A DESIGN FLOW FOR PREDICTABLE COMPOSABLE SYSTEMS

Author:
Ekrem Altinel

Examiner:
Prof. Ingo Sander

Supervisors:
Prof. Kees Goossens
Hosein Attarzadeh
Martijn Koedam

MASTERS’ DEGREE PROJECT
TRITA-ICT-EX-2013:137
July 18, 2013
Abstract

MPSoCs serve for the needs of the modern embedded systems by providing computationally powerful and flexible platforms. However, due to the design productivity gap and some architectural and methodological challenges, the successful design of real-time applications on these platforms is becoming a pressing concern. Methodologies starting with models at low levels of abstractions often are limited in their design space exploration. One way to improve the situation is by introducing formally analyzable models and entering the design process at a high level of abstraction. This approach enables the creation of correct-by-construction designs. ForSyDe is a modeling framework for embedded systems based on the theory of formal models of computation and it allows specification of systems at a high abstraction level. On the other hand, architectural challenges such as unpredictable timing behavior and interference between applications call for predictable and composable architectures. The CompSOC platform has a predictable and composable architecture and its design flow can map analyzable data-flow applications to an MPSoC in a way that guarantees the real-time requirements of the applications. Methodological challenges such as the automation of the design flows and tool interoperability are other major contributors of the design productivity gap, hence these aspects of a design flow are of paramount importance. By combining the ForSyDe and the CompSOC design flows, this thesis proposes a design flow that starts with a high level model of the system. By formal analysis, this design flow can produce a mapping of application tasks to an MPSoC platform. The design flow can implement an FPGA prototype of the system. The design flow is automated and as case studies, two image processing applications are implemented. These two applications are used to validate the design flow.
# Contents

Abstract i

List of Figures vii

List of Tables ix

Listings xi

List of Abbreviations xiii

1 Introduction 1
   1.1 Motivation .................................................. 2
      1.1.1 Problem Description ................................. 2
   1.2 Scope of the Study ......................................... 4
   1.3 Contributions ................................................ 4
   1.4 Overview ..................................................... 5

I Background 7

2 Approaches to Real-Time Systems Design 9
   2.1 A Primer of Real-Time Systems Design ..................... 10
      2.1.1 Software Design ....................................... 10
      2.1.2 Hardware Design ...................................... 11
      2.1.3 Design Space Exploration ............................ 12
      2.1.4 System Synthesis ..................................... 12
      2.1.5 Design Flows ......................................... 13
   2.2 Models of Computation ...................................... 14
      2.2.1 Kahn Process Networks ............................... 14
      2.2.2 Synchronous Data-Flow ............................... 15
      2.2.3 Cyclo-Static Data-Flow .............................. 15
List of Figures

1.1 Illustration of the ForSyDe-CompSOC design flow at an abstract level based on the y-chart ................................................. 3
2.1 A typical software stack of real-time systems ........................................ 10
2.2 Typical hardware architectures observed in computer systems ............. 11
2.3 An example mapping of tasks to processing elements ...................... 12
2.4 Steps of a generic design flow .................................................. 13
2.5 An example Kahn Process Network ............................................. 15
2.6 An example Synchronous Data-Flow Graph .................................. 16
2.7 An example Cyclo-Static Data-Flow Graph ................................... 16
2.8 Hardware/software co-design double roof model [1] .......................... 20
2.9 Platform based design flow [2] .................................................. 22
2.10 Y-chart design flow [3] .......................................................... 23
2.11 Y-chart abstraction pyramid and y-chart stacks [3] .......................... 23
2.12 The Daedalus$^R$ design flow [4] ............................................... 26
2.13 The MAMPS design flow [5] ................................................... 27
2.14 The DOL design flow [6] ....................................................... 28
2.15 The SHE design flow [7] ....................................................... 29
3.1 An example ForSyDe process network with domain interfaces between different MoCs. P3 is modeled as a composite process. . 32
3.2 Illustration of the simulation steps ForSyDe process constructors execute. ................................................................. 34
3.3 Illustration of the generated XML files for a given process network .... 35
4.1 Illustration of the CompSOC architecture [8] ............................... 38
4.2 Illustration of the CompSOC software platform ............................ 41
5.1 High level view of the ForSyDe-CompSOC design flow ................... 45
5.2 The code conversion and process network tools in the adaptation layer ................................................................. 47
5.3 The back-annotation phases in the adaptation layer ....................... 50
LIST OF FIGURES

5.4 The connection of the execution log and the binary file to the adaptation layer ........................................ 53
5.5 Addition of the platform dependent property extraction units in the adaptation layer ...................................... 53
5.6 The addition of the ForSyDe helper utilities .................. 54
5.7 The high level view of the automated design flow .......... 55
5.8 The complete automated design flow ......................... 57

6.1 An example ForSyDe SDF process network .................. 59
6.2 The main parts of the PN-to-SDF conversion utility ......... 62
6.3 The steps involved in the process network’s conversion into an SDF graph .............................................. 63
6.4 The flattened ForSyDe model .................................. 64
6.5 The actor-to-actor connections(left) and their expected conversion results(right) ........................................... 65
6.6 The conversion result of the process network shown in Figure 6.1 ................................................................. 67
6.7 The functional blocks in the code generator tool ............. 68
6.8 An example callgraph generated by opt ........................ 72

7.1 Example input(left) and output(right) images for SUSAN edge detection algorithm ............................. 75
7.2 SDF graph of SUSAN [9] ........................................ 76
7.3 Illustration of the ForSyDe process network of SUSAN .... 76
7.4 The input and output images for the edge detection algorithm ................................................................. 78
7.5 SDF graph of JPEG Decoder .................................. 79
7.6 Illustration of the ForSyDe process network of JPEG Decoder ............................................................ 79
7.7 The input and output image used for the JPEG decoder application .......................................................... 79
7.8 The system architecture used for case studies ................ 80
7.9 The execution time charts of the actors of the SUSAN application ....................................................... 82
7.10 The execution time charts of the actors of the JPEG decoder application ............................................. 83

A.1 Illustration of the ForSyDe design flow ...................... 97
A.2 Illustration of the CompSOC design flow ................... 98
A.3 Illustration of the ForSyDe-CompSOC design flow ........ 99
List of Tables

2.1 Design Flows and Tools ........................................ 30
7.1 Maximum measured execution times for SUSAN .............. 81
7.2 Maximum measured execution times for JPEG decoder ....... 81
Listings

3.1.1 Sample ForSyDe function used to model a process . . . . . . . . . 33
6.1.1 The implementation of the actor K . . . . . . . . . . . . . . . . . 60
6.1.2 Additional definitions and a top level block . . . . . . . . . . . . . 61
6.1.3 Parts of the process network produced with the help of the
definitions given in Listing 6.1.2 . . . . . . . . . . . . . . . . . . . . . . . 61
6.3.4 Code generation result for the actor implementation in Listing 6.1.1 70
7.1.1 The ForSyDe helper utilities used in the SUSAN models . . . . . . 77
B.1.1 The process network of SUSAN represented in XML . . . . . . . . . 100
B.2.2 The process network of JPEG decoder represented in XML . . . . 103
C.1.1 The CompSOC SDF graph of SUSAN . . . . . . . . . . . . . . . . . 105
C.2.2 The CompSOC SDF graph of JPEG decoder . . . . . . . . . . . . . 112
D.1.1 The converted SUSAN model . . . . . . . . . . . . . . . . . . . . . . . 117
D.2.2 The converted JPEG decoder model . . . . . . . . . . . . . . . . . . . 119
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoMik</td>
<td>Composable micro-kernel</td>
</tr>
<tr>
<td>CCSP</td>
<td>Credit-Controlled Static-Priority</td>
</tr>
<tr>
<td>CSDF</td>
<td>Cyclo-Static Data-Flow</td>
</tr>
<tr>
<td>CT</td>
<td>Continuous Time</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DOL</td>
<td>Distributed Operation Layer</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>DT</td>
<td>Discrete Time</td>
</tr>
<tr>
<td>ELF</td>
<td>Executable and Linkable Format</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-in first-out</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field programmable gate array</td>
</tr>
<tr>
<td>FRT</td>
<td>Firm real-time</td>
</tr>
<tr>
<td>FSL</td>
<td>Fast Simplex Link</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General-purpose graphics processing unit</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware abstraction layer</td>
</tr>
<tr>
<td>HRT</td>
<td>Hard real-time</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction set architecture</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>KPN</td>
<td>Kahn Process Network</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low Level Virtual Machine</td>
</tr>
<tr>
<td>MAMPS</td>
<td>Multi-Application Multi-Processor Synthesis</td>
</tr>
<tr>
<td>MCU</td>
<td>Minimal Coded Units</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of computation</td>
</tr>
<tr>
<td>MPSOC</td>
<td>Multi-Processor System-On-Chip</td>
</tr>
<tr>
<td>NI</td>
<td>Network Interface</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on chip</td>
</tr>
<tr>
<td>NRT</td>
<td>Non-real-time</td>
</tr>
<tr>
<td>PN</td>
<td>Process Network</td>
</tr>
<tr>
<td>POOSL</td>
<td>Parallel Object-Oriented Specification Language</td>
</tr>
<tr>
<td>PPI</td>
<td>Partition-programming-interface</td>
</tr>
<tr>
<td>PPN</td>
<td>Polyhedral Process Networks</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>SANLP</td>
<td>Static Affine Nested Loop</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data-Flow</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous dynamic random access memory</td>
</tr>
<tr>
<td>SHAPES</td>
<td>Scalable Software/Hardware Architecture Platform for Embedded Systems</td>
</tr>
<tr>
<td>SHE</td>
<td>Software/Hardware Engineering</td>
</tr>
<tr>
<td>SOC</td>
<td>System-on-chip</td>
</tr>
<tr>
<td>SRT</td>
<td>Soft real-time</td>
</tr>
<tr>
<td>SUSAN</td>
<td>Smallest Unvalue Segment Assimilating Nucleus</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SY</td>
<td>Synchronous</td>
</tr>
<tr>
<td>TDM</td>
<td>Time division multiplexing</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time division multiple access</td>
</tr>
<tr>
<td>TIFU</td>
<td>Timer-interrupt-frequency unit</td>
</tr>
<tr>
<td>TT</td>
<td>Time Triggered</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>USAN</td>
<td>Univalue Segment Assimilating Nucleus</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very-high-speed integrated circuit hardware description language</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst case execution time</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

As a result of the Moore’s Law[10], the emerging technologies in microelectronics allow us to fit multiple processing elements and various co-processors onto a single chip. However, due to these advancements, the gap between the microelectronics technology and the design productivity has grown even larger and one major contributor to this fact is seen as the absence of the design tools and appropriate methodologies for designing electronic systems[11]. The situation is aggravated in embedded systems and real-time systems which are electronic systems that, in general, contain both hardware and software components and sometimes have strict constraints.

Traditionally, the design of hardware in contrast to the design of software had been seen as separate processes. However, the design of embedded systems and real-time systems requires a unified methodology that is tailored to the application domain to satisfy these strict constraints. Satisfaction of these strict constraints depends on having precise knowledge about the behaviors of both hardware and software components of the system. With this precise knowledge about the behavior of the system, the automation of the unified methodology for the design of real-time systems can be achieved.

Behavior of the software components of a Multi-Processor System-On-Chip (MPSOC) can be captured using appropriate models of computation (MoC) for the application domain. MoCs have well-defined execution semantics which allow various aspects of the modeled applications to be analyzed. Different MoCs have different suitable application domains and each vary in their analyzability, expressiveness and level of implementation complexity. Hence it is infeasible to model any arbitrary application from any domain using a single MoC. Furthermore, having a framework which supports modeling and simulation of applications with various MoCs helps to improve productivity when designing applications for different domains. ForSyDe[12] is such an application
modeling framework developed at Royal Institute of Technology to help design applications by adhering to one or more of the MoCs it supports.

On the other hand, capturing the behavior of the hardware requires a rather holistic approach. MoCs can be used to model the hardware components of a system to determine their timing behavior. Nevertheless, if the architecture has unreliable timing behavior, having these models would serve no purpose when the applications developed require certain timing constraints to be met. Hence, time predictability is an important property of the hardware architecture. A time predictable architecture performs primitive operations within bounded time, hence the worst case execution time (WCET) of all its primitive operations can be calculated without running the applications.

Unfortunately, in a real-time system however, having analyzable models of a time predictable architecture by itself does not earn us very much either, since we need to preserve these properties at the Real-Time Operating System (RTOS) level which provides system services to the applications. Hence the design of hardware components also needs to deal with software support layers; leading to the concept of a platform which includes both hardware and software components. CompSOC [13] is a platform developed at the Eindhoven University of Technology, with these ideas in mind. In addition to the ideas discussed here, CompSOC also supports execution of multiple applications in temporal isolation which is also known as composability. Composability is the ability of the platform to run multiple applications without interfering with each other’s timing behavior.

1.1 Motivation

1.1.1 Problem Description

As the first part of this chapter has outlined, the design of real-time systems requires well defined methodologies and certain system level properties to successfully satisfy the application constraints.

This thesis investigates the integration of an application modeling framework such as ForSyDe with a composable predictable platform such as CompSOC to devise a design flow for real-time systems. The matches and mismatches between the design flows of these two research projects should be identified and solutions to the identified mismatches should be proposed in order to enable the automation of the design flow.

The intended design flow this thesis project aims to achieve is shown in Figure 1.1. This flow is based on the y-chart approach [3] which is explained in subsection 2.4.3.
1.1. MOTIVATION

ForSyDe is a framework for modeling real-time applications with formal MoCs and it allows simulation of the modeled applications. Moreover, CompSOC is an eminent example of a platform for development of real-time systems with a fully automated design flow incorporating design space exploration (DSE) tools for satisfying real-time constraints of applications.

The coupling of ForSyDe and CompSOC results in a real-time system design flow with an entry point allowing modeling at a high abstraction level using analyzable MoCs. Additionally, since both projects are aligned along similar principles, their integration is possible to achieve. The anticipated outcome of this thesis is an automated design flow integrating ForSyDe and the CompSOC platform, which is validated by case study applications.

The work described in this thesis is inviting for the individuals involved in these projects and relevant fields. Thesis work should provide sufficient introductory information to fellow students in the same field as a guide to ForSyDe and CompSOC.
1.2 Scope of the Study

ForSyDe supports a number of MoCs to be used for application modeling, whereas CompSOC can execute support Kahn Process Networks (KPN), Cyclo-Static Data-Flow (CSDF) and Time Triggered (TT) MoCs. Along with the time limitations of the thesis project and technical reasons, there exist theoretical restrictions on the integration of all of the MoCs into the intended design flow. Synchronous Data-Flow (SDF) is an analyzable and thoroughly tested MoC on the CompSOC platform, hence achieving the integration of ForSyDe and CompSOC for the SDF MoC is a realistic goal. Therefore, among the MoCs ForSyDe supports, SDF is chosen to be the MoC for integration of ForSyDe and CompSOC design flows.

Synchronous MoC is of particular interest for integration as a continuation to this project. Hence investigation of this MoC’s integration into the design flow is seen as a future work in this thesis.

1.3 Contributions

In light of the motivation, contributions of this thesis are listed in this section.

- **Presentation of the general approaches in real-time system design flows:** Preliminary information about real-time system design and design flows that are relevant to ForSyDe and CompSOC are presented as a literature study.

- **Explanation of ForSyDe and CompSOC design flows:** As part of the background study, the design flows of ForSyDe and CompSOC are presented. Firstly, for ForSyDe, the modeling concepts are introduced and design flow of ForSyDe is explained. Secondly, for CompSOC, brief information about the architecture is given and the design flow components are explained briefly.

- **Integration of ForSyDe design flow with CompSOC design flow for SDF MoC:** As part of the main contribution of this thesis, the ForSyDe framework is integrated into the design flow of CompSOC. The mismatches between two design flows are identified and steps to achieve integration for applications with SDF MoC are explained.

- **Development of necessary tools to automate the ForSyDe-CompSOC design flow:** For implementation of the proposed design flow in this thesis, a number of technical development tasks were carried out. These technical aspects of the project are explained.
1.4. OVERVIEW

- **Identification and development of case study applications:** For validation of the design flow proposed in this thesis, two case study applications are modeled using the SDF MoC from the ForSyDe framework.

- **Validation of the ForSyDe-CompSOC design flow using case study applications:** The design flow developed is validated in a validation environment consisting of simulations and test-benches.

1.4 Overview

Chapter 2: Approaches to Real-Time Systems Design introduces some of the basic concepts encountered in designing applications for real-time systems. The chapter starts with evaluation of classical approaches to design flows and continues with important aspects of the design flows. This chapter concludes by giving information on overview of the design flows relevant to this thesis.

Chapter 3: ForSyDe introduces the ForSyDe methodology, its modeling concepts and the design flow. This chapter is presented as part of the background study of the thesis, hence only the parts relevant to this thesis are discussed.

Chapter 4: The CompSOC Platform includes the overview of the CompSOC platform. The hardware and software architecture of the CompSOC platform are introduced in this chapter. The design flow of CompSOC is summarized here as well.

Chapter 5: Stepwise Development of the Design Flow contains the development steps taken to devise the design flow proposed in this thesis. The overview of the design flow is presented with the major design decisions and motivations for the tools developed throughout the thesis work.

Chapter 6: Tool Development explains the modeling conventions in the ForSyDe-CompSOC design flow and the internal components of the tools developed.

Chapter 7: Case Studies presents the case study applications developed for validation and the results of the experiments.

Chapter 8: Conclusions and Future Work suggests possible future improvements to the design flow and presents the conclusions for the thesis.
Part I

Background
Chapter 2
Approaches to Real-Time Systems Design

An embedded system is an electronic system generally found in a part of a larger system. Compared to personal computers, embedded systems are designed for a specific purpose and they are often restricted in their capabilities due to limited resources. These resources can be computational resources such as processing elements, memory and peripherals, as well as less tangible resources such as power.

Some embedded systems also have timing constraints where a specific task started in response to an event must be completed before its deadline. These systems are known as real-time systems. These systems are used in various application areas, and the failure to complete a task before its deadline may have serious consequences depending on the application area. Depending on the severity of the consequences of missing the deadline of a task in a real-time system, these systems can be classified as hard real-time (HRT), firm real-time (FRT), soft real-time (SRT) or non-real-time (NRT) systems respectively, with HRT systems having the greatest severity, whereas severity level is the least for the NRT systems.

As a result of the consistent advances in the microelectronics technology, the integration of multiple computational units on a single chip had been possible. This brought about the concept of the system-on-chip (SOC). A SOC is an integrated circuit that can contain a complete computer system or a subsystem of a larger electronic system on a single chip. SOCs are often designed as customizable platforms, where the end-user can modify the architecture template to fit the needs of the target application. Design of SOCs relies on methodologies and tools to help deal with both hardware and software aspects of the design.
2.1 A Primer of Real-Time Systems Design

This section provides a brief background information in design of real-time systems.

By its nature real-time systems design requires both hardware and software components to be designed together and in contrast to the designs targeting personal computers, the designs in real-time systems are customized greatly to achieve the best fit for the target applications within the given constraints.

2.1.1 Software Design

A typical software subsystem of a real-time system is structured as a stack of software components, where components at each layer provide an abstraction to the upper layers as shown in Figure 2.1.

![Diagram of software stack of real-time systems]

Upper layers in this stack are considered to be the application software, whereas the lower layers are most often implemented by the RTOS.

These applications are generally modeled as concurrent tasks that are triggered by certain events, such as a timer interrupt. The concurrency among the tasks is facilitated by the RTOS constructs such as semaphores and message queues. RTOS is also responsible for scheduling of the application tasks.

Apart from system level services the RTOS provides, it implements an abstraction for the hardware, which is widely known as the hardware abstraction.
layer (HAL). The HAL helps the development of applications that are less dependent on the underlying hardware architecture. This allows the applications to become portable to the different architectures the RTOS supports.

2.1.2 Hardware Design

A typical hardware architecture for a computer system encloses one or more processing elements connected to a bus or a more complex interconnect to connect memory elements and peripherals to the processing element(s). Figure 2.2 illustrates these commonly seen hardware architectures.

Uni-processor systems feature a system architecture as depicted in Figure 2.2(a). This architecture includes a single processing element, a memory unit and peripherals all communicating via a shared bus.

Historically, the typical multi-processor system as shown in Figure 2.2(b) has evolved from uni-processors for higher performance, better resource utilization and power efficiency. Multi-processor systems may include a more sophisticated interconnect for communication such as a network on chip (NoC).

The pressing performance demands of the applications have driven the development of technologies such as caches, branch prediction and speculative execution. These technologies allowed significant improvements in the average performance of the computer systems.
2.1.3 Design Space Exploration

After the design of software components and hardware architectures, the design of real-time systems is essentially concerned with the mapping of the application tasks onto the processing elements in the architecture. Mapping is the effort of finding a valid placement and schedule for the tasks on each processing element while also satisfying the real-time requirements of the system. Mapping can be seen as a part of a larger system level design phase known as the design space exploration (DSE).

The design space exploration efforts are carried out in order to find the best fit for the application in view of its functional and non-functional constraints, such as performance constraints or power constraints. Furthermore, in general case, this task is very laborious to accomplish, since the problem of exploring the entire design space is known to be NP-hard [14].

2.1.4 System Synthesis

System synthesis can be seen as the generation of hardware and software components. Hence it involves two separate processes known as hardware and software synthesis. During the hardware synthesis, depending on the technology used, specific vendor provided tools are used to generate the hardware for the
system. Software synthesis on the other hand, may involve code generation, compilation and customization of the software architecture to have the best fit for the target application.

### 2.1.5 Design Flows

In the design of real-time systems, methodologies or design flows are used to standardize the way of correctly implementing and satisfying all the requirements of the target applications.

In overall view, the design steps explained in the previous sections constitute the main elements of a design flow. At an abstract level, the steps of a design flow can be illustrated as shown in Figure 2.4.

![Fig. 2.4: Steps of a generic design flow](image)

- **System design**: System design involves the design of both hardware and software components of a system. Based on the functional specifications provided, the applications are developed using programming languages such as Ada, C/C++ and in some cases modeling languages such as Unified Modeling Language (UML). In this phase, the hardware models can also be generated and they can be seen as part of the functional models generated at the end of this phase.

- **Design space exploration**: At this step, the functional models are used to find a mapping between the application tasks and the resources available
in the architecture. These resources do not always have to be physical resources, they may as well be resources such as time slots for execution or bandwidth for communication. This phase results in a mapping that satisfies all the constraints while guaranteeing the functional correctness of the system. Another goal of this step is the optimization of the resource usage.

- **System synthesis**: This final phase, strives to reach an efficient implementation for a given mapping and it involves the hardware and software synthesis tasks explained in subsection 2.1.4.

Depending on the complexity of the systems developed, a design flow may accommodate many tools to automate the design process. From the design productivity point of view automation and interoperability are important aspects of a design flow.

### 2.2 Models of Computation

Models of computation (MoC) allow to abstract the implementation of computation and communication in the system. Once a system functionality is expressed using a MoC, this model can be subjected to transformations and analysis to reach to an efficient implementation. Design of real-time systems benefits from analyzable models with high abstraction levels. Hence, the selection and use of a suitable MoC for a specific application domain is of great importance.

There are many MoCs in the literature, however investigation of all the existing MoCs is not the intention of this thesis. Hence, in the following sections, only the relevant data-flow MoCs and the synchronous MoC are presented.

#### 2.2.1 Kahn Process Networks

Kahn Process Networks (KPN) \[15\] consist of independent concurrent processes communicating via unbounded first-in first-out (FIFO) channels. These processes are also known as actors. Figure 2.5 illustrates a KPN where A, B, C and D are actors.

Execution principles of a KPN can be listed as:

- Each actor in the network reads tokens from its inputs in a blocking manner, and writes to its outputs in a non-blocking manner.

- When an actor tries to read from an empty input channel, that actor becomes blocked and stays blocked until a token is written to that input channel.
2.2. MODELS OF COMPUTATION

![Diagram of an example Kahn Process Network](image)

Fig. 2.5: An example Kahn Process Network

- At any instance, an actor may be executing or waiting for an input token.
- Channels may contain initial tokens. Initial tokens on the channels prevent the network from becoming deadlocked.

Due to its data-driven execution order, KPN MoC is a suitable MoC in design of data-flow applications. Hence, the advantage of using KPN MoC for data-flow applications is the succinctness and expressiveness of the models.

On the other hand, since KPN requires non-blocking write operations, the memory requirements for FIFO channels can be infinite. Another disadvantage of using KPN is, since there are no restrictions on the production and consumption of tokens, KPN actors need to be scheduled dynamically, which introduces a computational overhead.

2.2.2 Synchronous Data-Flow

Synchronous Data-Flow (SDF) [16], [17] is a restricted version of KPN in which actors consume or produce fixed amount of tokens at each execution. Execution of an actor is also called firing. Figure 2.6 shows an example SDF graph. The production and consumption rates of actors in an SDF network are represented by the numbers on the channels between the actors.

The main advantage of using SDF MoC is its analyzability. Due to the constant number of tokens produced and consumed in SDF MoC, the static schedules for actors and worst-case buffer sizes for channels can be computed.

2.2.3 Cyclo-Static Data-Flow

Cyclo-Static Data-Flow (CSDF) [18] is a more relaxed version of SDF where each actor’s production and consumption rates in consecutive iterations are allowed to change in a cyclic pattern.
The advantage of CSDF over SDF is that it allows succinct modeling of data-flow applications with variable rate actors since variable rates can be expressed easily using CSDF. Just as SDF, CSDF is also analyzable and it is still possible to find static schedules for the actors, since the rates are known \textit{a priori}.

### 2.2.4 Synchronous

Synchronous (SY) MoC [19] assumes the existence of an implicit time interval for execution. Synchronous MoC defines the basic mathematical principles underlying in the design of synchronous languages.

According to the synchrony hypothesis, at every tick or time instant marking the beginning of a time interval, inputs are read and outputs are generated instantaneously. The outputs generated may have a special token for denoting absent values.

In contrast to data-flow MoCs, in synchronous MoC, the event in the input channels of a process must be \textit{totally ordered}. In other words, a synchronous
process must wait for tokens on all of its inputs and react instantaneously to a value change.

Important to notice here is that, not all applications can be modeled with a single model of computation. The way the concurrency and control handled in a specific MoC influences the design of the application greatly. For instance, control oriented applications with many decision points in its execution flow may be infeasible to model using data-flow based MoCs.

2.3 Some Challenges in the Design of Real-Time Systems

Within the context of this thesis work, some challenges in the design of real-time systems are identified and categorized.

2.3.1 Design Challenges

In contrast to applications designed to execute sequentially, real-time applications are in general concurrent in nature. Hence, the way of thinking required to develop these applications is fundamentally different from that of the sequential applications. Even when the designers master this way of thinking to parallelize these applications, the design process still needs to be supported by tools to validate the correctness of the designs.

Another design challenge is in the static analysis of the application models, which provides knowledge about concurrency in the applications. The knowledge of concurrency opens up a range of possibilities at the DSE step of the design flow for reaching to a more efficient mapping. Moreover, due to the use of inappropriate MoCs or modeling languages, the static analysis may not be possible. Hence, the MoC used to design the applications must provide some mathematical tools to analyze the concurrency in the design.

2.3.2 Architectural Challenges

Architectural challenges originate from the system level properties observed in the overall architecture. As explained in subsection 2.1.3, the DSE involves finding valid schedules for the tasks that satisfy given constraints. However, this is only possible if the exact timing behavior of the hardware architecture is known. Hence, for real-time systems design, this timing behavior is imperative,
and without this knowledge, it is not possible to satisfy the real-time constraints of the system.

Predictability

For hardware components, having the precise knowledge about the timing properties, sometimes, may not be possible due to the technologies used in the interest of improving the average performance of the system. Technologies such as caches, branch prediction and speculative execution were introduced to improve the average performance of the computer systems, however they cause the prediction of the exact temporal behavior of the system to be very hard. Use of these technologies in real-time systems limits the DSE severely. This is why hardware architectures for real-time systems often eliminate these structures in favor of predictability.

Composability

Yet another source of unpredictability, even when hardware has temporal predictability, is caused by the applications running on the system, regardless of the formal models used in the design of the software. This unpredictability is observed when two applications interfere with each other’s temporal behavior due to their accesses to resources that are shared between them. Providing dedicated resources to all applications is not always a feasible solution, thus for guaranteeing the consistency of the timing characteristics of an application, having temporal predictability by itself is inadequate. Another system level property known as temporal isolation or composability is also needed to preserve temporal predictability of individual applications. Composability allows mapping of independent applications to the system in a way such that none of the applications are affected by the co-existing applications.

Predictability and composability together allow independent development and formal verification of a system, reducing the effort spent on integration and verification, ultimately reducing the overall development cost of the system.

Predictability and composability have severe implications on the designs of both hardware and software components in a system. Some techniques and rules to achieve predictability and composability in a platform are explained in [8] and [20]. Åkesson et al. [8] demonstrate how to ensure predictability and composability with five techniques. More information on how these principles for predictability and composability are applied on the CompSOC platform can be found in chapter 4. In [20], the predictability is seen as an inherent property of the instruction set architecture (ISA).
2.3. Methodological Challenges

The methodological challenges are directly caused by the tools, methods and work-flows adopted to design real-time systems. Methodologies help to deal with the complexity of designing real-time systems by providing standardized methods to correctly implement the systems and by improving the design productivity. The economic budgets of these development projects are profoundly influenced by the design productivity. Hence, the methodological challenges determine the economic outcome of the projects.

The level of abstraction at the entry point to the design flow influences the decisions taken in the design process. Thus, starting from a low level of abstraction causes many possible design choices to become infeasible in later stages of the design flow; consequently, leading to many limitations in the DSE step. To solve this problem, the abstraction level can be raised by using formally analyzable MoCs at the entry point to the design flow.

Automation of the design tools is a major contributor of the design productivity. Embedding of standardized methods into automated design flows reduces the time spent on recurrent design activities, since it minimizes the human involvement in the design flows. Nevertheless, automation of crucial steps like DSE is not always a trivial task. Successful mapping of tasks to processing elements is only possible if both software models and hardware models are analyzable. Hence, having analyzable models for the software and hardware components of a system is also an important contributor of the design productivity.

Automation of design flows

The real-time systems design flows involve many steps for transformations and analysis of intermediate models. In addition, the overall work-flow may have many decision points, where alternative transformations may be needed depending on the system being modeled. Hence, the overall design effort may become rather laborious and error-prone process for humans. The lack of appropriate tools and limited automation in the design flow of real-time systems are big contributors of the design productivity challenge [11].

Automation of the design flows improves the productivity of the overall design effort by allowing quick prototyping and consequently reducing the development cost of the real-time systems. Automation in design flows can be viewed as two stage process. At the first stage, the individual tools to perform the transformation and analysis steps need to be automated. At the second stage, the overall flow of control between the tools needs to be automated.
Interoperability between tools

A more fundamental cause of the design productivity problem can be seen as the lack of interoperability between the tools in the design flows.

Pimentel et al. address the problem of interoperability among the design flow tools in the context of the Daedalus design flow in [21]. As explained in this work, the main goal of achieving interoperability of tools is to reduce the effort spent on developing customized tools in the design flows by standardizing the interaction between these tools and development of an infrastructure to establish the flow of control among the tools.

2.4 Holistic Approaches to Real-Time Systems Design

2.4.1 Hardware/Software Co-design

Hardware/software co-design has been a very active research area in the past two decades and a general historical overview of this approach is given in [1]. This approach has emerged as a remedy to the complexity of handling the design of embedded systems. This approach requires that the system is specified at a high abstraction level. After the system specification is defined, the abstraction levels of hardware and software components are lowered in an iterative manner.

![Hardware/software co-design double roof model](image)

Figure 2.8: Hardware/software co-design double roof model [1]

Figure 2.8 shows the double roof model of co-design. Left side of this model shows the synthesis steps associated with software components and the right side
2.4. HOLISTIC APPROACHES TO REAL-TIME SYSTEMS DESIGN

shows the hardware related synthesis steps. The behavioral descriptions of the system components are on the upper roof and the structural descriptions are at the lower roof. The design efforts in this model are represented as the arrows. In this picture, the implementations are represented by downward directed arrows. For example, in the hardware part, at the logic level, this arrow may represent the synthesis of an RTL model to a gate level description. The horizontal arrows show the transfer of information to the following lower level of abstraction. This information is generally related to the implementation details of the model to satisfy certain requirements.

For successful implementation of real-time systems, hardware/software co-design approach places the DSE at the heart of this methodology. DSE is seen as a system level exploration, hence it can evaluate many alternative architectures for mapping of the software tasks.

2.4.2 Platform Based Approach

Keutzer et al. [2] present and demonstrate some important concepts that have been integrated into a methodology based on the concept of a platform. A major contribution of this work includes the separation of concerns in the overall design methodology of embedded systems. In this proposed methodology, the system function, system micro-architecture, mapping and system implementation phases are independent from each other. The platform is seen as a combination of a hardware platform comprising of a class of micro-architectures enabling software reuse and a software platform containing an RTOS, device drivers and a network communication subsystem.

Figure 2.9 illustrates the design space of a system and the arrow shown represents the mapping of application onto the platform which constitutes an abstraction level in this design space. In this methodology, the first step is to map an application instance to a platform by refinements guided by the designer. In the second phase, the main concern in this methodology is to reach an efficient implementation by exploration of possible solutions.

The advantage of using a platform based methodology is that it allows high volume production of hardware platforms targeted to a specific application domain, thus reducing the production costs of embedded systems. By allowing high level of software reuse and integration of advanced debugging techniques, this methodology also helps to reduce the development costs. The disadvantage of using this approach is that the degree of freedom in modeling a system is limited to the capabilities of the platform.

For this methodology to be effective, it can be argued that design should start at high levels of abstraction with formal MoCs and the design activity must be
well defined. By exploiting the formalisms in the functional specification, this methodology enables automated verification and synthesis.

2.4.3 Y-chart Approach

Y-chart approach is based on the quantitative exploration of the design space on a platform template. Kienhuis et al. [3] present a methodology based on the y-chart approach and discuss some aspects of the design space exploration, stacks of y-charts, mapping and abstraction pyramid in this methodology. This methodology favors performance models for evaluation of trade-offs between different architectural choices the designer makes. The overview of this approach can be observed in Figure 2.10.

The y-chart approach starts with the description of a particular architecture instance and generation of the performance model of this particular architecture by using performance analysis. This performance model is used for mapping a set of applications on the architecture instance, and as a result, the performance numbers are obtained. At this point, until the performance numbers satisfy all the constraints of the system, in an iterative fashion, the designer makes refinements on the architecture instance or the application specification or on the mapping.

In theory, the design space exploration can be performed systematically by iterating over the set of all possible configurations of parameters of the
architecture template. In practice however, a stepwise refinement of the design space is needed due to the large design space of the architecture templates.

To cope with the design complexity, the design starts at a high abstraction level and the abstraction level is lowered in multiple steps. At each step the design space is explored with a different y-chart environment based on a more detailed model than the one used in previous iteration as shown in Figure 2.11.
CHAPTER 2. APPROACHES TO REAL-TIME SYSTEMS DESIGN

2.5 Position of the ForSyDe-CompSOC Design Flow

The ForSyDe-CompSOC design flow is the integrated design flow of the modeling framework ForSyDe and the predictable composable platform CompSOC. This section puts the ForSyDe-CompSOC design flow into a context by providing some background information on ForSyDe, CompSOC and the relevant design flows and tools. To better understand these two parts of this design flow, firstly, ForSyDe and CompSOC are briefly introduced. Chapter 3 and chapter 4 explain more in depth aspects of ForSyDe and CompSOC respectively.

2.5.1 ForSyDe

Real-time system design requires software to be designed in a way that can be formally analyzed. Depending on the target application, suitable MoCs that offer useful mathematical analysis tools are good candidates to use for the design of software components of a real-time system. Furthermore, a design flow for real-time systems must consider the inclusion of design automation tools for software modeling.

ForSyDe is an overall system design methodology that helps to design applications with mixed MoCs and supports the design process of real-time systems. ForSyDe offers several MoCs that are commonly encountered in design of real-time systems, such as SDF, SY, and several others.

Designs in ForSyDe are expressed as networks of processes. ForSyDe allows creation of hierarchical and modular designs of applications, hence a ForSyDe process network may contain the implementation for an entire application or a subsystem of it. These process networks consist of various kinds of ForSyDe processes such as combinational elements, state machine elements, delay elements, source elements and sink elements. These ForSyDe processes communicate with each other using FIFO buffers and they execute with well-defined semantics with respect to the MoC they are picked from. At the design phase, each instance of a ForSyDe element is given its associated parameters, such as functions for combinational elements or state values for elements with state.

As of writing of this thesis, ForSyDe framework has two alternative implementations; one using the language Haskell, and another based on SystemC [22] using the language C++. ForSyDe allows the simulation of application models on designers’ workstations.

ForSyDe framework is seen as part of a bigger design flow, where ForSyDe plays the role of an application modeling framework at the entry point to the design flow. During the simulation of ForSyDe models, it is possible to extract the
structure of the process network of the application model in an XML format. This feature is referred as introspection and its main purpose is to provide information to the DSE tools about the application’s structure.

2.5.2 CompSOC

CompSOC faithfully adheres to the principles of predictability and composability for the implementation of real-time systems and it provides predictable services to applications while executing applications in a composable fashion.

Each application on the platform is given a virtual platform consisting of a set of resource budgets. At run time, when an application exhausts its budget for a resource, the application is no longer allowed to use that resource. This is the essential mechanism behind the composability feature of CompSOC.

Software subsystem of CompSOC contains a micro-kernel and an optional library to facilitate the execution of applications with specific MoCs. Currently, in addition to applications that are not designed with a specific formal MoC, CompSOC is also able to execute applications designed with CSDF MoC and TT MoC.

The hardware subsystem of the platform is handled by the design flow of CompSOC to automatically generate the detailed architecture models for the hardware from an abstract representation of the intended architecture provided by the designer. Currently, CompSOC hardware architecture features Microblaze cores from Xilinx [23] as processing elements that are connected to each other via a NoC. For the DSE of real-time applications, CompSOC uses the SDF³ [24] tool for generating mappings between application tasks and the hardware architecture. The design flow also handles the generation of the middle-ware for the software stack which includes the micro-kernel and the support libraries needed by the applications. The entire system can be automatically synthesized, and run on Xilinx field programmable gate arrays (FPGAs).

2.5.3 Relevant Design Flows and Tools

This section briefly introduces some of the existing design flows and relevant tools for real-time systems with the purpose of establishing the position of the ForSyDe-CompSOC design flow. More extensive surveys of the existing design flows and tools are carried out by Densmore et al. [25], Sangiovanni-Vincentelli [26], Gerstlauer et al. [27] and by Haid et al. [28].
Daedalus$^{\text{RT}}$  

Daedalus$^{\text{RT}}$ is a design methodology that targets streaming applications. Bammakhrara et al. introduce and demonstrate the design flow in [4]. Daedalus$^{\text{RT}}$ flow features a front-end for converting a set of sequential applications in Static Affine Nested Loop (SANLP) form to Polyhedral Process Networks (PPN). The PPN representation of the applications are then converted into their CSDF equivalents to perform hard-real-time schedulability analysis. CSDF actors are scheduled in a way to allow multiprocessor deployment and ensure temporal isolation between applications. The overall design methodology of Daedalus$^{\text{RT}}$ is given in Figure 2.12.

The back-end of the Daedalus$^{\text{RT}}$ flow utilizes the ESPAM tool for system synthesis. This tool takes in a number PPN models of the applications, a platform specification generated by the analysis phase and a mapping specification derived from the converted CSDF models. The platform specification assumes a tiled homogeneous MPSOC with distributed memory. ESPAM supports a number of synthesis back-ends including Xilinx Platform Studio for FPGAs. Except the WCET analysis step, the Daedalus$^{\text{RT}}$ flow is automated.

MAMPS

Multi-Application Multi-Processor Synthesis (MAMPS) design flow is based on the SDF MoC and it differentiates itself from relevant design flows by accom-
2.5. POSITION OF THE FORSYDE-COMPSOC DESIGN FLOW

modating multiple use cases of applications to be mapped onto a multiprocessor [5]. MAMPS design flow utilizes the SDF³ tool for design space exploration and has an implementation back-end for Xilinx FPGAs [23]. The overall design flow can be seen in Figure 2.13.

As the inputs to the design flow, an XML file describing the topology of the actor graph and the source code for the individual actors are provided. The XML document contains information about each actor’s execution time, memory requirements and relevant source code identifier. For the channels between actors, this file also contains information about the buffer sizes and initial token sizes.

MAMPS design flow maps applications onto the target platform in a way such that, each actor of a single application is mapped to a single processing element and each processor executes only one actor per application. And each edge in the SDF specification of the application is mapped to a dedicated FIFO buffer. Hence the mapping performed by the MAMPS flow follows the natural fitness concept discussed in [3]. However, this scheme has the possible disadvantage of over-designing the system.

The hardware generation part of the flow synthesizes the system based on Xilinx Microblaze processors and communication between processors is accomplished by Fast Simplex Links (FSLs). The MAMPS back-end also performs software synthesis to generate the software to allow correct execution of the actors in the application.
DOL

The Distributed Operation Layer (DOL) is a framework for designing parallel applications targeting multiprocessor streaming applications that can be mapped to scalable, heterogeneous MPSOCs with distributed memory and advanced interconnects. The framework has been developed within the Scalable Software/Hardware Architecture Platform for Embedded Systems (SHAPES) project [29]. The DOL is a suitable modeling framework for a design flow based on the y-chart approach discussed in subsection 2.4.3. The overall design methodology is given in Figure 2.14.

![Fig. 2.14: The DOL design flow [6]](image)

The applications are modeled as KPNs using C/C++ and the parallel structure of the applications is described in XML format.

The DOL framework performs mapping and design space exploration based on the application specification, architecture description and a set of mapping constraints. DOL can simulate the applications designed for multiprocessor architectures using a simulation engine based on SystemC. DOL has a software synthesis back-end for several multiprocessor platforms such as Atmel Diopsis 940, Mparm and Cell Broadband Engine.

SHE/POOSL

Software/Hardware Engineering (SHE) is a model-driven design methodology utilizing UML and the Parallel Object-Oriented Specification Language (POOSL) for modeling and analysis of software/hardware systems [7]. The overall design methodology can be seen in Figure 2.15.

In the modeling and analysis phase, SHE has steps of formulation, formalization and evaluation. In the formulation step, design concepts are described
2.5. **POSITION OF THE FORSYDE-COMPSOC DESIGN FLOW**

in UML and design properties are formulated by annotating the UML models or using plain text by the designer. In the formalization step, the UML descriptions are converted to executable formal models expressed in POOSL and design properties are formalized by monitors. Finally in the evaluation phase the expected design properties are checked against the executable formal models. Until the design properties are satisfied, the modeling and analysis phases iterate.

SHE also has a realization phase where the models are converted into synthesizable descriptions of hardware and software. After the design properties are met, the flow commences to the realization phase in which the current version of SHE can synthesize software for a single processor platform.

**2.5.4 Position of ForSyDe-CompSOC Design Flow**

The ForSyDe-CompSOC design flow starts with a ForSyDe model of an SDF application, hence the flow is a good candidate for modeling streaming applications. Since the applications are modeled using SDF MoC, the parallelism is explicitly expressed in the application description. Since ForSyDe models are executable, this allows quick prototyping and early identifications of possible solutions at the design flow entry.

Fig. 2.15: The SHE design flow [7]
Real-time requirements are satisfied by mapping the applications with the SDF$^3$ tool. Mapping is done automatically. CompSOC platform offers predictability and composability, which enable independent development and verification of applications, consequently reducing the integration cost of the system. The CompSOC hardware-flow can generate an FPGA prototype on Xilinx FPGAs. Finally, the ForSyDe-CompSOC design flow is automated. It starts with a ForSyDe model and produces the implementation for the given set of applications without the intervention of the designer.

<table>
<thead>
<tr>
<th>MoCs</th>
<th>ForSyDe</th>
<th>CompSOC</th>
<th>Daedalus$^{RT}$</th>
<th>MAMPS</th>
<th>DOL</th>
<th>SHE/POOSL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SDF, SY, CT, DT, TT</td>
<td>CSDF, TT</td>
<td>SANLP</td>
<td>SDF</td>
<td>KPN</td>
<td>UML</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Backend</th>
<th>NA</th>
<th>NoC based MPSOC</th>
<th>Xilinx MPSOC</th>
<th>Xilinx MPSOC</th>
<th>Several MPSOC platforms</th>
<th>SW synthesis for single processor</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Automation</th>
<th>NA</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Unknown</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>DSE</th>
<th>NA</th>
<th>SDF$^3$</th>
<th>Hard real-time scheduling techniques</th>
<th>SDF$^3$</th>
<th>Built-in</th>
<th>Unknown</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Executable abstract specs.</th>
<th>Yes</th>
<th>No</th>
<th>Yes</th>
<th>No</th>
<th>Yes</th>
<th>Unknown</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Yes</th>
<th>No</th>
<th>Unknown</th>
<th>No</th>
<th>Has a built-in simulation engine</th>
<th>Unknown</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Predictab.</th>
<th>NA</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>NA</th>
<th>NA</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Composab.</th>
<th>NA</th>
<th>Yes</th>
<th>Yes</th>
<th>No</th>
<th>NA</th>
<th>NA</th>
</tr>
</thead>
</table>

Table 2.1: Design Flows and Tools
Chapter 3

ForSyDe: Application Modeling Framework

ForSyDe methodology [12] is based on formal MoCs that can be used to model hardware/software systems at a high abstraction level to overcome the complexities of designing such systems. Since modeling in ForSyDe is done using formal MoCs, it is easier to partition the design into software and hardware once the functionality of the system is established. ForSyDe has two modeling front-ends based on Haskell and SystemC respectively. In this chapter, the ForSyDe-SystemC [30] version is introduced and different aspects of using ForSyDe as a modeling framework are discussed. ForSyDe-SystemC lends itself as a viable modeling framework to overcome the difficulties experienced in developing applications for heterogeneous MPSOCs since the system model is already expressed in C++ and it is straightforward to synthesize it for a particular platform. ForSyDe design flow allows the designer to co-simulate an application with existing tools, legacy code or IPs by the method of co-simulation and wrapper processes [31].

ForSyDe-Haskell [32] already has synthesis back-ends for synthesizable VHDL and GPGPUs. There is ongoing work on integrating additional back-ends such as Nostrum NoC and GPGPUs for ForSyDe-SystemC. In the following sections application modeling, some important concepts and finally the design flow of ForSyDe are discussed.

3.1 Application Modeling and Simulation

ForSyDe-SystemC is a C++ header library that can be referenced, compiled and linked to produce an executable abstract specification of the system. In this regard, it allows quick prototyping of systems and provide early results for
verifier. It is also possible to integrate foreign models specified in C++ for simulation and verification. ForSyDe-SystemC allows to express design of a system in several MoCs, mainly synchronous MoC, SDF MoC and Continuous Time MoC. To be able to model heterogeneous systems, ForSyDe also allows to have mixed MoC designs by using domain interfaces between processes belonging to different MoCs.

ForSyDe models can be seen as hierarchical process networks where communication among processes is facilitated via FIFO channels. In ForSyDe, processes are instantiated by a construct called a process constructor. There are various process constructors for each MoC. A process constructor's sole purpose is to allow the designer to focus on functional correctness instead of being concerned about execution semantics of the MoC that the design is being specified in. ForSyDe-SystemC library provides these process constructors and handles the execution semantics of the created processes. This enables the designer to concentrate on creating the correct topology for the process network. Figure 3.1 illustrates a process network with domain interfaces to communicate between different MoCs.

ForSyDe does allow composing hierarchical process networks. Hierarchy in the design reduces the complexity and enables component reuse. Processes in ForSyDe are restricted to have a single output channel whereas the number of input channels can be arbitrarily large.

Some of the most frequently used process constructors in SDF MoC are `comb` for stateless processes with combinational behavior, `delay` for storing values produced by a process, `zip` for bundling multiple channels into a single channel.
3.1. APPLICATION MODELING AND SIMULATION

and unzip to separate individual channels bundled into a single channel. The main reason for using zip processes is the limited number of input channels in existing process constructors and by bundling input channels into a single channel even a process constructor with a single input channel can be provided all of its necessary inputs. The reason for using an unzip process would be to separate multiple channels a process might have zipped to output through its single output channel. The production and consumption rates for the zipped and unzipped channels are taken into account by the ForSyDe library when executing the connected processes.

Modeling a process: Process constructors with behavior such as comb, take in a function pointer to execute whenever that process needs to fire. When simulating a ForSyDe-SystemC model, the input and output values are passed as function arguments and the functions have return type of void. The provided output variable is a reference to the actual value object in the memory that is used by the ForSyDe-SystemC library and at the end of the execution of the process the result is written into the output FIFO. Listing 3.1.1 shows the function prototype expected by ForSyDe.

```cpp
void adder_func ( std::vector<double>& out1 ,
    const std::vector<double>& inp1 ,
    const std::vector<double>& inp2 ) {
#pragma ForSyDe begin adder_func
    out1[0] = inp1[0] + inp2[0];
#pragma ForSyDe end
}
```

Listing 3.1.1: Sample ForSyDe function used to model a process

When the software compilation flows of most design flows are inspected, it can be seen that there is an important fact that should not be missed and it is that most vendors providing processing platforms only provide C compilers for them or even if there are C++ compilers they may not be suitable for the system designed because of the system’s constrained environment. Hence a subset of the C++ language may have to be used by the designers to ensure portability among different platforms. When modeling the behavior of a process in ForSyDe, especially in view of a possible software synthesis step in the design flow, it is required from the designer to mark the beginning and the end of the synthesizable part of the functions using C++ pragma preprocessor. This is especially useful in automating the software synthesis phase.

Simulation: Process constructors are implemented as C++ classes and they are derived from the sc_module class of SystemC. Technically, all processes in ForSyDe models are individual SystemC processes. The channels are
implemented as classes derived from the `sc_fifo` class of SystemC. All process constructors are executed in a five stage flow as illustrated in Figure 3.2. In the \textit{init} stage, the process constructor initializes the data structures needed for the process. In the \textit{prep} stage, the inputs to the process are read from the input channels of the process. In the \textit{apply} stage, the process is executed; for processes with behavior, this involves calling the function provided by the designer. In the \textit{prod} stage, the output produced by the \textit{apply} stage is written to the output channel. After this stage, the execution flow normally continues with the \textit{prep} stage until the end of the simulation and the execution continues by iterating this loop. At the end of the simulation, the \textit{clean} stage is initiated and the internal data structures created by the process constructor are destroyed.

![Fig. 3.2: Illustration of the simulation steps ForSyDe process constructors execute.](image_url)

3.2 Introspection

While ForSyDe simulation is enough for functional verification, for possible hardware/software partitioning and mapping phases in the design flow, derivation of the structural information about the process network is crucial. ForSyDe allows extracting the process network of the system model by introspection, and the generated topology is expressed in an XML file.

Introspection is achieved by overriding the `end_of_elaboration` method of the `sc_module` class belonging to SystemC. When SystemC simulation kernel executes this callback method after the elaboration phase, ForSyDe collects information about the process name, function name it executes and consumption and production rates of inputs and outputs respectively. The function name is derived from the process name with all the trailing numbers removed and by adding a marker text \texttt{_func}. For instance, if the process name is given as `adder0` by the designer, the derived name could be `adder_func`.

34
3.3. **FORSYDE DESIGN FLOW**

Since ForSyDe allows design of hierarchical modules, the generated structure may span several XML files each representing the processes in one level in the hierarchy encapsulated by one composite process. Figure 3.3 illustrates the generated XML files for a given process network.

![Diagram of XML files](image)

*Fig. 3.3: Illustration of the generated XML files for a given process network*

### 3.3 ForSyDe Design Flow

ForSyDe itself is a modeling framework, however the y-chart approach is a suitable design flow in which ForSyDe can be used as an entry point to the design flow. The flow starts with the design of executable model of the system and the designer verifies the functional correctness of the system by simulation. Section A.1 depicts the overview of the design flow where ForSyDe acts as the modeling framework. The functional specification in C/C++ with the structural representation obtained by introspection allow the design space exploration phase to map the application to an architecture instance derived from a platform template.

A platform template is seen as a library of platforms with both micro-architecture and software that are parametrizable in order to instantiate a platform with specific performance characteristics. Currently, ForSyDe design flow does not have an established standard way to represent the platform templates. The design space exploration tool(s) iterates over the possible platform templates and their respective configurations to satisfy the design constraints provided by the designer. The design space exploration tool relies on the WCET results of processes acquired from either static code analysis or from
measurements. If design space exploration tool reaches a mapping that satisfies all the design constraints, the flow commences with the platform generation and software synthesis phase.

The software synthesis phase takes in a mapping description produced by the design space exploration phase, relevant platform API and libraries and the functional specification in C/C++ provided by the designer. The main objective of the software synthesis phase is to generate boilerplate code for the runtime environment, initialization and process execution for each processor in the platform in a way described in the mapping description. The platform generation phase synthesizes the hardware platform.
Chapter 4

The CompSOC Platform

This chapter introduces the CompSOC platform and its design flow. The CompSOC platform is built on the principles of predictability and composability as explained in [8]. The design flow of CompSOC consists of several parts as explained in section 4.4.

4.1 Predictability and Composability

CompSOC offers predictable and composable services by implementing the techniques discussed in [8], [33], [13] and [34] covering both hardware and software aspects of the system. As described in [34], the RTOS of CompSOC implements a two level scheduling of applications and tasks. The RTOS allows inter-application scheduling using TDMA for composability and intra-application scheduling responsibility is left to the task scheduler of the application, hence the application developer may choose whichever scheduling policy he or she sees fit. This decoupling allows the RTOS to be kept as small and coherent as possible by only switching between applications while enforcing composability and providing basic system functionalities for the applications. Each application is given its own virtual platform which is an abstraction of the underlying platform with a set of resource budgets allocated for resource requirements of that particular application.

4.2 Hardware Architecture

The overall view of the CompSOC platform hardware architecture is illustrated in Figure 4.1. This figure illustrates the processing tiles, memory tiles and
interconnect of the platform. The individual components in this overall figure are explained below.

Fig. 4.1: Illustration of the CompSOC architecture [8]
4.2. HARDWARE ARCHITECTURE

**Processor Tiles:** Processor tiles are the main processing elements in the CompSOC platform and they contain the micro processor Microblaze [23] from Xilinx, instruction and data memories, a configurable number of direct-memory-access (DMA) units, a timer-interrupt-frequency unit (TIFU) and a pair of communication memories allocated per DMA. Processor tiles also contain bus elements to connect to the interconnect. To guarantee predictability, Microblaze cores do not have memory caches and DMA units are not shared across applications. Hence, each FRT application requiring a DMA unit for communication must be given its own DMA unit. Composability is achieved by two-level application and task scheduling as explained in section 4.3.

**Interconnect:** The interconnect consists of mainly protocol shells, network interfaces and routers. Protocol shells serialize the parallel protocol to a sequence of words. Network interfaces (NI) determine the route and packetize (or depacketize) the data to be transferred through the network into a number of flits. Each packet starts with a header flit with routing information so that routers forward the packets according to this route. NIs are responsible for scheduling and buffering packets. To achieve composability, NIs schedule every flit independently to make transactions preemptive, and every flit has a constant size of three words to ensure constant worst case scheduling interval and finally, each NI injects flits using a TDM arbiter such that no contention occurs in the network. As explained in [35], the temporal behavior of an interconnect connection can be included in the data-flow graph of an application, allowing it to be analyzed.

**Memory Tiles:** Memory tiles are divided into two parts, namely front-end and back-end. The front-end contains a number of blocks to achieve composability while the back-end guarantees the predictability of the resource [8]. Different kinds of memories require different back-end implementations. SDRAM back-end is realized by making use of predictable memory patterns [36] and combining them at run-time. At the front-end, Credit-Controlled Static-Priority (CCSP) arbiter [37] is used to achieve predictable arbitration. Atomizers enable the front-end to be able to preempt a requester who may issue large requests, by splitting the requests into smaller atomic service units that can be served by the resource in bounded time. Delay blocks allow composability by scheduling execution of requests with an interval equal to the WCET.
CHAPTER 4. THE COMPSOC PLATFORM

4.3 Software Platform

For inter-application and intra-application scheduling, CompSOC uses two level scheduling as explained in [34]. First level, the inter-application scheduler, allows the platform to execute applications composably, while the second level, intra-application scheduler (task scheduler), supports executing application tasks with semantics of a particular MoC. Having a separation of these two levels allows the RTOS implementation to be minimalistic, coherent and decoupled from the execution semantics of different applications. In fact, the intra-application scheduler is not enforced and the designer is free to implement the application as he or she sees fit with or without a scheduler.

The first level scheduler in CompSOC platform is a part of the minimalistic RTOS implementation named CoMik. In CompSOC, applications are given a set of resource budgets, which are referred to as virtual platforms. Applications are scheduled in TDM fashion and individual TDM time-slots for applications are called partitions. An application may be given multiple partitions to ensure its real-time guarantees. Each virtual platform executes composably and may use basic RTOS calls to issue requests. Figure 4.2 illustrates the software platform of CompSOC.

At the initialization phase, applications register three functions for CoMik to run the partition code, handle partition interrupts and exceptions. CoMik exposes a partition-programming-interface (PPI) to partitions to access underlying resource managers and drivers. At the boot time, the partitions are created, partition scheduler is initialized and the micro-kernel is started by the user provided code (this code can be generated by the software synthesis tool genapp). After this stage, the timer interrupts determine the execution flow. If an incoming interrupt is detected to be a system interrupt, then the partition scheduler is called to schedule the next partition for execution. Otherwise, if the interrupt is directed to the currently executing partition, the partition interrupt handler registered by the application is invoked. System interrupts are handled in a non-interruptible fashion, whereas partition interrupts are interruptible once the control is dispatched to the partition interrupt handler function.

A user library, called libcompose, is provided for the designer to choose from existing task schedulers to facilitate the required execution semantics. Based on the triggering mechanism of application tasks, the designer can choose one of four task schedulers implemented in libcompose. Static-order and round-robin task schedulers rely on cooperation of tasks and they are non-preemptively executed in the partition time. They are suitable for data-flow applications with MoCs such as KPN and CSDF. On the other hand, time-triggered domains may use preemptive schedulers, namely TDM and fixed-priority schedulers.
CSDF applications are scheduled according to the firing rules given by the designer of the application. Tasks with multiple input and output edges are supported. Eligibility of a task’s execution is determined by the availability of its required number of tokens at every input edge. Each task requires differing firing sequences in consecutive iterations of the data-flow graph.

CompSOC platform also supports execution of time-triggered applications although mapping of these applications is not automated yet. Time-triggered application tasks do not need firing rules but they require task properties, such as period, priority and deadline, to be provided by the designer. The realization of composable execution of time-triggered applications is achieved by interrupt virtualization.

As explained before, applications in CompSOC are given a set of resource budgets, one of which is the power budget. By application’s cooperation, different power management policies can be put in action to allow applications to manage their allocated power or energy budgets independent of other
CHAPTER 4. THE COMPSOC PLATFORM

applications. The realization of composable power management is achieved by keeping track of power budgets and providing a number of power management policies to applications; the details can be found in [38].

4.4 CompSOC Design Flow

CompSOC design flow is partitioned into several sections, namely hardware generation flow, mapping flow and software compilation flow. A number of input files supplied by the designer are needed to start the design flow. These files are communication and architecture model of the platform, application model and the associated source files to be compiled. The overview of the flow is given in section A.2.

Hardware generation flow takes in the communication and architecture models of the platform as inputs, where communication model is expressed as connections between applications and the architecture model is expressed as processing tiles in the platform. Five main steps in hardware generation flow are design dimensioning, allocation, verification, instantiation and synthesis. The first four of these steps are explained in detail in [39]. The fifth step involves creating a project environment for the designer to develop the applications and synthesizing the platform. The hardware generation flow generates all the necessary interconnect related IP blocks and allocates resources to satisfy requirements of the applications. It outputs the final architecture model of the platform, and additional platform description files used by the Xilinx tool-flow and the compilation flow.

The mapping flow involves design space exploration and software synthesis tools. It takes in the SDF graph of the application and the final architecture model of the platform and produces a mapping of application tasks to the tiles in the platform. The software synthesis tool uses this information to generate the initialization code for each tile to be used by the compilation flow.

Compilation flow is invoked consecutively for each tile in the platform and code compilation for each tile includes generation of a static library (per tile) by invoking the Xilinx tools, compilation and linking of source files for the tile and inclusion of libcompose to generate the binary file (ELF) for the tile. At the end of the flow, the generated binary is merged with the platform bit-stream file generated by the hardware generation flow.
Part II

A Design Flow for Predictable Composable Systems
Chapter 5
Stepwise Development of the Design Flow

This chapter presents the development process in a constructive manner by introducing the necessary adaptations and tools in the design flow step by step. At each step, the abstraction level is lowered until the individual tools and the interconnections between them are identified. In doing so, the mismatches between the ForSyDe design flow and the CompSOC design flow are identified and the motivations for developing the introduced tools are presented.

From a high level perspective, the missing part that connects the design flows of ForSyDe and CompSOC, can be seen as the adaptation layer shown in Figure 5.1.

![Figure 5.1: High level view of the ForSyDe-CompSOC design flow](image)
CHAPTER 5. STEPWISE DEVELOPMENT OF THE DESIGN FLOW

The adaptation layer can be seen as an abstraction for the tools that need to be developed for integrating the ForSyDe and CompSOC design flows.

Mismatch 1 - Modeling language and source files: The modeling language of ForSyDe is C++, however the CompSOC software design flow requires the functional models to be in the C language. Furthermore, additional source files need to be generated to integrate the converted ForSyDe models with the CompSOC RTOS.

Mismatch 2 - Structural representation: The structural representation acquired by the ForSyDe simulation contains information about the process network of the ForSyDe SDF models. However, the format used for this representation is not the same format used in CompSOC. Additionally, the process network contains ForSyDe process constructors such as zip, unzip, delay, fanout, source and constant. These processes are not trivially convertible to their purely SDF equivalents.

5.1 Step 1: The Conversion Tools

In light of mismatch 1 and mismatch 2, a closer look at this adaptation layer reveals that two conversions need to be performed.

1. ForSyDe C++ source files need to be converted to C files that CompSOC software flow can compile. Additionally, the integration source files need to be generated.

2. The ForSyDe SDF process network represented in an XML format needs to be converted into the CompSOC SDF application representation in XML.

Tools performing these conversions are represented as individual components in Figure 5.2.

There are several aspects of these conversions that need to be taken into consideration.

Firstly, the code generator tool needs to take in a directory path where the ForSyDe C++ source files are and scan these files to find all the functions that belong to the processes in the structural representation of the application model. Additionally, the necessary data structures and additional functions in the dependent source files need to be found.

After these functions and data structures are found, the code generator tool needs to produce code for the processes in the target SDF graph. In addition, the code generator needs to know about the data types of the tokens, initial token values, function names for processes and the kinds of processes in the
5.1. **STEP 1: THE CONVERSION TOOLS**

![Diagram](image.png)

Fig. 5.2: The code conversion and process network tools in the adaptation layer

process network. Hence, it is evident that this step cannot start without knowing the result of the conversion from the application’s process network to the SDF graph.

The code generation can be performed in several ways including source-to-source transformation from C++ to C. However, this approach is problematic in several regards.

- Even though, tools exist to transform C++ code to C code, the generated C code may still have dependencies to C++ constructs in external libraries. This would cause the parts of these libraries to be included in the produced binary, hence unnecessarily increasing the memory usage of the application.

- The C files produced as a result of using source-to-source transformation are not maintainable by humans. This also complicates the software synthesis step later, where specific function blocks or top level blocks are put together for a specific purpose.

Another way to achieve the same goal would be to use well-defined conventions in the design and use the C language for the parts that can be mapped to the target platform. Most importantly, this approach is safe and as long as the dependencies of the functions are correctly specified by the designer the program is guaranteed to have the functional correctness. This approach also makes it easy to produce a composition of the actor functions to produce the C files for the CompSOC platform.
The conventions to design the ForSyDe models must allow easy identification of the source blocks that can be used to generate the C files for the target platform. Furthermore, the design process must be supported with helper utilities that allow generation of the C files without performing any transformations on the code blocks.

Secondly, the process network conversion tool needs to take in a set of XML files generated during the ForSyDe simulation and it needs to convert these process networks into the SDF graphs that can be used by the mapping tool of CompSOC. A comparison of the information available in the ForSyDe process network and the CompSOC SDF representations yields the mismatches and the missing information to construct the SDF graph.

A CompSOC SDF graph contains:

- Actors
- Function name of each actor
- Channels connecting the actors
- Production and consumption rates of each actor
- Number of initial tokens in channels
- Actor memory requirements for data and instruction sections
- Sizes of token data types
- Worst case execution times of actors

On the other hand, a ForSyDe process network contains information about the following:

- Processes
- Process names
- Function names of the processes
- Ports for the modules
- Signals that connect the processes
- Consumption and production rates of the processes
- Values of the initial tokens in a string format
5.1. **STEP 1: THE CONVERSION TOOLS**

- Names of the data types for tokens

By comparing the information available in these two representations, it is evident that the information available in the ForSyDe process network is not sufficient to construct the SDF graph expected by the CompSOC mapping flow.

**Mismatch 3 - Missing information about the number of initial tokens in signals of ForSyDe SDF models:** The CompSOC SDF graph requires the number of initial tokens in the channels. However, the ForSyDe process networks do not contain this information.

Although, the ForSyDe process network does not include the number of tokens, it includes the values of the initial tokens in a string format. However, due to the lack of reflection capabilities in the C++ language, this information is not always generated correctly for tokens of uncommon types or user defined struct types. If this information can be generated by the introspection in an analyzable way, it can be used to extract the individual token values and also for determining the number of initial tokens in signals. Since the CompSOC software synthesis step relies on the initialization functions to produce the initial tokens in the channels, the extraction of the initial token values is especially important for the code generation tool. Hence, these initialization functions need to be generated by the code generation tool. As a byproduct, the code generation tool can produce the information about the number of initial tokens.

**Mismatch 4 - Missing information about properties of processes and channels:** The CompSOC SDF graph needs to have some properties of actors and channels such as WCET and memory requirements of actors and token data sizes for each channel. Moreover, these properties are dependent on the architecture of the target platform and the compiler used to produce the executable for the target platform. The ForSyDe process network is generated as a result of the simulation run on the designer's workstation. Hence, the platform dependent information about processes and channels cannot be produced by the ForSyDe simulation.

Since the CompSOC SDF graph needs to have these properties, to be able to produce a mapping that satisfies the real-time constraints of the system, the information about these properties needs to be found. The required properties of the actors and the channels are:

- Memory requirements of each actor
- Token data size of each channel
- Worst case execution time of each actor

As can be seen from these mismatches, the code generator and the process network converter have interdependencies and the process network conversion
tool needs information about the implementations of the actors and channels. However, since the ForSyDe representation contains all the information needed to convert the topology, it is possible to perform the transformation in a partial way. Therefore, the conversion tool can produce an intermediate representation of the SDF graph containing the topological information.

Furthermore, assuming the missing properties are found at a later stage, the generation of the SDF graph for CompSOC mapping tool can be done in multiple phases where each phase annotates the necessary properties in the SDF graph.

5.2 Step 2: Back-annotation Phases

Although, they depend on each other’s behavior, the code generation and process network conversion can be performed by splitting the overall conversion process into multiple phases. At this step, the adaptation layer contains the blocks shown in Figure 5.3.

Fig. 5.3: The back-annotation phases in the adaptation layer

At this point, assuming the code generator produces the C files for the actors and initialization functions, the missing parts in the adaptation layer are the sources of the platform dependent properties of the SDF graph.
5.2. **STEP 2: BACK-ANNOTATION PHASES**

**The token data sizes** can be found by analyzing the binary file produced by the CompSOC software synthesis step. DWARF [40], is a standardized debugging information format used to support source level debugging of several languages, and its specification has been implemented by many compilers and associated tool-chains including the GNU tool-chain.

GNU tools can produce the DWARF information in a text format from a compiled object file or a linked binary. This produced output contains detailed information about the functions, variables, primitive and user defined data types and their architecture specific properties, such as the size in bytes. Hence, this information can be used to determine the data sizes of tokens in the SDF channels.

More information about the extraction of the token data sizes can be found in section 6.5.

**The memory requirements of the actors** can be found by invoking GNU tools to produce information about the memory sizes of functions in an object file or a binary. However, finding only the size of the actor function may not be sufficient to find the correct memory requirements of that actor. If the actor function calls other functions, the compiled binary would also contain the instructions and data for the functions the actor depends on. Hence, these dependencies must be found as well and the actual memory requirements for an actor would be the sum of the function sizes of these dependencies.

These dependencies can be found by finding the call-graph of each actor. The function call-graph can be generated statically from the source code or from the binary, or it can be generated dynamically when the application is run. However, since at this point the application cannot be run on the platform, dynamic generation of the call-graph is not a trivial task. To limit the complexity of generating the call-graphs, the static call-graph generation can be used. For source codes in programming languages with dynamic dispatching mechanisms, the statically computed call-graphs are approximations of the actual call-graphs. Therefore, to be able to find the precise call-graphs for the actors, the use of such mechanisms should not be allowed in the modeling of the applications.

Clang [41] and LLVM [42] tool-chains can produce the call-graphs in DOT format from the source files of the applications. This file can be used to determine what the dependencies of each actor are, and then by using the GNU tools to find the memory sizes of each dependency, the actual memory requirements of each actor can be found.

More information about the extraction of the actor memory requirements can be found in section 6.4.
The worst case execution time of actors can be found by measurement or static analysis and a general overview of the techniques in determining WCET values is given by Wilhelm et al. in [43]. To acquire the WCET values statically, the precise knowledge of the timing behavior of the system is required to find bounds on the execution times of the actors. Additionally, the control flows of the actors must be analyzed to determine the paths contributing to the WCET of the actors. On the other hand, the measurement based techniques give the most accurate results for WCET of the actors.

Since the CompSOC platform features the system level properties of predictability and composability, a measurement based technique would produce the same execution values for an actor in different runs. In a measurement run, an actor may execute multiple times and the case where it takes the longest time to complete its execution can be considered as a candidate for WCET value of that actor. However, since the execution flow might not have gone through the longest possible execution path in the application, the precision of this measurement may not be perfect.

To rectify this situation, the measurement based approach can be mixed with static methods where the actor source code or the binary can be analyzed to determine the longest possible execution path. A less precise, yet simpler alternative to this solution can be to introduce a margin of error to the measurement results. This is a common practice in the industry and it does not involve the complexities of the hybrid solution mentioned above. However, this solution may require manual intervention of the designer to reduce the overestimated WCET values for optimization or increase the underestimated WCET values for guaranteeing the real-time constraints.

The CompSOC platform produces an execution log of the system after each system run on the FPGA prototype. The actors can use the RTOS functions to write debug information to this execution log. During a measurement run, the actors can write the system time value or a counter value at the beginning and at the end of the actors’ executions. This execution log can be analyzed to find the execution times of the actors. Among the executions of each actor, the greatest execution time can be found and with an addition of an error margin, the approximate WCET value can be obtained.

Finally, to put together all of the above mentioned ways to find the memory requirements and execution times of the actors, the adaptation layer must be provided the binary file and the execution log file. Connection of these files to the adaptation layer is illustrated in Figure 5.4.

After adding the units to extract the platform dependent properties the adaptation layer can be seen as illustrated in Figure 5.5.
5.2. **STEP 2: BACK-ANNOTATION PHASES**

Fig. 5.4: The connection of the execution log and the binary file to the adaptation layer

Fig. 5.5: Addition of the platform dependent property extraction units in the adaptation layer
5.3 Step 3: ForSyDe Helpers

To make the process network conversion and the code generation possible, the process network produced by simulating the ForSyDe models, must contain specific information about the initial values and the token types. Furthermore, as explained in section 5.1, the designer must be given utilities to model the applications in a way that the developed functions for the actors are extracted from C++ source files without performing any transformations. Hence, for these reasons the ForSyDe models and introspection must be supported by several helper utilities. These helper utilities are shown as the ForSyDe helper utilities box in Figure 5.6 and with this new addition the adaptation layer is completed.

The most important utility that helps to model the applications in a C-compliant way is the signal flattener utility used to access tokens in ForSyDe signals.

**Signal Flattener Utility:** When the application code is generated for the CompSOC platform, the accesses to the input and output channels must be done
in an array like syntax. However, since input and output tokens of ForSyDe processes are encapsulated in vectors, tuples and nested combinations of these, the designer is normally forced to use C++ helper functions to get the tokens in these data structures.

This, however, results in actor functions with C++ function calls, which contradicts the assumption that the source code developed by the designer is easily transferable to C files.

To avoid performing any transformations on the source code, the designer must be provided with a utility that would allow accessing the input and output tokens in the same way they are accessed in the CompSOC platform. This utility can be implemented using C++ template meta-programming techniques to provide a type safe interface to access the tokens in the input and output arguments of the actor functions. The use of the helper utilities is demonstrated in section 6.1.

### 5.4 Step 4: Automation

The adaptation layer illustrated in Figure 5.6 shows the flow of information between different components involved in producing the source files and the SDF graph for the CompSOC software flow. However, for the automation of the design flow, these components and the existing CompSOC design flow tools need to be used in a specific order to produce these outputs.

The high level view of the automated design flow can be seen as shown in Figure 5.7.

![Fig. 5.7: The high level view of the automated design flow](image-url)
The design flow consists of three main phases each producing an artifact needed for the next phase in the design flow. The automated design flow allows multiple applications to be mapped to the target platform. Each application in the system goes through the first two phases of the design flow to generate the final CompSOC SDF graph representation. In the final phase, all of the applications are mapped to the platform and the final binary that can be executed on the platform is produced.

Since CompSOC offers composability, the independent way of finding the missing property values in the SDF graphs is possible. Otherwise, all of the applications would have to be merged at the beginning of the design flow.

- **Initial Build:** The initial build phase takes in the ForSyDe models for an application, performs the conversion operations with constant values for platform dependent properties and maps the actors to a single processor. This mapping is used to generate the executable binary file for the application.

  Since the platform dependent properties are only constant values, the produced mapping is not optimal. However, the produced binary can be analyzed for extracting information about the memory requirements of the actors and the token data sizes.

- **Measurement Run:** The binary produced and the source files generated for the application at the end of the first phase are analyzed to find the memory requirements of the actors and the token data sizes. This information is back-annotated to the SDF graph obtained in the first phase.

  A new mapping is produced using this enriched SDF graph. This time, however, the mapping produced may be a multi-processor mapping. For this mapping, the CompSOC flow produces a new binary and the application is run on the platform for measurement of the execution times of the actors.

- **Final Build:** Once the execution times of the actors are obtained, these values can be back-annotated to the SDF graph generated in the second phase. At this point, all the applications have been run to obtain their final SDF graphs. Thus, it is possible to merge the SDF graphs of all of the applications into a single CompSOC SDF graph representation. Hence, the SDF graph for the entire system is given to the mapping tool one last time to produce the mapping of actors from multiple applications to the platform. As the final step in the design flow, the binary that can be run on the platform is generated.

The complete automated design flow is illustrated in Figure 5.8. The design flow can be invoked via the designer's workstation. Since the design flow can
access the remote development server where the CompSOC tool-flow is installed and configured, the designer’s workstation does not have to have the CompSOC related tools.
Chapter 6

Tool Development

6.1 Modeling ForSyDe Processes with the ForSyDe Helper Utilities

This section presents the use of the ForSyDe helper utilities by a simple use case. In Figure 6.1, a simple hierarchical ForSyDe SDF process network with zip and comb processes is given.

Zip processes have their own consumption rates for their inputs as specified in Figure 6.1. Since the zip processes produce tokens of type tuple, the input channel of the process K has tokens with nested tuples. The implementation of the process K is given in Listing 6.1.1.
The C++ \texttt{pragma} preprocessor directives are used to mark the beginning and the end of the code blocks that are eligible for code conversion. The designer is free to use C++ constructs outside of these code blocks. However within these blocks the C++ syntax and calls to C++ functions are not allowed.

As can be seen, this function has two input arguments, \texttt{inp1} and \texttt{inp2}. These arguments are used by the \texttt{FLATTEN\_INPUTS} macro to allow access to the tokens in the nested structure in a linear fashion. On line 18 in Listing 6.1.1, it is demonstrated how the tokens in the input channels are accessed and how the tokens are written to the output channels. On this line, the last read input token is from the channel 3, which corresponds to the \texttt{inp2} argument.

The introspection helpers must be used when an uncommon or user defined type such as \texttt{structs} are used in the design. Assuming the token type of the channel connecting \texttt{z2} and \texttt{K} is \texttt{ZippedToken}, the designer introduced definition on line 21 in Listing 6.1.2 helps producing the type information in the format shown on line 4 of Listing 6.1.3. This type information is then used in the process network to SDF conversion. Similarly, the \texttt{struct} definition on line 28 in Listing 6.1.2 allows the introspection to put the content of an initial token on the line 10 of Listing 6.1.3. This is of vital importance since the code generation tool is going to use this information for generating the functions that CompSOC uses to inject the initial tokens into the channels.
6.1. MODELING FORSYDE PROCESSES WITH THE FORSYDE HELPER UTILITIES

Listing 6.1.2: Additional definitions and a top level block

```c
// A top level code block for code generation
#pragma ForSyDe begin
typedef struct {
    double d;
    char *n;
} InternalStructType;
typedef struct {
    char a;
    float f;
    InternalStructType i;
} UserType;
#pragma ForSyDe end
typedef std::tuple<std::vector<float>, std::vector<int>> T2;
typedef std::tuple<std::vector<T2>, std::vector<UserType>> ZippedToken;

// For correctly producing the type name information in the introspection output these signals must be defined
DEFINE_SIGNAL_TYPE(T2)
DEFINE_SIGNAL_TYPE(ZippedToken)

// These are needed for each uncommon or struct type correctly producing the initial token values in the process network
DEFINE_STRUCT_TYPE(InternalStructType) {
    os << "{" << obj.d << ", " << obj.n << "}" ;
}
DEFINE_STRUCT_TYPE(UserType) {
    os << "{" << obj.a << ", " << obj.f << ", " << obj.i << "}" ;
}
```

Listing 6.1.3: Parts of the process network produced with the help of the definitions given in Listing 6.1.2

```xml
<process_network name="top">
    ...
    ...
    <signal name="fifo_5" moc="sdf" type="float.int.UserType" source="z2" source_port="oport1" target="k" target_port="iport1"/>
    ...
    ...
    <leaf_process name="c">
        <port name="oport1" type="UserType" direction="out"/>
        <process_constructor name="constant" moc="sdf">
            <argument name="init_val" value="'(A, 23.3, {12, '012345678'})'"/>
        </process_constructor>
    </leaf_process>
    ...
    ...
</process_network>
```
6.2 Process Network to SDF Converter

The ForSyDe simulation produces the process networks of the modeled ForSyDe applications. However, this representation may contain some of the ForSyDe process kinds that are not SDF compliant. Moreover, some of these processes do not have corresponding equivalents in the SDF MoC. For instance, the ForSyDe processes such as `zip` and `unzip` are designed to deal with the limited number of input and output ports of ForSyDe process constructors. However, these constructs are normally not found in regular SDF models. The tools used in the design flow of CompSOC require this structural representation to be purely in SDF MoC. To be able to use these tools, the ForSyDe process network structure needs to be transformed into its CompSOC equivalent.

Both ForSyDe and CompSOC structural representations are essentially graph-like data structures expressed in XML format. However, the XML schemas used by two projects differ significantly. Therefore, in addition to the structural representations, these schemas also need to be transformed. For portability, and also for robustness against future changes in the representations, the conversion can be done on an intermediate graph representation. The PN-to-SDF converter can be visualized with three main parts as depicted in Figure 6.2.

![Fig. 6.2: The main parts of the PN-to-SDF conversion utility](image)

The first part in Figure 6.2 deals with the ForSyDe process network representation and converts it into an intermediate representation that is internally used by the conversion tool. This part allows the conversion tool to be adaptable to future representations of ForSyDe process networks. By only modifying this part to handle the possible changes, the actual conversion logic can be left unchanged.
Since ForSyDe supports hierarchical models, the introspection generates a process network for each module in the model. Each of these process networks is represented in a separate XML file. Hence, the process network of an application may span multiple XML files. To be able to easily process these hierarchical models that span multiple files, the hierarchy must be flattened into a single module.

The second part in Figure 6.2 is the part where the actual conversion of the ForSyDe process network is done. Detailed explanation on how this conversion is performed is explained in subsection 6.2.1.

The same internal graph representation mentioned above for an SDF graph can be used as the output of the actual conversion part; and at the end of the conversion, this intermediate SDF graph can be converted to the CompSOC representation. The third part in Figure 6.2 illustrates the step that converts the intermediate SDF representation to the CompSOC representation.

6.2.1 Conversion of the Process Networks to SDF Graphs

The process network to SDF conversion tool performs the transformations needed in multiple steps and these are illustrated in Figure 6.3.

- **Hierarchy Flattening:** The composite processes in the process network of an application are linked to other process networks in different XML files. The connections between the top level processes and processes inside of a composite process are represented as signals connected to the ports

---

Fig. 6.3: The steps involved in the process network’s conversion into an SDF graph

- **Hierarchy Flattening:** The composite processes in the process network of an application are linked to other process networks in different XML files. The connections between the top level processes and processes inside of a composite process are represented as signals connected to the ports
of the composite process. The hierarchy flattening is achieved by adding the processes and signals inside of a composite process to the top level process network. All the port information is discarded after finding the connections between the top level processes and processes added from the composite processes. Finally, all the composite processes are removed from the process network. As a result of flattening the hierarchical process network shown in Figure 6.1 the process network shown in Figure 6.4 is obtained.

![Image of Figure 6.4: The flattened ForSyDe model](image)

- **Convert to Internal Graph Representation:** The flattened process network is still a valid ForSyDe model. It still contains the original ForSyDe processes and connections among them. However, the data structure used to represent this graph needs to be converted to an internal representation so that possible future changes in the ForSyDe process network structure would not affect the entire conversion utility. This is also beneficial since the internal representation can have a more appropriate format for processing the graph elements.

- **Find Actor-to-Actor Paths:** In cases where *zips* or *combs* with *tupled* outputs are used in the ForSyDe models, the consumption and production rates given in the process networks do not reflect the actual amount of tokens exchanged between actors connected to each other via a path passing through one of the *zip* and/or *unzip* processes. For example in Figure 6.4, actor A is connected to actor K via the *zip* processes z1 and z2, which have consumption rates of 2 and 4 respectively. Actor A produces 1 token at each firing, and the actor K consumes 2 tokens before each execution. At the end of the conversion, this path connecting the two actors will become a single channel. However, since each of the *tupled* tokens in the original channel contains 8 tokens from the actor A and since
K consumes 2 of these \texttt{tupled} tokens, the consumption rate of the actor K from this channel must be in fact 16. This anomaly complicates the removal of the \textit{zip} and \textit{unzip} elements especially in combination with \textit{fanout} and \textit{delay} elements.

As a solution to this problem, all the paths connecting two actors in the graph are found. What is meant by the word actor here is one of the processes of kind, \textit{comb}, \textit{sink}, \textit{constant} or \textit{source}. When finding these paths between actors, the information about the ForSyDe processes occurring along these paths are also tracked. Using this information, the correct consumption and production rates of the actors can be found in a simple way as explained in subsection 6.2.2. This also allows simple removal of \textit{zip}, \textit{unzip} and \textit{fanout} elements.

The actor-to-actor connections and their expected SDF conversion results for the process network shown in Figure 6.4 are illustrated in Figure 6.5.

\begin{itemize}
    \item \textbf{Remove Zips and Unzips}: With the actor-to-actor path information, which also contains the processes encountered on the paths, the removal of the \textit{zip} and \textit{unzip} elements becomes as trivial as removing these elements from the paths in the actor-to-actor path list.
\end{itemize}
• **Remove Fanouts:** Just as in the case of *zip* and *unzip* removal, *fanout* removal also involves removing the *fanout* elements from the actor-to-actor path list.

• **Constant and Source Workaround:** The *constant* and *source* elements are different from other processes since they have state values of their own. A *constant* element has an initial token in its output, and a self edge for the state value of how many time this process must be fired. These anomalies can be solved by introducing a delay element at the output of this actor and creating a new actor-to-actor path with a delay element in its path where source and destination are the *constant* process itself.

The same method also can be applied to the *source* elements with the addition of the self edge for the current state of the output value. An important detail to notice is, if the *source* process produces *tupled* tokens, then a self edge for each of these individual outputs must be added to the actor-to-actor paths.

• **Reconstruct the Internal Graph Representation:** After removing the *zip*, *unzip* and *fanout* elements from the actor-to-actor paths, the internal graph can be reconstructed using the information available in the initial graph created using the flattened ForSyDe process network. At this step, the delay elements are preserved in the graph representation since this information is going to be used by the code generator tool. The produced graph representation is then supplied to the code generator to produce the source files and the information about the number of initial tokens in the channels.

• **Remove Delay Elements:** Once the numbers of the initial tokens are found, the delay elements in the graph can be removed.

• **Convert to CompSOC:** At this step, the graph contains all the information necessary to convert it to the CompSOC SDF graph representation. Although the implementation properties are not available at this time, they can be assumed to have constant values to produce the CompSOC SDF graph. This produced initial CompSOC graph is then used in the automated design flow.

The result of the conversion for the process network shown in Figure 6.1 are illustrated in Figure 6.6
6.2.2 Determination of the Consumption and Production Rates

Since the ForSyDe models contain zip and unzip processes the production and consumption rates of processes given in the process network XML files may not be correct when the zip and unzip elements are removed from the design. The implemented way to find these values is by finding the actor-to-actor paths between processes in the network. Once the actor-to-actor links are known the production and consumption rates can be found with the following algorithm.

**Algorithm 1** Determination of the consumption and production rates

\[
\begin{align*}
\text{a2a\_links} & \leftarrow \text{all actor-to-actor links} \\
\text{for all } \text{link} \in \text{a2a\_links do} \\
\text{zip\_factor} & \leftarrow \text{multiply(zip rates in link)} \\
\text{unzip\_factor} & \leftarrow \text{multiply(unzip rates in link)} \\
\text{channel\_factor} & \leftarrow \text{zip\_factor div unzip\_factor} \\
\text{if channel\_factor} > 1 \text{ then} \\
\text{target\_rate} & \leftarrow \text{target\_rate} \ast \text{channel\_factor} \\
\text{else if channel\_factor} < 1 \text{ then} \\
\text{source\_rate} & \leftarrow \text{source\_rate} \ast (1/\text{channel\_factor}) \\
\text{else if both actors' ports are tuples then} & \\
\text{Report error: Unbalanced zip and unzip sequence} \\
\text{else} & \\
\text{Keep the original source and target rates} \\
\text{end if}
\end{align*}
\]

67
6.3 Code Generation

The code generation utility must deal with the application source code to transform it into a C compliant form that can be used by the compilation flow of the CompSOC. It also needs to generate some additional files to be used by the rest of the design flow.

Fig. 6.7: The functional blocks in the code generator tool

The code generation tool can be split into several functional units as depicted in Figure 6.7.

- **Read SDF Graph**: The intermediate SDF graph generated by the PN-to-SDF conversion utility must be provided to the code generation utility to be able to extract the relevant parts of the application source code. This graph in XML format is given as an input to the code generation tool and it is used as the source of information for actor properties and initial token values. This information is provided to the other functional units inside the code generation tool.

- **Find Token Values**: The initial token values in ForSyDe process network XML files are represented in their serialized forms. For this to be possible, the designer needs to provide some definitions in the source code so that ForSyDe can extract this serialized representation of the initial tokens by introspection. This serialized representation can then be parsed to acquire
the values given by the designer. This functional unit parses the initial token information given as delay processes in the SDF graph to produce the information about the initial token values and the number of initial tokens in channels.

• **Create Constants:** The constant ForSyDe processes are not explicitly implemented by the designers. These process constructors are implemented by the ForSyDe library. However, when generating code for the target platform, these processes need to be implemented by the code generation utility. This functional unit is responsible for generating the implementation for such actors in the SDF graph provided. Since a constant process produces constant tokens, the SDF graph has the serialized version of this token value. This information is given to this functional unit by the unit *Find Token Values*.

• **Make Function:** The application source files are annotated by the designers to mark the beginning and the end of relevant code sections that should be compiled by the CompSOC compilation flow. These code sections are identified and are put into the generated source files. Some additional declarations for accessing input and output channels are also added to the function bodies.

• **Add Fanouts:** The functions generated by the unit *Make Function* may also require some additional output channels after the PN-to-SDF conversion. These new channels may be needed due to the *fanout* processes introduced by the designer. The *fanout* processes allow multi-casting of tokens to multiple destination actors. They can be eliminated completely from the final SDF graph. To achieve this, the tokens to be multi-cast must be copied to the input channels of the actors that were originally connected to this originating actor via *fanout* processes.

• **Create a Header File:** The CompSOC design flow generates the boilerplate code for initialization of the processing elements of the MPSOC. And by convention, this code requires the existence of a C header file that contains the forward declarations for the actor and initialization functions along with the necessary type information that may be needed for the compilation of the boilerplate code generated. Since the SDF graph contains the names of the actors and the function names for them, this information by itself would be enough to generate the forward declarations in the header file. The type information can be obtained from the source files given as input.
• **Write Tokens and Types:** The final phase of the PN-to-SDF conversion requires the number of initial tokens which can only be found by parsing the serialized form of the initial token values. The parsing is done by the *Find Token Values* unit. For each channel in the SDF graph, the number of initial tokens and the type of the tokens are written by this block.

• **Create Initialization Functions:** The CompSOC design flow handles the injection of the initial tokens in a channel by calling that channel’s initialization function that is specified by the designer. Essentially, this function copies the initial token values into the buffer area reserved for the channel. The code generation tool needs to create these initialization functions for the channels that contain initial tokens and these functions are put into a single C file.

The generated code for the implementation of the actor K in Listing 6.1.1 is given in Listing 6.3.4.

```
void k_func () {
    FLATTEN_INPUTS();
    FLATTEN_OUTPUTS();
    INPUT_VAR(0, float);
    INPUT_VAR(1, int);
    INPUT_VAR(2, UserType);
    OUTPUT_VAR(0, float);
    OUTPUT_VAR(1, float);

    // Inputs are read using I(S, N),
    // S: the index of the channel
    // N: index of the token in the buffer
    // Outputs are written by assigning to O(S, N)
    O(0, 0) = process_1(I(0, 0), I(1, 0), I(2, 0), I(3, 0));

    // Fanouts
    for (int __k = 0; __k < 1; __k++) {
        O(1, __k) = O(0, __k);
    }
}
```

Listing 6.3.4: Code generation result for the actor implementation in Listing 6.1.1

### 6.4 Determination of the Actor Memory Requirements

The SDF graph needed by the CompSOC design flow must contain the memory requirements of the actors to be able to map these actors to the processing
6.4. DETERMINATION OF THE ACTOR MEMORY REQUIREMENTS

elements in an efficient way. The memory requirements of the actors affect the placement of the actors on the processing elements and ultimately this influences the real-time characteristics of the application. Therefore, it is important to find the correct amount of memory needed by an actor.

The memory size of a function can only be found after compilation of the source files and linking of all software components into the executable binary. The size information for functions and other symbols in an executable can be found using the output generated by the GNU command-line tool \texttt{nm} as shown below.

\begin{verbatim}
$ mb-nm --print-size --size-sort --radix=d binary.elf
...
00270604 00000001 b initialized
00270605 00000001 b last_MCU
00269776 00000004 D heap_start
00022636 00000796 T splitIntoMCUs
00019756 00002720 T susanUsan
00023412 00003552 T takeMCU
00015372 00003888 T stitch
00026964 00005896 T susanThin
00006872 00008116 T susanDirection
...
\end{verbatim}

For each symbol in the executable file, this output contains a line of text indicating the start address, the size and the type of the symbol. In the output, the lines containing the letter \texttt{T} on the type column indicate the symbols or functions in the text section of the binary. Using this output, the memory requirements of the functions can be found.

However, most actor functions actually contain calls to other functions as well. Finding the size of the actor function alone would yield an incorrect result. Therefore, all of the functions required by each actor must be found. One way to find this information is to determine which functions are called by each actor function. This information can be found by obtaining the call-graphs of actor functions. Call-graphs contain the dependencies of the actor function on other functions in the application. Once these dependencies are found, the aggregate size of all dependent functions would yield the correct size information.

The call-graphs are generated by analyzing the source files using the tool-set of the LLVM, specifically \texttt{clang} and \texttt{opt}. \texttt{Clang} is used to convert the source files into the LLVM intermediate representation (IR) as shown below.

\begin{verbatim}
$ clang -S -emit-llvm susan.c -I ../libcompose/include/ -o susan.c.bc
$ opt -analyze -dot-callgraph susan.c.bc
\end{verbatim}
The generated IR can then be used by the LLVM tool \texttt{opt} to generate a call-graph of the original source file. The generated call-graph is in \texttt{DOT} format and an example call-graph generated is shown in Figure 6.8.

![Call Graph](image)

**Fig. 6.8: An example callgraph generated by opt**

The call-graph generated contains extra information that is not needed for determining the function size of the actors. In the call-graph illustrated in Figure 6.8, the actual actor function is the \texttt{wrapUp} function. However, there are some additional nodes such as the \texttt{external node}, \texttt{Node0x88f680}, \texttt{llvm.memset}, and \texttt{llvm.memcpy}. These nodes and their arcs are discarded when finding the memory requirements of the actor functions.

Once the call-graph of each source file is generated, these call-graphs are traversed to find all the dependencies of actor functions. Hence every actor has a set of functions it depends on and this information is used to find the aggregate memory size needed for each actor.

An important limitation is that the implemented solution only deals with the code size of the functions. However, for more efficient actor to processor mappings, the exact requirements from the memory need to be determined. The implication of this limitation is that the designer is discouraged from the use of global and static variables in the models. The global and static variables are normally placed in the data section of the executable binary.

Another limitation of this way of determining the memory requirements of actor functions is that, this method relies on explicit calls to functions. Since the call-graphs are generated by statically analyzing the source files, in cases where dynamic dispatching methods are used, the call-graphs generated are not accurate. For example, the use of function pointers to invoke functions is a disruptive design effort when the target of the call is important to know at design time.
6.5 Determination of the Token Sizes

The CompSOC design flow relies on the information provided in the SDF graph for the token sizes in bytes. The CompSOC design flow uses this information to generate the boilerplate code for initialization and allocation of memory buffers for the channels. Therefore, the memory sizes of the tokens in the channels must be determined.

The token sizes are essentially the size of the data type of the tokens and this information is platform dependent, hence it cannot be automatically found at design time for user defined types such structs and unions. It can be found after the compilation of the source files by analyzing the binary produced using the GNU tool readelf. The readelf command line tool can generate the debugging information in the DWARF format. The command used and the produced DWARF output is shown below.

```
$ mb-readelf -wi binary.elf
...
<i><231>: Abbrev Number: 2 (DW_TAG_base_type)
  DW_AT_name : long int
  DW_AT_byte_size : 4
  DW_AT_encoding : 5 (signed)
<i><2e2>: Abbrev Number: 2 (DW_TAG_base_type)
  DW_AT_name : unsigned char
  DW_AT_byte_size : 1
  DW_AT_encoding : 8 (unsigned char)
<i><1458>: Abbrev Number: 7 (DW_TAG_structure_type)
  DW_AT_sibling : <147a>
  DW_AT_name : (indirect string, offset: 0xc0): EdgeDirection_struct
  DW_AT_byte_size : 1092
  DW_AT_decl_file : 11
  DW_AT_decl_line : 110
<i><147a>: Abbrev Number: 11 (DW_TAG_typedef)
  DW_AT_name : (indirect string, offset: 0xc0): EdgeDirection
  DW_AT_decl_file : 11
  DW_AT_decl_line : 115
  DW_AT_type : <1458>
...
```

As can be seen in output shown above, the base data types and user defined types are listed in this output. Information about each data type is given as a record in this output. The records for the base types, structure types and
the union types contain the data field `DW_AT_byte_size` which contains the size information for the particular data type.

The base types, user defined structs and unions may have multiple names caused by the use of the `typedef` keyword. These aliases are represented with the `DW_TAG_typedef` records in the DWARF output. For instance, the data type `EdgeDirection` has a `typedef` record in the output shown above. This record does not have the `DW_AT_byte_size` record but it has a reference to the record where the type is defined. The `DW_AT_type` field in this record indicates the address of the record where the actual data type is defined. Hence, if a given data type is an alias of another data type then the record for the original data type must be found to find the size of the given data type.
Chapter 7

Case Studies

This chapter presents the modeled applications and the experiments run to validate the ForSyDe-CompSOC design flow. The validation of the design flow is carried out using two case study applications. These applications are image processing applications which are suitable dataflow applications that can be developed using the design flow proposed in this thesis.

7.1 Application Modeling

7.1.1 SUSAN: Smallest Univalue Segment Assimilating Nucleus

SUSAN is a low level image processing approach and its theoretical foundations are given in [44]. SUSAN has three different algorithms, however the SUSAN algorithm used here is the one for the edge detection. Edge detection is an important image processing application used in many areas, such as medical imaging. Example input and output images for the edge detection algorithm are shown in Figure 7.1.

Fig. 7.1: Example input(left) and output(right) images for SUSAN edge detection algorithm
The ForSyDe model given in this section is based on the parallelized reference implementation of the edge detection algorithm of SUSAN given in [9]. A brief description of the algorithm is given below.

The input image is partitioned into smaller blocks of images and the steps listed below are applied to each pixel in these blocks.

- A mask is applied centered in the pixel of interest.
- The number of pixels within the mask that have similar intensity as the center pixel is counted. This area of the mask is known as USAN. Those pixels that have their USAN smaller than the threshold are good candidates for being edge points.
- The direction of the edge is detected by calculating the momentums of the USAN area.
- Thinning on the edges is applied to remove unwanted edge points.

The edge detection algorithm can be realized using SDF MoC by making each of these steps an actor in an SDF graph as shown in Figure 7.2.

![Fig. 7.2: SDF graph of SUSAN [9]](image)

Figure 7.3 depicts the process network of the ForSyDe model for the edge detection algorithm. As it can be seen that it resembles the SDF graph given in Figure 7.2 with additional ForSyDe processes such as unzip processes.

![Fig. 7.3: Illustration of the ForSyDe process network of SUSAN](image)

SUSAN ForSyDe model contains tupled token types and declarations of these types are shown in Listing 7.1.1. The modeling constructs and conventions explained in section 6.1 are used in the development of this application.
7.1. APPLICATION MODELING

```cpp
typedef std::tuple<std::vector<MCU_BLOCK>, std::vector<EdgeStrength>> SusanUsanOutputType;
typedef std::tuple<std::vector<MCU_BLOCK>, std::vector<EdgeStrength>, std::vector<EdgeDirection>> SusanDirectionOutputType;
typedef std::tuple<std::vector<MCU_BLOCK>, std::vector<EdgeDirection>> SusanThinOutputType;
DEFINE_SIGNAL_TYPE(SusanUsanOutputType)
DEFINE_SIGNAL_TYPE(SusanDirectionOutputType)
DEFINE_SIGNAL_TYPE(SusanThinOutputType)
DEFINE_STRUCT_TYPE(MCU_BLOCK) {}
DEFINE_STRUCT_TYPE(EdgeStrength) {}
DEFINE_STRUCT_TYPE(EdgeDirection) {}

void direction_func(std::vector<SusanDirectionOutputType> &out, const std::vector<MCU_BLOCK> &inp1, const std::vector<EdgeStrength> &inp2) {
    out = V(1, T(C(1, MCU_BLOCK), C(1, EdgeStrength), C(1, EdgeDirection)));
    FLATTEN_INPUTS(inp1, inp2);
    FLATTEN_OUTPUTS(out);
    #pragma ForSyDe begin direction_func
    susanDirection(&I(0, 0), &I(1, 0), &O(0, 0), &O(1, 0), &O(2, 0));
    #pragma ForSyDe end
}
```

Listing 7.1.1: The ForSyDe helper utilities used in the SUSAN models

The use of the ForSyDe helper utilities in development of the edge detection algorithm are demonstrated in Listing 7.1.1. The signal flattener utility is used to be able to access the tokens in the channels using the macros \(I\) and \(O\). The code section that can be extracted by the code generator is marked with the `pragma` keywords. This code section within the pragmas must be C compliant. In this actor function, the actual implementation is in another function named `susanDirection`. The function `susanDirection` exists in a different C file.

The application operates on a given input image shown in Figure 7.4(a). As the result of the simulation the image shown in Figure 7.4(b) is obtained. Another artifact produced at the end of the simulation is the structural representation of the application in XML format. This XML document is automatically generated using the introspection ability of ForSyDe. The generated XML file can be seen in Listing B.2.2 and it contains the data structure for the graph shown in Figure 7.3.
7.1.2 JPEG decoder

The ForSyDe model given in this section is based on the parallelized version of the JPEG [45] decoding algorithm given in [46].

JPEG decompression is done in five steps as explained below.

- **VLD**: The VLD process parses the image headers and performs decompression and as a result it produces a series of a minimal coded units (MCUs) which consists of one or more (up to ten) 8x8 pixel blocks of data. This process also outputs the header information to be used by the CC and Raster steps.

- **IQZZ**: The IQZZ process takes blocks in the MCUs produced by the VLD step and performs inverse quantization and reordering on the blocks on these blocks.

- **IDCT**: The IDCT process calculates an inverse discrete cosine transform of the 8x8 pixel blocks.

- **CC**: The CC process combines the 8x8 pixel blocks inside an MCU into RGB pixel values.

- **Raster**: The Raster process puts the pixel values for the MCU in place in the image.
7.1. APPLICATION MODELING

The ForSyDe model of the JPEG decoding application is based on the parallelized JPEG decoding algorithm given in [46] and the SDF model of the original application is given in Figure 7.5.

![Fig. 7.5: SDF graph of JPEG Decoder](image)

Figure 7.6 depicts the process network of the ForSyDe model for the JPEG decoding algorithm. Just as in case of the first application explained, the JPEG decoder also uses the ForSyDe helper utilities for accessing the tokens and producing the appropriate structural representation. The input image to the JPEG decoder application is shown in Figure 7.7.

![Fig. 7.6: Illustration of the ForSyDe process network of JPEG Decoder](image)

Fig. 7.7: The input and output image used for the JPEG decoder application
### 7.2 System Architecture

The automated design flow is run for a CompSOC platform with two processing tiles each containing three DMA units. The arrangement of the tiles in the system is shown in Figure 7.8.

![System Architecture Diagram](image)

**Fig. 7.8: The system architecture used for case studies**

Each processing tile in the system owns a set of instruction and data memories each having 256 KB size. The monitor is responsible for synchronization of the tiles and capturing the debug information from the tiles. The system also features a shared memory tile which is not utilized by the case study applications.

The system contains both SUSAN and JPEG Decoder applications. The design flow is invoked from the command line using a `make` command. The first two main stages of the design flow, namely the initial build and the measurement run stages are executed for the two applications separately. The final phase is executed once for both applications and it produces the bit-stream file that can be downloaded onto the FPGA board.

### 7.3 Results

The design flow step of the PN-to-SDF conversion results in SDF graphs as shown in Figure 7.2 and Figure 7.5 for the SUSAN and JPEG Decoder applications respectively. The XML file containing the resultant CompSOC SDF graphs can be seen in Listing C.1.1 and Listing C.2.2.

After the code generation step, the ForSyDe models are transformed into C files and the top module of the transformed source codes for applications can be seen in Appendix D.
After the applications are compiled and the memory requirements are back annotated, the applications are separately run on the platform to collect the execution times of the actors. The maximum measured execution times (in terms of clock cycles) for the applications are given in Table 7.1 and Table 7.2. The measured execution times for the actors of the SUSAN and the JPEG Decoder applications can be seen in Figure 7.9 and Figure 7.10 respectively. As can be seen from these charts, the execution times of some actors vary greatly. This variation is caused by the input dependent execution flow of the actor functions.

<table>
<thead>
<tr>
<th>getImage</th>
<th>USAN</th>
<th>Direction</th>
<th>Thin</th>
<th>putImage</th>
</tr>
</thead>
<tbody>
<tr>
<td>20077</td>
<td>1177105</td>
<td>833912</td>
<td>35843</td>
<td>15866</td>
</tr>
</tbody>
</table>

Table 7.1: Maximum measured execution times for SUSAN

<table>
<thead>
<tr>
<th>VLD</th>
<th>IQZZ</th>
<th>IDCT</th>
<th>CC</th>
<th>Raster</th>
</tr>
</thead>
<tbody>
<tr>
<td>62684</td>
<td>4294</td>
<td>15505</td>
<td>21284</td>
<td>1327</td>
</tr>
</tbody>
</table>

Table 7.2: Maximum measured execution times for JPEG decoder

After the back annotation of the execution times, the applications are merged and the final mapping is obtained. The automated design flow has produced a single tile mapping to run both of the applications. This final mapping is used to generate the final binary and the bit-stream files. When the system is run, both of the applications produce the correct outputs.
Fig. 7.9: The execution time charts of the actors of the SUSAN application
7.3. RESULTS

(a) Actor VLD
(b) Actor IQZZ
(c) Actor IDCT
(d) Actor CC
(e) Actor raster

Fig. 7.10: The execution time charts of the actors of the JPEG decoder application
Part III

Conclusions and Future Work
Chapter 8

Conclusions and Future Work

A general summary of the thesis work is presented in the following.

**General approaches to design real-time systems:** The thesis work has started with the investigation of the general approaches to design real-time systems. This study included the design challenges of real-time systems, the holistic approaches to designing real-time systems and why abstraction, predictability and composability are important concepts in the design of these systems. This background study has concluded with an investigation of the position of the design flow proposed in this thesis among relevant design flows reported in the academia.

**Investigation of ForSyDe and CompSOC design flows:** As the two main components of the thesis project, ForSyDe methodology and the CompSOC platform design flow are investigated as a background study. The ways to model applications using ForSyDe and how CompSOC operates to provide real-time guarantees to these applications are studied.

**Integration of ForSyDe and CompSOC design flows for synchronous data-flow model of computation:** As the main contribution of this thesis, an automated design flow is developed using ForSyDe as the modeling framework, and the CompSOC as the predictable and composable platform. The design flow requires the development of new tools to adapt the ForSyDe SDF models to the form the CompSOC software flow expects. Additionally, the design flow is automated.

**Development of the case study applications and validation:** To validate the design flow developed, two case study applications are developed and they are run on the platform using the automated design flow.
8.1 Future Work

The design flow developed within the context of this thesis focused on connecting the ForSyDe modeling framework with the predictable and composable platform CompSOC. In addition the thesis was restricted to the synchronous data-flow model of computation.

As a future work, integration of the ForSyDe and CompSOC design flows for the synchronous model of computation can be investigated. The synchronous model of computation allows modeling control oriented applications, which is a desirable property in embedded systems design.

Another area of possible investigation is the execution of applications with mixed models of computation. This is particularly interesting in today's complex embedded systems where heterogeneous platforms are becoming the de facto execution environments for such systems.

The current way the memory requirements of actors are determined does not include the data sections used by the actors. This needs to be improved for obtaining optimal mappings of actors to processors.

Currently, the WCET values for the actors are found by measurement methods. This method is not guaranteed to produce the correct results. The precision of the execution times found by this method is influenced by the input data given to the applications. As a possible future work, the WCET values for actors can be determined using static methods.

8.2 Conclusions

Using conventional ways to design real-time systems is quite challenging because of the inexhaustible design space of such systems. System modeling at low levels of abstraction, by using models of computation that either are not suitable for the application domain or are too complex to analyze, leads to sub-optimal implementations. And system level unpredictabilities in timing behavior are major causes of recurrent design activities.

By using a modeling framework that allows modeling applications at a high level of abstraction using formal models of computation and by providing architectural properties for predictable timing behavior of the system, this thesis work has shown that, a design flow targeting real-time data-flow applications is possible to achieve. The significance of this design flow is that, the applications can be developed in isolation and the real-time guarantees can be given to applications. This eliminates the recurrent activities observed in conventional design flows and in consequence allowing quick prototyping with a reduced cost.
Bibliography


Part IV

Appendices
Appendix A

ForSyDe and CompSOC Design Flows

A.1 ForSyDe Flow

Fig. A.1: Illustration of the ForSyDe design flow
A.2 CompSOC Flow

Fig. A.2: Illustration of the CompSOC design flow
A.3 The ForSyDe-CompSOC flow

Fig. A.3: Illustration of the ForSyDe-CompSOC design flow
Appendix B

ForSyDe Process Networks

B.1 Process Network of SUSAN

```xml
<?xml version="1.0"?>
<!DOCTYPE process_network SYSTEM "forsyde.dtd">
<process_network name="top">
  <signal name="fifo_0" moc="sdf" type="int" source="dummyInputConst" source_port="oport1" target="getimage" target_port="iport1"/>
  <signal name="fifo_1" moc="sdf" type="MCU_BLOCK" source="getimage" source_port="oport1" target="usan" target_port="iport1"/>
  <signal name="fifo_2" moc="sdf" type="MCU_BLOCK.EdgeStrength" source="usan" source_port="oport1" target="susanUsanOutUnzipper" target_port="iport1"/>
  <signal name="fifo_3" moc="sdf" type="MCU_BLOCK" source="susanUsanOutUnzipper" source_port="oport1" target="direction" target_port="iport1"/>
  <signal name="fifo_4" moc="sdf" type="EdgeStrength" source="susanUsanOutUnzipper" source_port="oport1" target="thin" target_port="iport1"/>
  <signal name="fifo_5" moc="sdf" type="MCU_BLOCK.EdgeStrength.EdgeDirection" source="susanUsanOutUnzipper" source_port="oport1" target="thin" target_port="iport1"/>
  <signal name="fifo_6" moc="sdf" type="MCU_BLOCK" source="susanDirectionOutUnzipper" source_port="port_2" target="thin" target_port="iport1"/>
  <signal name="fifo_7" moc="sdf" type="EdgeStrength" source="susanDirectionOutUnzipper" source_port="port_1" target="thin" target_port="iport1"/>
  <signal name="fifo_8" moc="sdf" type="EdgeDirection" source="susanDirectionOutUnzipper" source_port="port_0" target="thin" target_port="iport1"/>
  <signal name="fifo_9" moc="sdf" type="MCU_BLOCK.EdgeDirection" source="thin" source_port="oport1" target="susanThinOutUnzipper" target_port="iport1"/>
  <signal name="fifo_10" moc="sdf" type="MCU_BLOCK" source="susanThinOutUnzipper" source_port="oport1" target="putimage" target_port="iport1"/>
  <signal name="fifo_11" moc="sdf" type="EdgeDirection" source="susanThinOutUnzipper" source_port="oport2" target="putimage" target_port="iport2"/>
  <signal name="fifo_12" moc="sdf" type="int" source="putimage" source_port="oport1" target="report" target_port="iport1"/>
</leaf_process>
</process_network>
</DOCTYPE>
```
B.1. PROCESS NETWORK OF SUSAN

```xml
</process_constructor>
<leaf_process name="getImage">
  <port name="import1" type="int" direction="in"/>
  <port name="export1" type="MCU_BLOCK" direction="out"/>
  <process_constructor name="comb" moc="sdf">
    <argument name="func" value="getImage_func"/>
    <argument name="o1toks" value="1"/>
    <argument name="i1toks" value="1"/>
  </process_constructor>
</leaf_process>

<leaf_process name="usan">
  <port name="import1" type="MCU_BLOCK" direction="in"/>
  <port name="export1" type="MCU_BLOCK. EdgeStrength" direction="out"/>
  <process_constructor name="comb" moc="sdf">
    <argument name="func" value="usan_func"/>
    <argument name="o1toks" value="1"/>
    <argument name="i1toks" value="1"/>
  </process_constructor>
</leaf_process>

<leaf_process name="susanUsanOutUnzipper">
  <port name="import1" type="MCU_BLOCK. EdgeStrength" direction="in"/>
  <port name="port_2" type="MCU_BLOCK" direction="out"/>
  <port name="port_1" type="EdgeStrength" direction="out"/>
  <port name="port_0" type="EdgeDirection" direction="out"/>
  <process_constructor name="unzipN" moc="sdf">
    <argument name="otoks" value="[1, 1, 1]"/>
  </process_constructor>
</leaf_process>

<leaf_process name="direction">
  <port name="import1" type="MCU_BLOCK" direction="in"/>
  <port name="import2" type="EdgeStrength" direction="in"/>
  <port name="import1" type="MCU_BLOCK. EdgeStrength. EdgeDirection" direction="out"/>
  <process_constructor name="comb2" moc="sdf">
    <argument name="func" value="direction_func"/>
    <argument name="o1toks" value="1"/>
    <argument name="i1toks" value="1"/>
    <argument name="i2toks" value="1"/>
  </process_constructor>
</leaf_process>

<leaf_process name="susanDirectionOutUnzipper">
  <port name="import1" type="MCU_BLOCK. EdgeStrength. EdgeDirection" direction="in"/>
  <port name="port_2" type="MCU_BLOCK" direction="out"/>
  <port name="port_1" type="EdgeStrength" direction="out"/>
  <port name="port_0" type="EdgeDirection" direction="out"/>
  <process_constructor name="unzipN" moc="sdf">
    <argument name="otoks" value="[1, 1, 1]"/>
  </process_constructor>
</leaf_process>

<leaf_process name="thin">
  <port name="import1" type="MCU_BLOCK" direction="in"/>
  <port name="import2" type="EdgeStrength" direction="in"/>
  <port name="import3" type="EdgeDirection" direction="in"/>
  <port name="export1" type="MCU_BLOCK. EdgeDirection" direction="out"/>
  <process_constructor name="comb3" moc="sdf">
    <argument name="func" value="thin_func"/>
    <argument name="o1toks" value="1"/>
    <argument name="i1toks" value="1"/>
    <argument name="i2toks" value="1"/>
    <argument name="i3toks" value="1"/>
  </process_constructor>
</leaf_process>
</leaf_process>
```
Listing B.1.1: The process network of SUSAN represented in XML
B.2 Process Network of JPEG Decoder

```xml
<?xml version="1.0" ?>
<!DOCTYPE process_network SYSTEM "forsyde.dtd">
<process_network name="top">
  <signal name="fifo_0" moc="sdf" type="int" source="vldCombOutUnzipper" source_port="port_2" target="init_vld_state" target_port="iport1"/>
  <signal name="fifo_1" moc="sdf" type="int" source="init_vld_state" source_port="oport1" target="vld" target_port="iport1"/>
  <signal name="fifo_2" moc="sdf" type="FValue.int.SubHeader1.SubHeader2" source="vld" source_port="oport1" target="vldCombOutUnzipper" target_port="iport1"/>
  <signal name="fifo_3" moc="sdf" type="FValue" source="vldCombOutUnzipper" source_port="port_3" target="vldCombOutUnzipper" target_port="iport1"/>
  <signal name="fifo_4" moc="sdf" type="SubHeader1" source="vldCombOutUnzipper" source_port="port_1" target="cc" target_port="iport1"/>
  <signal name="fifo_5" moc="sdf" type="SubHeader2" source="vldCombOutUnzipper" source_port="port_0" target="rasterInputZipper" target_port="port_1"/>
  <signal name="fifo_6" moc="sdf" type="FBlock" source="iqzz" source_port="oport1" target="idct" target_port="iport1"/>
  <signal name="fifo_7" moc="sdf" type="PBlock" source="idct" source_port="oport1" target="rasterInputZipper" target_port="port_0"/>
  <signal name="fifo_8" moc="sdf" type="ColorBuffer" source="cc" source_port="oport1" target="rasterInputZipper" target_port="port_0"/>
  <signal name="fifo_9" moc="sdf" type="SubHeader2.ColorBuffer" source="rasterInputZipper" source_port="oport1" target="raster" target_port="iport1"/>
  <leaf_process name="init_vld_state">
    <port name="iport1" type="int" direction="in"/>
    <port name="oport1" type="int" direction="out"/>
    <process_constructor name="delay" moc="sdf">
      <argument name="init_val" value="1"/>
    </process_constructor>
  </leaf_process>
  <leaf_process name="vld">
    <port name="iport1" type="int" direction="in"/>
    <port name="oport1" type="FValue.int.SubHeader1.SubHeader2" direction="out"/>
    <process_constructor name="comb" moc="sdf">
      <argument name="func" value="vld_func"/>
      <argument name="otoks" value="1"/>
      <argument name="i1toks" value="1"/>
    </process_constructor>
  </leaf_process>
  <leaf_process name="vldCombOutUnzipper">
    <port name="iport1" type="FValue.int.SubHeader1.SubHeader2" direction="in"/>
    <port name="port_2" type="FValue" direction="out"/>
    <port name="port_3" type="FValue" direction="out"/>
    <port name="port_1" type="SubHeader1" direction="out"/>
    <port name="port_0" type="SubHeader2" direction="out"/>
    <process_constructor name="unzipN" moc="sdf">
      <argument name="otoks" value="[10, 1, 1, 1]"/>
    </process_constructor>
  </leaf_process>
  <leaf_process name="iqzz">
    <port name="iport1" type="FValue" direction="in"/>
    <port name="oport1" type="FBlock" direction="out"/>
    <process_constructor name="comb" moc="sdf">
      <argument name="func" value="iqzz_func"/>
      <argument name="i1toks" value="1"/>
      <argument name="i1toks" value="1"/>
    </process_constructor>
  </leaf_process>
</process_network>
```
Listing B.2.2: The process network of JPEG decoder represented in XML
Appendix C

PN-to-SDF Conversion Results

C.1 PN-to-SDF conversion results for SUSAN

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" type="sdf" version="1.0"
     xsi:noNamespaceSchemaLocation="usecase.xsd">
    <applicationGraph name="susan" type="sdf">
        <csdfgraph>
            <actor name="susan_dummyInputConst">
                <executionTime time="312"/>
                <port name="take_val_iport_1" type="in" rate="1"/>
                <port name="oport1_1" type="out" rate="1"/>
                <port name="take_val_oport_1" type="out" rate="1"/>
            </actor>
            <actor name="susan_getImage">
                <executionTime time="20077"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="oport1_1" type="out" rate="1"/>
            </actor>
            <actor name="susan_usan">
                <executionTime time="1177105"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="oport1_1" type="out" rate="1"/>
                <port name="oport1_2" type="out" rate="1"/>
            </actor>
            <actor name="susan_direction">
                <executionTime time="833912"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="iport2_1" type="in" rate="1"/>
                <port name="oport1_1" type="out" rate="1"/>
                <port name="oport1_2" type="out" rate="1"/>
                <port name="oport1_3" type="out" rate="1"/>
            </actor>
            <actor name="susan_thin">
                <executionTime time="35843"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="iport2_1" type="in" rate="1"/>
                <port name="oport1_1" type="out" rate="1"/>
                <port name="oport1_2" type="out" rate="1"/>
            </actor>
        </csdfgraph>
    </applicationGraph>
</nest>
```
<actor name="susan_putImage">
  <executionTime time="15866"/>
  <port name="iport1_1" type="in" rate="1"/>
  <port name="iport2_1" type="in" rate="1"/>
  <port name="oport1_1" type="out" rate="1"/>
</actor>

<actor name="susan_report">
  <executionTime time="240"/>
  <port name="iport1_1" type="in" rate="1"/>
</actor>

<channel name="susan_dummyInputConst_oport1_1_susan_getImage_iport1_1" srcActor="susan_dummyInputConst"
  srcPort="oport1_1" dstActor="susan_getImage" dstPort="iport1_1" initialTokens="1"/>

<channel name="susan_getImage_oport1_1_susan_usan_iport1_1" srcActor="susan_getImage"
  srcPort="oport1_1" dstActor="susan_usan" dstPort="iport1_1" initialTokens="0"/>

<channel name="susan_usan_oport1_1_susan_direction_iport1_1" srcActor="susan_usan"
  srcPort="oport1_1" dstActor="susan_direction" dstPort="iport1_1" initialTokens="0"/>

<channel name="susan_direction_oport1_1_susan_thin_iport1_1" srcActor="susan_direction"
  srcPort="oport1_1" dstActor="susan_thin" dstPort="iport1_1" initialTokens="0"/>

<channel name="susan_direction_oport1_2_susan_thin_iport2_1" srcActor="susan_direction"
  srcPort="oport1_2" dstActor="susan_thin" dstPort="iport2_1" initialTokens="0"/>

<channel name="susan_direction_oport1_3_susan_thin_iport3_1" srcActor="susan_direction"
  srcPort="oport1_3" dstActor="susan_thin" dstPort="iport3_1" initialTokens="0"/>

<channel name="susan_thin_oport1_1_susan_putImage_iport1_1" srcActor="susan_thin"
  srcPort="oport1_1" dstActor="susan_putImage" dstPort="iport1_1" initialTokens="0"/>

<channel name="susan_thin_oport1_2_susan_putImage_iport2_1" srcActor="susan_thin"
  srcPort="oport1_2" dstActor="susan_putImage" dstPort="iport2_1" initialTokens="0"/>
C.1. PN-TO-SDF CONVERSION RESULTS FOR SUSAN

```xml
<csdfgraph>
  <csdfProperties>
    <actorProperties actor="susan_dummyInputConst">
      <processor type="microblaze0" default="true">
        <memory>
          <memoryElement name=".instr">
            <size>984</size>
            <accessCnt>1</accessCnt>
            <accessType>IFetch</accessType>
            <accessSize>word</accessSize>
          </memoryElement>
          <memoryElement name=".data">
            <size>0</size>
            <accessCnt>1</accessCnt>
            <accessType>DRead, DWrite</accessType>
            <accessSize>halfword</accessSize>
          </memoryElement>
          <memoryElement name="sharedVar">
            <size>0</size>
            <accessCnt>1</accessCnt>
            <accessType>DRead, DWrite</accessType>
            <accessSize>byte</accessSize>
          </memoryElement>
        </memory>
        <implementation>
          <function symbol="susan_dummyInputConst_func">
            <argumentMapping>
              <argument port="take_val_oport_1" number="0"/>
              <argument port="oport1_1" number="0"/>
              <argument port="take_val_iport_1" number="1"/>
            </argumentMapping>
          </function>
        </implementation>
      </processor>
    </actorProperties>
    <actorProperties actor="susan_getImage">
      <processor type="microblaze0" default="true">
        <memory>
          <memoryElement name=".instr">
            <size>5740</size>
            <accessCnt>1</accessCnt>
            <accessType>IFetch</accessType>
            <accessSize>word</accessSize>
          </memoryElement>
          <memoryElement name=".data">
            <size>0</size>
          </memoryElement>
        </memory>
      </processor>
    </actorProperties>
  </csdfProperties>
  <channel name="susan_putImage_oport1_1_susan_report_iport1_1" srcActor="susan_putImage">
    <srcPort>oport1_1</srcPort>
    <dstActor>susan_report</dstActor>
    <dstPort>iport1_1</dstPort>
    <initialTokens>0</initialTokens/>
  </channel>
  <channel name="susan_dummyInputConst_take_val_oport_1_susan_dummyInputConst_take_val_iport_1" srcActor="susan_dummyInputConst">
    <srcPort>take_val_oport_1</srcPort>
    <dstActor>susan_dummyInputConst</dstActor>
    <dstPort>take_val_iport_1</dstPort>
    <initialTokens>1</initialTokens/>
  </channel>
</csdfgraph>
```
APPENDIX C. PN-TO-SDF CONVERSION RESULTS

<accessCnt>1</accessCnt>
<AccessType>DRead, DWrite</AccessType>
<accessSize>halfword</accessSize>
</memoryElement>
<memoryElement name="sharedVar">
<size>0</size>
<accessCnt>1</accessCnt>
<AccessType>DRead, DWrite</AccessType>
<accessSize>byte</accessSize>
</memoryElement>

<function symbol="getImage_func">
<argumentMapping>
<argument port="iport1_1" number="0"/>
<argument port="oport1_1" number="0"/>
</argumentMapping>
</function>

<memory>
<memoryElement name=".instr">
<size>4088</size>
<accessCnt>1</accessCnt>
<AccessType>IFetch</AccessType>
<accessSize>word</accessSize>
</memoryElement>
<memoryElement name=".data">
<size>0</size>
<accessCnt>1</accessCnt>
<AccessType>DRead, DWrite</AccessType>
<accessSize>halfword</accessSize>
</memoryElement>
<memoryElement name="sharedVar">
<size>0</size>
<accessCnt>1</accessCnt>
<AccessType>DRead, DWrite</AccessType>
<accessSize>byte</accessSize>
</memoryElement>
</memory>

<implementation>
</implementation>
</processor>
</actorProperties>
<actorProperties actor="susan_usan">
<processor type="microblaze0" default="true">
<memory>
<memoryElement name=".instr">
<size>9896</size>
<accessCnt>1</accessCnt>
<AccessType>IFetch</AccessType>
<accessSize>word</accessSize>
</memoryElement>
</memory>
<br>
</implementation>
</processor>
</actorProperties>
<actorProperties actor="susan_direction">
<processor type="microblaze0" default="true">
<memory>
</memory>
<br>
</implementation>
</processor>
</actorProperties>
C.1. PN-TO-SDF CONVERSION RESULTS FOR SUSAN

```xml
<memoryElement name=".data">
  <size>0</size>
  <accessCnt>1</accessCnt>
  <accessType>DRead, DWrite</accessType>
  <accessSize>halfword</accessSize>
</memoryElement>

<memoryElement name="sharedVar">
  <size>0</size>
  <accessCnt>1</accessCnt>
  <accessType>DRead, DWrite</accessType>
  <accessSize>byte</accessSize>
</memoryElement>

<implementation>
  <function symbol="direction_func">
    <argumentMapping>
      <argument port="iport1_1" number="0"/>
      <argument port="iport2_1" number="1"/>
      <argument port="oport1_1" number="0"/>
      <argument port="oport1_2" number="1"/>
      <argument port="oport1_3" number="2"/>
    </argumentMapping>
  </function>
</implementation>

<processor type="microblaze0" default="true">
  <memory>
    <memoryElement name=".instruction">
      <size>6848</size>
      <accessCnt>1</accessCnt>
      <accessType>IFetch</accessType>
      <accessSize>word</accessSize>
    </memoryElement>
    <memoryElement name=".data">
      <size>0</size>
      <accessCnt>1</accessCnt>
      <accessType>DRead, DWrite</accessType>
      <accessSize>halfword</accessSize>
    </memoryElement>
    <memoryElement name="sharedVar">
      <size>0</size>
      <accessCnt>1</accessCnt>
      <accessType>DRead, DWrite</accessType>
      <accessSize>byte</accessSize>
    </memoryElement>
  </memory>
  <implementation>
    <function symbol="thin_func">
      <argumentMapping>
        <argument port="iport1_1" number="0"/>
        <argument port="iport2_1" number="1"/>
        <argument port="iport3_1" number="2"/>
        <argument port="oport1_1" number="0"/>
        <argument port="oport1_2" number="1"/>
      </argumentMapping>
    </function>
  </implementation>
</processor>
</actorProperties>
```

109
APPENDIX C. PN-TO-SDF CONVERSION RESULTS

```xml
<actorProperties actor="susan_putImage">
  <processor type="microblaze0" default="true">
    <memory>
      <memoryElement name=".instr">
        <size>6028</size>
        <accessCnt>1</accessCnt>
        <accessType>IFetch</accessType>
        <accessSize>word</accessSize>
      </memoryElement>
      <memoryElement name=".data">
        <size>0</size>
        <accessCnt>1</accessCnt>
        <accessType>DRead,DWrite</accessType>
        <accessSize>halfword</accessSize>
      </memoryElement>
      <memoryElement name="sharedVar">
        <size>0</size>
        <accessCnt>1</accessCnt>
        <accessType>DRead,DWrite</accessType>
        <accessSize>byte</accessSize>
      </memoryElement>
    </memory>
    <implementation>
      <function symbol="putImage_func">
        <argumentMapping>
          <argument port="iport1_1" number="0"/>
          <argument port="iport2_1" number="1"/>
        </argumentMapping>
      </function>
    </implementation>
  </processor>
</actorProperties>

<actorProperties actor="susan_report">
  <processor type="microblaze0" default="true">
    <memory>
      <memoryElement name=".instr">
        <size>832</size>
        <accessCnt>1</accessCnt>
        <accessType>IFetch</accessType>
        <accessSize>word</accessSize>
      </memoryElement>
      <memoryElement name=".data">
        <size>0</size>
        <accessCnt>1</accessCnt>
        <accessType>DRead,DWrite</accessType>
        <accessSize>halfword</accessSize>
      </memoryElement>
      <memoryElement name="sharedVar">
        <size>0</size>
        <accessCnt>1</accessCnt>
        <accessType>DRead,DWrite</accessType>
        <accessSize>byte</accessSize>
      </memoryElement>
    </memory>
    <implementation>
      <function symbol="report_func">
        <argumentMapping>
          <argument port="iport1_1" number="0"/>
        </argumentMapping>
      </function>
    </implementation>
  </processor>
</actorProperties>
```

110
C.1. PN-TO-SDF CONVERSION RESULTS FOR SUSAN

```xml
Listing C.1.1: The CompSOC SDF graph of SUSAN

```
C.2 PN-to-SDF conversion results for JPEG decoder

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" type="sdf" version="1.0"
    xsi:noNamespaceSchemaLocation="usecase.xsd">
    <applicationGraph name="jpeg_decoder" type="sdf">
        <csdgraph>
            <actor name="jpeg_decoder_vld">
                <executionTime time="62684"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="oport1_1" type="out" rate="10"/>
                <port name="oport1_2" type="out" rate="1"/>
                <port name="oport1_3" type="out" rate="1"/>
                <port name="oport1_4" type="out" rate="1"/>
            </actor>
            <actor name="jpeg_decoder_iqzz">
                <executionTime time="4294"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="oport1_1" type="out" rate="1"/>
            </actor>
            <actor name="jpeg_decoder_idct">
                <executionTime time="15505"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="oport1_1" type="out" rate="1"/>
            </actor>
            <actor name="jpeg_decoder_cc">
                <executionTime time="21284"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="iport2_1" type="in" rate="10"/>
                <port name="oport1_1" type="out" rate="1"/>
            </actor>
            <actor name="jpeg_decoder_raster">
                <executionTime time="1327"/>
                <port name="iport1_1" type="in" rate="1"/>
                <port name="iport1_2" type="in" rate="1"/>
            </actor>
        </csdgraph>
        <channel name="jpeg_decoder_vld_oport1_1_jpeg_decoder_iqzz_iport1_1">
            srcActor="jpeg_decoder_vld"
            srcPort="oport1_1"
            dstActor="jpeg_decoder_iqzz"
            dstPort="iport1_1"
            initTokens="0"/>
        </channel>
        <channel name="jpeg_decoder_vld_oport1_2_jpeg_decoder_vld_iport1_1">
            srcActor="jpeg_decoder_vld"
            srcPort="oport1_2"
            dstActor="jpeg_decoder_vld"
            dstPort="iport1_1"
            initTokens="1"/>
        </channel>
        <channel name="jpeg_decoder_vld_oport1_3_jpeg_decoder_cc_iport1_1">
            srcActor="jpeg_decoder_vld"
            srcPort="oport1_3"
            dstActor="jpeg_decoder_cc"
            dstPort="iport1_1"
            initTokens="0"/>
        </channel>
        <channel name="jpeg_decoder_vld_oport1_4_jpeg_decoder_raster_iport1_1">
            srcActor="jpeg_decoder_vld"
            srcPort="oport1_4"
            dstActor="jpeg_decoder_raster"
            dstPort="iport1_1"
            initTokens="0"/>
    </applicationGraph>
</nest>
```
<memoryElement name=".data">
  <size>0</size>
  <accessCnt>1</accessCnt>
  <accessType>DRead, DWrite</accessType>
  <accessSize>halfword</accessSize>
</memoryElement>

<memoryElement name="sharedVar">
  <size>0</size>
  <accessCnt>1</accessCnt>
  <accessType>DRead, DWrite</accessType>
  <accessSize>byte</accessSize>
</memoryElement>

<implementation>
  <function symbol="iqzz_func">
    <argumentMapping>
      <argument port="iport1_1" number="0"/>
      <argument port="oport1_1" number="0"/>
    </argumentMapping>
  </function>
</implementation>

<implementation>
  <function symbol="idct_func">
    <argumentMapping>
      <argument port="iport1_1" number="0"/>
      <argument port="oport1_1" number="0"/>
    </argumentMapping>
  </function>
</implementation>

<implementation>
  <function symbol="jpeg_decoder_idct">
    <argumentMapping>
      <argument port="iport1_1" number="0"/>
      <argument port="oport1_1" number="0"/>
    </argumentMapping>
  </function>
</implementation>

<implementation>
  <function symbol="jpeg_decoder_cc">
    <argumentMapping>
      <argument port="iport1_1" number="0"/>
      <argument port="oport1_1" number="0"/>
    </argumentMapping>
  </function>
</implementation>

<implementation>
  <function symbol="jpeg_decoder_idct">
    <argumentMapping>
      <argument port="iport1_1" number="0"/>
      <argument port="oport1_1" number="0"/>
    </argumentMapping>
  </function>
</implementation>

<implementation>
  <function symbol="jpeg_decoder_cc">
    <argumentMapping>
      <argument port="iport1_1" number="0"/>
      <argument port="oport1_1" number="0"/>
    </argumentMapping>
  </function>
</implementation>

<processor type="microblaze0" default="true">
  <memory>
    <memoryElement name=".instr">
      <size>2288</size>
      <accessCnt>1</accessCnt>
      <accessType>IFetch</accessType>
      <accessSize>word</accessSize>
    </memoryElement>
  </memory>
</processor>

<processor type="microblaze0" default="true">
  <memory>
    <memoryElement name=".data">
      <size>0</size>
      <accessCnt>1</accessCnt>
      <accessType>DRead, DWrite</accessType>
      <accessSize>halfword</accessSize>
    </memoryElement>
  </memory>
</processor>

<processor type="microblaze0" default="true">
  <memory>
    <memoryElement name="sharedVar">
      <size>0</size>
      <accessCnt>1</accessCnt>
      <accessType>DRead, DWrite</accessType>
      <accessSize>byte</accessSize>
    </memoryElement>
  </memory>
</processor>

<processor type="microblaze0" default="true">
  <memory>
    <memoryElement name=".inst">
      <size>2036</size>
    </memoryElement>
  </memory>
</processor>
C.2. PN-TO-SDF CONVERSION RESULTS FOR JPEG DECODER

```xml
<accessCnt>1</accessCnt>
<accessType>IFetch</accessType>
<accessSize>word</accessSize>
</memoryElement>
<memoryElement name=".data">
<size>0</size>
<accessCnt>1</accessCnt>
<accessType>DRead, DWrite</accessType>
<accessSize>halfword</accessSize>
</memoryElement>
<memoryElement name="sharedVar">
<size>0</size>
<accessCnt>1</accessCnt>
<accessType>DRead, DWrite</accessType>
<accessSize>byte</accessSize>
</memoryElement>
</memory>
<implementation>
<function symbol="cc_func">
<argumentMapping>
<argument port="iport1_1" number="0"/>
<argument port="iport2_1" number="1"/>
<argument port="oport1_1" number="0"/>
</argumentMapping>
</function>
</implementation>
</actorProperties>
<processor type="microblaze0" default="true">
<memory>
<memoryElement name=".instr">
<size>2176</size>
<accessCnt>1</accessCnt>
<accessType>IFetch</accessType>
<accessSize>word</accessSize>
</memoryElement>
<memoryElement name=".data">
<size>0</size>
<accessCnt>1</accessCnt>
<accessType>DRead, DWrite</accessType>
<accessSize>halfword</accessSize>
</memoryElement>
<memoryElement name="sharedVar">
<size>0</size>
<accessCnt>1</accessCnt>
<accessType>DRead, DWrite</accessType>
<accessSize>byte</accessSize>
</memoryElement>
</memory>
<implementation>
<function symbol="raster_func">
<argumentMapping>
<argument port="iport1_1" number="0"/>
<argument port="iport1_2" number="1"/>
</argumentMapping>
</function>
</implementation>
</processor>
</actorProperties>
<channelProperties channel="jpeg_decoder_vld_oport1_1_jpeg_decoder_iqzz_iport1_1">
```

115
Listing C.2.2: The CompSOC SDF graph of JPEG decoder
Appendix D

The code generation results

D.1 Code generation results for SUSAN

```c
#include "susan-common.h"

void getImage_func() {
  FLATTEN_INPUTS();
  FLATTEN_OUTPUTS();

  INPUT_VAR(0, int);
  OUTPUT_VAR(0, MCU_BLOCK);

  splitIntoMCUs(&O(0, 0));
}

void usan_func() {
  FLATTEN_INPUTS();
  FLATTEN_OUTPUTS();

  INPUT_VAR(0, MCU_BLOCK);
  OUTPUT_VAR(0, MCU_BLOCK);
  OUTPUT_VAR(1, EdgeStrength);

  susanUsan(&I(0, 0), &O(0, 0), &O(1, 0));
}

void direction_func() {
  FLATTEN_INPUTS();
  FLATTEN_OUTPUTS();

  INPUT_VAR(0, MCU_BLOCK);
  INPUT_VAR(1, EdgeStrength);
  OUTPUT_VAR(0, MCU_BLOCK);
  OUTPUT_VAR(1, EdgeStrength);
```
susanDirection(&I(0, 0), &I(1, 0), &O(0, 0), &O(1, 0), &O(2, 0));
}

void thin_func() {
  FLATTEN_INPUTS();
  FLATTEN_OUTPUTS();
  INPUT_VAR(0, MCU_BLOCK);
  INPUT_VAR(1, EdgeStrength);
  INPUT_VAR(2, EdgeDirection);
  OUTPUT_VAR(0, MCU_BLOCK);
  OUTPUT_VAR(1, EdgeDirection);
  susanThin(&I(0, 0), &I(1, 0), &I(2, 0), &O(0, 0), &O(1, 0));
}

void putImage_func() {
  FLATTEN_INPUTS();
  FLATTEN_OUTPUTS();
  INPUT_VAR(0, MCU_BLOCK);
  INPUT_VAR(1, EdgeDirection);
  OUTPUT_VAR(0, int);
  
  wrapUp(&I(0, 0), &I(1, 0));
  
  O(0, 0) = 1; //dummy output
}

void report_func() {
  FLATTEN_INPUTS();
  FLATTEN_OUTPUTS();
  INPUT_VAR(0, int);
  
  ;
  ;
}

Listing D.1.1: The converted SUSAN model
D.2 Code generation results for JPEG decoder

```c
#include "jpeg_decoder-common.h"

void vld_func()
{
    FLATTEN_INPUTS();
    FLATTEN_OUTPUTS();

    INPUT_VAR(0, int);
    OUTPUT_VAR(0, FValue);
    OUTPUT_VAR(1, int);
    OUTPUT_VAR(2, SubHeader1);
    OUTPUT_VAR(3, SubHeader2);

    if (I(0, 0) == 1)
    {
        init_header_vld(NULL, NULL, NULL);
        header_vld(&O(0, 0), &O(2, 0), &O(3, 0));
        O(1, 0) = 0;
    }
}

void iqzz_func()
{
    FLATTEN_INPUTS();
    FLATTEN_OUTPUTS();

    INPUT_VAR(0, FValue);
    OUTPUT_VAR(0, FBlock);

    iqzz(&I(0, 0), &O(0, 0));
}

void idct_func()
{
    FLATTEN_INPUTS();
    FLATTEN_OUTPUTS();

    INPUT_VAR(0, FBlock);
    OUTPUT_VAR(0, PBlock);

    idct(&I(0, 0), &O(0, 0));
}

void cc_func()
{
    FLATTEN_INPUTS();
    FLATTEN_OUTPUTS();

    INPUT_VAR(0, SubHeader1);
    INPUT_VAR(1, PBlock);
}
Listing D.2.2: The converted JPEG decoder model