KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Technology

Second cycle, 30 credits

Software Synthesis For ForSyDe-IO

YIHANG ZHAO

Stockholm, Sweden, 2022

Software Synthesis For ForSyDe-IO

YIHANG ZHAO

Master's Programme, Embedded Systems, 120 credits
Date: June 14, 2022

Supervisor: Rodolfo Jordão
Examiner: Ingo Sander
School of Electrical Engineering and Computer Science
Swedish title: Mjukvarusyntesen av ForSyDe-IO

Abstract

The implementation of embedded software applications is a complex process. The complexity arises from the intense time-to-market pressures; power and memory constraints. To deal with this complexity, an idea is to automatically construct the applications based on the high-level abstraction model. Synchronous data flow (SDF) is a high-level model of computation, and is used to model the embedded applications. Formal System Design (ForSyDe), developed by ForSyDe group at KTH Royal Institute of Technology, is a methodology for modeling and designing heterogeneous systems-on-chip. The aim of Formal System Design (ForSyDe) is to automatically generate the detailed software implementation or hardware implementation according to the high-level system specification. Formal System Design (ForSyDe) starts from the high-level system specification and specifies the system model in Haskell language. Synchronous data flow is supported by ForSyDe. ForSyDe-IO is an intermediate representation of the high-level system specification. This master thesis focuses on the software synthesis of ForSyDe-IO and synchronous data flow, and aims to produce an automatic tool that can generate software applications in C code for different platforms based on ForSyDe IO. In this project, a software synthesis method for ForSyDe-IO was proposed. Then, based on the software synthesis method, a code generator, written in Java and Xtend, was designed. The derived code generator was tested on two examples. The experiment results show that the ForSyDe-IO is successfully synthesized into C code.

Keywords

Synchronous Data Flow, Software Synthesis, ForSyDe, ForSyDe-IO

Sammanfattning

Implementeringen av inbäddade mjukvaruapplikationer är en komplex process. Komplexiteten beror på det intensiva trycket på tid-till-marknad; kraft- och minnesbegränsningar. För att hantera denna komplexitet är en idé att applikationerna automatiskt kan konstrueras den högnivåabstraktionsmodellen. Synkront dataflöde (SDF) är en beräkningsmodell på hög nivå som används för att modellera inbäddade applikationer. Formell systemdesign (ForSyDe), utvecklad av ForSyDe-gruppen vid KTH, Kungliga Tekniska Högskolan , är en metodik för modellering och design av heterogena system på chipp. Syftet med formell systemdesign (ForSyDe) är att automatiskt generera den detaljerade mjuk- eller hårdvaruimplementationen enligt systemspecifikationen på hög nivå. Formell systemdesign (ForSyDe) utgår från systemspecifikationen på hög nivå och specificerar systemmodellen på Haskell-språket. Synkront dataflöde stöds av ForSyDe. ForSyDe-IO är en mellanrepresentation av systemspecifikationen på hög nivå. Detta examensarbete fokuserar på mjukvarusyntesen av ForSyDe-IO och synkront dataflöde, och syftar till att producera ett automatiskt verktyg som kan generera mjukvaruapplikation i C-kod för olika plattformar baserat på ForSyDe-IO. I detta projekt föreslås en mjukvarusyntesmetod för ForSyDe-IO. Sedan, baserat på mjukvarusyntesmetoden, designas en kodgenerator skriven i Java och Xtend. Den härledda kodgeneratorn testas på två exempel. Experimentresultaten visar att ForSyDe-IO framgångsrikt har syntetiserats till C-kod.

Nyckelord

Synkront dataflöde, Mjukvarusyntes, ForSyDe, ForSyDe-IO

Acknowledgments

I would like to thank professor Ingo for giving me this master thesis opportunity. Through this master thesis, I have learned a lot about the SDF mode of computation and had great training in scientific research. I would like to thank my supervisor Rodolfo Jordão. He gives me a lot of help in the ForSyDe-IO . I also would like to thank Rui Chen. When I had difficulties in the CompSoC board, he gave me many help.

Stockholm, June 2022
Yihang Zhao

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

ForSyDe	Formal System Design
ForSyDe-IO	Formal System Design-IO
KTH	KTH Royal Institute of Technology
MMU	Memory Management Unit
MoC	Model of Computation
PL	Programmable Logic
PS	Processing System
SAS	Single Appearance Schedule
SDF	Synchronous Data Flow
SDFG	Synchronous Data Flow Graph
SoC	System on Chip
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
WCET	Worst Case Execution Time

Chapter 1

Introduction

1.1 Background

In modern society, embedded systems are almost everywhere, from military weapons to daily life equipment, and from the big particle collider to a smartphone. The implementation of embedded applications is a complex process. The complexity arises from the intense time-to-market pressures, power and memory constraints, and so on. To deal with this complexity, an idea is to automatically construct the applications based on the high-level abstraction model. In the embedded system design field, this idea is called *correct-by-construction* design flow, which contains *system modeling* phase, *design space exploration* phase, and *synthesis and compilation* phase.

In *system modeling* phase, the functionality of each application is modeled at high-level. Synchronous Data Flow model [1, 2, 3, 4, 5] is a paradigm for modeling embedded applications in *system modeling* phase.

Synchronous Data Flow (SDF) model [1] is expressed in the directed graph, in which the node (actor) represents applications and arc (edge) represents the token channel. Whenever the actors have enough input tokens, it would be executed (fire)—reading tokens from input channels, and producing output tokens into output channels. For each actor, the production and consumption of tokens are specified. SDF model [1, 2] has both the expression ability and analysis ability. The existence of feasible static periodic schedules, on both uniprocessor platforms and multiprocessor platforms, can be determined by the topology matrix [2]. Furthermore, the number of invocations of each actor in one period can also be determined by the repetition vector [2]. Wang *et al.,* [6] proposed an improved scheduling algorithm for SDF on multi-core systems. Halbwachs *et al.,* [7] proposed a

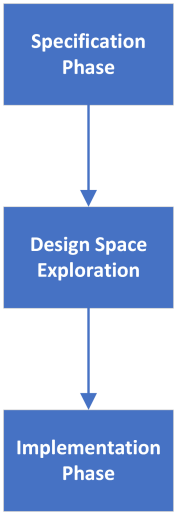


Figure 1.1: Workflow of ForSyDe

synchronous programming language LUSTRE. Guernic *et al.,* [8] proposed a synchronous programming language Signal, which is a data-flow oriented real-time synchronous language. Ghamarian *et al.,* [9] proposed the thourghput analysis of SDF.

There are some commercial or open-source products which support the high-level, graphical design environments for SDF, such as Cadence System-to-IC Flow [10], CoCentric System Studio [11] from Synopsys.

After *system modeling* phase, the next phase the called *design space exploration* [12]. In this phase, in order to satisfy the given design constraints, such as cost, power and real-time, mapping from applications to the hardware platforms is carefully explored. The final phase is *synthesis* phase. In *synthesis* phase, based on the results from *system modeling* phase, applications are implemented on the specific hardware platforms.

Formal System Design (ForSyDe) [13, 14] is a methodology for modeling and designing complicated embedded systems and is developed by ForSyDe group at KTH Royal Institute of Technology (KTH). ForSyDe is constructed on the Model of Computation (MoC), and supports SDF model.

In Figure 1.1, ForSyDe starts from the *specification phase*. In the specification phase, the system model is specified in high-level, without considering any detailed software implementations and hardware architectures.

Later, by DeSyDe [15], which is a design space exploration tool of ForSyDe [13], the design space exploration is carried out. The *design space*

exploration takes software model, hardware model, and design constraints as input, maps software applications on the hardware platforms and outputs the Formal System Design-IO (ForSyDe-IO) files.

Finally, in the *implementation phase* of ForSyDe [13], the intermediate ForSyDe-IO files are synthesized into the corresponding target code, such as C, C++, Ada, and Very High-Speed Integrated Circuit Hardware Description Language (VHDL). In this way, ForSyDe [13] eliminates the gaps between the high-level design and low-level implementation. This project focuses on the software synthesis of ForSyDe-IO on different platforms.

1.2 Problem

In the ForSyDe [13], after designing the application at high-level and the design space exploration, the ForSyDe-IO, containing the scheduling and mapping of software applications and hardware platforms, is generated. In this project, the problems are: 1) how to correctly synthesize the ForSyDe-IO into C code on uniprocessor bare-metal platform, multiprocessor bare-metal platform, and RTOS platform. 2) How to design a code generator which takes ForSyDe-IO as input and outputs the synthesized C code on different platforms.

1.3 Purpose

This project can speed up the embedded system development workflow and help developers in the embedded industry to quickly produce the products and to launch the products into the market in a short time. This thesis explores the software synthesis for ForSyDe-IO. Researchers who are interested in ForSyDe, software synthesis of SDF may be interested in this project.

1.4 Goals

The goals of this project are to design a software synthesis method which synthesizes ForSyDe-IO containing SDF model into C code and to design a code generator which takes ForSyDe-IO as input and automatically generates C codes for uniprocessor bare-metal, multiprocessor bare-metal, and RTOS platform. The code generator should be extensible.

The term ”software synthesis” consists of three parts:

1. The correct synthesis of each SDF element.
2. The binding of SDF actors to specific cores.
3. The correct implementation of SDF scheduling on each core.

The goal can be divided into the following four sub-goals:

- Subgoal 1: Design the software synthesis for ForSyDe-IO on bare metal with single processor.
- Subgoal 2: Design the software synthesis for ForSyDe-IO on bare metal with multiple processors.
- Subgoal 3: Design the software synthesis for ForSyDe-IO on RTOS platform.
- Subgoal 4: Implement the code generator which automatically generates C code for ForSyDe-IO.

1.5 Research Methodology

The focus of this research is to design a software synthesis for ForSyDe-IO and implement the code generator.

In this thesis, the deduction approach is used. The hypothesis is that the designed software synthesis method and code generator could work for ForSyDe-IO containing only the SDF model.

In order to prove hypothesis, data such as output data stream, code size and execution time need to be collected and analyzed. As a result, the quantitative research is used in this thesis.

1.6 Delimitations

This project only focuses on the code generation for the software applications, the hardware codes such as VHDL are not concerned. Secondly, the ForSyDe-IO files are supposed to contain the SDF models, and other MoC are excluded.

1.7 Structure of the thesis

Chapter 1 defines the problems and delimitations of this master thesis. Chapter 2 introduces the basic concepts of SDF, the software synthesis and optimization of SDF, ForSyDe, ForSyDe-IO, Xtend language, and testing boards used in this project. Chapter 3 includes the software synthesis model for ForSyDe-IO, the research process of this thesis, and the design of two experiments. Chapter 4 discusses the data flow in ForSyDe-IO . Chapter 5 introduces the software synthesis model for ForSyDe-IO in detail. Chapter 6 shows the design of the automatic code generator. The design pattern and the overall structure of the code generator are proposed. In Chapter 7, the results of experiments designed in Chapter 3 are shown. Chapter 8 analyzes the experiment results. In the end, Chapter 9 is concerned with the final conclusion and the future work.

Chapter 2

Background

This chapter provides necessary background about synchronous data flow (SDF), software synthesis of SDF, ForSyDe and ForSyDe-IO, Xtend, and the information of testing boards used in this project.

2.1 Synchronous Data Flow Graph

In this section, the basic concepts of graph theory and synchronous data flow [1, 2] are presented.

2.1.1 Graph Concepts

A graph G is a pair sets $G = (V, E)$, where V is a set of vertexes and E is a set of edges (A edge is a pair of vertexes). A directed graph is a graph in which the edge has the direction.

$V(G)$ represents the set of all vertexes in G. $E(G)$ represents the set of all edges in G. For an edge e in directed graph G, $src(e)$ represents the source vertex of edge e, $snk(e)$ represents the sink (destination) of edge e.

A subgraph of G is formed by vertex set $Z \subseteq V(G)$ together with the set of edges $\{e \in V(G) | src(e) \in Z, snk(e) \in Z\}$, and this subgraph is denoted as $subgraph(Z, G)$.

A path in directed graph G is a nonempty sequence $e_1, e_2, ... \in E$ such that $snk(e_i) = src(e_{i+1})$. A cycle represents a path that starts from a vertex to itself. A graph is acyclic if it does not have cycles.

A directed graph G is strongly connected if for each vertex pair v_i, v_j, there exists one path from v_i to v_j and another path from v_j to v_i. A strongly connected component of directed graph G is a strongly connected subgraph

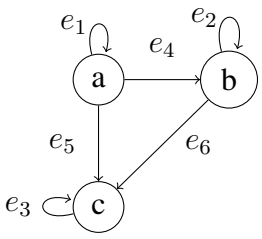


Figure 2.1: A directed graph example

$subgraph(Z, G)$ such that there is no other strongly connected subgraphs of G which contains $subgraph(Z, G)$.

Given a directed graph G, a vertex v is a root vertex if there is no edge in G such that $snk(e) = v$.

For Example, Figure 2.1 shows a directed graph G. It contains three vertexes(nodes) and six directed edges.

$$V(G) = \{a, b, c\} \tag{2.1}$$
$$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6\} \tag{2.2}$$
$$src(e_4) = a \tag{2.3}$$
$$snk(e_4) = b \tag{2.4}$$

2.1.2 Data Flow Graph

Many streaming media applications can be represented by the data flow graph [16]. In the data flow graph [16], the node or actor represents the application and the edge or channel represents the stream buffer. The data on the buffer is called a token. A node (actor) can be executed (fired) whenever there are enough tokens on the input channels (buffers). The vertex (actor) consumes tokens in the input channels and produces tokens on the output channels. Normally, the channels are represented as FIFO queues with infinite capacity. The token, which is firstly input into the channel, must firstly be consumed.

2.1.3 Fundamentals of Synchronous Data Flow Graph

Synchronous data flow (SDF) graph [1] is a restricted version of the data flow graph, in which the number of tokens consumed by actors and the number of tokens produced by actors are static and known at the compile time. Formally, the source actor of edge e is $src(e)$, and the sink actor of edge e is $snk(e)$. The edge of the SDF graph has three properties:

1. production $prd(e)$, the number of tokens generated onto channel e when executing actor $src(e)$.

2. consumption $cns(e)$, the number of tokens consumed from channel e when actor $snk(e)$ fires.

3. delay $delay(e)$, is a non-negative integer that represents the initial number of tokens on channel e.

For example, Figure 2.2 shows an SDF graph.

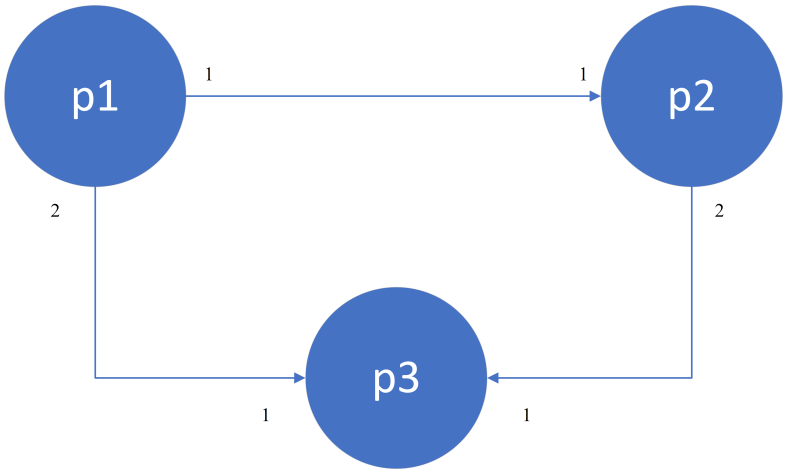
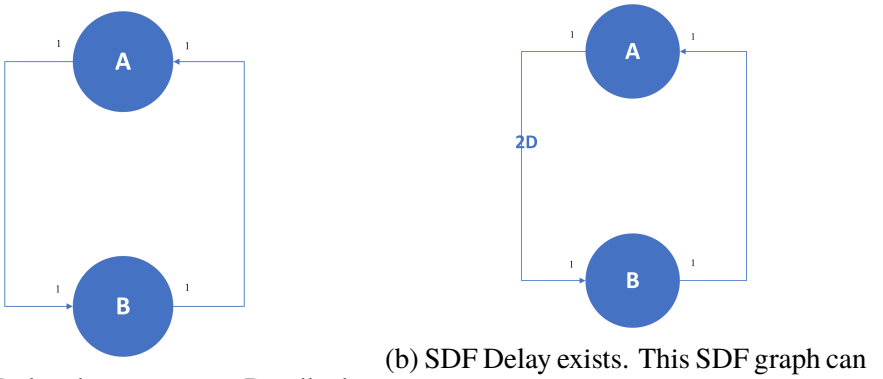


Figure 2.2: A SDF graph example

In SDF, Delay "D" represents the initial token in the channel. For example, 2D means there are 2 tokens at the beginning. In Figure, there is no delay, actor A tries to read tokens from the input channel, which is empty. Actor B tries to read tokens from its input channel, which is also empty. As a result, it is a deadlock and the SDF graph in Figure 2.3a can not run. However, in Figure 2.3b, the SDF graph can run because of "2D".

2.1.4 Topology Matrix and Repetition Vector

Compared with the data flow graph, the SDF graph's analysability is enhanced. An SDF graph can be depicted by the topology matrix [1, 2]. A topology matrix is a $m \times n$ matrix Γ, whose row is indexed by edges (external edges are discarded) and whose column is indexed by actors. Each element is defined by Equation 2.5. In matrix Γ, element $\Gamma(i, j)$ represents the production by the



(a) SDF Delay does not exist. Deadlock

(b) SDF Delay exists. This SDF graph can run.

Figure 2.3: Two SDF Delay examples

actor j on edge i.

$$\Gamma(i,j) = \begin{cases} prd(e), \text{if actor } j \text{ is a } src(e) \\ -cns(e), \text{if actor } j \text{ is a } snk(e) \\ 0, \text{other} \end{cases} \tag{2.5}$$

where e is the edge i. However, the topology matrix does not express the external edges and self-loop edges. The delay is not included in topology matrix.

For example, for the SDF Graph in Figure 2.2, the topology matrix is

$$\Gamma = \begin{pmatrix} 1 & -1 & 0 \\ 2 & 0 & -1 \\ 0 & 2 & -1 \end{pmatrix} \tag{2.6}$$

2.1.5 SDF Graph Scheduling

When discussing SDF graph scheduling, the capacity of FIFO channels is finite. An SDF periodic schedule is a feasible schedule which fires each actor at least once and causes no change in the state of SDF graph. Obviously, for an edge α, in one period,

$$i \times prd(\alpha) = j \times src(\alpha) \tag{2.7}$$

where i (j) is the number of invocation of actor $src(\alpha)$ ($snk(\alpha)$) in one period.

Equation 2.7 means that for an SDF graph with a feasible periodic

schedule, the number of tokens produced on edge e is equal to the number of tokens consumed on edge e.

2.1.5.1 Dynamic Scheduling

The scheduling approaches consist of dynamic scheduling and static scheduling. In dynamic scheduling, the tasks are scheduled by a scheduler and are scheduled at run-time. On a platform with OS, the SDF graph is dynamically scheduled.

Round-robin scheduling is a kind of dynamic scheduling method. Tasks with the same priority are assigned with a time slice. When the time slice expired, the task is switched.

2.1.5.2 Static Scheduling

In contrast to dynamic scheduling, static scheduling is decided at compile time. The advantage of static scheduling is that there is no extra overhead, which is caused by the scheduler.

For an SDF graph, a feasible static schedule on a uniprocessor platform is called periodic admissible sequential schedule (PASS) [2] and a feasible schedule on a multiprocessor platform is called periodic admissible parallel schedule (PAPS) [2]. Lee et al [2] proposed the necessary condition the for the existence of PASS:

$$\text{rank}\,(\Gamma) = n - 1 \tag{2.8}$$

where n is the number of actors in the SDF graph.

For the SDF graph with PASS, the number of invocations of actors in one period is decided by the repetition vector. Repetition vector is a $n \times 1$ vector v, whose element $v(i)$ is the number of invocations of actor i in one period. n is the number of actors and $i \in \{1, 2, ..., n\}$.

$$\Gamma q = \mathbf{0} \tag{2.9}$$

The repetitions vector is the null space of topology matrix. Given an SDF graph α, if a feasible schedule on the uniprocessor platform exists, the repetitions vector satisfies Equation 2.9.

For example, for the SDF graph in Figure 2.2, the topology matrix is

Equation 2.6. The rank and the repetition vector are as follows:

$$rank(\Gamma) = 2 \tag{2.10}$$

$$q = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} \tag{2.11}$$

The necessary condition of PAPS is also Equation 2.8. The first step of building a periodic admissible parallel schedule (PAPS) [2] is to construct the precedence graph.

The precedence graph specifies the precedence relationships between actor invocations. In the precedence graph, each vertex corresponds to an actor firing. If, in the SDF graph, the firing B_j consumes at least one token produced by firing A_i, then, there is a directed edge in the precedence graph from vertex A_i to vertex B_j.

For a small SDF graph, the PAPS can be constructed directly from the precedence graph. But for the large and complex SDF graphs, Lee et al [2] proposed an algorithm that generates precedence graph.

2.1.6 SDF Graph Buffer

The SDF graph buffers are FIFO queues (first in first out) with finite capacity. Given an SDF graph G, $max\_token(e, G)$ denotes the maximum number of tokens queued on the buffer e. The buffer memory of the SDF graph G is given by

$$buffer\_memory(G) = \sum_{e \in G} max\_token(e, G) \tag{2.12}$$

Normally, buffers are evaluated by four qualities: the logical size of the buffer, whether the buffer is contiguous, whether accessing the buffer is static, and whether the buffer is circular or linear.

2.1.6.1 Static Buffer

A buffer can be classified into the static buffer and dynamic buffer. For an SDF buffer α, if the ith token in one period resides in the same memory location as ith token accessed in any other period, this buffer is static, otherwise, this buffer is a dynamic.

The method [1] to find the static buffer size is as follows: Static buffer α

with size N occurs if and only if, there exists a positive integer K such that

$$i \times q = K \times N \tag{2.13}$$

where node A fires q times in one cycle and the production or consumption rate is i, and the buffer size is N. Node A can be either source node or target node, if node A is the source node, i is production, otherwise, i is consumption.

2.1.6.2 Contiguous Buffer Versus Scattered Buffer

According to the knowledge of data structure, the FIFO can be implemented by both a circular linked list (scattered) and a circular array (contiguous). The linked list is appropriate for the FIFO whose size is unknown. The scattered method may introduce the memory fragment problem and thus causes low memory efficiency. For the circular buffer, two pointers pointing to the $begin\_location$ and $end\_location$ should be maintained as the inner state. Wrapping around the buffer needs to be carefully maintained.

2.1.7 SDF Graph Clustering

Different nodes in SDF graph G can be clustered together into a subgraph Z. This subgraph Z and the remaining nodes in G compose a new graph G'.

For each node $A \in Z$,

$$q_G(A) = q_G(Z) \times q_Z(A) \tag{2.14}$$

where q_G is the repetitions vector of graph G and q_Z is the repetitions vector of subgraph Z.

$q_G(Z)$ represents the number of times subsystem Z is invoked.

$$q_G(Z) = gcd(\{q_G(A_1), q_G(A_2), ..., q_G(A_s)\}) \tag{2.15}$$

where gcd represents the greatest common divisor

Each input arc α to the subgraph Z is replaced by an arc α' such that

$$snk(\alpha') = \Omega \tag{2.16}$$
$$src(\alpha') = src(\alpha) \tag{2.17}$$
$$prd(\alpha') = prd(\alpha) \tag{2.18}$$
$$delay(\alpha') = delay(\alpha) \tag{2.19}$$
$$cns(\alpha') = cns(\alpha) \times q_G(sink(\alpha))/q_G(Z) \tag{2.20}$$

Each output arc β from the subgraph Z is replaced by an arc β' such that

$$snk(\beta') = snk(\beta) \tag{2.21}$$
$$src(\beta') = src(\Omega) \tag{2.22}$$
$$delay(\beta') = delay(\beta) \tag{2.23}$$
$$c(\beta') = c(\beta) \tag{2.24}$$
$$prd(\beta') = prd(\beta) \times q_G(sink(\beta))/q_G(Z) \tag{2.25}$$

2.1.8 SDF Software Synthesis

SDF graph software synthesis or SDF graph compilation represents the transformation from an SDF graph to the target code. Generally, there are approaches [17], one is called threading, the other is called synthesis.

In threading, the schedule is firstly generated, then, code generator steps through the schedule and sequentially insert actor code block which is predefined in the actor library. In the storage allocation phrase, buffers and assigned variables in memory are inserted into the sequence of code blocks.

In synthesis, the SDF graph is firstly transferred into an intermediate format, and then this intermediate format is compiled into C or other high-level language.

In paper [18], a software synthesis method is proposed. In Figure 2.4, the whole software synthesis task is divided into the process layer, subsystem layer, and system layer [18].

Steps in Figure 2.5 are proposed by the paper [18]. In the process layer, the skeleton templates (functions) are firstly constructed, and the process synthesis is based on the created skeleton templates. For example, for an actor with two input ports and one output port, the skeleton template is `actor21SDF(...)`. The data types are mapped into the corresponding target language. Combinatorial functions are passed into the skeleton

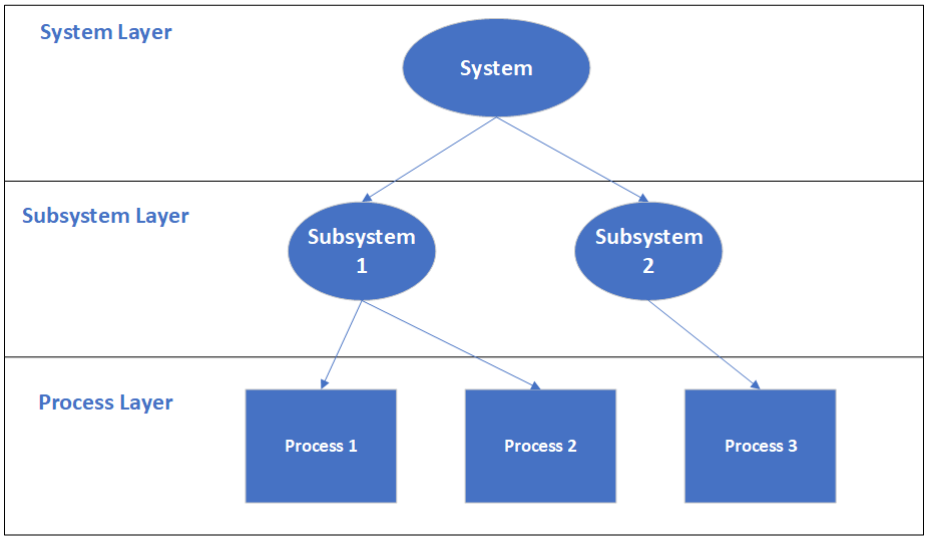


Figure 2.4: Three-layer software synthesis

templates. In subsystem layers, the processes form the subsystem according to the static process' schedule. In system layers, subsystems form the system according to subsystems' schedule.

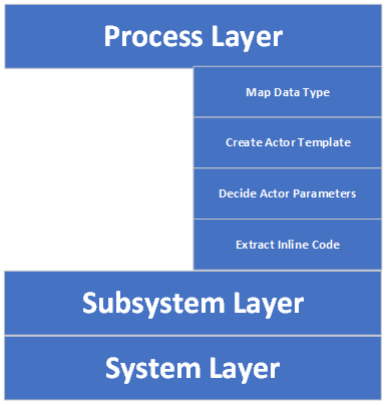


Figure 2.5: Detailed software synthesis procedure [18]

2.2 Code Size Optimization in SDF Software Synthesis

In this section, technology about code size optimization in SDF software synthesis is presented.

In software synthesis, inline code can eliminate the overhead of function call. However, it may introduce the code size explosion. As a result, code size optimization aims to find a schedule which makes the inline code size minimum.

2.2.1 Basic Concepts about Single Appearance Schedule

Single Appearance Schedule (SAS) is a special SDF graph schedule. In a single appearance schedule, each actor appears only once. For example, the schedule $(ABAB)$ is not a single appearance schedule, but $(2AB)$ is a single appearance schedule. The single appearance schedule can minimize the code size.

Definition 2.2.1 (Schedule Loop and Looped Schedule [19]) *A schedule loop is $(nS_1S_2...S_m)$, where S_i is either an actor or a schedule loop, and n is a positive integer and is called iteration count. $(nS_1S_2...S_m)$ represents a loop with n times iteration, inside the loop is the firing sequence $S_1S_2...S_m$. A looped schedule is a sequence $V_1V_2...V_k$ where each V_i is either an actor or a schedule loop.*

For example, (2AB) represents ABAB, and (2A(3BC)) represents ABCBCBCABCBCBC. Given a looped schedule L , a node A or a schedule loop A, $appearance(A, L)$ denotes the number of times that A appears in schedule L. For example, $appearance(A, (2AB)) = 1$

Definition 2.2.2 (Single Appearance Schedule [19, 20]) *Given a looped schedule L, if for each node A in L, $appearance(A, L)$ is 1, then, schedule L is a single appearance schedule.*

For an acyclic SDF graph G, constructing a single appearance schedule is simple. Because there is no loop in the directed graph, each time, fire the root node $q(A)$ times (q is the repetition vector). The single appearance schedule can be written as $(n_1A_1)(n_2A_2)...(n_mA_m)$ where n_i is the iteration count for node A_i. For an arc α, We denote the total number of tokens consumed on arc α in one period as $total\_consume(\alpha)$. $total\_consume(\alpha) = q_G(snk(\alpha)) \times c(src)$.

Definition 2.2.3 (subindependence [19, 20]) *Given an SDF graph G, the node set of G is denoted as $V(G)$, If $Z_1 \subseteq V(G)$ and $Z_2 \subseteq V(G)$, Z_1 and Z_2*

are disjoint. We say that "Z_1 is subindependent of Z_2" if for every arc α where $src(\alpha) \in Z_2$ and $snk(\alpha) \in Z_1$, we have $delay(\alpha) \geq total\_consume(\alpha)$. If $Z_1 \bigcup Z_2$ is G, then we have Z_1 is subindependent of G and we denote it as $(Z_1|GZ_2)$.

Z_1 is subindependence of Z_2 represents that no tokens produced from Z_2 are consumed by Z_1.

Definition 2.2.4 (loosely interdependent and tightly interdependent [19, 20]) *Given an nontrivial strongly connected SDF graph G. We say G is loosely interdependent if there exists a node partition X and Y for G, such that $X|GY$. Otherwise, we say G is tightly interdependent.*

Theorem 1 (existence of single appearance schedule [19, 20]) *An SDF graph has the single appearance schedule if and only if each strongly connected component has a single appearance schedule. An strongly connected SDF graph G has a single appearance schedule only if there exists a partition P_1, P_2 such that P_1 is subindependent of G.*

Theorem 1 means the existence of a single appearance schedule is equivalent to the absence of tightly interdependent subgraphs. If the γ is the greatest common divisor of $\{n_1, n_2, ..., n_k\}$, then the generated schedule is fully reduced. However, this method increases the runtime cost of starting loops. Additionally, factoring is not a legal transformation in the schedule which is not a single appearance schedule.

2.3 ForSyDe

ForSyDe [13], is a methodology for modeling and designing heterogeneous systems-on-chip. ForSyDe aims to let designers build a systems-on-chip at the high-level and without considering implementation details. ForSyDe is firstly written in the functional language Haskell, whose high-order function supports concepts in system modeling. Then, ForSyDe is ported to SystemC and is called ForSyDe-SystemC.

Figure 2.6 shows the design flow of ForSyDe. ForSyDe begins with the specification phase. The *specification* phase is an abstraction of system functionality. It separates the computation from communication and separates system function from detailed architecture. In this phase, the system model is refined and validated.

The next phase is the *design space exploration*. with the help of DeSyDe, the software applications and hardware platforms are mapped together.

The third phase in ForSyDe is *implementation* phase. The input of this phase contains all the implementation details, for instance, the size of each buffer and the static schedule. In this phase, the code generation for both software and hardware is implemented.

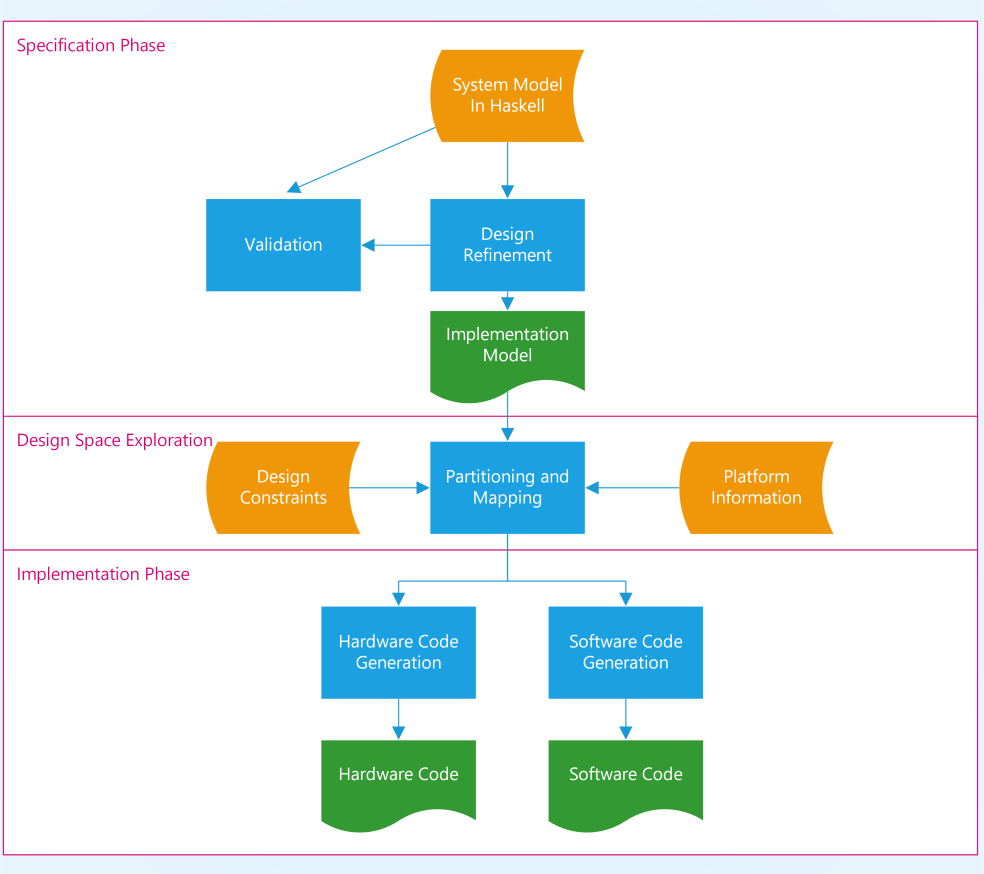


Figure 2.6: Design flow of ForSyDe

2.3.1 ForSyDe System Modeling

A ForSyDe system model consists of signals and processes. The signal is an infinite or finite list and is defined as:

$$\vec{s} = \langle v_1, v_2, v_3, ... \rangle \tag{2.26}$$

A process is defined as a mapping, which maps m input signals $\vec{i}_1, \vec{i}_2..., \vec{i}_m$ into n output signals $\vec{o}_1, \vec{o}_2..., \vec{o}_n$:

$$P(\vec{i}_1, \vec{i}_2..., \vec{i}_m) = (\vec{o}_1, \vec{o}_2..., \vec{o}_n) \tag{2.27}$$

The communication between processes is modeled by signal.

2.3.2 ForSyDe-Shallow

Forsyde-Shallow [14], a sub-library of ForSyDe, contains the main concepts of ForSyDe and is written in the functional programming language Haskell. Synchronous computational model, synchronous data flow MoC, and continuous-time MoC are contained in the Forsyde-Shallow. This project only focuses on the synchronous data flow MoC.

In ForSyDe, processes are constructed by the process constructor provided by the library ForSyDe-Shallow. In Figure 2.7, each process constructor takes functions and values as input parameters and returns a process function.

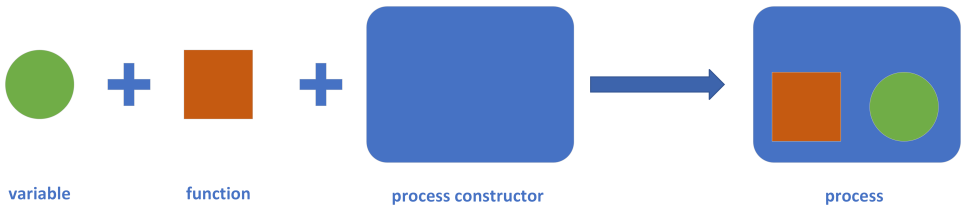


Figure 2.7: Function, variable, and processor constructor together build a process.

In ForSyDe, the functions are classified into two categories: 1) Combinatorial function 2) Sequential function. Combinatorial functions are functions without internal states. While, sequential functions are functions with internal states, such as infinite state machine.

2.3.2.1 Signal

In Forsyde-Shallow, the signal is defined as a sequence of events. Each event has a tag and value which represent the order and value of the corresponding event. Signals are used to transmit information between processes.

Listing 2.1: An example of a signal in ForSyDe-Shallow

```
let s1 = signal [7,8,9]
```

2.3.2.2 Process Constructor For Synchronous MoC

This section briefly introduce the API about the synchronous MoC.

- **Processor Constructor: Combinatorial**

  In Figure 2.8, $combXSY(f)$ is a process constructor for the synchronous model of computation. It takes a combinatorial function as a parameter and returns a process which takes X input signals and produces 1 output signal.

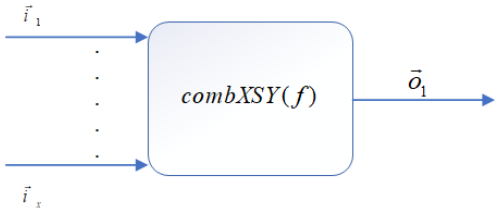


Figure 2.8: Process constructor $combXSY(f)$ for combinatorial function. X is an integer.

  In the ForSyDe.Shallow.MoC.Synchronous library, $combSY$, $comb2SY$, $comb3SY$, $comb4SY$ are provided.

- **Process Constructor: Sequential**

  In Figure 2.9, $delaySY$ delays the input signal for several cycles and inserts the initial tokens into the prefix of the output signal. Besides

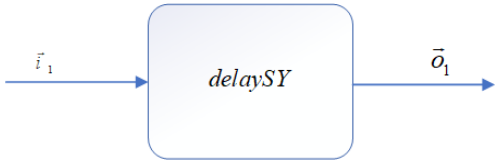


Figure 2.9: Process constructor $delaySY$.

  the $delaySY$, there are still some process constructors for sequential functions. For example, in Figure 2.10, $scanlXSY(f)$ returns a finite state machine process, which takes X input signals and produces one output signal which is computed by both the function and inner state.

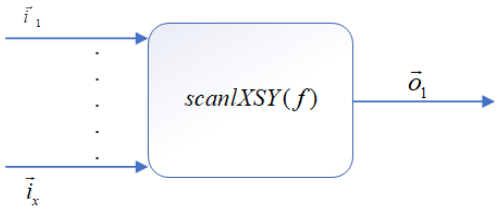


Figure 2.10: Process constructor $scanlXSY(f)$, X is an integer.

- **Process Constructor: Others**

  Some process constructors which provide convenient functionalities are provided. For example, in Figure 2.11, $zipXSY$ returns a process which takes X input signals and outputs a signal of a tuple of these X signals.

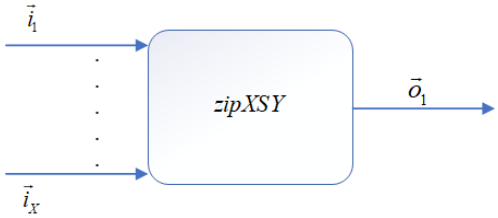


Figure 2.11: Process constructor $zipXSY$ for combinatorial function. X is an integer.

2.3.2.3 Process Constructor For Synchronous Dataflow MoC

This section contains the API for the SDF MoC. In the SDF model, the only sequential process constructor is the $delaySDF$.

- **Process Constructor: Combinatorial**

  In Figure 2.12, process constructor $actorXYSDF(f)$ returns a process which takes X signals as input and outputs Y signals.

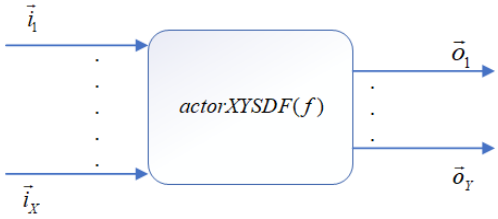


Figure 2.12: Process constructor $actorXYSDF(f)$ for combinatorial function. X and Y are integers.

- **Process Constructor: Delay**

  In Figure 2.13, $delaySDF$ is similar to $delaySY$. It delays the input signal for several cycles and inserts the initial tokens into the prefix of the delayed input signal.

Figure 2.13: Process constructor $delaySDF$.

2.3.3 DeSyDe

DeSyDe [15, 21] is a subproject under ForSyDe [14] and is a design space exploration tool. It takes platform.xmi, application.xmi, WCETs.xmi as input and produces the mapping of software and hardware, together with the static schedule. The design constraints are satisfied. The functionalities of platform.xmi, application.xmi, wcet.xmi are as follows:

1. platform.xmi: information about the platform.
2. application.xmi: software applications.
3. WCETs.xmi: worst case execution time.

2.4 ForSyDe IO

ForSyDe IO [22] is the output of the *design space exploration* and is input into the *implementation* phase in ForSyDe. A JAVA library forsyde-io-java and a python library forsyde-io-python are maintained for ForSyDe-IO.

ForSyDe IO can be stored in both xmi files and fiodl files. In this section, only the fiodl format is introduced.

ForSyDe-IO is a directed graph with loops and multiple edges between vertexes. In this directed graph, each node contains: (i) a unique identifier (ii) a set of ports (iii) a set of properties (iv) a set of traits. Each edge contains: (i) a unique identifier (ii) a set of ports (iii) a set of traits.

Listing 2.2 shows a ForSyDe-IO vertex in fiodl format. In the brackets are the vertex's traits. In the parenthesis are the vertex's ports. Inside the brace are the vertex's property name and the corresponding property value.

Listing 2.2: ForSyDe-IO vertex in fiodl format

```
vertex name
[vertex_trait_1, vertex_trait_2]
(port_1, port_2, port_3)
{
    property_1: property_1_value,
    property_2: property_2_value,
}
```

Listing 2.3 shows an edge in ForSyDe-IO. In the brackets are the edge's traits. The `src_vertex` is the source vertex, and the `src_port` is the source port. The `snk_vertex` is the sink vertex, and the `snk_port` is the sink port.

Listing 2.3: ForSyDe-IO edge format in fiodl format

```
edge [edge_trait_1, edge_trait_2,...] from
  ↪ src_vertex port src_port to snk_vertex port
  ↪ snk_port
```

2.4.1 SDF Channel in ForSyDe-IO

In ForSyDe-IO, the channel in the SDF model is expressed by a vertex with the trait "moc::sdf::SDFChannel", and it must have two ports: producer port, and consumer port.

The ”producer” port is connected to the source actor’s node and the ”consumer” port is connected to the sink actor’s node. For example, Listing 2.4 shows how an SDF channel is stored in the fiodl. In Listing 2.4:

- The SDF channel’s identifier is `s1`
- The element `tokenSizeInBits` shows the size of token type in bits. In this channel, each token has 32 bits.
- The `maximumTokens` shows the max number of tokens in the channel, and in this channel, the max number of tokens is 20.
- The `maxSizeInBits` shows the memory size this channel needs, in this channel, it needs at least 32 bits memory space.
- The `numOfInitialTokens` represents the number of SDF delays. In this example, there is no SDF delays in `s1`.

Listing 2.4: An example of a SDFChannel named s1 in ForSyDe-IO fiodl file

```
vertex s1
[decision::sdf::BoundedSDFChannel, moc::sdf::
    ↪ SDFChannel, visualization::Visualizable]
(consumer, initialTokenValues, producer)
{
    ”numOfInitialTokens”: 0_i,
    ”maxSizeInBits”: 32_l,
    ”maximumTokens”: 20_i,
    ”tokenSizeInBits”: 32_l
}
```

2.4.2 SDF Actor and Functions in ForSyDe-IO

In ForSyDe-IO, the SDF actor is represented by a vertex with the trait ”moc::SDF::SDFComb”. Normally, an SDF Actor vertex contains some input ports, some output ports, and a port named ”combFunctions”.

For the SDF Actor vertex, each input port is connected to the input SDF channel vertex. Each output port connects to an SDF output channel vertex. The input tokens are read from the input channels via input ports and output tokens are written to the output channels via output ports.

When an SDF Actor fires, its functions are executed. The SDF Actor's functions are not inside the SDF Actor, but in other vertexes, which are connected to the SDF Actor via SDF Actors' "combFunctions" port.

The function in ForSyDe-IO is represented by a vertex, which, in this paper, is called the *combFunction vertex*. The *combFunction vertex*'s input ports and output ports are the input parameters and output parameters of this function. Further, these ports connect the vertexes representing data types. The definition of the function is contained in *combFunction vertex*'s "inlineCodes" property.

Listing 2.5: An example of a SDF Actor in ForSyDe-IO fiodl file

```
vertex p1
[decision::sdf::PASSedSDFActor,
  ↪ moc::sdf::SDFActor,
  ↪ visualization::Visualizable]
(combFunctions, s1_port, s2_port)
{
  "production": {
    "s1_port": 1_i
  },
  "consumption": {
    "s2_port": 2_i
  },
  "firingSlots": [
    0_i,
    4_i
  ]
}
```

Listing 2.5 shows an example of SDF Actor:

- vertex name is `p1`
- the consumption of port `s2_port` is 2, and the production of port `s1_port` is 1.
- `firingSlots` represents the static schedule of this SDF Actor on uniprocessor bare-metal. The schedule on multiprocessor bare-metal is introduced later.

Listing 2.6: An example of a SDF CombFunction vertex in ForSyDe-IO fiodl file

```
  vertex p1Body
  [impl::ANSICBlackBoxExecutable,
    ↪ typing::TypedOperation]
  (inputPortTypes, outputPortTypes, s1, s2)
  {
    "inlinedCode": "s1=s2;",
  }
edge [moc::AbstractionEdge] from p1 port
  ↪ combFunctions to p1Body
```

Listing 2.6 shows a *combFunction vertex*:

- vertex name is `p1Body`.
- `inlinedCode` represents the code inside this function.
- the `p1Body` vertex is connected to the vertex `p1`'s combFunctions port.

Figure 2.14 shows the connection among SDF Channel, SDF Actor, *combFunction vertex* and data types in ForSyDe-IO.

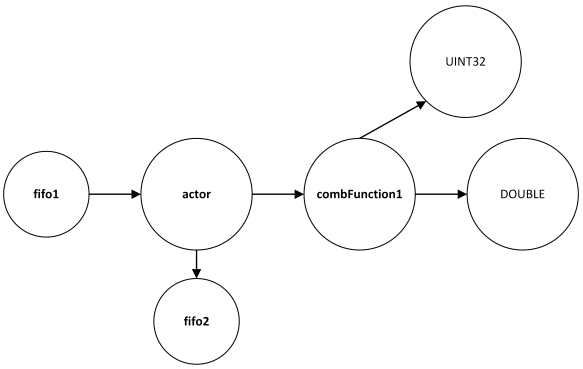


Figure 2.14: An example of vertex connection in ForSyDe-IO. The ports in the vertex are not drawn. `fifo1` and `fifo2` are SDF Channel vertexes. `actor` is the SDF Actor vertex. `UINT32` and `DOUBLE` are data-type vertexes. `comnFunction1` is the *combFunction vertex*.

2.4.3 Schedule of Multiprocessor Platform in ForSyDe-IO

In Listing 2.7, a schedule vertex is proposed.

- the schedule's name is `order_0`.

- it has trait `platform::runtime::RoundRobinScheduler`.
- the schedule `order_0` is on `tile0`.
- the SDF Actor `p1` is on `slot_0`.

. In this schedule vertex, it has ports named as slot[0], slot[1], slot[2], *etc.,* Each slot connects an SDF actor vertex.

Listing 2.7: An example of a schedule vertex in ForSyDe-IO fiodl file

```
vertex order_0
[platform::runtime::RoundRobinScheduler]
(contained, slot_0, slot_1, slot_2, slot_3,
  ↪ slot_4)
{
}

edge [AbstractScheduling] from order_0 port
  ↪ slot_0 to p1
edge [decision::AbstractAllocation] from tile0
  ↪ port execution to order_0
```

2.4.4 Processor in ForSyDe-IO

Listing 2.8 shows a processor vertex named `tile0`.

Listing 2.8: An example of a platform vertex in ForSyDe-IO fiodl file

```
vertex tile0
[platform::GenericProcessingModule,
  ↪ visualization::GreyBox,
  ↪ visualization::Visualizable]
(communication, contained, execution)
{}
```

Figure 2.15 shows how the processor vertex, schedule vertex, and actor vertexes are connected.

2.4.5 ForSyDe-IO Trait and Viewer

Recalling the previous sections, ForSyDe-IO traits are the labels for vertexes and edges. A vertex with a specific trait must have specific ports and properties. For example:

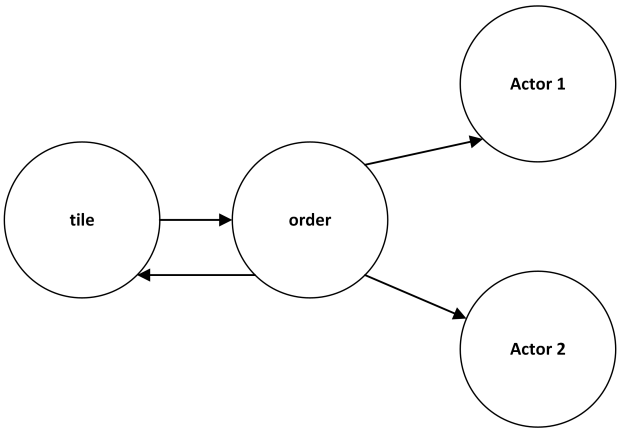


Figure 2.15: An example of vertex connection in ForSyDe-IO. The `tile` vertex is a generic processing platform, the `order` is the schedule on the tile. The vertexes' ports are not drawn. `Actor1` and `Actor 2` are the SDF Actor vertexes.

- a vertex with trait `moc::sdf::SDFComb` is the SDF Actor, which must have the `production` property, `consumption` property, and the `combFunctions` port.
- a vertex with trait `moc::sdf::SDFChannel` is the SDF Channel, which must have the `producer` port and `consumer` port.
- a vertex with trait `impl::ANSICBlackBoxExecutable` must have the `inlinedCode` property.

In order to access these special properties conveniently, the viewer hierarchy is supported in `forsyde-io-java` library. Viewer hierarchy provides methods to access these properties. For example:

- intereface `SDFComb` is the viewer for trait `moc::sdf::SDFComb`. It provides function `getConsumption()` which returns the consumption, `getProduction()` which returns the production, and the `getCombFunctionsPort()` which returns the *combFunction vertexes*.
- interface `ANSICBlackBoxExecutable` provides the method `getInlinedCode()` returning the inline code.

2.5 Xtend Language Introduction

In this project, the programming language Xtend [23] is used. Xtend is a general-purpose language based on the JVM and is a flexible dialect of Java. When compiling, the Xtend code is actually compiled into Java code. The Xtend's most important feature used in this project is the **Template Expressions**, which provides an easier way to construct a readable string.

Listing 2.9: Xtend template expression example

```
def String hello()
    var content = "helloworld!"
    '''
    <html>
    <body>«content»</body>
    </html>
    '''
```

The **Template Expressions** are surrounded by the triple single quotes (’’’), and the variables inside the template expression are surrounded by « (shift+<) and » (shift+>). For example, in Listing 2.9, the function `hello()` would return a string:

```
<html>
<body>helloworld!</body>
</html>
```

«variable» is replaced by the variable's actual value. In Listing 2.9, the «content» is replaced by `helloworld!`.

The template expression also provides the for each functionality and if else functionality.

Listing 2.10: Xtend template expression For Loop example

```
def String for_example()
    var String actor_name_set={"A", "B","C"}
    '''
    «FOR name: actor_name_set»
    void «name»();
    «ENDFOR»
    '''
```

In Listing 2.10, the function `for_example()` returns the following string:

```
void A();
```

```
void B();
void C();
```

Listing 2.11 shows a if else example in xtend.

Listing 2.11: Xtend template expression If Else Loop example

```
def String if_example()
    var int a=1
    '''
    «IF a==1»
        true
    «ELSE»
        false
    «ENDIF»
    '''
```

In Listing 2.11, the function *if_example()* returns the following string:

```
true
```

For more information, please visit Xtend webpage [23].

2.6 NucleoF401RE Board

Nucleo401RE board contains a STM32f401re microcontroller. STM32f401re microcontroller is a chip with Cortex-M4 architecture, without Memory Management Unit (MMU). Figure 2.16 shows the 4G physical memory space of NucleoF401RE board.

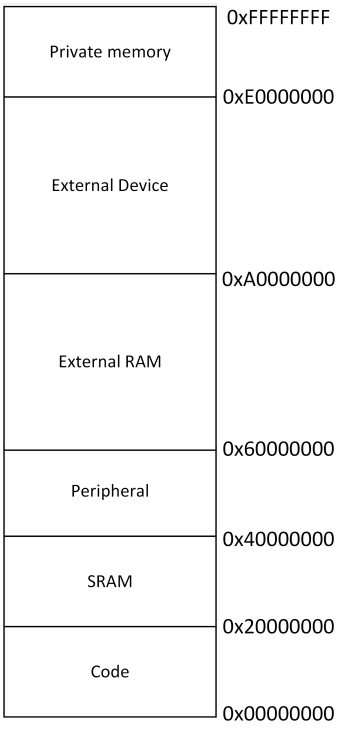


Figure 2.16: 4G physical memory map of stm32f401re microcontroller

2.7 CompSoC Board

The CompSoC board is based on the PYNQ board, which has one System on Chip (SoC) chip containing Processing System (PS) part and Programmable Logic (PL) part. On CompSoc board, the Linux runs on the PS part, and the PL part consists of three RISC-V processors, which are called tile0, tile1, tile2. For these three tiles, because of the usage of MMU, the memory addresses are all the virtualized memory addresses, instead of the physical addresses.

The communications between processors on PL are via the shared memory:

- MEM01 is a 32KB memory area shared by tile0 and tile1. For tile0 and tile1, the start addresses of MEM01 are both 0x80020000.
- MEM12 is a 32KB memory area shared by tile1 and tile2. For tile1 and tile2, the start addresses of MEM01 are both 0x80030000.
- MEM02 is a 32KB memory area shared by tile0 and tile2. For tile0, the start address is 0x80030000. For tile2, the start address is 0x80020000.

Figure 2.17 shows the memory architecture of CompSoC board.

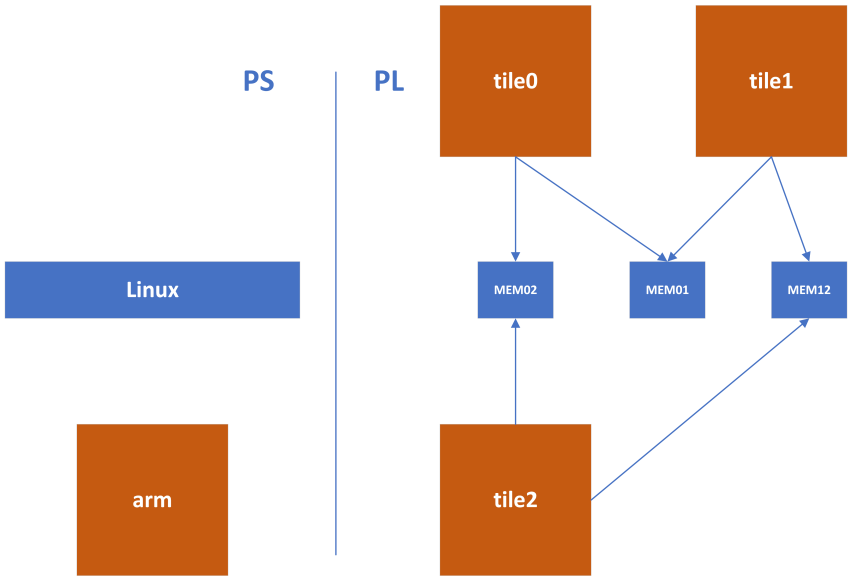


Figure 2.17: The memory architecture of CompSoC board

2.8 Summary

In this chapter, the necessary background about this project is given. First of all, the concepts of SDF are presented. Secondly, the software synthesis of SDF is shown. Thirdly, the ForSyDe and ForSyDe-IO are introduced. Then, the xtend language is proposed. Finally, the memory maps of two testing boards used in this project are proposed.

Chapter 3

Method

Section 3.1 describes the software synthesis method for ForSyDe-IO. Section 3.2 describes the research process. Section 3.3 presents the procedure of the testing experiment, and the two designed experiments. Section 3.4 the data collection.

3.1 Research Model

Zhonghai *et al.* [18] (described in Chapter 2) proposed a software synthesis method in ForSyDe. In paper [18], the software synthesis is divided into three layers. The bottom layer is the process layer. Processes in process layer can be glued together to build the middle layer—subsystem layer. Subsystem layers are further combined and form the system layer. This idea is realized in the `ForSyDe.Shallow` Haskell library. In this project, the software synthesis for ForSyDe-IO is developed from the software synthesis for ForSyDe in the paper [18], but some modifications are made.

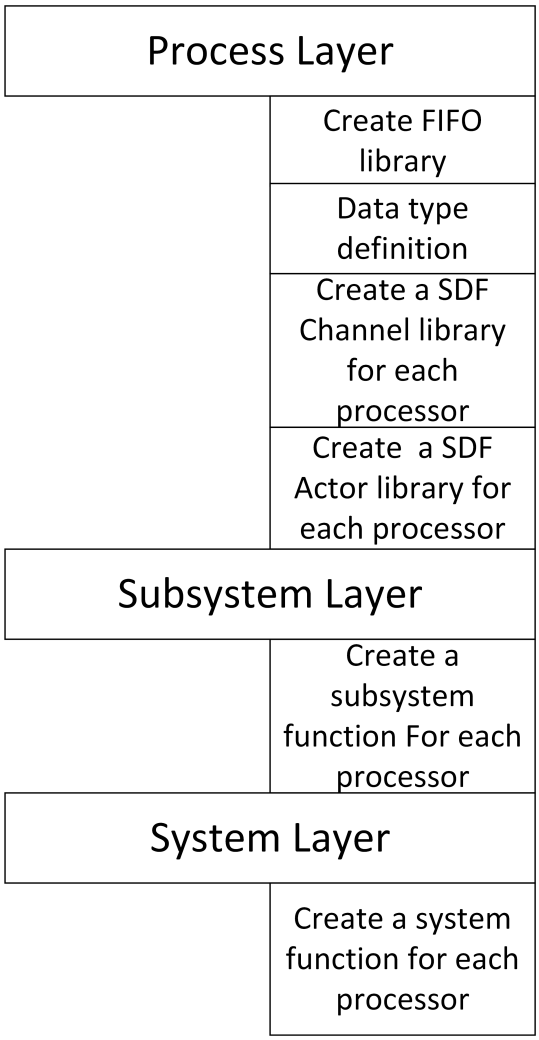


Figure 3.1: Software synthesis for ForSyDe-IO.

In this project, to synthesize the ForSyDe-IO into C code, steps in Figure 3.1 are followed. Figure 3.1 reflects the layered structure of the model. The synthesis task is divided into three sub-tasks, each of which is mapped to one layer in Figure 3.1. Because the ForSyDe-IO is the output of *desige space exploration*, the mapping of applications and hardware platforms must be considered in the software synthesis ForSyDe-IO. This is the main difference between software synthesis for ForSyDe [18] and software synthesis for ForSyDe-IO in this project.

In the process layer sub-task:

1. The FIFO library shared by all generic processing platforms (processors) is first created.
2. The data types defined in the ForSyDe-IO are extracted, defined and shared by all generic processing platforms (processors).
3. Thirdly, for each generic processing platform (processor), an SDF Channel library is created based on the created FIFO library.
   (a) All the SDF Channel vertexes on this generic processing platform are identified.
   (b) For each identified SDF channel vertex, the global FIFO variables are declared inside the SDF Channel library on this generic processing platform (processor).
4. In the end, for each generic processing platform (processor), a SDF Actor library is constructed.
   (a) All the actors (processes) on this generic processing platform are identified.
   (b) Each identified actor is constructed by a function, which is called *actor function*.
   (c) Inside the *actor function*, the memory variables are first declared. Then, the *actor function* reads data from input channels, executes the inlined codes from *combFunction vertex*es,and writes tokens to the output channels.

In the subsystem layer sub-task:

1. Firstly, for each generic processing platform (processor), a *subsystem initial function* is created. In the *subsystem initial function*, the corresponding SDF Channels (FIFOs) are initialized with the corresponding SDF Delay elements.
2. Secondly, for each generic processing platform (processor), a *subsystem function* is created. Inside this *subsystem function*, actors mapped into this platform are called according to the schedule of this platform.

The final sub-task is the system layer sub-task, which is similar to the subsystem layer sub-task. As the subsystem element is a netlist of actors, the system is a netlist of subsystems.

1. Firstly, for each generic processing platform (processor), a *system initial function* is created. In the *system initial function*, the FIFOs between subsystems are initialized with the corresponding SDF Delay elements.

2. Secondly, for each generic processing platform (processor), a *system function* is created. Inside this *system function*, subsystems mapped into this platform are called according to the schedule of this platform.

Thus, in this way, the ForSyDe-IO with SDF model is hierarchically synthesized into C code.

Since, in ForSyDe, SDF actors only take the combinatorial functions as input, the *actor functions* in Figure 3.1 do not contain the state variables. However, in ForSyDe-IO, for the actors which need the state variables, state variables are implemented by self-looped FIFOs. For each state variable:

1. The actor reads the value of this state variable from a FIFO.

2. The actor updates this state variable.

3. The actor writes the updated state variable value back to the same FIFO.

3.2 Research Process

This section introduces the main process of the master thesis and the further division of some steps.

3.2.1 Overall Process

The overall process of this project can be divided as follow steps:

Step 1 literature study.

Step 2 implement a software synthesis method transforming ForSyDe-IO to C code and implement the automatic code generator.

Step 3 conduct the experiment.

Step 4 discuss the results.

In step 1, the literature study is focused on the SDF software synthesis. Several materials about ForSyDe, ForSyDe-IO, and Xtend are also studied. In step 2, the software synthesis is going to be implemented based on Section 3.1, and the code generator will be implemented in Xtend and Java. The derived code generator will be tested on two ForSyDe-IO examples.

The step 2 is further divided based on platform's type. In Figure 3.2, for each platform type in tier 2, the software synthesis model in Section 3.1 is carried out.

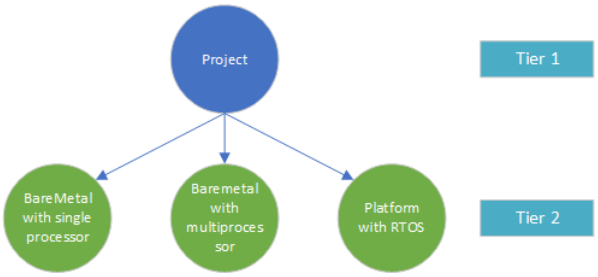


Figure 3.2: Divide the project according to the platform.

3.3 Experimental Design

The derived code generator is tested on two ForSyDe-IO examples. The goals of experiment are as follows:

1. Test if the code generator can derive the expected C code.
2. Test if the derived C code works as expected.

Figure 3.3 shows the overall experiment procedure. In Figure 3.3:

1. ForSyDe-IO is input into the code generator, which outputs the C code according to the platform and ForSyDe-IO. The testing platforms are uniprocessor bare-metal platform, multiprocessor bare-metal platform, and RTOS platform.
2. Secondly, the input data streams are put into the derived C code, and the output data streams are collected.
3. Finally, to determine if the C code runs correctly, the collected output data streams are compared with the expected output data streams.

3.3.1 Experiment One

The first experiment example is the sobel example in Figure 3.4, which computes the grayscale for one RGB pixel and applies the sobel operator for exact one pixel.

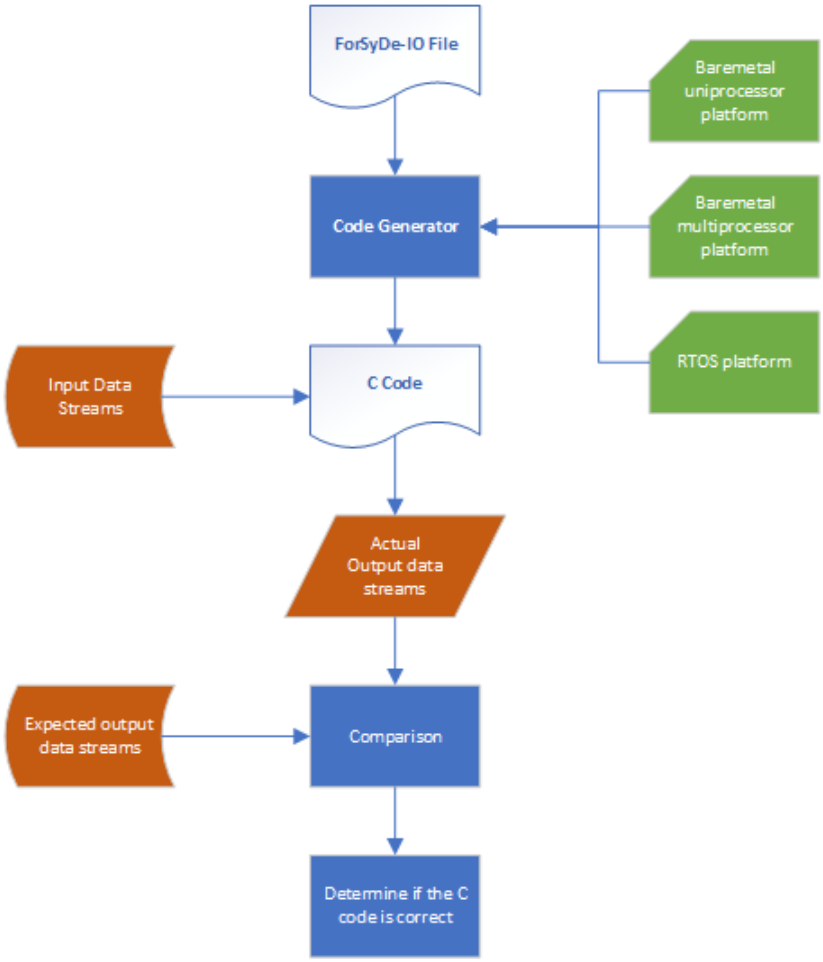


Figure 3.3: Experiment procedure.

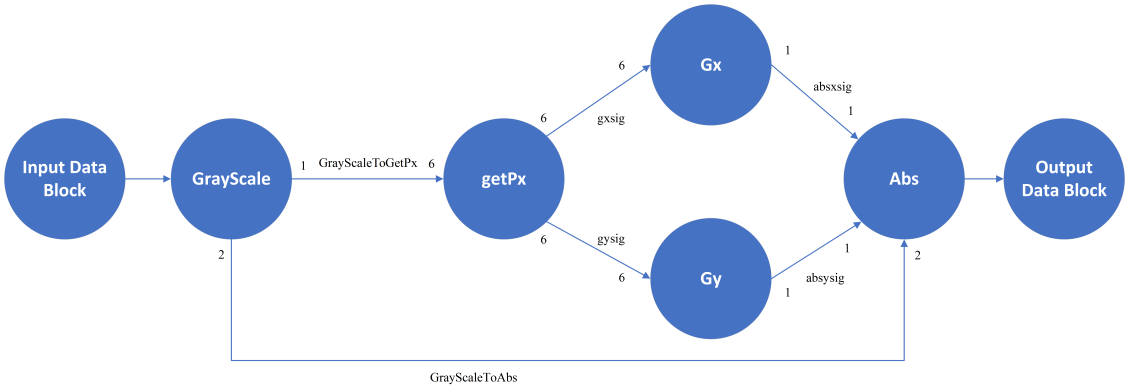


Figure 3.4: SDF graph of experiment one.

1. actor *GrayScale* computes the gray scale of one RGB pixel.
2. the computed gray scale is passed to the actor *getPx*, which further passes the gray scale into actor *Gx* and actor *Gy*.
3. the actor *Gx* and actor *Gy* implement sobel operator in x-direction and y-direction, and pass the values to actor *Abs*.
4. in actor *Abs*, the result gradient magnitude is computed by adding the approximations in x-direction and y-direction, and is written to the output data block.

Figure 3.5 shows how experiment one is partitioned on the uniprocessor bare-metal. All the SDF actors and SDF channels are on the same CPU.

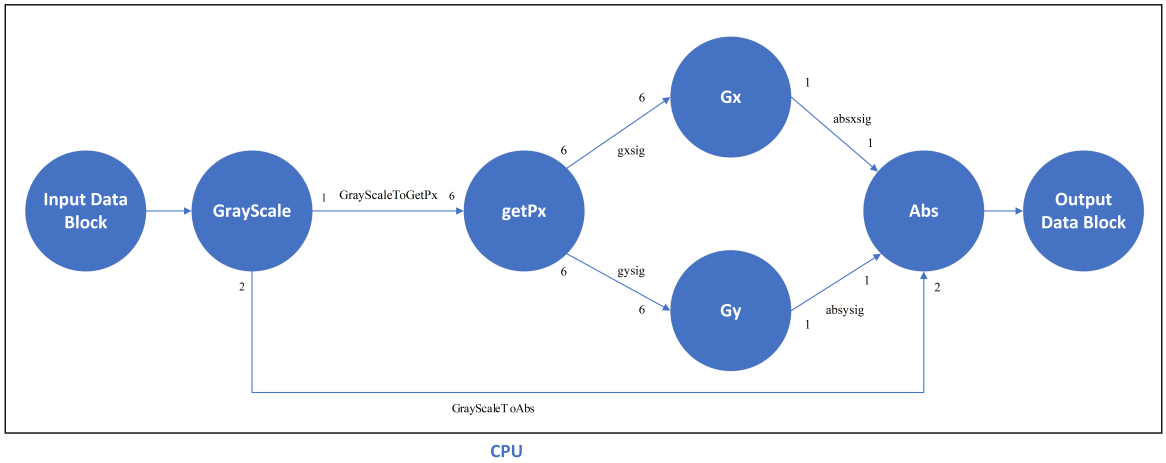


Figure 3.5: Experiment one partitioned on uniprocessor bare-metal.

Figure 3.6 shows how example one is partitioned on the RTOS platform.

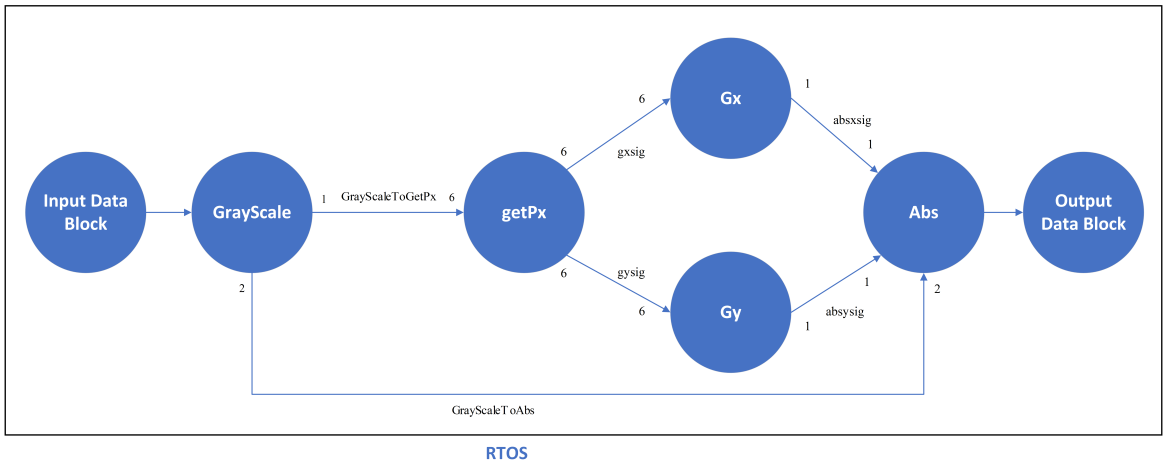


Figure 3.6: Experiment one partitioned on RTOS platform.

Figure 3.7 shows how example one is partitioned on the multiprocessor bare-metal. The *gxsig*, *gysig*, and *GrayScaleToAbs* are the channels between CPU0 and CPU1. The *Input Data Block* is on CPU0, and *Output Data Block* is on CPU1.

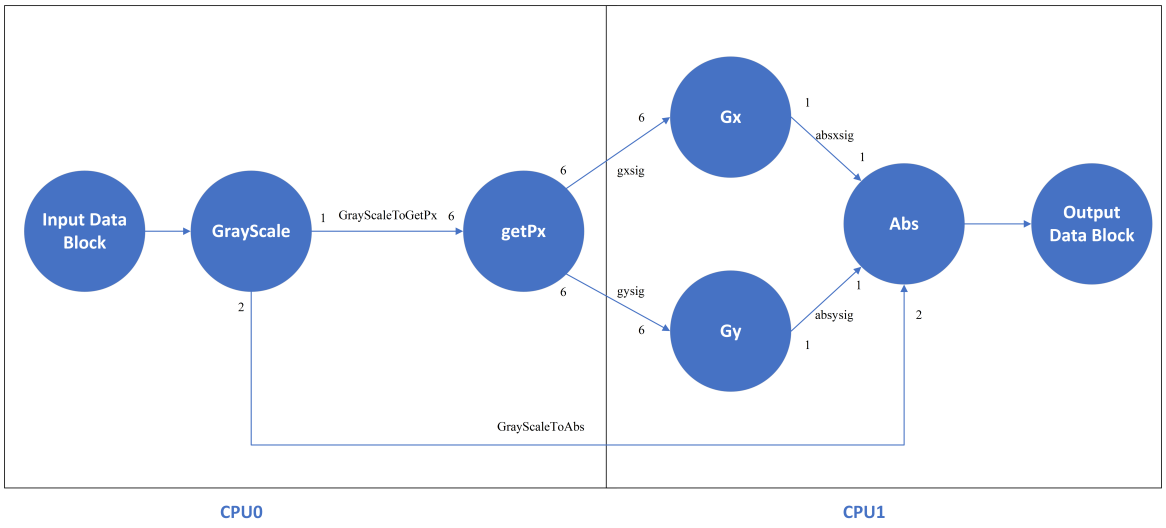


Figure 3.7: Experiment one partitioned on bare-metal with multiprocessors.

3.3.2 Experiment Two

The second experiment is shown in Figure 3.8. The input data stream is put into FIFO s_{in}, the output data stream is in FIFO s_{out}. The $2D$ on edge s_6 represents that there are two delays on edge s_6.

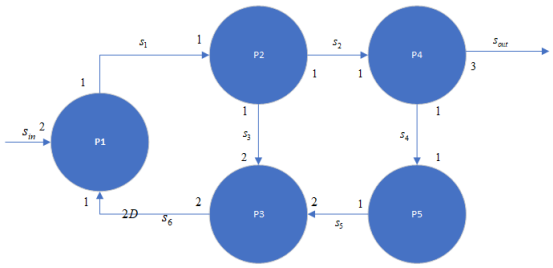


Figure 3.8: SDF graph of Experiment two

Figure 3.9 shows how experiment two is partitioned on the uniprocessor bare-metal. All the SDF actors and SDF channels are on the same CPU.

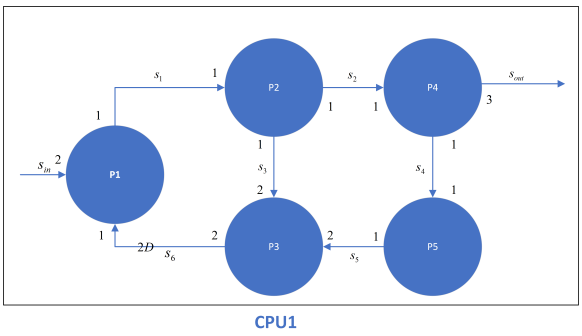


Figure 3.9: Experiment two partitioned on the uniprocessor bare metal.

Figure 3.10 shows how experiment two is partitioned on the RTOS platform.

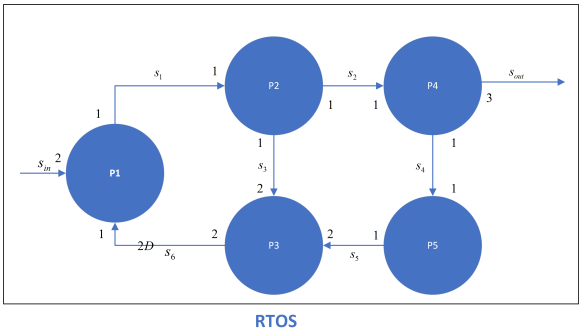


Figure 3.10: Experiment two partitioned on RTOS platform.

Figure 3.11 shows how experiment two is partitioned on the multiprocessor bare-metal. The s_5 and s_2 are the channels between CPU0 and CPU1. Other channels are on the one CPU.

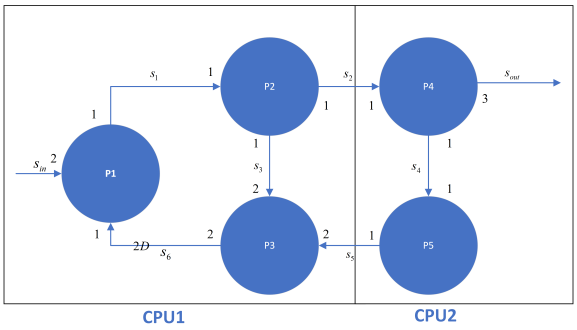


Figure 3.11: Experiment two partitioned on multiprocessor bare metal.

3.3.3 Hardware/Software To Be Used

Table 3.1 shows the software and hardware equipment used in this project.

Table 3.1: Software and hardware equipment

Language	Java 11, Xtend, C
Boards	STM32 Nucleo401RE board, CompSoc board
RTOS	FreeRTOS

The CompSoc board (introduced in Section 2), Nucleo401RE board (introduced in Section 2), and FreeRTOS are chosen as the testing platform.

3.4 Data Collection

The data to be collected is the output data streams, generated code size and execution time. It has nothing to do with the social and ethical concerns.

1. The output data streams are printed on console, and then collected.
2. The code size is the size of the compiled binary file, instead of the elf file.
3. For the execution time, only the execution time of C code, on uniprocessor bare-metal in experiment one, is collected. The C code runs 100000 iterations on Linux operating system, and then the execution time is collected and analyzed by the GCC profiling tool.

Chapter 4

Discussion of Data Flow in ForSyDe-IO

To better understand the ForSyDe-IO, the data flow is discussed in this Chapter. An accurate understanding of the data flow in ForSyDe-IO makes the software synthesis of ForSyDe-IO easier.

4.1 Discussion of Data Flow in ForSyDe-IO

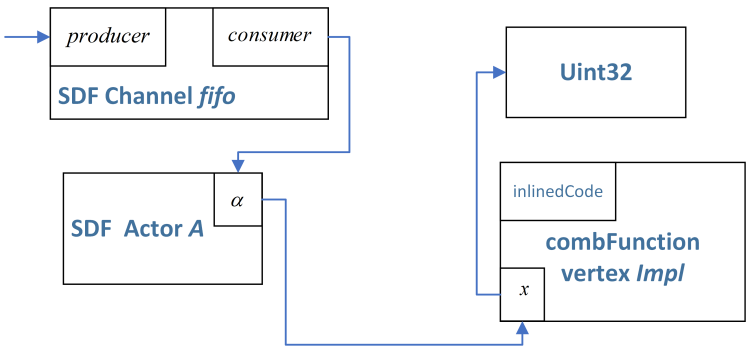


Figure 4.1: Input data flow example in ForSyDe-IO

Figure 4.1 shows an example of input data flow in ForSyDe-IO. The vertex named *fifo* is the SDF Channel, the vertex named *A* is the SDF Actor, the vertex named *Impl* is the *combFunction vertex* containing function body and the *Uint32* vertex is the data type vertex. In Figure 4.1:

1. the connection from *fifo* vertex's $consumer$ port to *actor* vertex's α port represents that the actor reads data from the input channel *fifo* via port α.

2. The data read by α port is then passed to the *x* port in *Impl* vertex, which means the data read from input channel is actually stored in a variable named *x*.

3. The connection between port *x* to vertex *Uint32* means that the *x* variable's data type is *Uint32*.

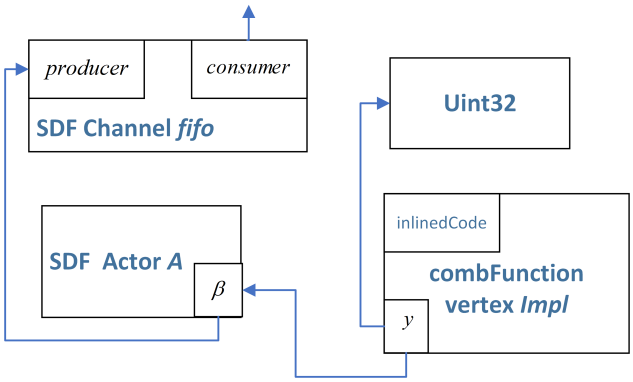


Figure 4.2: Output data flow example in ForSyDe-IO

Figure 4.2 shows an example of output data flow in ForSyDe-IO. The vertexes' names in Figure 4.2 are the same as vertexes' names in Figure 4.1. In Figure 4.2, the data stored in y variable, via the β port in *actor* vertex, is passed to the *fifo* vertex *producer* port. This path represents that the y variable's value is written to the *fifo* channel.

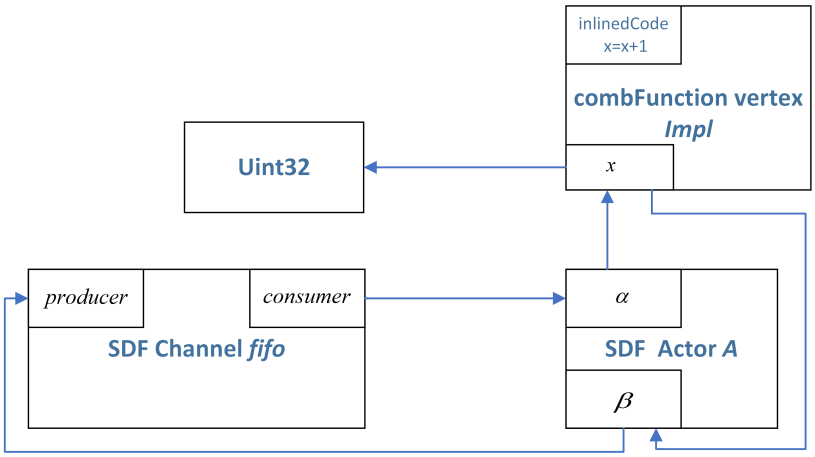


Figure 4.3: Self-looped data flow example in ForSyDe-IO

Figure 4.3 shows an example of self-looped data flow in ForSyDe-IO. The data is read from the SDF Channel *fifo* and stored in the memory variable *x*. After `x=x+1`, the data in memory variable *x* is then written back to the SDF Channel *fifo*.

Besides transferring data with SDF Channels, SDF Actors can also directly read data from source data blocks and write data directly to sink data blocks. In ForSyDe-IO, data blocks are regarded as the external global variables.

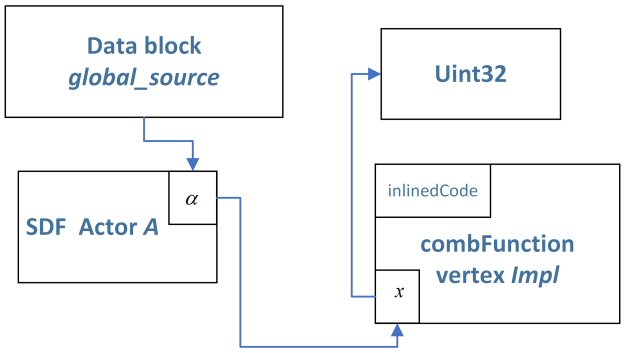


Figure 4.4: Connection between SDF Actor and global source in ForSyDe-IO

In Figure 4.4, the data block vertex *global_source* is connected to *actor* vertex α port, which is connected to the *Impl* vertex *x* port. This means that the actor function reads data from *global_source*, and the memory variable *x* is initialized with *global_source*.

In Figure 4.5, data block vertex *global_sink* is connected to the *actor* vertex β port, which connects the *Impl* vertex *y* port. This means that the actor

function writes data to *global_sink*, and the memory variable y is initialized with *global_sink*.

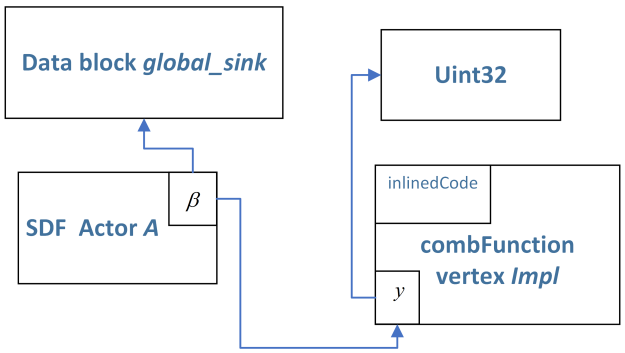


Figure 4.5: Connection between SDF Actor and global sink in ForSyDe-IO

4.2 Conclusion

In conclusion, SDF Actor's memory variables are the *combFunction vertexes*' ports which have a connection with the ports of this SDF Actor. The SDF Actor's memory variables can either be allocated on the stack or be the static global variable. The data block vertexes are global variables and are used to initialize the SDF Actor's memory variables.

For SDF actor A, if there exists a 2-length path p such that

$$\begin{aligned} p = (&\text{SDF Channel Vertex } fifo \text{ port } consumer, \\ &\text{SDF Actor vertex } A \text{ port } \alpha, \\ &\text{combFunction vertex } impl \text{ port } x \\ &) \end{aligned} \tag{4.1}$$

then, SDF actor A reads data from channel $fifo$ and stores data in memory variable x.

For SDF actor A, if there exists 2-length path p such that

$$p = (\text{Data Block Vertex } input\_data\_block, \text{SDF Actor Vertex } A \text{ Port } \alpha, \text{combFunction vertex } impl \text{ port } x) \tag{4.2}$$

then, in SDF actor A, memory variable x is initialized with $input\_data\_block$.

For SDF actor A, if there exists a 2-length path p such that

$$p = (\text{ combFunction vertex } impl \text{ port } y, \text{SDF Actor vertex } A \text{ port } \beta, \text{SDF Channel Vertex } fifo \text{ port } producer) \tag{4.3}$$

then, in SDF actor A, data stored in memory variable y is written into channel $fifo$

For SDF actor A, if there are two paths p_1 and p_2 such that

$$p_1 = (\text{SDF Actor Vertex } A \text{ Port } \beta, \text{Data Block Vertex } output\_data\_block) \tag{4.4}$$

$$p_2 = (\text{SDF Actor Vertex } A \text{ Port } \beta, \text{combFunction vertex } impl \text{ port } y) \tag{4.5}$$

then, in SDF actor A, memory variable y is initialized with $output\_data\_block$.

Chapter 5

Software Synthesis Method for ForSyDe-IO in Detail

In this Chapter, the software synthesis for ForSyDe-IO in Figure 3.1 is discussed in detail. However, because, for now, the ForSyDe-IO does not contain the mapping of subsystem and hardware, and the corresponding schedules, only the process layer, and subsystem layer are discussed in this Chapter.

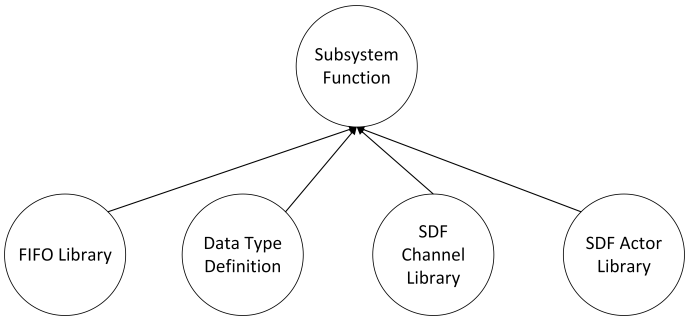


Figure 5.1: The five components in the process layer and subsystem layer.

In Figure 5.1, software synthesis in the process layer and subsystem layer contains five components: (i) Data Type Definition (ii) FIFO library (iii) SDF Channel Library (iv) SDF Actor library (v) Subsystem Function.

5.1 FIFO Library Synthesis

The FIFO Library is shared by all generic processing platforms (processors), and is used to construct the SDF Channel Libraries. Regardless of hardware

platform's type, FIFOs can be divided into two categories:

- **FIFO on One CPU**: Source actor and sink actor are on the same CPU.
- **FIFO across CPUs**: Source actor and sink actor are not on the same CPU.

Table 5.1 shows the FIFOs' types on different hardware platforms. For the uniprocessor bare-metal platform, all the FIFOs belong to **FIFO on One CPU**. For the multiprocessor bare-metal platform, FIFOs belong to either **FIFO on One CPU** or **FIFO across CPUs**. For the RTOS platform, there is no need to consider the FIFO type, since RTOS provides the necessary library, such as message queue, and message box.

Table 5.1: Platform type and its FIFO types.

Platform Type	**FIFO Type**
baremetal with single processor	FIFOs on One CPU
baremetal with multiple processors	FIFOs on One CPU , FIFOs across CPUs
RTOS	——

In this project, the FIFO libraries are designed as circular FIFOs.

5.1.1 FIFO Library for FIFOs on One CPU

In this section, for **FIFOs on one CPU**, two FIFO libraries named **FIFO Library One** and **FIFO Library Two** are proposed.

FIFO Library One and FIFO Library Two do not use the lock or semaphore. This is because for **FIFOs on One CPU**, once the static schedule is determined, there is no concurrency or race condition.

5.1.1.1 FIFO Library One

FIFO Library One is based on the token's data type inside the FIFO, which means, for each data type in the FIFOs, a FIFO library for that data type is created. For example, if in the ForSyDe-IO, FIFOs contain `int` and `double` data types, then, the FIFO library int and the FIFO library double are created. This is a very straightforward method. The xtend templates are stored in `template.fifo.fifo1.FIFOInc1` and `template.fifo.fifo1.FIFOSrc1`.

In Listing 5.1, the `circular_fifo_<DataType>` defines a circular FIFO control block. For each FIFO, the "<DataType>" is replaced by the actual data type of tokens in that FIFO.

The circular FIFO control block contains:

1. `buffer`: a pointer which points to the data buffer.
2. `front`: the position of the first element in FIFO.
3. `rear`: the last element's next position in FIFO.
4. `size`: the maximum number of tokens in the buffer.
5. `count`: how many tokens are inside the FIFO now.

The functionalities of functions are as follows:

- `init_channel_<DataType>(...)`: initialize the fifo.
- `read_fifo_<DataType>(...)`: read `number` tokens from channel to dst.
- `write_fifo_<DataType>(...)`: write `number` tokens from src to channel.

A circular FIFO which maximumly contains n tokens needs a token buffer with size $n + 1$. If the FIFO is empty, `front==rear` is `true`; If the FIFO is full, `front==(rear+1)%size` is `true`. When a token is read, `front=(front+1)%size`. When a token is written into FIFO, `rear=(rear+1)%size`.

Listing 5.1: Pseudocode of FIFO Library One

```
typedef struct
{
    <DataType>* buffer;
    int front;
    int rear;
    int size;
    int count;
}circular_fifo_<DataType>;

void init_fifo_<DataType>(circular_fifo_<DataType>
  ↪ *channel ,<DataType>* buffer, int capacity);
```

```
/* read number tokens from channel to dst */
void read_fifo_<DataType>(circular_fifo_<DataType>*
    ↪  channel,<DataType>* dst, int number);

/* write number tokens from src to channel */
void write_fifo_<DataType>(circular_fifo_<DataType
    ↪ >* channel,<DataType>* src, int number);
    ↪
```

The detailed implementation is in Listing A.1.

5.1.1.2 FIFO Library Two

FIFO Library Two is built based on the `memcpy()` function. For all the data types, only one FIFO library is created. The corresponding templates are stored in `template.fifo.fifo2.FIFOInc2` and `template.fifo.fifo1.FIFOSrc2`.

Listing 5.2 shows the **FIFO Library Two**.

Listing 5.2: FIFO Library Two pseudocode

```
typedef struct{
        void* buffer;
        int front;
        int rear;
        int capacity; // the max number of token
        int token_size; // size of one token
        int count;
}circular_fifo;

void init_fifo(circular_fifo* fifo_ptr, void* buf,
    ↪ int token_number, int token_size);

void read_fifo(circular_fifo* channel, void* dst,
    ↪ int number);

void write_fifo(circular_fifo* channel,void* src,
    ↪ int number);
```

In the `circular_fifo` structure:

- `buffer`: a pointer which points to the data buffer.

- `front`: the position of the first element in FIFO.
- `rear`: the last element's next position in FIFO.
- `capacity`: the maximum number of tokens in the buffer.
- `token_size`: the size of one token in byte.
- `count`: how many tokens are inside the FIFO now.

When initializing this fifo, the size of the token must be passed into the function.

- `read_fifo(...)`: copy `number` tokens from the FIFO to memory address `dst`.
- `write_fifo(...)`: copy `number` tokens from memory address `src` into the FIFO.

`read_fifo(...)` and `write_fifo(...)` are realized by the `memcpy` function. The detailed implementation is in Listing A.2.

5.1.2 FIFO Library for FIFOs Across CPUs

In this section, a spinlock is firstly designed. Then, a FIFO library for **FIFOs across CPUs** is proposed.

5.1.2.1 Spinlock

For FIFOs across CPUs, the lock must be used, in order to maintain the correctness of data. The lock relies on the atomic operations provided by the processor, such as test-and-set, and compare-and-swap.

In Listing 5.3:

1. function `spinlock_get`: gets the lock. If the lock is not acquired, the processor would be blocked and "spin".
2. function `spinlock_release`: releases the lock.

Listing 5.3: Spinlock Prototype

```
typedef struct{
    volatile    int flag;
}spinlock;
```

```
void spinlock_get(spinlock* lock);
void spinlock_release(spinlock* lock);
```

Listing 5.4 shows the definition of spinlock.

Listing 5.4: Spinlock Prototype

```
void spinlock_get(spinlock* lock){
        while(ATOMIC_TEST_AND_SET(&lock->flag,1)
          ↪ ==1){

        }
}
void spinlock_release(spinlock* lock){
        ATOMIC_TEST_AND_SET(&lock->flag,0);
}
```

5.1.2.2 FIFO Library for FIFOs across CPUs

Listing 5.5 shows the definition of a FIFO library for **FIFOs across CPUs**. Compared with the FIFO library for **FIFOs on One CPU** in Listing 5.1, there is a `lock` inside the `circular_fifo_<datatype>_across_cpu`.

Listing 5.5: FIFO Library pseudocode for FIFOs across CPUs

```
typedef struct{
    <datatype>* buffer;
    int front;
    int rear;
    int capacity;
    spinlock lock;
}circular_fifo_<datatype>_across_cpu;

void init_fifo_<DataType>_across_cpu(circular_fifo_
  ↪ <DataType> *channel,<DataType>* buffer, int
  ↪ capacity);

/* blocking read number tokens from channel to dst
  ↪ */
void read_fifo_<DataType>_across_cpu(circular_fifo_
  ↪ <DataType>* channel,<DataType>* dst, int
  ↪ number);
```

```
/* blocking write number tokens from src to channel
   ↪ */
void write_fifo_<DataType>_across_cpu(
   ↪ circular_fifo_<DataType>* channel,<DataType>*
   ↪ src, int number);

/* return the capacity of the channel */
int get_capacity(circular_fifo_<DataType>* channel)
   ↪ ;
```

However, in this project, because the multiprocessor platform CompSoC provides the *CHeap* library for **FIFO across CPUs**, the FIFO library in Listing 5.5 is not implemented. The *CHeap* library is directly used in the software synthesis.

5.2 SDF Channel Library Synthesis

The SDF Channel libraries are constructed on FIFO Library defined in the Section 5.1. A SDF channel in ForSyDe-IO is synthesized into a FIFO in C code with blocking read/write and finite capacity. If there is no enough data in the FIFO, the sink actor should be blocked. For each generic processing platform, an SDF Channel library should be constructed.

5.2.1 SDF Channel Library on Uniprocessor Bare-metal

Because the uniprocessor bare-metal only contains one processor, only one SDF Channel library is constructed. Listing 5.6 shows the SDF Channel library based on **FIFO library One** in Listing 5.1. Listing 5.7 shows the SDF Channel library based on **FIFO library Two** in Listing 5.2.

Listing 5.6: SDF Channel Library constructed on FIFO Library One on uniprocessor bare-metal

```
circular_fifo_<DataType> fifo_<SDFChannelName>;

volatile <DataType> buffer_<SDFChannelName>[<
   ↪ maxNumOfToken>+1];
```

```
int channel_<SDFChannelName>_size = <maxNumOfToken
    ↪ >;

/*Because of circular fifo, the buffer_size=
    ↪ channel_size+1 */
int buffer_<SDFChannelName>_size = <maxNumOfToken
    ↪ >+1;
```

Listing 5.7: SDF Channel Library constructed on FIFO Library Two on uniprocessor bare-metal

```
circular_fifo fifo_<SDFChannelName>;

volatile <DataType> buffer_<SDFChannelName>[<
    ↪ maxNumOfToken>+1];

int channel_<SDFChannelName>_size=<maxNumOfToken>;

/*Because of circular fifo, the buffer_size=
    ↪ channel_size+1 */
int buffer_<SDFChannelName>_size = <maxNumOfToken>
    ↪ + 1;
```

5.2.2 SDF Channel Libraries on Multiprocessor Bare-metal

Multiprocessor bare-metal platform contains both SDF Channels on one CPU and SDF Channels across CPUs:

- For the **SDF Channels on one CPU**, the SDF Channel libraries (Listing 5.6 and Listing 5.7) on uniprocessor bare-metal are reused.
- For the **SDF Channels across CPUs**, Listing 5.8 shows the SDF Channel library constructed on *CHeap* Library.

Listing 5.8: SDF Channel library constructed on *CHeap* Library on multiprocessor bare-metal platform

```
/* FIFOs Between Two Processors */
 volatile cheap const fifo_admin_<channelName>;
```

```
volatile <DataType> * const fifo_data_<channelName
    ↪ >;

unsigned int buffer_<channelName>_size = <Number
    ↪ of Tokens>;

unsigned int token_<channelName>_size = sizeof(
    ↪ Tokens);
```

5.2.3 SDF Channel Library on RTOS Platform

In RTOS, the SDF channel is synthesized to the message queue. Listing 5.9 shows the pseudocode.

Listing 5.9: SDF Channel library in FreeRTOS

```
/*
==========================================
    SDFChannel ChannelName Message Queue
==========================================
*/
QueueHandle_t msg_queue_ChannelName;
size_t queue_length_ChannelName = NumberOfTokensInFIFO;
long item_size_ChannelName = sizeof(DataType);
```

5.3 SDF Actor Library Synthesis

In this section, the SDF Actor Libraries on different kinds of platforms are proposed.

5.3.1 SDF Actor Library on Uniprocessor Bare Metal

On the uniprocessor bare metal , each SDF actor is synthesized into a C function. However, compared to the software synthesis method in paper [18], this C function here does not contain any function parameters, since each actor knows its own SDF channels when the ForSyDe-IO is finally decided.

Listing 5.10: SDF Actor library pseudocode on uniprocessor bare-metal platform

```
/*
```

```
/*
=============================================
    External Channal Variables
=============================================
*/
extern circular_fifo_DataType fifo_channelName;
    or
extern circular_fifo fifo_channelName;
...
/*
=============================================
        Actor Function
=============================================
*/
void ActorName(){
    /* declare stack memory inside actor */

    /* read from input channels */

    /* code blocks from combFunctions vertexes */

    /* write to the output channels */

}
```

5.3.2 SDF Actor Library on Multiprocessor Bare Metal

On the multiprocessor platform, the SDF Actor library is similar to the SDF Actor library in Listing 5.10. However, some necessary modifications are made in Listing 5.11.

- **Modification 1**: When reading data from FIFOs connecting two processors, `cheap_claim_tokens` and `cheap_release_spaces` should be used.
- **Modification 2**: when writing data to FIFOs which connects two processors, the API `cheap_claim_spaces` and `cheap_release_tokens` should be used.

Listing 5.11: SDF Actor library on the multiprocessor platform

```
/*
==========================================
    External Channal Variables
==========================================
*/
extern circular_fifo_DataType fifo_channelName;
    or
extern circular_fifo fifo_channelName;

extern cheap...
/*
==========================================
        Actor Function
==========================================
*/
void ActorName(){
    /* declare stack memory inside actor */

    /* read from input FIFOs on one CPU */

    /* read from input FIFOs across CPUs */

    /* code blocks from combFunctions vertexes */

    /* write to the output FIFOs on one CPU */

     /* write to the output FIFOs across CPUs */

}
```

5.3.3 SDF Actor Library on RTOS Platform

In RTOS, each SDF Actor is synthesized to a task. Listing 5.12 shows the pseudocode for the actor task in FreeRTOS.

For each SDF actor:

- since the schedule is periodic, a soft timer `timer_ActorName` and a `timer_sem_ActorName` are defined. The period is Worst Case Execution Time (WCET).

- In the soft timer callback function, the semaphore `timer_sem_ActorName` is posted.
- If the message queue is empty(full), the actor's read (write) is blocked forever.

Listing 5.12: SDF Actor library pseudocode in FreeRTOS

```
/*
==========================================================
    Define Task Stack
==========================================================
*/
StackType_t task_ActorName_stk[ActorName_STACKSIZE];
StaticTask_t tcb_ActorName;
/*
==========================================================
    Define Soft Timer and Soft Timer Semaphore
==========================================================
*/
TimerHandle_t timer_ActorName;
SemaphoreHandle_t timer_sem_ActorName;
...
/*
==========================================================
    External Message Queue Handler
==========================================================
*/
extern QueueHandle_t msg_queue_ChannelName;
...
/*
==========================================================
    Actor Task Definition
==========================================================
*/
void actor(void* pdata){
    /* declare stack memory inside actor */
    while(1){
        /* read from input channels */
```

```
        /* code blocks from combFunctions vertexes */

        /* write to the output channels */

        /* pend soft timer semaphore */
    }
}
/*
=============================================
    Soft Timer Callback Function
=============================================
*/
void timer_ActorName_callback(TimerHandle_t xTimer){
        xSemaphoreGive(timer_sem_ActorName);
}
```

5.4 Subsystem Software Synthesis

Subsystem corresponds to the subsystem layer in the software synthesis steps for ForSyDe-IO in Section 3. Each input ForSyDe-IO can be regarded as a subsystem, which is called either as a whole system or forming a larger system with other subsystems. In this section, the subsystem is synthesized according to the platform type.

5.4.1 Subsystem Software Synthesis on Uniprocessor Bare Metal

A subsystem is composed of processes and is synthesized into a `subsystem function` and a *subsystem initial function*. The aim of *subsystem initial function* is to initialize all the FIFOs and write the SDF Delays into the FIFOs. The aim of `subsystem function` is to call the *actor functions* according to the schedule. Listing 5.13 shows the corresponding pseudocode.

Listing 5.13: Software synthesis for subsystem layer on uniprocessor bare metal

```
void subsystem(){
    while(1){
    /* Call Actors */
```

```
    }
}

void init_subsystem(){
    /* Extern variables */

    /* initialize the channels */

    /* SDF Delay */
}
```

5.4.2 Subsystem Software Synthesis on Multiprocessor Bare Metal

On multiprocessor bare metal, for each processor(tile), subsystem is synthesized into the subsystem_tileName.c. Each subsystem_tileName.c contains the function `subsystem_tileName()` which calls actors according to the schedule on this tile, and `init_tileName()` function which initializes all the FIFOs whose source actor is on this processor (**FIFOs on One CPU** and **FIFOs across CPUs**). The `init_tileName()` should be invoked before the function `subsystem_tileName()`. Listing 5.14 shows the corresponding pseudocode.

Listing 5.14: Software synthesis for subsystem layer on multiprocessor bare-metal.

```
int init_tileName(){
    /* extern variables */

    /* initialize FIFOs (FIFO on one CPU) whose src
       ↪ actor and snk actor are on this
       ↪ processor */

    /* initialize FIFOs (FIFO across CPUs) whose
       ↪ src is on this processor and snk is on
       ↪ different processor */

    /* waiting for the completion of initialization
       ↪ of the FIFOs whose source actor is on
```

```
        ↪ another processor, while the sink actor
        ↪ is on this processor */


}

void subsystem_tileName(){
        /* call actors according to the schedule*/
}
```

5.4.3 Subsystem Software Synthesis on RTOS Platform

In Listing 5.15, the function `subsystem` is a function which creates all the actor tasks, message queues, soft timers, and semaphores. The message queues are initialized according to the Delay elements in Synchronous Data Flow Graph (SDFG). When created, the actor tasks are ready and the executions of actors are determined by the RTOS scheduler.

Listing 5.15: Software synthesis for subsystem layer on the RTOS platform

```
void subsystem(){
        /* Initialize Message Queue      */

        /* Initialize Software Timer     */

        /* Initialize Timer's Semaphore */

        /* Initialize Actor Tasks        */

        /* Start Software Timer          */
}
```

5.5 Data Type Definition

The aim of Data Type Definition is to define all the data types, and to declare all the global variables in the ForSyDe-IO. The generated data types' definitions and global variables' declarations are shared by all generic processing platforms.

Listing 5.16 is the header file, and Listing 5.17 is the source file.

Listing 5.16: Data type definition, header file

```
typedef actual_datatype
    ↪ datatype_name_in_forsyde_io;
...
```

Listing 5.17: Data type definition, source file

```
datatype_name_in_forsyde_io
    ↪ value_name_in_forsyde_io;
...
```

Chapter 6

Implementation of Code Generator

In this chapter, first of all, the structure, and the design pattern of the code generator are proposed. Then the designed structure of generated C code is shown.

6.1 Structure of Code Generator

The code generator is written in both Xtend and Java. Figure 6.2 shows the structures of classes that build up the code generator. These classes are described in detail later. However, please note that not all the classes are drawn.

In Figure 6.2, there are three important class modules: **Generator Module**, **Processing Module**, and **Template Module**.

The **Template Module** is the bottom layer of the whole code generator and is responsible for containing the C templates for each SDF element. On the top of the Template Module is the **Processing Module**.

Processing Module contains the processing classes. The aims of **Processing Module** are

- extracting SDF elements from ForSyDe-IO .
- invoking template classes' `create()` method, which returns the generated C code string .
- saving the generated C code string according to the relative path returned by template classes' `savePath()` method.

Generator Module is the top layer. Its aims are as follows:

- contain the saving root information, FIFO library type, platform type information and ForSyDe-IO model.
- contain and invoke all the classes in processing module.

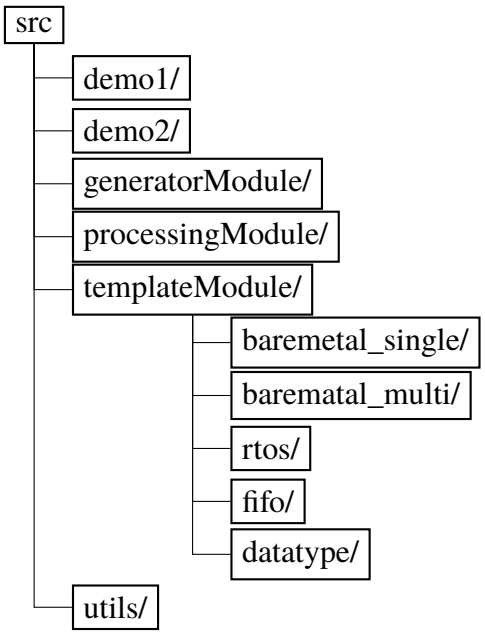


Figure 6.1: Package structure

Figure 6.1 shows the code generator' structure from an aspect of package.

- The package `demo1` contains the demos for experiment one.
- The package `demo2` contains the demos for experiment two.
- The package `generatorModule` contains classes in **Generator Module** layer.
- The package `processingModule` contains classes in **Precessing Module** layer.
- The package `templateModule` contains template classes in **Template Module** layer.
- The package `utils` contains some functions extracting information from ForSyDe-IO , and the save function.

6.1.1 Design Pattern

The aim of the code generator is to automatically generate the executable C code based on the software synthesis strategy in the previous chapter. The derived code generator is supposed to be open to code extension—users should be able to easily extend the templates. Therefore, the strategy design pattern is exploited. Figure 6.2 shows the strategy pattern. Some classes are not drawn in Figure 6.2.

In Figure 6.2, each processing module class implements the interface `ProcessingModule`. Each class of template module implements the corresponding template interface according to the SDF element type. Users can easily replace the class.

6.1.2 Classes in Template Module

The template module contains five smaller packages shown in Figure 6.3.

- The package `template.baremetal_single` contains SDF actor templates, SDF channel templates, and subsystem templates for baremetal uniprocessor platform.
- The package `template.baremetal_multi` contains SDF actor templates, SDF channel templates, and subsystem templates for baremetal multiprocessor platform.
- The package `template.rtos` contains SDF actor templates, SDF channel templates, and subsystem templates for RTOS platform.
- The package `template.fifo` contains the templates FIFO Library One and FIFO Library Two proposed in the Chapter 5. The template classes for FIFO Library One are FIFOInc1.xtend and FIFOSrc1.xtend. The template classes for FIFO Library Two are FIFOInc2.xtend and FIFOSrc2.xtend.
- The package `template.datatype` contains the templates for data type definition.

Figure 6.2: UML diagram of code generator

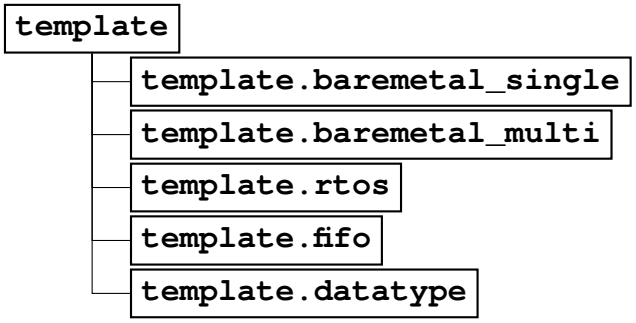


Figure 6.3: Package structure in **Template Module Layer**

In the **template module layer**, the template classes are divided into four categories based on their functionalities. Each category has its corresponding interface. Classes in that category is supposed to implement that interface. Table 6.1 shows the category and the interface.

Table 6.1: Template class categories and interfaces

Category	**Corresponding Interface**	**Example Classes**
SDFActor	interface `ActorTemplate`	SDFActorSrc
SDFChannel	interface `ChannelTemplate`	SDFChannelInc, SDFChannelSrc
Subsystem	interface `SubsystemTemplate`	SybsystemSrc
Initialization	interface `InitTemplate`	FIFOInc, DataTypeSrc

The template classes, whose name ends with ”Src”, are source file templates. The template classes, whose name ends with ”Inc”, are header file templates.

6.1.3 Classes in Processing module

In the processing module, there are four classes:

- `SDFChannelProcessing`
- `SDFActorProcessing`
- `SubsystemProcessing`

- `InitProcessing`

All these four classes implement the interface `ProcessingModule`. Table 6.2 shows the classes in the processing module and their corresponding functionalities.

Table 6.2: The classes in **Processing Module Layer** and their functionalities.

Class in Processing Module	**Functionality**
Class **`SDFChannelProcessing`**	Generate and save .c file and .h file for each sdf channel vertex.
Class **`SDFActorProcessing`**	Generate and save .c file and .h file for each sdf actor vertex.
Class **`SubsystemProcessing`**	generate and save subsystem.c and subsystem.h for the subsystem.
Class **`InitProcessing`**	generator and save other necessary files such as circular_fifo_lib.c, datatype_definition.h,*etc.,*

6.1.4 Classes in Generator Module

Generator Module only contains one class: `Generator` class. In class `Generator`, there are two important static variables: 1) `Generator.fifoType` 2) `Generator.platform`.

Table 6.3 shows the possible value and corresponding meaning of `Generator.fifoType`. Table 6.4 shows the possible value and corresponding meaning of `Generator.platform`.

Table 6.3: Possible value of `Generator.fifoType` and the corresponding meaning.

Value	**Meaning**
`Generator.fifoType=1`	When generating C code, use FIFO Library One
`Generator.fifoType=2`	When generating C code, use FIFO Library Two

Table 6.4: Possible value and corresponding representation of `Generator.platform`.

Possible Value	**Meaning**
`Generator.platform=1`	The target platform belongs to bare metal with a single processor
`Generator.platform=2`	The target platform belongs to bare metal with multiple processors
`Generator.platform=3`	The target platform belongs to RTOS platform

The `generator` class contains a list of `ProcessModule` classes. Each processing module class has a list of template module classes. When `generator.create()` is called. Each `ProcessModule` class in that list calls its method `create()`. The `ProcessModule.create()` further calls the template classes' `create()`, which returns the generated C code string. The process module class saves the c code string.

6.2 Structure of Generated C Code

On the uniprocessor bare metal, the C code directory structure is designed as Figure 6.4.

- The directory circular_fifo_lib contains the circular FIFO library.
- The directory datatype contains the `datatype_definition.c` and `datatype_definition.h`, which are responsible for defining customized data types and declaring the global variables, such as `typedef unsigned short UInt16` and `UInt16 ZeroValue=0;`.
- The tile directory contains the `subsystem.c`, `subsystem.h`, SDF actor library (in sdfactor directory) and SDF channel library (in sdfchannel directory).

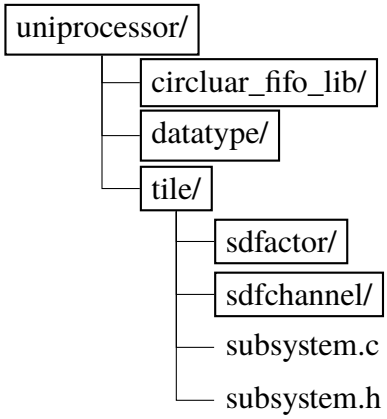


Figure 6.4: Structure of generated C code on uniprocessor bare metal.

Figure 6.5 shows the generated C code structure on multiproccessor bare metal.

- The directory circular_fifo_lib, and directory datatype respectively contain files for the circular FIFO library, and data type definition and declaring global variables.
- For directory tile$i$, it contains the `subsystem.c`, `subsystem.h`, sdf channel library and sdf actor library on this tile i. For example, if the channel $sig1$, $sig2$, and actor $actor1$ are on tile 1, then their files are in directory tile1.

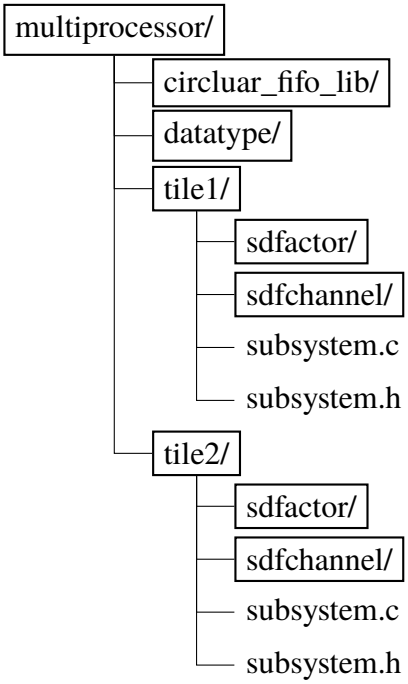


Figure 6.5: Structure of generated C code on baremetal multiprocessor platform.

For the RTOS platform, the c code structure is shown is Figure 6.6.

- Directory sdfactor contains SDF Actor library.
- Directory sdfchannel contains SDF Channel library.
- Directory timer contains the soft timers and their semaphores.
- File configRTOS.h contains the stack size for each actor's task.
- The start_task.h and start_task.c contain a function which creates all the actor tasks, message queues, timers, and semaphores.

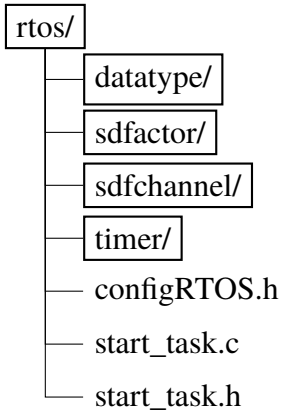


Figure 6.6: Structure of generated C code on RTOS platform.

Chapter 7

Results

7.1 Experiment Environment

Table 7.1 shows the experiment environment.

Table 7.1: Testing Platforms

Platform	Test Board	Processor Architecture
Uniprocessor bare-metal	CompSoC board	Risc-v32
Multiprocessor bare-metal	CompSoC board	Risc-v32
RTOS platform	FreeRTOS+ Nucleo401RE	Arm Cortex-M4

The produced C codes for uniprocessor bare-metal were tested on CompSoC board's tile0 core. The produced C code for RTOS was tested on FreeRTOS platform. The produced C codes for multiprocessor bare-metal were tested on CompSoc board's tile0 core and tile1 core.

When testing C codes for multiprocessor bare-metal, the memory address of shared memory by tile0 and tile1 was manually added into the produced C Code, since the ForSyDe-IO does not contain the memory address information. Table 7.2 and Table 7.3 show the memory address setting in experiment one and experiment two.

Table 7.2: Memory addresses of FIFOs cross two CPUs in experiment one.

CPU	FIFO Across Two CPUs	Virtuliazed Address
CPU0	gxsig	0x80020000
	gysig	0x80020100
	GrayScaleToAbs	0x80020200
CPU1	gxsig	0x80020000
	gysig	0x80020100
	GrayScaleToAbs	0x80020200

Table 7.3: Memory addresses of FIFOs cross two CPUs in experiment two.

CPU	FIFO Across Two CPUs	Virtuliazed Address
CPU0	s_2	0x80020000
	s_5	0x80020100
CPU1	s_2	0x80020000
	s_5	0x80020100

7.2 Major results

In this section, the results of experiment one and experiment two are presented.

7.2.1 Results of Experiment One

Table 7.4 shows the results of experiment one on different platforms. In experiment one, ForSyDe-IO was synthesized with both FIFO Library One (Section 5.1.1.1) and FIFO Library Two (Section 5.1.1.2). The input and output data are both 5×5 two-dimension arrays. In the output array, values of pixel $(0, 0)$, $(0, 1)$, $(0, 2)$ were computed.

Table 7.4: Results of experiment one.

Platform	**FIFO Library Used in Synthesis**	**Input**	**output**
Uniprocessor bare-metal	FIFO Library One	$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$	$\begin{pmatrix} 38 & 38 & 38 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
	FIFO Library Two	$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$	$\begin{pmatrix} 38 & 38 & 38 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
Multiproceessor bare-metal	FIFO Library One , CHeap	$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$	$\begin{pmatrix} 38 & 38 & 38 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
	FIFO Library Two , CHeap	$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$	$\begin{pmatrix} 38 & 38 & 38 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
RTOS	Message queue api proveded by RTOS	$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$	$\begin{pmatrix} 38 & 38 & 38 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$

Table 7.5 shows the code size of produced C code on different platforms. The code size is the size of the binary file, instead of the elf file. When compiling, the produced elf file was transformed into binary file by `objcopy` in GCC.

Table 7.5: The code size of generated C code in experiment one.

Platform	**FIFO Library Used in Synthesis**	**Code Size of Generated C Code**
Uniprocessor bare-metal	FIFO Library One	16KB
	FIFO Library Two	16KB
Multiprocessor bare-metal	FIFO Library One + CHeap	10KB on CPU0 9KB on CPU1 total 19KB
	FIFO Library Two + CHeap	14 KB on CPU0 14 KB on CPU1 total 28KB
RTOS	Message queue	30KB

The C codes generated for uniprocessor bare-metal, synthesized with both FIFO Library One and FIFO Library Two , were tested on the Linux OS. The C codes ran 100000 iterations on Linux, and later, were analyzed by the GCC profiling tool. Table 7.6 and Table 7.7 contains part of the profiling results.

In Table 7.6 and Table 7.7, the ”self seconds” shows the execution time of this function alone. The ”percentage” shows the percentage of total running time used by this function. The ”cumulative seconds” shows the sum of this function’s ”self seconds”, and ”self seconds” of the functions above this function.

Table 7.6: Execution time of 100000 iterations on Linux. The C codes were synthesized with FIFO Library One .

Percentage	**Cumulative Seconds**	**Self Seconds**	**Function Name**
22.27%	0.02	0.02	read_fifo_UInt16
22.27%	0.04	0.02	write_fifo_DoubleType
22.27%	0.06	0.02	test
11.14%	0.07	0.01	write_fifo_UInt16
11.14%	0.08	0.01	read_fifo_DoubleType
11.14%	0.09	0.01	actor_GrayScale
0	0.09	0	actor_Abs
0	0.09	0	actor_Gx
0	0.09	0	actor_Gy
0	0.09	0	actor_getPx
0	0.09	0	init_fifo_DoubleType
0	0.09	0	init_fifo_UInt16
0	0.09	0	init_subsystem
0	0.09	0	subsystem

Table 7.7: Execution time of 100000 iterations on Linux. The C codes were synthesized with FIFO Library Two .

Percentage	Cumulative Seconds	Self Seconds	Function Name
80.19%	0.04	0.04	write_fifo
20.05%	0.05	0.01	read_fifo
0	0.05	0	actor_Abs
0	0.05	0	actor_GrayScale
0	0.05	0	actor_Abs
0	0.05	0	actor_Gx
0	0.05	0	actor_Gy
0	0.05	0	actor_getPx
0	0.05	0	test
0	0.05	0	init_fifo
0	0.05	0	init_subsystem
0	0.05	0	subsystem

7.2.2 Results of Experiment Two

Table 7.8 shows the results of experiment two.

1. On uniprocessor bare-metal, SDF Channels in ForSyDe-IO were synthesized with both FIFO Library One and FIFO Library Two .

2. On multiprocessor bare-metal, **FIFOs on one CPU**, were synthesized with both FIFO Library One and FIFO Library Two . For the FIFOs across CPUs (**FIFOs across CPUs**), they were synthesized with the *Cheap* library provided by CompSoC board.

3. On RTOS platform, the SDF Channels were synthesized with message queues provided by FreeRTOS.

In Table 7.8, the input data stream contains 20 data. In each period, 4 data in input data stream s_{in} were consumed by SDF, and 6 data were produced and written to the output data stream s_{out}. Thus, the Table 7.8 shows the results produced in the first five periods.

Table 7.9 shows the code size of generated C code. Same as experiment one, the code size is the size of the binary machine code.

Table 7.8: Results of experiment two.

Platform	**FIFO Library Used in Synthesis**	**Input Data Stream in s_in**	**Output Data Stream in s_out**
Uniprocessor platform	FIFO Library One	{1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16, 17,18,19,20}	{3,4,5,7,8,9 23,24,25,27,28,29, 71,72,73,75,76,77, 175,176,177,179,180,181, 391,392,393,395,396,397}
	FIFO Library Two	{1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16, 17,18,19,20}	{3,4,5,7,8,9 23,24,25,27,28,29, 71,72,73,75,76,77, 175,176,177,179,180,181, 391,392,393,395,396,397}
Multiprocessor platform	FIFO Library One, CHeap	{1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16, 17,18,19,20}	{3,4,5,7,8,9 23,24,25,27,28,29, 71,72,73,75,76,77, 175,176,177,179,180,181, 391,392,393,395,396,397}
	FIFO Library Two, CHeap	{1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16, 17,18,19,20}	{3,4,5,7,8,9 23,24,25,27,28,29, 71,72,73,75,76,77, 175,176,177,179,180,181, 391,392,393,395,396,397}
RTOS platform	Message queue provided by RTOS	{1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16, 17,18,19,20}	{3,4,5,7,8,9 23,24,25,27,28,29, 71,72,73,75,76,77, 175,176,177,179,180,181, 391,392,393,395,396,397}

Table 7.9: Code size in experiment two.

Platform	**FIFO Library Used in Synthesis**	**Code Size of Generated C Code**
Uniprocessor bare-metal	FIFO Library One	9KB
	FIFO Library Two	10KB
Multiprocessor bare-metal	FIFO Library One + CHeap	9KB on CPU0 9KB on CPU1 total 18KB
	FIFO Library Two + CHeap	9 KB on CPU0 9 KB on CPU1 total 18KB
RTOS	Message queue	22KB

Chapter 8

Discussion

In summary, the code generator (in Chapter 6), based on the software synthesis for ForSyDe-IO (in Chapter 5), successfully synthesizes the input ForSyDe-IO into C codes on uniprocessor bare metal , mutliprocessor bare metal , and RTOS platform. The generated C codes work correctly on the target platforms. The results show that the FIFO Library Two has larger code size than FIFO Library One , but has less execution time.

8.1 Discussion About Output Results

The results of experiment one are presented in Table 7.4. In Table 7.4, the output matrixes are the same as the expected outputs matrixes, which were manually calculated according to the ForSyDe-IO.

Table 7.8 shows the results of experiment two. In Table 7.8, the output data streams are the same as the output data streams simulated by ForSyDe.

Additionally, in experiment two, it was observed that after executing 5 periods, the code codes were blocked. This is correct, because, after 5 periods, the data in input data stream s_{in} were all consumed. According to the concepts of SDF, the actor p_1 was blocked.

Thus, the results in Table 7.4 and Table 7.8 show that the code generator, according to the ForSyDe-IO, successfully generated the C codes on the uniprocessor bare-metal, the multiprocessor bare-metal, and the RTOS platform, and the software synthesis method for ForSyDe-IO is correct.

However, the WCET is only supported in the software synthesis for RTOS platform. The software synthesis for the uniprocessor bare-metal and multiprocessor bare-metal does not support the WCET in ForSyDe-IO.

8.2 Discussion About Code Size

Table 7.5 and Table 7.9 respectively show the code size in experiment one and experiment two.

A finding in Table 7.5 and Table 7.9 is that, for the same ForSyDe-IO, the code size on RTOS is larger than the code size on multiprocessor bera-metal, which is larger than the code size on uniprocessor bare-metal.

The explanation for this finding is that, on RTOS platform, the generated C codes, together with RTOS codes, were compiled into one binary file. On multiprocessor bare-metal, the generated C codes, and the codes of *Cheap* library provided by CompSoC board, were compiled together. On uniprocessor bare-metal, only the generated C codes were compiled. As a result, the code size on RTOS is the largest, and the code size on uniprocessor bare-metal is the smallest.

Surprisingly, another interesting finding of code size is that, in Table 7.5 and Table 7.9, the code size synthesized with FIFO Library Two , is not less than but even larger than the code size synthesized by the FIFO Library One , on both uniprocessor bare metal and mutliprocessor bare metal . This finding is unexpected.

The expectation is that code size synthesized with FIFO Library Two , should be smaller than the code size synthesized with FIFO Library One . Because the FIFO Library One creates a FIFO library for each data type. While, the FIFO Library Two , for all data types, only creates one FIFO library. For example, for the data types `UInt32`, `UInt16`, and `DoubleType`, three FIFO libraries `circular_fifo_UInt32`, `circular_fifo_UInt16`, and `circular_fifo_DoubleType` are respectively created by the FIFO Library One template. However, by FIFO Library Two , only `circular_fifo` library is created.

The explanation about this contradiction is that, the FIFO Library Two is constructed by the `memcpy()` function, but the FIFO Library One does not call any extra functions, as a result, the code size of FIFO Library Two is actually larger than code size of FIFO Library One . Only when there are many data types, the code size synthesized with FIFO Library Two , is smaller than the code size synthesized with FIFO Library One .

8.3 Discussion About Execution Time

In Table 7.6, the execution time of c code synthesized with FIFO Library One and generated for uniprocessor bare-metal, is 0.09 seconds. The total execution time of reading, and writing of FIFOs is 0.07 seconds, accounting for approximately 65% of the total running time.

In Table 7.7, the execution time of c code synthesized with FIFO Library Two and generated for uniprocessor bare-metal is 0.05 seconds. The total execution time of reading, writing of FIFOs is 0.05 seconds, accounting for approximately 100% of total running time.

This finding indicates, the FIFO Library Two , built on `memcpy()` function, runs faster than the FIFO Library One .

For ForSyDe-IO whose SDF Channels contain many kinds of data types, the FIFO Library Two has less running time and less code size, compared with the FIFO Library One .

For ForSyDe-IO whose SDF Channels contain only one or two kinds of data types, the choosing of FIFO Library One and FIFO Library Two is a trade-off between code size and running speed.

Chapter 9

Conclusions and Future work

This chapter presents the conclusions and future work.

9.1 Conclusions

Within this thesis, the software synthesis for ForSyDe-IO has been explored. The automatic code generator, synthesizing ForSyDe-IO into C code on uniprocessor bare metal , mutliprocessor bare metal , and RTOS platform, has been developed. This thesis provides a code generation tool which can help researchers to quickly generate C code from the high-level SDF model. The whole thesis consists of the following parts:

- theoretical background study.
- software synthesis for ForSyDe-IO.
- implementation of code generator based on the software synthesis for ForSyDe-IO .
- testing code generator on two ForSyDe-IO examples.

The software synthesis for ForSyDe-IO contains the processor layer, subsystem layer, and system layer. However, only the components in processor layer and subsystem layer were synthesized, because for now, the ForSyDe-IO does not contain the mapping of subsystem and hardware platform, and does not contain the schedules of the subsystem.

In the software synthesis for ForSyDe-IO , there are five important components:

- FIFO Library

- Data Type Definition
- SDF Channel Library
- SDF Actor Library
- Subsystem function

On the uniprocessor bare metal and mutliprocessor bare metal , the FIFO library and data type definition were shared by all the generic processing platforms. Two FIFO libraries—FIFO Library One and FIFO Library Two —were created. While, the SDF Channel library, SDF Actor library, and subsystem function were created on each generic processing platform. Inside the subsystem function, the actors were called according to the generic processing platform's schedules.

On the RTOS platform, the FIFO library was not created, because the message queue was provided by the RTOS. The *start_task* function, instead of subsystem function, was created on RTOS platform. The aim of *start_task* function was to create all the soft timers, semaphores, message queues, and actor tasks. The actor tasks' priorities were set to the same value.

The results of two experiments indicate that the generated C code on different platforms worked correctly. Testing results also show that, although the size of FIFO Library Two is larger than FIFO Library One , it runs faster than FIFO Library One .

9.2 Future work

However, there are still many things to be done in the future.

- The software synthesis method and code generator produced by this thesis only work for the SDF model in the ForSyDe-IO . For other models of computation, the code generator does not work. In the future, the software synthesis and code generator should be further extended to other MoCs.
- In this thesis, the ForSyDe-IO is only synthesized into C code. In the future, more templates could be added, in order to synthesize the ForSyDe-IO into other languages.
- When synthesizing the FIFO library, in this project, only the circular FIFO data structure is implemented. The generated FIFO is the dynamic

FIFO. In the future, a static FIFO library could be implemented, in order to reduce the execution time of reading and writing of FIFOs.

- This thesis does not implement the methods to optimize the code size of generated C code and the FIFO's memory space. In the future, memory sharing technology could be used to reduce the FIFOs' memory space.

References

[1] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. doi: 10.1109/PROC.1987.13876 [Pages 1, 7, 8, 9, and 12.]

[2] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987. doi: 10.1109/TC.1987.5009446 [Pages 1, 7, 9, 11, and 12.]

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of Embedded Software from Synchronous Dataflow Specifications," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 21, no. 2, pp. 151–166, Jun. 1999. doi: 10.1023/A:1008052406396. [Online]. Available: https://doi.org/10.1023/A:1008052406396 [Page 1.]

[4] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij, "Throughput Analysis of Synchronous Data Flow Graphs," in *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, Jun. 2006. doi: 10.1109/ACSD.2006.33 pp. 25–36, iSSN: 1550-4808. [Page 1.]

[5] R. de Groote, *Throughput analysis of dataflow graphs*, 10 2018, pp. 751–786. [Page 1.]

[6] L. Wang, C. Wang, and H. Wang, "Improved scheduling algorithm for synchronous data flow graphs on a homogeneous multi-core systems," *Algorithms*, vol. 15, no. 2, 2022. doi: 10.3390/a15020056. [Online]. Available: https://www.mdpi.com/1999-4893/15/2/56 [Page 1.]

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, pp. 1305 – 1320, 10 1991. doi: 10.1109/5.97300 [Page 1.]

[8] P. Guernic, A. Benveniste, P. Bournai, and T. Gautier, "Synchronous data flow programming with the language signal," *IFAC Proceedings Volumes*, vol. 20, pp. 359–364, 07 1987. doi: 10.1016/S1474-6670(17)55987-X [Page 2.]

[9] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij, "Throughput analysis of synchronous data flow graphs." 01 2006. doi: 10.1109/ACSD.2006.33 pp. 25–36. [Page 2.]

[10] "IC Package Design Flows." [Online]. Available: https://www.cadence.com/en_US/home/tools/ic-package-design-and-analysis/ic-package-design-flows.html [Page 2.]

[11] S. A. Edwards, "CoCentric System Studio," in *Languages for Digital Embedded Systems*. Boston, MA: Springer US, 2000, pp. 259–265. ISBN 978-1-4615-4325-1. [Online]. Available: https://doi.org/10.1007/978-1-4615-4325-1_17 [Page 2.]

[12] A. D. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Design & Test*, vol. 34, no. 1, pp. 77–90, 2017. doi: 10.1109/MDAT.2016.2626445 [Page 2.]

[13] I. Sander and A. Jantsch, "System modeling and transformational design refinement in forsyde [formal system design]," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, 2004. doi: 10.1109/TCAD.2003.819898 [Pages 2, 3, and 17.]

[14] "ForSyDe." [Online]. Available: https://forsyde.github.io/ [Pages 2, 19, and 22.]

[15] "DeSyDe," Mar. 2021, original-date: 2016-10-05T08:21:53Z. [Online]. Available: https://github.com/forsyde/DeSyDe [Pages 2 and 22.]

[16] J. Dennis, *Data Flow Graphs*. Boston, MA: Springer US, 2011, pp. 512–518. ISBN 978-0-387-09766-4. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_294 [Page 8.]

[17] S. Bhattacharyya, P. Murthy, and E. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI Signal Processing Systems*, vol. 21, 10 1999. doi: 10.1023/A:1008052406396 [Page 14.]

[18] Z. Lu, I. Sander, and A. Jantsch, "A case study of hardware and software synthesis in forsyde," in *15th International Symposium on System Synthesis, 2002.*, 2002. doi: 10.1145/581199.581219 pp. 86–91. [Pages xi, 14, 15, 33, 34, and 56.]

[19] S. S. Bhattacharyya and E. A. Lee, "Looped schedules for dataflow descriptions of multirate dsp algorithms," USA, Tech. Rep., 1993. [Pages 16 and 17.]

[20] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, *Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms*. USA: Kluwer Academic Publishers, 2001, p. 452–464. ISBN 1558607021 [Pages 16 and 17.]

[21] K. Rosvall and I. Sander, "A constraint-based design space exploration framework for real-time applications on mpsocs," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014. doi: 10.7873/DATE.2014.339 pp. 1–6. [Page 22.]

[22] "Forsyde io webpage." [Online]. Available: https://forsyde.github.io/forsyde-io/ [Page 23.]

[23] "Xtend - Modernized Java." [Online]. Available: https://www.eclipse.org/xtend/ [Pages 29 and 30.]

Appendix A

FIFO Library Implementation

This Chapter contains the implementation of FIFO Library One and FIFO Library Two .

A.1 FIFO Library One

Listing A.1 shows the implementation of FIFO Library One .

Listing A.1: FIFO Library One

```
void init_channel_<DataType>(circular_fifo_<
    ↪ DataType> *channel ,<DataType>* buffer, int
    ↪ size){

    channel->buffer = buffer;
    channel->size=size;
    channel->front = 0;
    channel->rear = 0;
    channel->count=0;
}
void read_fifo_<DataType>(circular_fifo_<DataType>*
    ↪  channel,<DataType>* dst, int number){

        //if there is no enough tokens in FIFO,
            ↪ block.
    while( channel->count < number );

    for(int i=0; i<number;++i){
```

```
    dst[i] = channel->buffer[channel->front];
    channel->front= (channel->front+1)%channel->
      ↪ size;
    --(channel->count);
        }
}

void write_fifo_<DataType>(circular_fifo_<DataType
  ↪ >* channel,<DataType>* src, int number){

    for(int i=0; i<number; ++i){
    channel->buffer[channel->rear] = src[i];
    channel->rear= (channel->rear+1)%channel->size;
    ++(channel->count);
}

}
```

A.2 FIFO Library Two

Listing A.2 shows the implementation of FIFO Library Two for FIFOs on One CPU. The prototype is in Listing 5.2

Listing A.2: Implementation of FIFO Library Two for FIFOs on One CPU

```
#include <string.h>
#include <stdio.h>

void init_fifo(circular_fifo* fifo_ptr, void* buf,
  ↪ size_t capacity, size_t token_size){
        fifo_ptr->buffer=buf;

        fifo_ptr->front=0;
        fifo_ptr->rear=0;
        fifo_ptr->capacity=capacity;
        fifo_ptr->token_size=token_size;
        fifo_ptr->count=0;
}
```

```
void read_fifo(circular_fifo* channel, void* dst,
    ↪ size_t number){
        while(channel->count< number);

        char* memcpy_dst,*memcpy_src;
        for(int i=0; i<number;++i){
                memcpy_dst=(char*)dst+i*channel->
                    ↪ token_size;
                memcpy_src=((char*)channel->buffer+
                    ↪ channel->front*channel->
                    ↪ token_size);

                memcpy(memcpy_dst, memcpy_src ,
                    ↪ channel->token_size);
                channel->front= (channel->front+1)%
                    ↪ channel->capacity;
                    ↪
                --(channel->count);
        }
}
void write_fifo(circular_fifo* channel,void* src,
    ↪ size_t number){
        char* memcpy_dst,*memcpy_src;
        for(int i=0; i<number;++i){

                memcpy_dst=(channel->rear*channel->
                    ↪ token_size+ (char*)channel->
                    ↪ buffer);
                memcpy_src = (char*)src+i*channel->
                    ↪ token_size;
                memcpy(memcpy_dst, memcpy_src,
                    ↪ channel->token_size);

                channel->rear= (channel->rear+1)%
                    ↪ channel->capacity;
                ++(channel->count);
        }
}
```

TRITA-EECS-EX- 2022:00

€€€€ For DIVA €€€€

{
"Author1": { "Last name": "Zhao",
"First name": "Yihang",
"Local User Id": "u100001",
"E-mail": "yihangz@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Cycle": "2",
"Course code": "DA248X",
"Credits": "30.0",
"Degree1": {"Educational program": "Master's Programme, Embedded Systems, 120 credits"
,"programcode": "TEBSM"
,"Degree": "Masters degree"
,"subjectArea": "Technology"
},
"Title": {
"Main title": "Software Synthesis For ForSyDe-IO",
"Language": "eng" },
"Alternative title": {
"Main title": "Mjukvarusyntesen av ForSyDe-IO",
"Language": "swe"
},
"Supervisor1": { "Last name": "Jordão",
"First name": "Rodolfo",
"Local User Id": "u1iplwom",
"E-mail": "jordao@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Department of Electrical Engineering" }
},
"Examiner1": { "Last name": "Sander",
"First name": "Ingo",
"Local User Id": "u1l8osh2",
"E-mail": "ingo@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Department of Electrical Engineering" }
},
"National Subject Categories": "20207, 10206",
"Other information": {"Year": "2022", "Number of pages": "1,97"},
"Series": { "Title of series": "TRITA-EECS-EX" , "No. in series": "2022:00" },
"Opponents": { "Name": "Baiheng Chen"},
"Presentation": { "Date": "2022-06-22 11:00-12:00"
,"Language":"eng"
,"Room": "via Zoom https://kth-se.zoom.us/j/62570527175"
,"Address": "eecs_Kistagången 16, west, floor 3, room 3347, Amiga (20 seats)"
,"City": "Stockholm" },
"Number of lang instances": "2",
"Abstract[eng]": €€€€

The implementation of embedded software applications is a complex process. The complexity arises from the intense time-to-market pressures; power and memory constraints. To deal with this complexity, an idea is to automatically construct the applications based on the high-level abstraction model. Synchronous data flow (SDF) is a high-level model of computation, and is used to model the embedded applications. Formal System Design (ForSyDe), developed by ForSyDe group at KTH Royal Institute of Technology, is a methodology for modeling and designing heterogeneous systems-on-chip. The aim of Formal System Design (ForSyDe) is to automatically generate the detailed software implementation or hardware implementation according to the high-level system specification. Formal System Design (ForSyDe) starts from the high-level system specification and specifies the system model in Haskell language. Synchronous data flow is supported by ForSyDe. ForSyDe-IO is an intermediate representation of the high-level system specification. This master thesis
focuses on the software synthesis of ForSyDe-IO and synchronous data flow, and aims to produce an automatic tool that can generate software applications in C code for different platforms based on ForSyDe IO. In this project, a software synthesis method for ForSyDe-IO was proposed. Then, based on the software synthesis method, a code generator, written in Java and Xtend, was designed. The derived code generator was tested on two examples. The experiment results show that the ForSyDe-IO is successfully synthesized into C code.

€€€€,
"Keywords[eng]": €€€€
Synchronous Data Flow, Software Synthesis, ForSyDe, ForSyDe-IO €€€€,
"Abstract[]": €€€€

Implementeringen av inbäddade mjukvaruapplikationer är en komplex process. Komplexiteten beror på det intensiva trycket på tid-till-marknad; kraft- och minnesbegränsningar. För att hantera denna komplexitet är en idé att applikationerna automatiskt kan konstrueras den högnivåabstraktionsmodellen. Synkront dataflöde (SDF) är en beräkningsmodell på hög nivå som används för att modellera inbäddade applikationer. Formell systemdesign (ForSyDe), utvecklad av

ForSyDe-gruppen vid KTH, Kungliga Tekniska Högskolan , är en metodik för modellering och design av heterogena system på chipp. Syftet med formell systemdesign (ForSyDe) är att automatiskt generera den detaljerade mjuk- eller hårdvaruimplementationen enligt systemspecifikationen på hög nivå. Formell systemdesign (ForSyDe) utgår från systemspecifikationen på hög nivå och specificerar systemmodellen på Haskell-språket. Synkront dataflöde stöds av ForSyDe. ForSyDe-IO är en mellanrepresentation av systemspecifikationen på hög nivå. Detta examensarbete
fokuserar på mjukvarusyntesen av ForSyDe-IO och synkront dataflöde, och syftar till att producera ett automatiskt verktyg som kan generera mjukvaruapplikation i C-kod för olika plattformar baserat på ForSyDe-IO. I detta projekt föreslås en mjukvarusyntesmetod för ForSyDe-IO. Sedan, baserat på mjukvarusyntesmetoden, designas en kodgenerator skriven i Java och Xtend. Den härledda kodgeneratorn testas på två exempel. Experimentresultaten visar att ForSyDe-IO framgångsrikt har syntetiserats till C-kod.

€€€€,
"Keywords[]": €€€€
Synkront dataflöde, Mjukvarusyntes, ForSyDe, ForSyDe-IO €€€€,
}