considerations we will develop the mapping process in the following sections.

5.3.2.1. Dimensions of the mapping problem

The mapper has three dimensions to optimize the heartbeat of the system:

- Assign a process to a network node
- Execution order of processes on a network node
- Combining processes to a hyper process within a node

Although; these seem to be only a few dimensions, the resulting search space is huge, as the processes decoupled by delays are independent and many solutions are feasible. Thus, the mapping problem is intractable. To solve mapping problems, heuristic methods, such as simulated annealing and genetic algorithms, are usually applied. Mappings based on genetic algorithms are presented by Blickle et al. [10], Lei [27], Pop [35], and Shin [43]. The two main differences of the problems tackled in these papers and the given problem of mapping a synchronous system are: Firstly, the presented problems must consider the dependencies between the processes, and
5. System Model and Mapping

secondly many of the problems assume that only one process is mapped to one network node.

Given that most of the processes within the synchronous system does not depend on each other and several processes can be mapped on one execution unit, makes the the search space even larger than for problems presented in these papers. But, the independence of the processes also allows to split up the mapping process to different steps. By splitting up the problem the search space of the sub-problems is reduced and better solutions can be found for them. The here described mapping process splits the mapping problem to three different stages:

1. Logical Binding
2. Physical Binding
3. Scheduling

Prior these steps the processes that belong to the same hyper process are grouped together. As this does not include any optimization this is not considered as part of the mapping. Step one and two of the mapping process covers the assignment of nodes to the processors. Whereas in the first step the processes are bound to logical execution units which has no relation to network nodes. These logical execution units are placed to the network nodes in the second step. The third step is the scheduling of the processes on the execution units. For all the steps heuristics are used, therefore it is not guaranteed that the ideal solution is found.

5.3.3. Logical Binding

The first step of our mapping process handles the distribution to the available execution units of the target architecture. In this step, the location of the processors within the network and the properties of the communication links are ignored.

We formulate two optimization goals for this step:

- Minimize the number of the connections between the execution units,
- Distribute the execution time of the processes equal to the execution units.

The first goal should reduce the network congestion. Minimizing the number of connections between the execution units, reduces the possibility that two signals of originating from different processes must share a link. The total communication time of all processes will also decrease as more signals are sent by the cheaper communication links (type II and III).

Assuming that the execution time is dominant for the heartbeat evaluation, the execution time should be distributed as evenly as possible to the execution unit,
which is described by the second optimization goal. The first optimization goal also supports this by the reduction of the communication times.

As the application is represented by a graph problem graph), the logical binding is a graph partitioning problem. The worst-case execution time of the processes is used as the weight of the graph edges. The goal is to partition the graph into \( n \) pieces of the same size by reducing the total number of the edges between them. Whereas \( n \) is the number of processors in the system.

If hardware limitations will be considered in the future, it has to be included to the logical mapping. It is important that the mapper does not exceed the available memory and RNI channels of the nodes. Processes that need to access the same special hardware, such as IP-blocks, must be mapped to the same nodes.

5.3.3.1. Kernighan-Lin Algorithm

Kernighan and Lin [24] developed an algorithm to partition a graph into several parts of given sizes, while minimize the edge cost between the partitions. Since its publication the algorithm has been applied in different areas, such as scientific computing, VLSI design and last but not least mapping of processes. The implementation of the logical binding is based on this algorithm.

As a starting point for the Kernighan-Lin algorithm the graph is split into two initial partitions of equal sizes. Then an iteration phase starts. During each iteration a node from each partition is selected. If swapping these two nodes lowers the number of edges between the two partitions, then they are swapped. The iterations continue until no pair of nodes can be found that lowers the edge count between the partitions. When the algorithm terminates, a local minimum was found. To improve the solution the algorithm can be run with different initial partitions, and selecting the best local minimum found by these runs.

The algorithm described above is the simplest version. It describes also the partitioning of the graph into two parts. It is also possible to adapt the algorithm to split the graph in to a larger number of partitions. Many other extensions were also made, for example assigning weight to nodes and/or edges. The weight of the nodes is used in this thesis to distribute the execution times equally to the partitions. The partitions can also have different maximum sizes for different parameters. Different execution speeds, and limitations of memory sizes and number of communication channels, of the execution units can be considered by that.

5.3.4. Physical Binding

After the logical binding of the processes; the logical execution units must be assigned to nodes of the network-on-chip. We call this physical binding. With the position of
the nodes within the network, the communication time can be calculated accurately. As the hardware is generated automatically, there is no restriction where a node is placed within the NoC.

There are two optimization goals to minimize the communication time:

- Avoidance of network congestion on the network links,
- Short communication paths.

Whereas, the avoidance of network congestion must be the main goal to achieve a time-predictable behaviour. Therefore, no two signals must access the same network link at any time instant. As the signals within a process are initiated sequentially, only signals from different processes could interfere on a link. The limitation of the injection rate of a single node to 25% — as it is in the current network implementation — does not help to avoid congestion. In the optimal case, all logical execution units can be mapped that no congestion occurs. If this is possible, the communication time of all signals mapped to the NoC links depends only on the distance of the nodes and all the timing parameters of the system can be evaluated.

In the case congestion occurs on a network link, a closer analysis of the network is necessary. It is possible to exploit the properties of the deflection routing algorithm to ensure that a link is not used by multiple signals. Figure 5.8 shows an example with the implemented routing algorithm. Node ‘4’ sends a signal to node ‘6’, and node ‘5’ communicates with node ‘9’. According to ‘X first, then Y’ routing; it is possible that packets from node ‘4’ and node ‘5’ wants to access link A at the same time. The packet of node ‘5’ is deflected to link B and will reach node ‘9’ via link C. The packet still follows the minimal path and the communication time is kept at the minimum. The drawback is that it cannot be predicted if the packet from node 5 will be deflected; therefore also link D must be kept free for the signal coming from node ‘3’.

This method is very limited. If congestion freeness cannot be ensured, the evaluation of the arrival time of the flits is very difficult and basically needs the timing properties of all the signals sent during one heartbeat period. In general this means that the network’s timing behaviour is unpredictable and an upper bound for the type III communication, called worst-case communication time, has to be used for all signals in the network.

The second optimization goal is a trivial one; the shorter the communication path is, the shorter is also the communication time. But the effect on the heartbeat period is minimal, compared to the effect congestion could have. The transfer of a flit from one node to the next takes only 4 clock cycles in the actual network. So far this optimization goal is not considered by the physical binding algorithm.
5.3. The Mapping Process

Figure 5.8.: Example of a case in which congestion on link A can be tolerated, because links B and C are kept free for deflected flits. For the case the flits from node 3 does not get deflected, link D must also kept free.

5.3.4.1. Algorithm

Many papers were written about binding single tasks to network nodes with different goals; such as energy efficiency [21] [43], execution speed [30], and communication optimization [13] [34] [40]. As the main goal of the physical binding is to reduce the network congestion in the network, the algorithm used belongs to the last class. In [33] Murali and de Micheli present an algorithm for bandwidth-constrained static mapping of processors to a NoC. The algorithm was developed for networks with minimum-path routing, and by tailoring the cost functions it can be applied to the dimension-order routing used by the generated NoC. The experimental results presented in that paper show better results than the compared algorithms.

The algorithm consists of two phases, in the first phase an initial binding is calculated which is iteratively optimized by a second phase. It takes two directed graphs as an input; a graph $G_L$ of logical EUs $V_L$ connected by edges $E_L$ which represent the communication between the EUs, and an architecture graph $G_A$ with network nodes $V_A$ and the connecting links $E_A$. The initial binding, which is shown in Algorithm 5.1, is generated by placing first the logical execution unit $\text{maxL}$ with the most connections with other logical execution units to the node $\text{maxA}$ in the network with the most available network links. Then the logical EU $\text{nextL}$ with the most connections with the already mapped EUs is chosen. It is mapped to the node with the lowest communication cost with all nodes $\text{Placed}$ to which already a logical EU is bound. In the case of the used NoC the goal is to reduce the number of occupied network links. Therefore, the communication cost $\text{commcost}$ is the sum of the distances to all placed EUs with which the new logical EU communicates which represents the number of network links by which a signal travels. This is repeated until all the logical EUs are bound to the network.

The main Algorithm 5.2 first uses the algorithm described before to get an initial
5. System Model and Mapping

binding. This is improved by swapping repetitively two nodes $w_i$, $w_j$ in the actual binding $Placed$. After the swapping the communication cost of between all the placed nodes is evaluated. If this new binding has a better communication cost than the previous best binding $bestbinding$, this will be used as the starting point for the next swapping. As the main goal is to avoid network congestion. The communication cost of the network is evaluated by counting the links on which congestion occur. The calculation of the communication cost is shown in Algorithm 5.3.

Algorithm 5.1 Calculate initial binding

```
function INITIALIZE($G_L(V_L, E_L), G_A(V_A, E_A))$
    $Placed(W, F) \leftarrow \emptyset$
    $maxL \leftarrow V_L$ with maximum communication links
    $maxA \leftarrow V_A$ with maximum neighbour nodes
    map($maxL$) = $maxA$
    remove $maxL$ from $G_L$ and $maxA$ from $G_A$, add $maxA$ to $Placed(W, F)$
    while $|V_L| < 0$
        $nextL \leftarrow V_L$ with maximum connections with $\forall w_i \in W$
        for all $v_{a,j} \in G_A(V_A, E_A)$ do
            for all $w_i \in Placed(W, F)$ do
                if connection $v_{a,j} \rightarrow w_i$ then
                    $commcost(v_{a,j}) += distance(v_{a,j}, w_i)$
                end if
                if connection $w_i \rightarrow v_{a,j}$ then
                    $commcost(v_{a,j}) += distance(v_{a,j}, w_i)$
                end if
            end for
        end for
        $nextA \leftarrow v_{a,j}$ with minimum $commcost$
        map($nextL$) = $nextA$
        remove $nextL$ from $G_L$ and $nextA$ from $G_A$, add $nextA$ to $Placed(W, F)$
    end while
end function
```
Algorithm 5.2 Algorithm used for physical binding

function PHYSICAL_BINDING($G_L(V_L, E_L), G_A(V_A, E_A)$)
    initialize($G_L(V_L, E_L), G_A(V_A, E_A)$)
    bestbinding ← Placed
    bestcommcost ← netcommcost(Placed)
    for $i = 1$ to $|V_A|$ do
        for $j = i + 1$ to $|V_A|$ do
            $P_{temp}(W_t, F_t) ← Placed(W, F)$ with swapping $w_i$ and $w_j$
            commcost = netcommcost($P_{temp}(W_t, F_t)$)
            if commcost < bestcommcost then
                bestbinding ← Placed
                bestcommcost ← commcost
            end if
        end for
        Placed ← bestbinding
    end for
    return Placed
end function

Algorithm 5.3 Communication cost calculation

function NETCOMMCO flatten($G(V,E)$)
    cost ← 0
    for all $e_i ∈ E$ do
        if congestion at $e_i$ then
            $e_i ← e_i + 1$
        end if
    end for
    return cost
end function
5.3.5. Scheduling

There are two different types of scheduling. The scheduling of independent processes/hyper processes and the scheduling of the processes within a hyper process which need to keep a certain order to satisfy their data dependencies.

5.3.5.1. Scheduling on Execution Unit

The processes running on a processor are decoupled by heartbeat delays. Thus the execution order does not matter by the functional point of view and the order can be chosen in order to optimize heartbeat period. As described in the next section, hyper processes are scheduled separately. To determine the position among the other processes/hyper processes, the execution time of the first process is taken as execution time, and the communication overhead, defined in Section 5.2.1.1 as the communication time of the last process, is used as the communication time.

Figure 5.9 shows three different cases of schedules which show the optimal case for their set of processes. Depending whether the execution time or the communication time is dominant, there are two different ideal schedules. Subfigure a) has no waiting time for the processors. As the last signal cannot be sent parallel to the execution, it must be the one with the shortest execution time. Subfigure b) shows an ideal schedule

![Figure 5.9](image)

**Figure 5.9:** Examples of different optimal schedules: a) shows an ideal schedule for an execution time dominant set of processes, b) shows a communication dominant set of processes, and c) is the case where no ideal schedule can be found.

Figure 5.9 shows three different cases of schedules which show the optimal case for their set of processes. Depending whether the execution time or the communication time is dominant, there are two different ideal schedules. Subfigure a) has no waiting time for the processors. As the last signal cannot be sent parallel to the execution, it must be the one with the shortest execution time. Subfigure b) shows an ideal schedule
5.3. The Mapping Process

if the communication time is dominant. The communication link is always busy and the process with the shortest execution time is scheduled first. These schedules has to be seen as bounds, as it is not always possible for a given set of processes to get an ideal schedule as shown Subfigure c). With this set of processes both the execution and the communication must wait for the other once.

Assuming a number of processes $n$ assigned with a worst-case execution time $WCET_i$ and a communication time $c_i$, then the bounds for the two cases of optimal schedules can be calculated by

$$t_{\tau}^{WCET} = \sum_{i=1}^{n} (WCET_i) + c_{min} \quad (5.5)$$

and

$$t_{\tau}^c = \sum_{i=1}^{n} (c_i) + WCET_{min} \quad (5.6)$$

Whereas $WCET_{min}$ stands for the shortest WCET amongst all processes, and $c_{min}$ the shortest communication time accordingly. The larger of these bounds is the lower bound for the time of the schedule, $t_{\tau}$.

$$t_{HB, min} = \max \left[ t_{\tau}^{WCET}, t_{\tau}^c \right] \quad (5.7)$$

Considering these constraints, the scheduling algorithm must choose the process with the shortest execution time first if the schedule is dominated by the communication and vice versa if the schedule is dominated by the execution time. Thereby is guaranteed that the optimal schedule can be found. The order of the other processes should be chosen that the waiting time is reduced for an execution time dominant schedule, and the gaps between the signals is small for a communication time dominant schedule.

Summing up, there are three optimization goals for the scheduling:

- Communication runs as parallel as possible to the computation,
- The communication overhead of the last process or the execution time of the first process must be as short as possible.

Our heuristic algorithm is described by an execution time dominated schedule, a communication time dominated one works just vice versa. The algorithm starts with the process with the longest communication time, as this one can lead to a lot of waiting time if no process with a fitting execution time can be found. Then the process with the best fitting execution time is added after the first process, and so forth until the process with the shortest communication time is added. Then the remaining processes are added before the first placed in a similar manner. The listing 5.4 shows the pseudo-code of the algorithm for a execution time dominated schedule.
5. System Model and Mapping

Algorithm 5.4 Scheduling of processes on a physical execution unit.

function SCHEDULE EXEC DOMINANT(P)
    \( p_{c,\text{min}} \leftarrow p \) with minimum \( c \) of \( P \)
    \( p_{c,\text{max}} \leftarrow p \) with maximum \( c \) of \( P \)
    move \( p_{c,\text{max}} \) to \( \tau \)
    order \( P \) in incr. order of WCET
    while \( \tau_{\text{back}} \neq p_{c,\text{min}} \) do
        \( c_{\text{last}} \leftarrow c \) of \( \tau_{\text{last}} \)
        \( p_{\text{next}} \leftarrow \) first \( p \) in \( P \) where WCET > \( c_{\text{last}} \)
        if \( p_{\text{next}} = \emptyset \) then
            \( p_{\text{next}} \leftarrow P_{\text{last}} \)
        end if
        move \( p_{\text{next}} \) to back of \( \tau \)
    end while
    order \( P \) in decr. order of \( c \)
    while \( P \neq \emptyset \) do
        WCET_{first} \leftarrow WCET of \( \tau_{\text{first}} \)
        \( p_{\text{next}} \leftarrow \) first \( p \) in \( P \) where \( c < WCET_{first} \)
        if no \( p_{\text{next}} \) found then
            \( p_{\text{next}} \leftarrow P_{\text{last}} \)
        end if
        move \( p_{\text{next}} \) to front of \( \tau \)
    end while
end function
5.3.5.2. Scheduling within Hyper Process

In general the processes within a hyper process must be executed in the order given by the edges of the problem graph to fulfill their data dependencies. In spite of, it is possible to schedule groups of processes within a hyper process independently. Figure 5.10 shows an example where processes $P_1$ and $P_2$ are independent from each other. Both schedules $A$ and $B$ are feasible, because there is a fork ($P_3$) in the part of the problem graph represented by a hyper process. In this case, the same scheduling algorithm presented for independent process can be applied to determine which of the independent processes should run first. Schedule $B$ of the figure would be chosen, as the communication time of $P_2$ is scheduled earlier. In the example schedule $A$ would be chosen as it does not have any waiting time.

![Diagram of Hyper Process Scheduling]

Figure 5.10: Hyper process example that can be scheduled in two different ways.

5.3.5.3. Heartbeat Evaluation

Given a schedule for every execution unit the minimal heartbeat period of the system can be calculated. The base of this calculation is the run time of a process $r_i$ as already described in Section 5.1.3. Hyper processes does not need to be treated specially, as their single processes are part of the schedule. The run time is equal
5. System Model and Mapping

to the larger one of the worst-case execution time of the task WCET\(_i\) and the
communication time of the preceding task \(c_{i-1}\)

\[
  r_i = \max(\text{WCET}_i, c_{i-1})
\]  

(5.8)

The run time \(R\) of the execution unit \(j\) can be calculated by summing up all the run
times of the bound processes, plus the communication time of the last process in the
schedule \(c_N\)

\[
  R_j = \sum_{n=0}^{N} (r_n) + c_N
\]  

(5.9)

As the system cannot run faster than its slowest component, the heartbeat pe-
riod \((t_{HB})\) is equal to the run time of the slowest execution unit among all execution
units \(M\)

\[
  t_{HB} = \max_{m=0}^{M} (R_m)
\]  

(5.10)

In Figure 5.11 the evaluation of the heartbeat for a system with 11 processes on four
execution units is shown. The different run times are given below the execution unit’s
names. Execution unit \(E2\) has the longest run time among all execution units, and
therefore the minimal heartbeat period is 5.5 as the run time \(r_2\).

**Figure 5.11:** Heartbeat evaluation example: The heartbeat period is equal
to the longest run time among all execution units. In this example the heartbeat
period is 5.5.
5.4. Conclusion

By assuming that the processes $P_1$, $P_2$ and $P_3$ scheduled on $E_1$ are combined in a hyper process, i.e. they cannot be bound to different processors, we can observe a lower bound for the heartbeat period. If the given system is mapped to 5 instead of 4 execution units, process $P_6$ can be moved to $E_5$. The new heartbeat period is given by the hyper process on $E_1$ and is therewith 5. Even if the system is mapped to more execution units, the heartbeat cannot be further reduced. This demonstrates why a system designer should take care of the size of processes and hyper processes, to avoid that one of them is dominating the others too much.

5.4. Conclusion

The first step to achieve an efficient mapping is to model the system with the necessary accuracy. For this thesis, the time model of the processes consist of a computational and a communication part which can run in parallel as former occupies a processor and the latter the interconnection network. As the heartbeat delay functionality is provided by the NoC, it is included in the model of the communication. Processes must have the possibility to communicate directly without a heartbeat delay, therefore the hyper process construct has been introduced, which groups together these processes. All together there exist three different communication types on the generated platform.

The usage of the synchronous model of computation gives the mapper high flexibility for the binding and scheduling of the processes. Therefore, the whole process was split into three different phases that allow to optimize the partial problems better. An implementation of the mapping process following these phases is presented in the next chapter.
5. System Model and Mapping
6 Implementation of the Mapper

A basic implementation of the mapping process, called syncmapper has been programmed. It maps a synchronous system to a NoC target platform. This program does not fill the whole gap in the design process between ForSyDe-SystemC and the NoC System Generator as described in Chapter 4. It handles only the mapping: binding and scheduling. Mapping constraints given by limited memory of the nodes and special hardware, such as input and output devices, are not a part of the mapper yet. The usage and the main concepts of the program are described in this chapter. The source code can be downloaded from the ForSyDe webpage[3]. The METIS library has to be downloaded and linked separately.

6.1. Usage of syncmapper

The syncmapper is a small command line tool which allows to map a synchronous application modelled by processes and signals, as described in Section 5.1. This application binds the processes to nodes of a system created by the NoC System Generator and schedules them. It also calculates the shortest possible heartbeat period. The application and the configuration of the target architecture are described in a file with the ending .map (described in Section 6.1.1) and is passed to the mapper as the first command line argument:

    syncmapper system.app

While the mapper runs, it prints information about the different steps of binding and scheduling: building the application graph from the input file, creation of hyper processes, binding to logical execution units (EU), and finally the binding to physical EUs and their scheduling. As the scheduling of the hyper processes is not implemented yet, the user can be prompted to enter a feasible schedule for a hyper process. If this happens, it must considered that the processes within a hyper process are not decoupled and must run in the right order to fulfil the dependencies among the other processes within the same hyper process.
6. Implementation of the Mapper

The result of the mapper is written to two output files which have the same name as the input file but the endings .map and .tim. The former contains the process mapping for the usage as an input to the NoC System Generator. The latter contains detailed information, especially about the timing of the system. They will be explained in detail in Section 6.1.2.

6.1.1. System Description File

The .map-file used as the input to the mapper consists of three parts: the target architecture description, the processes of the system, and the signals connecting them. An example system and the corresponding description file are shown in Figure 6.1. The system consists of four processes which are connected by four signals. The signals containing a heartbeat delay are marked with the \( \Delta \) symbol.

```
#config
mesh2d
2 2 0
3
#processes
1 20 proc1.cpp
2 10 proc2.cpp
3 10 proc3.cpp
4 20 proc4.cpp
#signals
1 1 2 10 0 0
2 1 3 10 1 1
3 2 4 20 1 1
4 3 4 20 1 1
```

![Figure 6.1: Syncmapper System Description File example: The example consists of 4 processes connected by 4 signals, where all the signals have a heartbeat delay of 1, except signal '1'. The application is mapped to 3 cores of a 2x2 2D-mesh network.](image)

Some comments can be added in the beginning of the file. The line `#config` indicates the start of the target architecture configuration. The first line indicates the network type, the second line contains 3 parameters to configure the network. The last line indicates to how many nodes of the network the application will be mapped. In the shown example the system is mapped to a 2D-mesh network, which is the only network type currently implemented. In the case of the 2D-mesh network the first two parameters are the X-, respectively Y-dimensions of the network. The example
6.1. Usage of syncmapper

is thus a 2x2 2D-mesh network and the application is mapped to 3 of the 4 available nodes.

After the target configuration, the process section begins with the #processes line. The parameters of a process are separated by simple spaces. The first column is the process id, the second column is the worst-case-execution time in clock cycles, and the last column the file with the process code. If we have a closer look to the first process with the id 1. Its worst-case execution time is 20 cycles and the corresponding code file is proc1.cpp.

The line #signals indicates the start of the last section with the signals descriptions. The parameters are also separated by spaces. The first column is the signal id. The following two columns indicate the source and destination processes by their ids. The third column is the transmitted data volume in bytes. The last two parameters specify the heartbeat delay of the system. The fifth column indicates the delay mode, which is in the current implementation just '0' for no delay and '1' for delay necessary. The sixth column is the number of delays assigned to the signal. In the example system, the first two signals are used to send 10 bytes each from \( P_1 \). The first signal \( S_1 \) transmits data to \( P_2 \) without any delay, and signal 2 \( S_2 \) sends with a delay of 1 heartbeat to \( P_3 \).

6.1.2. Output Data

As already mentioned, the syncmapper outputs the results to two files. The first file contains the mapping of the system, and the second one contains statistics about the mapping and its timing properties.

An example of a mapping file (file ending .map) can be seen in Listing 6.1. The format is similar to the hardware description file used by the NoC System Generator. But, it cannot be copied one-to-one as hyper processes are not supported yet by the NoC System Generator.

Listing 6.1: Output file format (.map) of the syncmapper

```
<parameter name="GlobalSync" value="2745"/>
<hprocess name="p1" moc="Synchronous" node="0"
  sources="{}" targets="{p2,p3}" files="{proc1.cpp,proc2.cpp}"/>
<process name="p3" moc="Synchronous" node="1"
  sources="{p1,p2}" targets="{p4}" files="{proc4.cpp}"/>
<process name="p4" moc="Synchronous" node="1"
  sources="{p3}" targets="{}" files="{proc5.cpp}"/>
<process name="p2" moc="Synchronous" node="2"
  sources="{p1}" targets="{p3}" files="{proc3.cpp}"/>
```

The first line indicates the calculated heartbeat period in clock cycles. If possible congestion is detected in the network, the value indicates 'CONGESTION', as no
6. Implementation of the Mapper

heartbeat can be calculated in this case. To use this value in the hardware description file of the NoC System Generator, it must be converted to the heartbeat frequency. The following lines list all the processes and hyper processes in the order of their execution. For example, Process $P_3$ and process $P_4$ run both on node '1', as $P_3$ is mentioned first it must be executed first to avoid heartbeat violations. Of the processes within hyper process $P_1$, proc1.cpp must be executed before proc2.cpp to ensure the fulfilment of their dependencies.

The second output file has the ending .tim. Its purpose is to give additional information for the analysis of the quality of the mapping and to detect problems. On the one hand, it shows also the binding and scheduling of the processes and on the other hand it contains also detailed timing properties on every level of the mapping. Some bounds are also indicated within the file. The format is created in order to be readable by humans.

6.2. Program Structure

The syncmapper is implemented in C++, and the class structure follows closely the system model and mapping process described in Chapter 5. The whole design is developed modularly to allow changes of mapping algorithms and network topologies to be made easily. Many variables use specific types which allow to change these types easily if necessary.

Figure 6.2 shows the structure of the classes that build up the mapping application. These classes are explained in detail in the following sections. Please note, that not all the relations between the classes are drawn, some more references exist to simplify the programming. The central objects of the mapping are called layers. These are instantiated by the main() function and model the mapping process. The AppGraph class models the application and builds the base for the whole process. On top of the application layer there is a layer for each phase of the mapping; LayerHP class for the creation of hyper processes, a LayerLog class for the partitioning to logical execution units, and the LayerPhy class for the mapping to the physical network. These layers build up on the lower layers but to compare different mappings, different higher level layers can exist for one of the lower levels. However, the higher levels do not update automatically on changes of the lower levels.

Different libraries were used to simplify the programming. Firstly, the syncmapper makes heavy use of STL containers. To simplify the iterating over the container elements the Boost Foreach Library [1] is used. The Boost Graph Library [2] provides classes to construct graphs and is used to model both the application graph and the network topology. For the partitioning to logical EUs, METIS [4] a graph partitioning library is used. More about this library can be found in Section 6.3.2.
6.2. Program Structure

![Class Structure Diagram]

**Figure 6.2.** The class structure of the syncmapper program, to keep the diagram clear only the direct connections between the classes are drawn.
6. Implementation of the Mapper

6.2.1. Application model class: AppGraph

Equivalent to the application model described in Section 5.1, the AppGraph class models the application. Even though it can be seen as the lowest layer, it is not derived from the Layer class (Section 6.2.2) as the other layers are. The AppGraph object describes the communication and dependencies between the processes. The application is modelled as a directed graph by using the Boost Graph Library. The Process objects are used as vertices of the graph and the Signal objects are assigned to the edges. The Process and Signal classes store the properties of processes and signals, such as WCET, communication volume and delay. Additionally, they have an internal id which is used to recognize them in the different classes. The application graph is created by reading the system description file.

6.2.2. Mapping Layer class: Layer

On top of the application model, different layers representing the different phases of the mapping process are added. The Layer class is abstract and three different derived classes exist for the three different phases. The central function is map(), that implements the mapping algorithm from the lower layer to the current layer.

A layer consists of execution unit objects, described in the next section, which represent the mapping of the execution units (EU) from the lower layer to this layer. The links between the EUs are not modelled separately, and the layer derives them from the application graph structure which is referenced by the layer class.

As the timing properties of the processes and signals, such as running time, communication time, etc. depend on the mapping of the process to a certain EU; the functions to calculate these properties are also a part of the Layer class. The used models are described in Section 6.3.3. The parameters are stored locally in the timefactors.hpp header to make it easy to change them.

The physical layer (LayerPhy class) also provides the calculation of the heartbeat period and the output of the mapping and timing properties to the output files as described in Section 6.1.2.

6.2.3. Execution Unit class: ExecUnit

Another abstract class is used to model the execution units as they are defined in Section 5.2.1. For each layer a derivation of this class exist. The layers contain containers that save the EUs of the lower layer which are mapped to this. Each EU also contains a schedule which is modelled by the Schedule class. A schedule is simply the execution order of processes, the class provides functions to create a
6.3. Algorithms

Some of the algorithms which are not a central part of the mapping process, but more implementation specific, are described in the following sections. For the central mapping algorithms see Chapter 5.

6.3.1. Hyper Process handling

Just the minimum necessary to handle hyper processes is implemented so far. All processes without any heartbeat delay in between are combined to a single hyper processes. This is done by Algorithm 6.1.

Although the scheduling of hyper processes needs closer evaluation of the code structure (see Section 5.3.5.2), this is not implemented yet. If there exist hyper processes in a model, the user is prompted to enter the schedule to the console.
6. Implementation of the Mapper

**Algorithm 6.1 Communication cost calculation**

```plaintext
function CREATE_HYPERPROCS(G_P(V_P,E_P))
  for all \( P_i \in V_P \) do
    \( HPlist \leftarrow \emptyset \)
    \( NeighbList \leftarrow \) all neighbour nodes of \( P_i \)
    for all \( P_N \in NeighbList \) do
      if delay of signal between \( P_N \) and \( P_i \) = 0 then
        if \( P_N \) is in hyper process then
          add hyper process of neighbour to \( HPlist \)
        end if
      end if
    end for
    Remove all duplicates from \( HPlist \)
    if \(|HPlist| > 0\) then
      Combine all hyper processes with \( P_i \) to one
    else
      Create new hyper process with \( P_i \)
    end if
  end for
end function
```

### 6.3.2. METIS - Graph Partitioning Library

For the graph partitioning by the Kernighan-Lin algorithm (Section 5.3.3.1), as used by the logical mapping phase, a library called METIS is used. It is based on a paper written by Karypis and Kumar [23] which describes a fast multi-level algorithm for graph partitioning. From their findings they developed the METIS library. Multi-level means that the vertices of a large graph are firstly collapsed to groups of adjacent vertices, and then when the graph is small enough to allow fast execution, the Kernighan-Lin algorithm (Section 5.3.3.1) is applied to it. Then the graph is restored by projecting the partitions to the original graph. As the application graphs used for the mapping usually have small sizes of a few tens of vertices, the collapsing and projecting are not executed and the Kernighan-Lin algorithm is directly applied to them.

More details and the library for download can be found on its webpage [4]. To use the library, it must compiled separately and linked as a static library to the syncmapper.

The library provides two different partitioning methods: multilevel recursive bisectioning and multilevel k-way partitioning. We use the latter one as it provides more options to fine-tune the algorithms.
Weights can be assigned to vertices and edges. For the syncmapper, the vertices represent the hyper processes created by the previous step in the mapping process. Their weight is the sum of runtime of the processes of the hyper process. The METIS library would allow to assign other weights to the vertices and maximum values for these weights for each partition. In the future, this can be used to model the memory requirements of processes and memory availability of execution units.

The communication volumes are assigned to the weights of the edges. But, this has no effect as the METIS library is set to optimize the maximum degree of the nodes by setting parameter `METIS_OPTION_MINCONN` to ‘1’. This means that the METIS library tries to minimize the maximum number of connections a partition has with the other partitions. This optimization strategy was chosen to avoid network congestion. The `METIS_OPTION_UFACTOR` parameter specifies the imbalance between the parameters, e.g. how much the size of the generated partitions can differ. It is set to a relatively high value of 100 (default = 30). Thereby, the priority of the algorithm is set more to avoid network congestion than of achieving equal sized partitions.

The initial partitioning is done randomly by setting parameter `METIS_OPTION_IPTYPE` to `METIS_IPTYPE_RANDOM`. Together with the setting that 100 iterations (`METIS_OPTION_NCUTS = 100`) should be executed, this increases the chance of finding the optimal portioning. This is supported by detecting first the connected components of a graph (parts that have no connections with other parts) by setting the parameter `METIS_OPTION_CCORDER` to ‘1’.

### 6.3.3. Models for Communication Time and Execution Time Evaluation

For the evaluation of the schedule, to calculate the communication cost for the mapping algorithms, and for the calculation of the heartbeat frequency and other statistics, it is necessary to calculate the execution time and the communication time as accurate as possible. The evaluation of the communication and execution time depends not only on the properties of the processes and signals but also on their mapping. Therefore, different models exists for the different layers of the syncmapper. The unit for the time calculation is clock cycles.

The values that specifies the models are defined as pre-processor macros in a separate header file `timefactors.hpp`, which makes it easy to change them. The parameters which are layer specific will be described in the following Sections 6.3.3.1–6.3.3.3, but there are also two parameters used to describe the architecture. The flit width in bits of the target network is defined by `FLIT_WIDTH` and is set to 32 bits for the actual implementation of the NoC platform. The number of flits in a packet, which does not carry data of the message, are specified by the packet overhead `PACKET_OH`
6. Implementation of the Mapper

parameter. This is 2 for the current NoC: one for the header flit and one for the
global clock flit.

6.3.3.1. Hyper Process Layer

As the network is not mapped to a network in this phase, there is no communication
between nodes (type I). The communication within the same node via RNI (type II)
is going to the network and some communication time $c$ can be added to all signals
between hyper processes. The model uses a number of cycles that are added per packet
\texttt{HP\_NET\_PACKETCOST} and a factor \texttt{HP\_NET\_VOLCOST} that makes the communication
time dependent on the volume $v$ in flits.

$$c = (v + \text{PACKET\_OH}) \times \text{HP\_NET\_VOLCOST} + \text{HP\_NET\_PACKETCOST}$$  \hspace{1cm} (6.1)

Communication within a hyper process (type III) does not involve the network and
the communication time is zero. Thus, it represents the time of the communication
parallel to the execution time. But as the additional memory accesses and the context
switching between the processes take some time, additional clock cycles are added to
the execution time. Taking the set of all signals $|S|$ of a process. For every incoming
signal from the same hyper process an additional execution time $\epsilon_{HP, in}$ is calculated,
and the same applies also to the outgoing signals to the same hyper process $\epsilon_{HP, out}$.
The macros \texttt{HP\_EXECCOST\_IN} and \texttt{HP\_EXECCOST\_OUT} define the additional cycles for
every flit of the communication volume $v_i$.

$$\epsilon_{HP, in} = \sum_{i=0}^{S} (v_i + \text{PACKET\_OH}) \times \begin{cases} \text{HP\_EXECCOST\_IN} & \text{type III, incoming} \\ 0 & \text{otherwise} \end{cases}$$ \hspace{1cm} (6.2)

$$\epsilon_{HP, out} = \sum_{i=0}^{S} (v_i + \text{PACKET\_OH}) \times \begin{cases} \text{HP\_EXECCOST\_OUT} & \text{type III, outgoing} \\ 0 & \text{otherwise} \end{cases}$$ \hspace{1cm} (6.3)

These two values are added to the original execution time $\epsilon$ of the process together
with a general cost for the process belonging to a hyper process — defined by the
macro \texttt{HP\_EXECCOST\_GENERAL}.

$$\epsilon_{HP} = \epsilon + \epsilon_{HP, in} + \epsilon_{HP, out} + \text{HP\_EXECCOST\_GENERAL}$$ \hspace{1cm} (6.4)

6.3.3.2. Logical Layer

As the communication and execution times on the hyper process layer are not used
yet, they are not implemented.
6.3.3.3. Physical Layer

After mapping the processes to the nodes of the network the communication time $c$ of a signal between two nodes can be calculated. The used model takes into account the communication volume $v$ of the signal, the distance between the nodes $dist(S)$ and a general cost per packet.

\[
\begin{align*}
    c_v &= (v + \text{PACKET}_\text{OH}) \times \text{PHY}_\text{NET}_\text{VOLCOST} \\
    c_d &= dist(S) \times \text{PHY}_\text{NET}_\text{DISTCOST} \\
    c &= c_v + c_d + \text{PHY}_\text{NET}_\text{PACKETCOST}
\end{align*}
\]  

(6.5)

The volume (in flits) and the distance are multiplied by their corresponding factors $\text{PHY}_\text{NET}_\text{VOLCOST}$ and $\text{PHY}_\text{NET}_\text{DISTCOST}$. The general cost per packet reflects the composition and decomposition of the packet in the RNI, the factor is called $\text{PHY}_\text{NET}_\text{PACKETCOST}$.

The models for type II and type III communication are the same as described for the hyper process layer in Section 6.3.3.1, just the names of the factors start with PHY instead of HP, e.g. $\text{HP}_\text{NET}_\text{PACKETCOST}$ becomes $\text{PHY}_\text{NET}_\text{PACKETCOST}$ and $\text{HP}_\text{EXECCOST}_\text{IN}$ becomes $\text{PHY}_\text{EXECCOST}_\text{IN}$.

6.4. Conclusion

Although the implemented mapper does not allow yet the fully automatic system creation as described in Chapter 4, it shows that the derived mapping process is implementable. So far the input file uses an own format as input, but this can also be directly derived by parsing the system structure XML and reading out the process structure. On the back-end side the mapper creates a file which is similar to the input file of the NoC System Generator, but it also contains the description of hyper processes which are not supported yet by the NoC System Generator.

The different steps of the mapping process are modelled by layers. Together with the network topologies which are represented by separate objects, this design allows to adapt easily changes and additions to the mapping process in the future.
6. Implementation of the Mapper
Example Mappings and Results

To test the implementation and get a first impression of the algorithms’ efficiency some tests on synthetic examples were carried out. Through these test some limitations and problems of the current were detected.

7.1. Mapping Examples

Due to the lack of real examples which are large enough, and have their WCET analysed, synthetic application graphs were defined to test the mapping process. A first set of application graphs simulate different structures of applications, these will be called the structure examples. A second set of application graphs, the size examples, simulate applications with different process granularities. The third example is inspired by the video object plane decoder of the MPEG-4 decoder described in [46]. This example demonstrates that a real application can be mapped and what the effect of the hyper process creation is. All the details of the used application graphs can be found in Appendix A.

7.1.1. Structure Examples

These examples were created to test the influence, that different application structures have to the mapping. To keep them comparable, all structures consist of the same 20 processes. Although, the number of signals between the processes is different for the different structures, the total communication volume is the same for all applications. Figure 7.1 shows the three used structures symbolically. These are: (a) The chain where every process receives data from one process and sends the processed data to another one. (b) A Parallel structure where the data is split to several parallel chains and combined again in the end. (c) A tree structure where the execution is split up, but not combined in the end.
7. Example Mappings and Results

The first used structure is a simple chain of processes. It is the base structure of many real applications that take an input apply different processing steps and output the result. The different processes itself can have however a more complex structure. So could the parallel structure in sub-figure b), be simplified to a chain by combining the functionality of the three processes in the middle to one process. A chain is simple to map without congestion, because the number of connections between the processes and the degree of the nodes are minimal. The application graph is named chain20.

With the given situation where the effect of deflection is not easily predictable, it is important to avoid congestion in the network. It is not possible to ensure that no deflection occurs, if the number of incoming signals to an execution unit exceed the number of available incoming network links to a node. In a 2D-mesh network the maximum input links is four. The parallel5 application structure checks the capability of the mapper to deal with this problem. The execution is split up to five different paths and at the end they are merged together. In real applications, this is a typical structure if data parallelism exist.

The last structure is a the tree structure, this one also splits up the execution to five paths, but does not join them in the end. An example for such a system could be a multimedia system, that reads a video file by one process and the processing of the video and audio is performed by different execution paths.

7.1.2. Size Examples

These examples shows the effect of the granularity of an application to the result of the mapping. The expected result is that applications consisting of more processes could be mapped more efficiently. To make this comparison, the chain example described in the previous section is used. Given that, a chain is easy to map, the size should be the main effect on the results.
Three different versions of the chain exist: The chain20 structure is the basic one with the same 20 processes as the other structure examples. The chain10 version contains only the first 10 processes and the chain40 version consists of the duplicated process set.

### 7.1.3. MPEG-4 Decoder Inspired Example

As an example closer to a real application, the MPEG-4 video object plane decoder as described in [46] is used. The presented block diagram consist of 18 processes which build a structure with several parallel paths and loops. The communication bandwidths between the processes are given in the paper. They are used in this example as the communication volumes. Unfortunately, no indication about the computational costs is made. We estimated the WCET by the expected complexity of the blocks, i.e. blocks such as de-multiplexers and up-samplers got lower WCETs assigned than blocks with complex signal processing algorithms such as the inverse discrete cosine transform. The result is an application structure of processes with different execution times and a big range of different communication volumes.

Two versions of the MPEG-4 decoder example exist. The first one mpeg4, adds a delay in between all the processes. This allows the mapper the highest flexibility for the mapping. The mpeg4-loop version groups the loops by removing all the delays except the ones after memory blocks. The mapper creates hyper processes for those processes without a delay and its flexibility becomes more limited.

### 7.2. Experiment Setup

The results were generated by applying the described examples to the syncmapper from repository revision 89. For the structure and size examples a 2D-mesh network with the size of 4x4 was used. The MPEG-4 decoder example was mapped to a 3x3 2D-mesh. Tests have shown that the resulting heartbeat does usually not depend on the size of the chosen network, but only on the number of occupied processors. Therefore, the result of mapping to four processors on a 4x4 network can also be used as the result of mapping to all four processors of a 2x2 network.

The parameters used by the timing models, are shown in Table 7.1. The same values are applied to the hyper process layer as well as to the physical layer. The communication model is based upon the number of cycles of the FSMs of the NoC System Generator’s network nodes. The injection of a flit happens every 16 cycles and the processing of a flit takes 4 cycles at each node. The congestion cost is set to a very high value, as its consequence is non-predictability. If a process is executed within a hyper process additional 30 clock cycles are added to its WCET.
7. Example Mappings and Results

<table>
<thead>
<tr>
<th>Hyper Process Layer</th>
<th>Physical Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP_DEL_PACKETCOST</td>
<td>PHYS_DEL_PACKETCOST</td>
</tr>
<tr>
<td>HP_DEL_VOLCOST</td>
<td>PHYS_DEL_VOLCOST</td>
</tr>
<tr>
<td>HP_EXECCOST_IN</td>
<td>PHYS_NET_PACKETCOST</td>
</tr>
<tr>
<td>HP_EXECCOST_OUT</td>
<td>PHYS_NET_VOLCOST</td>
</tr>
<tr>
<td>HP_EXECCOST_GENERAL</td>
<td>PHYS_NET_DISTCOST</td>
</tr>
</tbody>
</table>

Table 7.1: The timing model parameters used for the evaluation of the results. The communication parameters base on the actual network implementation. The congestion penalty is set to a very high value to detect it. For the execution of a hyper process additional 30 clock cycles are assumed.

7.3. Mapping Results

To generate the results, the different applications were mapped to all possible numbers of execution units within the network. The mappings which cause congestion in the network are marked by con and are not considered during further analysis of the results. To compare the results of the different application graph examples, the speedup and the efficiency were calculated for each mapping. The speedup of a mapping is defined as the ratio between the heartbeat period of the application mapped to a single processor $t_{HB,1}$ and the heartbeat period of the application mapped to $p$ processors $t_{HB,p}$. It is calculated by the following formula:

$$S_p = \frac{t_{HB,1}}{t_{HB,p}}$$ (7.1)

The efficiency is the speedup normalized to the number of processors $p$.

$$E_p = \frac{t_{HB,1}}{p \times t_{HB,p}}$$ (7.2)

An efficiency of 1 means ideal mapping.

7.3.1. Mapping Results of the Structure Examples

The heartbeat periods of the different structure example mappings are presented in Table 7.2, the corresponding speedups are shown in Figure 7.2 and the efficiency in Figure 7.3. The curves of the speedup and the efficiency of the chain20 and the tree applications are similar. The efficiency is more than 0.8 for mappings up to 6 processors for the chain20 example, and up to 5 processors for the tree
7.3. Mapping Results

example and decreases then. This is also visible in the speedup graph where the speedup approaches a value around 8 for a larger number of processors. For this two applications it seems that the structure has just a minor effect on the efficiency of the mapping.

As expected, the mapper shows difficulties to map the parallel5 application without congestion. The used algorithms found only a solution up to 3 processors. A more detailed analysis of this problem showed that mappings without congestion exist if the processes are split in a proper way to the processors. The problem arises from the METIS library used for the logical mapping. It tries to minimize the total number of connections between the processes. But, congestion can only be caused by incoming signals to an execution unit. Within the generated network, the outgoing signals of one execution unit cannot cause congestion, as all the signals are sent serially, and therefore cannot interfere with each other. Thus, the algorithm should optimize the number of incoming signals to the execution units.

<table>
<thead>
<tr>
<th>processors</th>
<th>Heartbeat period [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>chain20</td>
</tr>
<tr>
<td>1</td>
<td>6,470</td>
</tr>
<tr>
<td>2</td>
<td>3,286</td>
</tr>
<tr>
<td>3</td>
<td>2,226</td>
</tr>
<tr>
<td>4</td>
<td>1,770</td>
</tr>
<tr>
<td>5</td>
<td>1,620</td>
</tr>
<tr>
<td>6</td>
<td>1,182</td>
</tr>
<tr>
<td>7</td>
<td>1,182</td>
</tr>
<tr>
<td>8</td>
<td>1,172</td>
</tr>
<tr>
<td>9</td>
<td>906</td>
</tr>
<tr>
<td>10</td>
<td>876</td>
</tr>
<tr>
<td>11</td>
<td>876</td>
</tr>
<tr>
<td>12</td>
<td>814</td>
</tr>
<tr>
<td>13</td>
<td>818</td>
</tr>
<tr>
<td>14</td>
<td>826</td>
</tr>
<tr>
<td>15</td>
<td>834</td>
</tr>
<tr>
<td>16</td>
<td>826</td>
</tr>
</tbody>
</table>

*acongestion*

Table 7.2: The results of mapping the structure examples to different numbers of processors on a 4x4 2D-mesh NoC. The parallel5 structure cannot be mapped without congestion to more than 3 processors.
7. Example Mappings and Results

Figure 7.2.: Speedup of the structure examples: Both the chain20 and the parallel5 examples show a similar trend and approach a speedup of 8. The tree example can only be mapped up to 3 processors, and is not visible as it has similar values as the other examples.

Figure 7.3.: Efficiency of the structure examples: As by the speedup the chain20 and parallel5 structures show a similar trend.
7.3.2. Mapping Results of the Size Examples

The comparison between different application sizes shows the expected result, that the applications split into more processes can be mapped in a more efficient way. The results are presented in Table 7.3, Figure 7.4 and Figure 7.5.

For a lower number of processors the speedup rises for the two larger applications chain20 and chain40 with a similar slope, only chain10 shows worse results from a low number of processors on. But from four processors on, chain20 shows also a lower speedup than chain40. The efficiency is more than 0.8 for the mapping to 3 processors for the chain10 application, 6 processors for the chain20 application, and 10 processors for the chain40 application.

The highest speedups achieved are 5.56 for chain10 which is also constant by mapping it to more than 10 processors. The cause of this is trivial, as from 10 processors on only one process runs on each processor. The maximum speedup is 7.95 for chain20, and 11.2 for chain40. If we have a look to the speedup (Figure 7.4), we clearly see some steps in the lines, for example between 12 and 13 processors for the chain40 example, and between 8 processors and 9 processors for the chain20 example. These steps are surrounded by flat areas, e.g. 6-8 processors for the chain20 application. All these effects are caused by the granularity of the applications’ processes. For example, if it is given that two processes two processes dominates the heartbeat period and cannot be split because there are not enough processors. There will be no change in the heartbeat period until enough processors are available to split them.

7.3.3. Mapping Results of the MPEG-4 Decoder

The calculated heartbeat periods for the mpeg4 and mpeg4-loop applications can be found in Table 7.4. The decoder example without the loop considerations mpeg4 showed some congestion if mapped to 7 processors. In the speedup graph (Figure 7.6) both applications have similar results up to 6 processors. Then the speedup value for the mpeg4-loop application stagnates. This comes from the size of the processes which must be combined to a hyper processes. As soon this hyper process is mapped with no other processes on the same processor, no improvement can be achieved by mapping to more processors. It is unexpected, that the result of the mpeg4-loop application is comparable or even slightly better than the mpeg4 application, although the mapper’s flexibility is reduced by the hyper processes.

For both applications, the efficiency drops below 0.8 for 5 processors. But the efficiency for the mpeg stays quite constant up to 9 processors.
### 7. Example Mappings and Results

<table>
<thead>
<tr>
<th>processors</th>
<th>Heartbeat period [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>chain10</td>
</tr>
<tr>
<td>1</td>
<td>3,190</td>
</tr>
<tr>
<td>2</td>
<td>1,670</td>
</tr>
<tr>
<td>3</td>
<td>1,182</td>
</tr>
<tr>
<td>4</td>
<td>1,060</td>
</tr>
<tr>
<td>5</td>
<td>876</td>
</tr>
<tr>
<td>6</td>
<td>814</td>
</tr>
<tr>
<td>7</td>
<td>826</td>
</tr>
<tr>
<td>8</td>
<td>822</td>
</tr>
<tr>
<td>9</td>
<td>590</td>
</tr>
<tr>
<td>10</td>
<td>574</td>
</tr>
<tr>
<td>11</td>
<td>574</td>
</tr>
<tr>
<td>12</td>
<td>574</td>
</tr>
<tr>
<td>13</td>
<td>574</td>
</tr>
<tr>
<td>14</td>
<td>574</td>
</tr>
<tr>
<td>15</td>
<td>574</td>
</tr>
<tr>
<td>16</td>
<td>574</td>
</tr>
</tbody>
</table>

Table 7.3: The results of mapping the size examples to different numbers of processors on a 4x4 2D-mesh NoC.

<table>
<thead>
<tr>
<th>processors</th>
<th>Heartbeat period [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mpeg4</td>
</tr>
<tr>
<td>1</td>
<td>121,919</td>
</tr>
<tr>
<td>2</td>
<td>64,537</td>
</tr>
<tr>
<td>3</td>
<td>44,039</td>
</tr>
<tr>
<td>4</td>
<td>35,754</td>
</tr>
<tr>
<td>5</td>
<td>34,758</td>
</tr>
<tr>
<td>6</td>
<td>28,466</td>
</tr>
<tr>
<td>7</td>
<td>con*</td>
</tr>
<tr>
<td>8</td>
<td>19,741</td>
</tr>
<tr>
<td>9</td>
<td>19,745</td>
</tr>
</tbody>
</table>

*congestion

Table 7.4: The results of mapping the MPEG-4 decoder examples to different numbers of processors on a 4x4 2D-mesh NoC. The mpeg4 example can not be mapped without congestion to 7 processors.
7.3. Mapping Results

**Figure 7.4.:** Speedup of the size examples: The achieved speedup is higher for the applications consisting of more processes.

**Figure 7.5.:** Efficiency of the size examples: Up to 3 processors the chain20 and the chain40 examples show a similar efficiency. Then chain40 with more processes shows the better result.
Figure 7.6.: Speedup of the mpeg4 examples: The speedup of the mpeg4-loop application does not increase for more than 7 processors as the biggest hyper process mapped to one processor is dominating the execution times.

Figure 7.7.: Efficiency of the mpeg4 examples.
7.4. Conclusion of Results

The most important condition to achieve an efficient mapping is the size of the processes, respectively the hyper processes as they are handled like large processes by the mapping steps after the hyper process creation. As soon one process dominates the heartbeat, no better results can be achieved by adding more processors. Therefore, a goal during the system modelling must be to split up the application as much as possible.

To get an idea of the efficiency that can be achieved for a given application, the different test applications were analysed, and the number of processors which result in a certain minimum efficiency are listed in Table 7.5. The analysis was done for the efficiencies of 0.7, 0.8 and 0.9. The number of processes necessary per processor to achieve an efficiency of at least 0.9 is 4.51 in average. To reach an efficiency of at least 0.8, the application granularity should allow in average 3.89 processes per processor, and for an efficiency of at least 0.7, in average 2.38 processors per processor are necessary.

<table>
<thead>
<tr>
<th></th>
<th>Efficiency &gt; 0.9</th>
<th>Efficiency &gt; 0.8</th>
<th>Efficiency &gt; 0.7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>processes</td>
<td>processes</td>
<td>processes</td>
</tr>
<tr>
<td>chain10</td>
<td>10</td>
<td>3</td>
<td>3.3</td>
</tr>
<tr>
<td>chain20</td>
<td>20</td>
<td>6</td>
<td>3.3</td>
</tr>
<tr>
<td>chain40</td>
<td>40</td>
<td>6</td>
<td>6.7</td>
</tr>
<tr>
<td>tree</td>
<td>20</td>
<td>5</td>
<td>4.0</td>
</tr>
<tr>
<td>mpeg4</td>
<td>18</td>
<td>3</td>
<td>6.0</td>
</tr>
<tr>
<td>mpeg4-loop</td>
<td>15</td>
<td>4</td>
<td>3.8</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td>4.51</td>
</tr>
<tr>
<td>Std Dev</td>
<td></td>
<td></td>
<td>1.45</td>
</tr>
</tbody>
</table>

*Table 7.5.*: Number of processors per processor which are necessary to achieve a certain efficiency. The smaller the number of processors/processor the better.
8 Conclusion and Future Work

This chapter summarizes the outcome of the thesis. In the second part there is an outlook to future research topics and improvements that can be made to ForSyDe and the NoC System Generator.

8.1. Conclusion

Within this thesis a process for mapping a synchronous application to a time-predictable platform was successfully developed. The goal of finding a mapping algorithm was achieved, moreover a first implementation was realized and some results were generated.

The following steps were carried out:

- a theoretical background study on ForSyDe and the NoC System Generator,
- the properties of the underlying platform were analysed,
- a process for the mapping was proposed,
- a first implementation was programmed,
- some problems of the mapper and the actual were identified.

The background study was pursued quite extensively, as it is important to detect all issues, that could have an effect on the predictability of the system. As the NoC System Generator was not well documented so far, the detailed chapter about it, should also help as a starting point for future projects.

By implementing the mapping process and applying it to some synthetic examples, it could be shown that feasible mappings can be created. Depending on the application, the algorithms are not always successful by finding a solution to avoid network congestion, even if such a solution exist. The main problem could be located in the METIS library that just takes into account the connections between the processes in
general and not the number of incoming edges, which are relevant for the network congestion.

As the system should ensure time-predictability the main goal must be: the avoidance of network congestion, as it makes the network behaviour unpredictable. Nevertheless high utilisation of the cores is also interesting. The results of the mapping of the synthetic examples give some idea of what could be achieved by the current mapping algorithms. So, a mapping efficiency of 0.8 is realistic if the number of processes of an application is four times larger than the number of cores it is mapped to. Given that, the granularity of the processes’ run times is important, the designer must be aware of that. An application must be therefore not only be split in as many processes as possible, single processes with a high run time must be also avoided.

The analysis of the NoC platform and the implementation showed that the behaviour of the network is difficult to predict if congestion occurs on a network link. To achieve the goal of time-predictability, it is necessary to solve these issues. There are two different paths that can be followed. Firstly, the congestion can be avoided, e.g. with more advanced routing or mapping algorithms that achieve better results. The problem with this path is, that it is not assured that a mapping to a specific number of cores without network congestion can be found for every application. Secondly, a possibility to determine the effect of congestion to the communication time in the NoC can be found. Some ideas to solve this problem are presented in the future work (Section 8.2).

This thesis is the first step to bring the ForSyDe design methodology and the hardware platform of the NoC System Generator together. Few more work has to be done to reach the goal of developing a complete process, as proposed in Chapter 4, that allows the design of predictable embedded systems. The next section present some ideas to finally reach this goal.

8.2. Future Work

As already mentioned in the conclusion, the main issue of the actual network is the missing predictability of the network behaviour in the case of network congestion. This must be solved to achieve the goal of a time-predictable system. Congestion avoidance could be realized by improving the mapping algorithms. One idea is to schedule the communication on the links. So, if it is guaranteed that one process finishes the communication on a link before a certain time, another process of a different node can send the data on this link afterwards. But, it must be also assured that the second process does not start the communication before that time. Another idea to increase the mapper’s flexibility is to introduce a combined X-Y and Y-X routing. Every signal would be mapped to one of these two possible paths. Although,
these ideas reduce congestion, they do not ensure, that it is possible to find a solution without congestion for any application.

The second way to go to reach time-predictability is to change the routing in the NoC. The deflection routing could be kept, but must be adapted that a upper bound for the arrival time of the flits can be calculated. This upper bound should also be tight enough, that the heartbeat period does not need to be chosen unnecessarily long. Else, it will be necessary to apply another routing mechanism that ensures time-predictability, e.g. some time-division-multiplex routing as described by Millberg et al. in [31].

The results produced in this thesis show the feasibility of the mapping algorithms and their efficiency. More detailed research can be conducted, to show how close the found mappings are to the best possible mapping. According to the results, the algorithms could be tuned. One problem that was already discovered is that the METIS library just optimize the total number of signals between the processes but does not consider that signals coming from the same node cannot cause congestion. The Kernighan-Lin algorithm could be newly implemented with optimizing the number of incoming edges to a node.

So far, the steps of the mapping process are executed just once. Iterative execution could help to reduce the number of congesting network links. By analysing the conditions that cause the network congestion, probably the congestion could be removed. A similar idea is to map the application first to all possible numbers of cores on a given topology and choose the best solution out of the different mappings. If the heartbeat period is short enough, even though all cores are occupied, it is maybe better to map to a lower number of cores to avoid congestion. As the results show, in some cases even the heartbeat period is shorter if the application is mapped to a lower number of cores.

One area that could be researched in more details are the issues arising from the necessity of delays between the processes of the application. For example; if a delay is included in one of two parallel paths, then a delay must also be included in the other parallel path. One solution would be to apply re-timing as it is used for hardware design. Another idea is to give tags to the signals which show on which other signals they depend. On the one hand this would allow to decouple the processes as much as possible before the mapping. On the other hand, the mapper could use this new freedom to optimize the mapping. We propose to research another technique used in digital design: multi-cycle paths. A multi-cycle path is used for long combinatorial logic blocks, which can not be pipelined, in order to decrease the clock cycle this combinatorial block gets two cycles time to finish. This could also be applied to the NoC System Generator platform to reduce the heartbeat period to a shorter time than the shortest hyper process. Another idea inspired by hardware, are clock domains. Processes could run on one node with a heartbeat period that is a multiple from the general heartbeat period.
8. Conclusion and Future Work

In the described model, the communication is always added after the execution of the whole process. Considering only one process running on one node, the computation and the communication are completely serial. But also on a core with several processes, the last process in the schedule causes some communication overhead. If the signal could be sent before the computation is sent this would increase the parallelism between the execution and communication. To allow early transmission of data, methods must be found that allow sending data before all the computations has performed.

If a process would not need to wait for the availability of the RNI to initiate the send, the waiting time of the processes could be reduced. There are two ideas to implement this. Firstly, the RNI could be changed that the sending could be initiated at any time. The RNI queues the send requests and executes them as soon as the network becomes available. Secondly, an active waiting could be implemented by software. This means the program keeps continuing with the calculations and checks regularly the RNI for availability.

So far the processes within a hyper process are grouped together to ensure the dependencies among them. This grouping is not necessary for the correct execution, just the dependencies between the processes must be ensured, but other processes could run in between the processes of a hyper process. To increase the scheduler’s flexibility, it can be re-implemented in this way.

The basic idea of the NoC System Generator project is to have the possibility to generate a network with heterogeneous nodes. This first implementation assumes a homogeneous NoC and does not take into account any special hardware. The mapping process must be extended to consider the nodes’ limitation of memory, the limited number of RNI channels, and special hardware components, such as IO devices. In a second step the mapping process could also be extended to heterogeneous NoCs, where the resource of a node could be also a hardware accelerator or a multi-core system by itself.

Finally, an application that combines the different parts of the design process, as described in Chapter 4, must be created. This application would allow to run many tests and present the big picture of the project. One critical point in the whole process will be for sure the WCET analysis, which must be researched carefully. Guidelines and methods that support the system designer to develop applications must also be elaborated. One topic to research would be the best size of process. Too large processes degrade the flexibility of the mapper, too small processes cause much communication overhead. One idea to to allow higher parallelism is to apply software pipelining.
8.3. Proposed Changes to Existing Projects

8.3.1. Proposed Additions to ForSyDe-SystemC

As the delays between the processes are an essential factor to achieve an efficient mapping, it is important to treat them early in the process. We propose that the delay configuration is a part of the XML file that is the input to the mapping process. A design refinement in ForSyDe could be defined that adds the information to the signals if a delay is: necessary, forbidden, possible, or possible under certain circumstances. By moving it to ForSyDe the resulting system could be simulated and the functionality compared with the specification model. A part of Raudvere's work [37] deals with this topic.

8.3.2. Changes to the Network Generator

The communication via the RNI but on the same node (type II) is realized the same way as all the other network communications. This means that a flit coming from the resource in the worst case is deflected to another node, if the receive port of the RNI is occupied. A solution could be that this internal communication is directly written to the receiving channel. This would also make the communication time 0 or close to 0 for type II communication.

As long as the arrival time of the packet is unpredictable in the case of congestion and congestion must be avoided by the mapping, it is not necessary to keep the injection rate at 25%. Increasing it to 100% would decrease the heartbeat period.

For debugging purposes and also as kind of a watchdog, it would be good if a violation of the heartbeat — e.g. a process runs too long — could be detected by the program running on a network node. So far, the system cannot distinguish between a heartbeat violation and an absent value of the synchronous model of computation.

No execution order of the processes is explicitly defined in the target description file of the NoC System Generator so far. Especially, for the implementation of hyper processes this would be helpful.

So far the value for the heartbeat period and the binding of the processes to the cores are fixed within the hardware. Therefore, if one of these should be changed the whole hardware platform must be generated again. This makes it very time consuming to check the differences between two different systems. We propose to make the heartbeat delay programmable by software. And a process addressing method which does not depend on the PIDs should be implemented, to allow the change of the process structure by software. A new addressing method could probably
also remove the limitation that only one signal between two hyper processes can be sent. With a variable heartbeat it would also be possible to adapt the heartbeat frequency to actual demand of the application, i.e. power-saving mechanisms could be implemented.

Some support for the hyper process execution is necessary if the calculated mappings should be realizable. An implementation of the communication within the heartbeat (type III) must be developed. The most comfortable way would be to implement it within the RNI, analogous to the communication with delay. As this would increase the size of the RNI, a software solution is probably preferred.
Bibliography

(visited: 2012-10-21).

(visited: 2012-10-21).


http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.
(visited: 2012-10-03).


Appendix
Example Application Graphs

A.1. Chain Examples

A.1.1. Chain10 Example

<table>
<thead>
<tr>
<th>Processes</th>
<th>exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>300</td>
</tr>
<tr>
<td>$P_2$</td>
<td>100</td>
</tr>
<tr>
<td>$P_3$</td>
<td>300</td>
</tr>
<tr>
<td>$P_4$</td>
<td>250</td>
</tr>
<tr>
<td>$P_5$</td>
<td>420</td>
</tr>
<tr>
<td>$P_6$</td>
<td>300</td>
</tr>
<tr>
<td>$P_7$</td>
<td>250</td>
</tr>
<tr>
<td>$P_8$</td>
<td>420</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>250</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signals</th>
<th>volume</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_2$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_3$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_4$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_5$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_6$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_7$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_8$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_9$</td>
<td>20</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A.1.: Process and Signal properties of the chain10 example.

Figure A.1.: Problem graph of the chain10 example.
### A.1.2. Chain20 Example

<table>
<thead>
<tr>
<th>Processes</th>
<th>exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>300</td>
</tr>
<tr>
<td>$P_2$</td>
<td>100</td>
</tr>
<tr>
<td>$P_3$</td>
<td>300</td>
</tr>
<tr>
<td>$P_4$</td>
<td>250</td>
</tr>
<tr>
<td>$P_5$</td>
<td>420</td>
</tr>
<tr>
<td>$P_6$</td>
<td>300</td>
</tr>
<tr>
<td>$P_7$</td>
<td>250</td>
</tr>
<tr>
<td>$P_8$</td>
<td>420</td>
</tr>
<tr>
<td>$P_9$</td>
<td>300</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>250</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>420</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>300</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td>250</td>
</tr>
<tr>
<td>$P_{14}$</td>
<td>420</td>
</tr>
<tr>
<td>$P_{15}$</td>
<td>300</td>
</tr>
<tr>
<td>$P_{16}$</td>
<td>250</td>
</tr>
<tr>
<td>$P_{17}$</td>
<td>420</td>
</tr>
<tr>
<td>$P_{18}$</td>
<td>130</td>
</tr>
<tr>
<td>$P_{19}$</td>
<td>280</td>
</tr>
<tr>
<td>$P_{20}$</td>
<td>210</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signals</th>
<th>volume</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_2$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_3$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_4$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_5$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_6$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_7$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_8$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_9$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_{10}$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_{11}$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_{12}$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_{13}$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_{14}$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_{15}$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_{16}$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_{17}$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_{18}$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_{19}$</td>
<td>60</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A.2.: Process and Signal properties of the chain20 example.
A.1. Chain Examples

Figure A.2.: Problem graph of the chain example.
### A.1.3. Chain40 Example

<table>
<thead>
<tr>
<th>Processes</th>
<th>Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$S_4$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$S_5$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$S_6$</td>
</tr>
<tr>
<td>$P_7$</td>
<td>$S_7$</td>
</tr>
<tr>
<td>$P_8$</td>
<td>$S_8$</td>
</tr>
<tr>
<td>$P_9$</td>
<td>$S_9$</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>$S_{10}$</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>$S_{11}$</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>$S_{12}$</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td>$S_{13}$</td>
</tr>
<tr>
<td>$P_{14}$</td>
<td>$S_{14}$</td>
</tr>
<tr>
<td>$P_{15}$</td>
<td>$S_{15}$</td>
</tr>
<tr>
<td>$P_{16}$</td>
<td>$S_{16}$</td>
</tr>
<tr>
<td>$P_{17}$</td>
<td>$S_{17}$</td>
</tr>
<tr>
<td>$P_{18}$</td>
<td>$S_{18}$</td>
</tr>
<tr>
<td>$P_{19}$</td>
<td>$S_{19}$</td>
</tr>
<tr>
<td>$P_{20}$</td>
<td>$S_{20}$</td>
</tr>
<tr>
<td>$P_{21}$</td>
<td>$S_{21}$</td>
</tr>
<tr>
<td>$P_{22}$</td>
<td>$S_{22}$</td>
</tr>
<tr>
<td>$P_{23}$</td>
<td>$S_{23}$</td>
</tr>
<tr>
<td>$P_{24}$</td>
<td>$S_{24}$</td>
</tr>
<tr>
<td>$P_{25}$</td>
<td>$S_{25}$</td>
</tr>
<tr>
<td>$P_{26}$</td>
<td>$S_{26}$</td>
</tr>
<tr>
<td>$P_{27}$</td>
<td>$S_{27}$</td>
</tr>
<tr>
<td>$P_{28}$</td>
<td>$S_{28}$</td>
</tr>
<tr>
<td>$P_{29}$</td>
<td>$S_{29}$</td>
</tr>
<tr>
<td>$P_{30}$</td>
<td>$S_{30}$</td>
</tr>
<tr>
<td>$P_{31}$</td>
<td>$S_{31}$</td>
</tr>
<tr>
<td>$P_{32}$</td>
<td>$S_{32}$</td>
</tr>
<tr>
<td>$P_{33}$</td>
<td>$S_{33}$</td>
</tr>
<tr>
<td>$P_{34}$</td>
<td>$S_{34}$</td>
</tr>
<tr>
<td>$P_{35}$</td>
<td>$S_{35}$</td>
</tr>
<tr>
<td>$P_{36}$</td>
<td>$S_{36}$</td>
</tr>
<tr>
<td>$P_{37}$</td>
<td>$S_{37}$</td>
</tr>
<tr>
<td>$P_{38}$</td>
<td>$S_{38}$</td>
</tr>
<tr>
<td>$P_{39}$</td>
<td>$S_{39}$</td>
</tr>
</tbody>
</table>

Table A.3.: Process and Signal properties of the chain40 example.
A.1. Chain Examples

Figure A.3.: Problem graph of the chain example.
A. Example Application Graphs

A.2. Parallel Example

<table>
<thead>
<tr>
<th>Processes</th>
<th>exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>300</td>
</tr>
<tr>
<td>P_2</td>
<td>100</td>
</tr>
<tr>
<td>P_3</td>
<td>300</td>
</tr>
<tr>
<td>P_4</td>
<td>250</td>
</tr>
<tr>
<td>P_5</td>
<td>420</td>
</tr>
<tr>
<td>P_6</td>
<td>300</td>
</tr>
<tr>
<td>P_7</td>
<td>250</td>
</tr>
<tr>
<td>P_8</td>
<td>420</td>
</tr>
<tr>
<td>P_9</td>
<td>300</td>
</tr>
<tr>
<td>P_{10}</td>
<td>250</td>
</tr>
<tr>
<td>P_{11}</td>
<td>420</td>
</tr>
<tr>
<td>P_{12}</td>
<td>300</td>
</tr>
<tr>
<td>P_{13}</td>
<td>250</td>
</tr>
<tr>
<td>P_{14}</td>
<td>420</td>
</tr>
<tr>
<td>P_{15}</td>
<td>300</td>
</tr>
<tr>
<td>P_{16}</td>
<td>250</td>
</tr>
<tr>
<td>P_{17}</td>
<td>420</td>
</tr>
<tr>
<td>P_{18}</td>
<td>130</td>
</tr>
<tr>
<td>P_{19}</td>
<td>280</td>
</tr>
<tr>
<td>P_{20}</td>
<td>210</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signals</th>
<th>volume</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>S_2</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>S_3</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>S_4</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>S_5</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>S_6</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>S_7</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>S_8</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>S_9</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>S_{10}</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>S_{11}</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>S_{12}</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>S_{13}</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>S_{14}</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>S_{15}</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>S_{16}</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>S_{17}</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>S_{18}</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>S_{19}</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>S_{20}</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>S_{21}</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>S_{22}</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>S_{23}</td>
<td>60</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A.4.: Process and Signal properties of the parallel example.
Figure A.4.: Problem graph of the parallel example.
A. Example Application Graphs

A.3. Tree Example

<table>
<thead>
<tr>
<th>Processes</th>
<th>exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>300</td>
</tr>
<tr>
<td>$P_2$</td>
<td>100</td>
</tr>
<tr>
<td>$P_3$</td>
<td>300</td>
</tr>
<tr>
<td>$P_4$</td>
<td>250</td>
</tr>
<tr>
<td>$P_5$</td>
<td>420</td>
</tr>
<tr>
<td>$P_6$</td>
<td>300</td>
</tr>
<tr>
<td>$P_7$</td>
<td>250</td>
</tr>
<tr>
<td>$P_8$</td>
<td>420</td>
</tr>
<tr>
<td>$P_9$</td>
<td>300</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>250</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>420</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>300</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td>250</td>
</tr>
<tr>
<td>$P_{14}$</td>
<td>420</td>
</tr>
<tr>
<td>$P_{15}$</td>
<td>300</td>
</tr>
<tr>
<td>$P_{16}$</td>
<td>250</td>
</tr>
<tr>
<td>$P_{17}$</td>
<td>420</td>
</tr>
<tr>
<td>$P_{18}$</td>
<td>130</td>
</tr>
<tr>
<td>$P_{19}$</td>
<td>280</td>
</tr>
<tr>
<td>$P_{20}$</td>
<td>210</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signals</th>
<th>volume</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_2$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_3$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_4$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_5$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_6$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_7$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_8$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_9$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_{10}$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_{11}$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_{12}$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_{13}$</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>$S_{14}$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_{15}$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_{16}$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>$S_{17}$</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$S_{18}$</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>$S_{19}$</td>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A.5.: Process and Signal properties of the tree example.
Figure A.5.: Problem graph of the tree example.
A.4. MPEG-4 Decoder Examples

A.4.1. mpeg4 Example

<table>
<thead>
<tr>
<th>Processes</th>
<th>exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>3432</td>
</tr>
<tr>
<td>$P_2$</td>
<td>13827</td>
</tr>
<tr>
<td>$P_3$</td>
<td>15511</td>
</tr>
<tr>
<td>$P_4$</td>
<td>18211</td>
</tr>
<tr>
<td>$P_5$</td>
<td>8703</td>
</tr>
<tr>
<td>$P_6$</td>
<td>5421</td>
</tr>
<tr>
<td>$P_7$</td>
<td>800</td>
</tr>
<tr>
<td>$P_8$</td>
<td>12622</td>
</tr>
<tr>
<td>$P_9$</td>
<td>260</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>5181</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>8532</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>260</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td>240</td>
</tr>
<tr>
<td>$P_{14}$</td>
<td>15224</td>
</tr>
<tr>
<td>$P_{15}$</td>
<td>11332</td>
</tr>
<tr>
<td>$P_{16}$</td>
<td>582</td>
</tr>
<tr>
<td>$P_{17}$</td>
<td>260</td>
</tr>
<tr>
<td>$P_{18}$</td>
<td>323</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signals</th>
<th>volume</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_2$</td>
<td>70</td>
<td>1</td>
</tr>
<tr>
<td>$S_3$</td>
<td>362</td>
<td>1</td>
</tr>
<tr>
<td>$S_4$</td>
<td>362</td>
<td>1</td>
</tr>
<tr>
<td>$S_5$</td>
<td>362</td>
<td>1</td>
</tr>
<tr>
<td>$S_6$</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>$S_7$</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>$S_8$</td>
<td>357</td>
<td>1</td>
</tr>
<tr>
<td>$S_9$</td>
<td>353</td>
<td>1</td>
</tr>
<tr>
<td>$S_{10}$</td>
<td>353</td>
<td>1</td>
</tr>
<tr>
<td>$S_{11}$</td>
<td>313</td>
<td>1</td>
</tr>
<tr>
<td>$S_{12}$</td>
<td>313</td>
<td>1</td>
</tr>
<tr>
<td>$S_{13}$</td>
<td>94</td>
<td>1</td>
</tr>
<tr>
<td>$S_{14}$</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>$S_{15}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{16}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{17}$</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>$S_{18}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{19}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{20}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{21}$</td>
<td>157</td>
<td>1</td>
</tr>
<tr>
<td>$S_{22}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{23}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{24}$</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A.6.: Process and Signal properties of the mpeg4 example.
Figure A.6: Problem graph of the mpeg4 example.
### A.4.2. mpeg4-loop Example

<table>
<thead>
<tr>
<th>Processes</th>
<th>exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>3432</td>
</tr>
<tr>
<td>$P_2$</td>
<td>13827</td>
</tr>
<tr>
<td>$P_3$</td>
<td>15511</td>
</tr>
<tr>
<td>$P_4$</td>
<td>18211</td>
</tr>
<tr>
<td>$P_5$</td>
<td>8703</td>
</tr>
<tr>
<td>$P_6$</td>
<td>5421</td>
</tr>
<tr>
<td>$P_7$</td>
<td>800</td>
</tr>
<tr>
<td>$P_8$</td>
<td>12622</td>
</tr>
<tr>
<td>$P_9$</td>
<td>260</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>5181</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>8532</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>260</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td>240</td>
</tr>
<tr>
<td>$P_{14}$</td>
<td>15224</td>
</tr>
<tr>
<td>$P_{15}$</td>
<td>11332</td>
</tr>
<tr>
<td>$P_{16}$</td>
<td>582</td>
</tr>
<tr>
<td>$P_{17}$</td>
<td>260</td>
</tr>
<tr>
<td>$P_{18}$</td>
<td>323</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signals</th>
<th>volume</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>$S_2$</td>
<td>70</td>
<td>1</td>
</tr>
<tr>
<td>$S_3$</td>
<td>362</td>
<td>1</td>
</tr>
<tr>
<td>$S_4$</td>
<td>362</td>
<td>1</td>
</tr>
<tr>
<td>$S_5$</td>
<td>362</td>
<td>1</td>
</tr>
<tr>
<td>$S_6$</td>
<td>49</td>
<td>-</td>
</tr>
<tr>
<td>$S_7$</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>$S_8$</td>
<td>357</td>
<td>1</td>
</tr>
<tr>
<td>$S_9$</td>
<td>353</td>
<td>1</td>
</tr>
<tr>
<td>$S_{10}$</td>
<td>353</td>
<td>1</td>
</tr>
<tr>
<td>$S_{11}$</td>
<td>313</td>
<td>-</td>
</tr>
<tr>
<td>$S_{12}$</td>
<td>313</td>
<td>-</td>
</tr>
<tr>
<td>$S_{13}$</td>
<td>94</td>
<td>1</td>
</tr>
<tr>
<td>$S_{14}$</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>$S_{15}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{16}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{17}$</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>$S_{18}$</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>$S_{19}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{20}$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$S_{21}$</td>
<td>157</td>
<td>-</td>
</tr>
<tr>
<td>$S_{22}$</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>$S_{23}$</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>$S_{24}$</td>
<td>16</td>
<td>-</td>
</tr>
</tbody>
</table>

Table A.7.: Process and Signal properties of the mpeg4-loop example.
Figure A.7: Problem graph of the mpeg4loop example. The shaded rectangles represent the hyper processes.