Title: A Translation Framework from RVC-CAL Dataflow Programs to OpenCL/SYCL based Implementations

Author: Florian Krebs

Supervisors: Prof. Dr. Klaus Schneider, M. Eng. Omair Rafique

Date: 2. Januar 2019
Erklärung

Ich erkläre hiermit, die vorliegende Masterarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Ich versichere, dass die eingereichte schriftliche Fassung der auf dem beigefügten Medium gespeicherten Fassung entspricht.

Kaiserslautern, den 2. Januar 2019

Florian Krebs
Abstract

Conventional programming languages nowadays still rely on sequential Models of Computation (MoC). However, the hardware makes more and more use of parallelism to increase the performance, e.g. an increasing number of cores. Nevertheless, programming languages, that still rely on sequential MoCs are not well suited to completely utilise this hardware. Dataflow programming languages like RVC-CAL that rely on Dataflow Process Networks (DPNs) as MoC are more appropriate for this purpose. DPNs explicitly require the programmer to specify different processes that can be executed concurrently. One of the most popular compilers for RVC-CAL is the Open RVC-CAL Compiler (ORCC). ORCC converts RVC-CAL code to various backends, preferably C. But the generated C code only uses the multithreading mechanism of the Operating System. However, using OpenCL, a standard for the programming of heterogeneous systems, or SYCL, an abstraction layer for OpenCL offers a much more fine-grained parallelism and, therefore, can further increase the performance.

This thesis provides a framework to convert RVC-CAL code to SYCL or OpenCL based implementations. This approach relies on the parallel execution of as many actions as possible within the actors. In comparison to other approaches, this framework does not entirely rely on synchronous dataflow. The execution of non-synchronous actors can also be parallelised.

The primary goal of this framework is to increase the performance by exploiting parallelism. This performance evaluation shows that for OpenCL based implementations performance gains up to 35% are achievable. For SYCL based implementations no performance gains are observable. Besides the performance, this framework also provides the ability to utilise OpenCL platforms other than the CPU, e.g. GPUs, and it enables a performance comparison of SYCL and OpenCL.
Zusammenfassung


Das Hauptziel dieses Frameworks ist es, die Performanz durch die Nutzung von Parallelität zu steigern. Die Performanzanalyse zeigt, dass bei OpenCL-basierten Implementierungen Performanzsteigerungen von bis zu 35% möglich sind. Für SYCL-basierte Implementierungen sind keine Leistungssteigerungen zu beobachten. Neben der Performanz bietet dieses Framework auch die Möglichkeit, andere OpenCL-Plattformen als die CPU, z.B. GPUs, zu nutzen und ermöglicht einen Performanzvergleich von SYCL und OpenCL.
# Contents

1 Introduction ........................................... 13
   1.1 Related Work ........................................... 16
   1.2 Outline ........................................... 17

2 Background ........................................... 19
   2.1 Dataflow Process Networks ......................... 19
   2.2 Dataflow Programming Languages ..................... 24
      2.2.1 CAL Actor Language ......................... 25
      2.2.2 RVC-CAL ....................................... 26
   2.3 Open RVC-CAL Compiler ............................. 26
   2.4 Open Computing Language .......................... 28
   2.5 SYCL ........................................... 29

3 Introduction to RVC-CAL ............................... 33
   3.1 Data Types ........................................... 33
   3.2 Procedures, Functions and Native Functions .......... 34
   3.3 Statements ........................................... 35
      3.3.1 Assignment ..................................... 35
      3.3.2 If-Statement ................................... 36
      3.3.3 Foreach-Statement ................................... 36
      3.3.4 While-Statement ................................... 37
   3.4 Expressions ........................................... 37
      3.4.1 List Comprehension ............................. 38
   3.5 Actions ........................................... 38
   3.6 Import Code Files ..................................... 40
   3.7 Actors ........................................... 40
      3.7.1 Priority Order ................................... 41
      3.7.2 Action Schedule ................................... 42
   3.8 Network Specification ................................... 42
## Contents

4 Methodology ........................................ 45
   4.1 Generation of SYCL and OpenCL based Code ........ 45
      4.1.1 Converting Actors with Actions with Guard Conditions ... 48
      4.1.2 Converting Actors with a unicyclic FSM ............. 49
   4.2 FIFO Buffer Design ............................... 50
      4.2.1 FIFO Buffer Access in SYCL and OpenCL ............ 52
   4.3 Scheduling ........................................ 54
   4.4 Model of Computation ............................. 56

5 Development of the Code Generator .................. 59
   5.1 Parsing the Network Files ......................... 59
   5.2 Import Handling ................................... 61
      5.2.1 Native Function Calls .......................... 61
   5.3 Code Generation for Actors ....................... 62
      5.3.1 Structure of a generated Actor .................. 62
      5.3.2 Code Generation for SYCL ....................... 64
      5.3.3 Code Generation for OpenCL ..................... 66
      5.3.4 Local Scheduler Generation ...................... 67
   5.4 Port and FIFO Buffer Code Generation ............. 68
   5.5 Global Scheduler Generation ...................... 69
   5.6 Conversion of Data Types ......................... 70

6 A Comprehensive Example ............................ 71
   6.1 Conversion of a Complete Network ................ 71
   6.2 Conversion of an Actor with Actions with Guard Conditions ... 81
   6.3 Conversion of an Actor with a Finite State Machine ....... 84

7 Evaluation ............................................. 87
   7.1 Evaluation of Image Processing .................... 88
   7.2 Evaluation of a Digital Filter ..................... 89
   7.3 Evaluation of a ZigBee Multitoken Transmitter ........ 90
   7.4 Evaluation of a JPEG Decoder ...................... 91

8 Conclusion and Future Work .......................... 93

Bibliography ............................................. 96

List of Abbreviations .................................... 101
## List of Figures

2.1 Dataflow Graph Example ........................................ 20
2.2 Synchronous Dataflow Example .................................. 23

4.1 Parallelization of Actions with Guard Conditions ............. 49
4.2 Unicyclic Finite State Machine Example .......................... 49
4.3 Parallelization of Action Schedules ............................... 50
4.4 Communication between two Actors ............................... 51
4.5 Irregular Memory Access Pattern Example ........................ 53
4.6 Regular Memory Access Pattern Example ........................ 53

5.1 Merging of Connections .......................................... 60
5.2 Structure of a SYCL or OpenCL Kernel ............................ 65

6.1 Top Network of ZigBee Multitoken ............................... 71
6.2 Network of ZigBee Multitoken .................................... 71
List of Tables

2.1 Firing Rules Example ........................................ 20
7.1 Results for Image Processing ............................. 88
7.2 Results for the FIR Filter ................................. 90
7.3 Results for ZigBee Multitoken .......................... 90
7.4 Results for the JPEG Decoder .......................... 91
## Listings

2.1 Vector Addition in SYCL ................................................. 30
3.1 min of two integers in RVC-CAL ...................................... 35
3.2 native function declaration for min .................................. 35
3.3 Inline If in RVC-CAL .................................................. 36
3.4 Foreach-Statement Example ........................................... 36
3.5 While-Statement Example .............................................. 37
3.6 List Comprehension Example .......................................... 38
3.7 Example Action ....................................................... 39
3.8 Example Actor ........................................................ 41
3.9 Instance XML Node Example ......................................... 42
3.10 Port XML Node Example .............................................. 43
3.11 Connection XML Node Example ..................................... 43
3.12 Network Specification for Array Addition ......................... 44
5.1 The General Structure of a generated C++ Actor ................. 62
5.2 The Structure of a local scheduler .................................. 67
6.1 Extract from the generated Global Scheduler for ZigBee Multitoken ........................................ 72
6.2 HeaderAdd Actor of ZigBee ........................................... 73
6.3 C++ Code generated from the HeaderAdd Actor .................. 74
6.4 Extract from the qpskMod Actor of ZigBee ......................... 77
6.5 Extract from the C++/SYCL Code generated from the qpskMod Actor .................................... 77
6.6 Extract from the C++/OpenCL Code generated from the qpskMod Actor ............................................ 80
6.7 Generated OpenCL Kernel for the qpskMod Actor ................ 81
6.8 RoundAndPack Actor .................................................. 82
6.9 Generated OpenCL Kernel for the RoundAndPack Actor ......... 82
6.10 A Test Actor with a unicyclic Finite State Machine ............ 84
6.11 C++/SYCL Code generated from the Test Actor .................. 84
1 Introduction

Nowadays, parallelisation plays an increasingly important role in computation. Computer architectures are not anymore designed mainly to increase single core performance with higher clock frequencies. Instead, modern computer architectures mainly rely on multiple separate computation units or cores to further increase the performance. CPUs evolve from Quad-Cores to Octa-Cores to Hexa-Cores. Often a System on Chip consists of multiple separate computation units or cores, sometimes even with different types of cores on a single chip. This shift enables hardware with more and more performance without increasing or even decreasing the frequency and supply voltage. This results in an increasing performance of CPUs while the power consumption remains stable and the performance per watt increases. From the hardware point of view, this works well. Nevertheless, most of the times it is up to the software to utilise the hardware and the potential performance gains.

Utilising parallel architectures, however, introduces new issues for software development. In the past, program development mainly relied on sequential programs. But sequential programs cannot utilise multicore architectures completely, and the potentially increased performance is not achievable. It is even possible that the performance decreases because the single cores might not perform as well as in the past due to the reduced supply voltage and clock frequency. Nevertheless, most of the modern programming languages still rely on a sequential Model of Computation (MoC). A MoC describes when and how operations are executed, how communication takes place and, therefore, how concurrent tasks are synchronised. But the software development domain is adapting to this change, more and more software is using multiple cores by defining multiple threads. But the underlying MoCs remain sequential. Using multiple cores, however, introduces new types of errors. For example, shared memory and wrong synchronisation of concurrent tasks can introduce errors that are hard to debug or even reproduce due to different scheduling of concurrent tasks. These issues make software development more complicated and error-prone. In this context, using sequential MoCs seems to be not optimal.

Fortunately, there are MoCs that address this issue, i.e. Dataflow Process Networks (DPNs) [1]. DPNs implicitly support the use of multiple different concurrent
tasks. Thus, DPNs are well suited to model distributed parallel systems. DPNs consist of processes and FIFO buffers in-between them. FIFO buffers are the only way of communication between the processes. This decouples the processes from each other and enables the concurrent execution of processes. Furthermore, this avoids common mistakes in multithreaded programs, like the access to shared resources without an appropriate synchronisation.

A programming language that utilizes this MoC is the CAL Actor Language (CAL) [2]. CAL can be used to specify dataflow actors using popular language constructs like if-statements, while and foreach loops or functions. The Reconfigurable Video Coding (RVC) framework uses a subset of CAL, often referred as the RVC-CAL dataflow language, for video coding development and specification [3], [4]. The RVC-CAL dataflow language is primarily meant to specify video codecs in a modular, reusable and concurrently executable way, but it can also be used to model any other suitable application like filters, communications protocols or cryptographic methods. The with RVC-CAL specified dataflow actors are then composed to complete dataflow networks with some communication channel between the actors. A well-known compiler for RVC-CAL is the Open RVC-CAL Compiler (ORCC) [5]. Besides the compiler, ORCC also provides a repository with sample implementations of different applications in RVC-CAL [6].

Another way of utilizing parallel architectures is the Open Computing Language (OpenCL) [7]. OpenCL is a specification for the programming of heterogeneous parallel architectures, i.e. graphics cards/GPUs. OpenCL enables the concurrent execution of code on devices that provide a OpenCL implementation. The host program can request thousands of instances of this code to be executed on a device as parallel as possible. Hence, OpenCL enables a much more fine-grained parallelism approach than the common multithreading utilised by many programming languages, and, therefore, OpenCL enables performance gains through the exploitation of parallelism.

However, ORCC only uses the operating system multithreading libraries for concurrent execution of the actors. The operating system libraries are sufficient for this task because the parallelisation of actors is a rather coarse-grained way of parallelisation. The parallelisation of actions or complete action schedules or executing an actor several times in parallel could offer a much more fine-grained way of parallelism. However, this would require a large number of operating system threads to execute multiple instances of an action in parallel. These threads are designed with the rather coarse-grained way of parallelisation in mind as required by programming languages that rely on sequential MoCs. Thus, using a large number of threads
for rather small tasks like a single action is not beneficial. Hence, multithreading libraries of operating systems and programming languages are not sufficient for the parallel execution of actions.

OpenCL is well suited to overcome this issue because it offers the ability to use a much more fine-grained way of parallelism than ORCC using the multithreading library of the operating system, i.e. by executing actions multiple times in parallel on a GPU. The ability to execute code on the GPU is another fact besides the fine-grained parallelism that further enhances the use of OpenCL. ORCC only uses the multithreading performed by the operating system that usually only utilises the CPU. But OpenCL code has to be written in OpenCL C and is stored in a separate file. Additionally, the programmer has to initialise the OpenCL device and program in an extensive fashion. An abstraction layer that offers an easier access to OpenCL and relies on OpenCL 1.2 is SYCL [8]. Using SYCL the kernel is written embedded into the C++ code, and the initialisation is hidden from the programmer.

This offers a more convenient way to use OpenCL because most of the programming overhead is managed by SYCL and need not be done by the programmer. Since OpenCL and SYCL enable a higher degree of parallelisation, this might allow a higher performance than the multithreading performed by the code generated by ORCC. Thus, the objective of this work is to design a code generator that converts RVC-CAL code to C++ code utilising more parallelism by the use of SYCL and OpenCL. For the code generation, a generic way to convert RVC-CAL code to C++ code is conceptualised and implemented. However, this thesis does not cover the parsing of RVC-CAL code. Hence, the RVC-CAL code, that is converted to C++, has to be developed with ORCC to ensure that the code is syntactically correct. The code generator converts the RVC-CAL code either to a SYCL based or an OpenCL based implementation or just C++. The first two versions also enable the comparison between SYCL and OpenCL.

The objective of using SYCL or OpenCL is to produce code that is more performant than the code generated by ORCC. This assumption is validated by benchmarks. As benchmarks serve some applications from the sample application repository of ORCC. For performance evaluation, a version without OpenCL or SYCL is compared to a OpenCL based version and a SYCL based version and the variant generated by ORCC. The benchmarks are executed on different available OpenCL devices.
1 Introduction

1.1 Related Work

Due to their modularity, reusability, and capability of implicit parallelism, DPNs
gained more and more interest in software development. The publication of the
CAL Actor Language constitutes a milestone in the application of DPNs in software
development. Building on this, frameworks and compilers were developed that enable
the translation of CAL or RVC-CAL code to languages that offer compilation to
executables, e.g. C. The OpenDF framework [9] can translate CAL code to C code
for the integration with SystemC. Additionally, OpenDF can generate VHDL/Verilog
code from CAL code.

Another compiler for CAL that generates C code is Cal2C [10], [11]. This compiler
generates C code and an associated SystemC model.

Also, the Ptolemy II framework [12], [13] that is part of the Ptolemy project in
which CAL was released provides a code generator. The code generator can produce
Java or C code. But the code generation is restricted to Synchronous Dataflow, finite-
state machines or Heterochronous Dataflow, an extension of synchronous dataflow,
models. The code generator of the Ptolemy II framework cannot generate code for
dynamic dataflow models.

However, all of these code generators or frameworks target mainly C as their back-
end. However, C relies on a sequential MoC and the main multithreading ability relies
on the multithreading performed by the operating system as ORCC does.

For the mapping of DPNs, especially Kahn Process Networks, Schor et al. [14]
introduced a distributed application layer. This layer maps streaming applications
onto heterogeneous many-core systems. In this approach, hierarchically organised
controllers perform the mapping at run-time.

Hautala et al. [15] showed that OS-managed threads with load balancing are more
performant than threads with a fixed CPU affinity. They created a certain number
of threads, each one looping through all the actors and executing them whenever
possible. With an increasing number of threads, they experienced an increasing
performance. Their reference was code generated by ORCC from RVC-CAL.

Based on their previous work where they introduced the distributed application
layer, Schor et al. [16], [17] and Scherer [18] provide a framework for the C/C++
and OpenCL code generation from synchronous dataflow models.

To increase the performance of the execution of dataflow models, Boutellier et
al. [19] proposed a method to parallelise for loops with OpenCL. Their approach
relies on the distributed application layer proposed by Schor et al..

In the same year, Lund et al. introduced a translation process from RVC-CAL to
C and OpenCL code. For this purpose, they reused the ORCC front-end. But their
appoach only works for synchronous dataflow actors without an action schedule and state variables. They state that this criterion is probably too strong, but otherwise the performance could decrease. They argue that for each work-item the overhead of finding the right input tokens and the place to store the output tokens in the OpenCL buffers is too high.

In comparison to all frameworks presented here, ORCC seems to be the most popular one. Additionally, ORCC provides a set of samples that can be used for benchmarking. Thus, ORCC will serve as the reference point of this thesis.

1.2 Outline

Chapter 2 gives an overview of the background of this thesis. This chapter introduces into DPNs, dataflow languages, ORCC, SYCL and OpenCL. Chapter 3 introduces into dataflow programming with RVC-CAL. After chapter 2 and chapter 3 gave an overview over all the necessary background, chapter 4 explains the methodology to translate dataflow networks specified in RVC-CAL to C++ and SYCL or OpenCL and summaries the resulting Model of Computation (MoC). The following chapter 5 discusses the implementation of the methodology. Based on this implementation, chapter 6 gives examples of the functionality of the implemented code generator. The performance of this code generator is evaluated in chapter 7. This chapter states the results of four different benchmarks and explains the difference of the benchmarks and its impact on the results. The last chapter, chapter 8, concludes this thesis by drawing some conclusions from the results of the benchmarks discussed in chapter 7.
2 Background

In this chapter the background of this thesis is introduced. The first section discusses Dataflow Graphs and Dataflow Process Networks in general, and afterwards, addresses some special forms. In the next section, the idea of Dataflow Programming Languages especially the CAL Actor Language and RVC-CAL is presented briefly. The next section gives a brief introduction to RVC-CAL. Next, in relation to RVC-CAL, a compiler for this language, the Open RVC-CAL Compiler is introduced. The last two sections give an introduction to the Open Computing Language (OpenCL) and SYCL.

2.1 Dataflow Process Networks

Dataflow graphs are directed graphs. The nodes represent actors and the arcs represent unidirectional streams of tokens [20], [21] or data elements [1]. An actor represents a function like multiplication or addition or another sub-graph. [22] The actors consume tokens from the incoming streams and produce tokens for the outgoing streams according to their function. In figure 2.1 such a dataflow graph is shown. This dataflow graph performs the function $Z = ((B + A) - (C * D)) * (E + F)$, but in a very fine-grained parallel way. For this reason, are dataflow graphs, however, not suitable to program von-Neumann-machines that rather rely on the sequential execution of code. Thus, Von-Neumann-machines provide a rather coarse-grained way of parallelism compared to the fine-grained way of dataflow graphs.

In Dataflow Process Networks (DPNs) the behaviour of the nodes in extended to complete processes and is not restricted to basic functions. In DPNs the processing of tokens is based on a set of firing rules each actor contains. The firing rules define how input tokens are mapped to output tokens, and mainly how much tokens are consumed and produced by each particular possible firing. There can also be a special rule that defines how the actor behaves when fired initially. An actor can only fire if at least one of those firing rules is satisfied. Table 2.1 shows an example of a table containing firing rules. This example represents an actor that consumes tokens from two input streams $x_1$ and $x_2$ and produces tokens for the output stream.
y as displayed in the first row. The row below defines the firing rule for the initial action that produces a one for the output stream y. In this case, the firing rules for the initial actions are separated from the other firing rules by a line. The following two rows define firing rules that can be fired repeatedly all the time. For the first rule of the two, a specific pattern must be fulfilled that the rule can be fired. The prefix or in this case, the first element that is read from the stream $x_1$ must be a one. The a or b describe that there must be at least one token in the input queue, but it does not matter what value it has. These two firing rules also define how the output is produced, either by just passing a consumed token on or by summing the values of the two consumed tokens up. However, this must not be as simple as this. The corresponding actions or functions defining the semantics of a firing rule can also be specified in a programming language like C or C++.

The streams the actors are consuming tokens from and producing tokens for are unidirectional communication channels between two actors. It is also possible that the source and the destination of the channel are equal and the actor is communicating with itself. But an actor could also achieve this with an internal state. A stream is accessed in a first in, first out (FIFO) order. Thus, the communication channels

```
x_1 | x_2 | y
---|---|---
A  | B  | 1
(a::A) | (b::B) | [a+b]
```

Table 2.1: Firing Rules Example
2.1 Dataflow Process Networks

are FIFO buffers between actors. The FIFO buffers are the only way of communication between two actors in this model and, therefore, they decouple the actors from each other. Thus, no clock or some other way of synchronisation is required.

Consequentially, an actor can but does not have to fire if there is at least one of its firing rules satisfied. If more than one firing rule is satisfied, the actor can fire any.

This property of DPNs introduces two significant problems: determinism regarding the actors and boundedness regarding the FIFO buffers.

Determinism is the property of an actor to produce an output stream that does not depend on the execution order, or from another point of view if knowledge about the firing rules and input streams is not sufficient to determine the output streams [1]. If every actor in the network is deterministic, the whole network is deterministic [23]. The example in table 2.1 shows such a non-deterministic actor. Consider the two input streams \( x_1 = [1,2,3] \) and \( x_2 = [3,2,1] \). The initial action does not really matter in this context because there are no scheduling choices possible. The next time the actor fires, it will consume a one from stream \( x_1 \) and a three from stream \( x_2 \). Thus, both firing rules are satisfied. If the first is fired, the output stream finally looks like this: \([1,3,4,4]\), if the second is fired, the output stream finally looks like this: \([1,4,4,4]\). Since the two output streams are different, the scheduling is relevant for the output of this actor. Thus, the actor is non-deterministic.

Boundedness of FIFOs means that the number of tokens that are buffered in the FIFO at every point in time for a given execution order or scheduling has an upper bound. Boundedness cannot always be guaranteed because in a given scheduling more tokens might be produced than consumed. Eventually, this would lead to an overflow of any fixed size buffer.

A special case of DPNs that addresses these issues are Kahn Process Networks [24]. Kahn Process Networks use infinite streams or buffers between the nodes or processes and enforces specific properties of the processes. The processes or their firing rules must have no conflicting firing rules, no fixed size patterns for input streams and it must be possible to read the different input streams sequentially. No conflicting firing rules means that for given input streams not more than one firing rule can be satisfied. If more firing rules are satisfied, a scheduling decision has to be made, like it is shown in the example for non-determinism. No fixed size patterns means that all patterns that must be satisfied to enable a firing rule should only consider the prefix of the input stream but not the entire stream. For instance, a fixed size pattern could be \([1,2]\) instead of \((1::2::A)\). In this case, this pattern would be satisfied if the stream contains a one and a two but nothing else. But the firing rule can be disabled
by the producer by producing more tokens. This again leads to a dependency of the output on the execution order or the time the scheduling is made. Sequential reading means that the different input queues can be accessed in a fixed sequential order. For instance, the tokens can always be consumed first from stream \( x_1 \) and then from \( x_2 \). This property is also essential because Kahn Process Networks impose that reads on buffers are blocking. If the FIFO is empty and the consumer tries to read, the consumer will block until there is an element it can consume. The actors cannot determine the size of the FIFOs or check for emptiness to avoid blocking reads. If this would be possible, the processes can implement non-deterministic behaviour because if no tokens are available, it just fires another rule instead of waiting. Sequential reading ensures that checking which rule to fire happens in a deterministic way. These properties ensure that the processes and the network are deterministic.

However, for real applications, infinite FIFOs are hard to implement. Unfortunately, fixed size FIFOs introduce some more constraints on scheduling or the execution order. In addition to the availability of tokens, in this case, also the availability of free space in the output FIFOs is relevant for scheduling. Fixed size FIFO buffers can lead to a synchronisation of the processes because they are now dependent on their predecessor and successor in the network and they can synchronise on the size of the buffers. If the buffer is full, the producer can only continue when the consumer consumed tokens from the buffer.

There are two special forms of DPNs: synchronous dataflow and cyclo-static dataflow. In synchronous dataflow networks, every firing of an actor produces the same amount of tokens and consumes the same amount of tokens. The number of consumed tokens, however, can differ from the amount of produced tokens. Different actors can have different numbers of tokens that are consumed and produced. But for every firing rule within one actor, they must remain the same. Thus, the number of tokens produced and consumed is independent of the executed action or firing rule and the values of the consumed tokens. Figure 2.2 shows an example for an synchronous dataflow network. The numbers on the incoming edges denote the number of consumed tokens from this buffer and the numbers on the outgoing edges denote the produced tokens every time the actor fires. The number of consumed or produced tokens is called token rate. Even if the token rate is the same every time the actor fires, different firing rules can be fired, and the functionality can differ within the actor \([25], [26]\). The constant token rates enable the creation of static schedules \([27], [28], [29], [30]\). In a static schedule, every token that is produced is also consumed. Thus, the number of tokens in all FIFO buffers after the execution
of the schedule is equal to the number of tokens in the same FIFOs before the execution of the schedule. A schedule for the example in figure 2.2 could be: \( a \) could fire four times, \( b \) twice, \( c \) five times and \( d \) twenty times. After all these firings, the number of tokens in the buffers is the same as it was before. This so-called repetition vector can be calculated by balance equations for each edge, for example, 

\[
\text{#firings } a \times 5 - \text{#firings } b \times 10 = 0
\]

for the edge between \( a \) and \( b \).

If these equations are not fulfilled, more or fewer tokens might be in the buffers afterwards, and the repeated execution of the static schedule might not be possible due to too few tokens or an infinitely increasing number of tokens in a FIFO buffer. Hence, resulting from a static schedule, bounds for the FIFOs can be determined.

Cyclo-static dataflow networks extend this behavior to different states [31], [32]. Every actor can have multiple states. In comparison to synchronous dataflow, between each state, the number of produced and consumed tokens can vary. When an actor is fired, the state of the actor transitions to the next state. The state transitions are cyclic. For example, instead of a constant token output rate of five, there can be \( \frac{5}{3}/\frac{4}{1} \). This means, in the first state five tokens are produced, then the state is transitioned to the next state, in this state three tokens are produced and so on. After the state is reached in which one token is produced, the state again transitions to the state in which five tokens are produced. The state transitions always happen in the same order. Thus, also for cyclo-static dataflow networks, a static schedule can be created. The tokens consumed and produced in one complete cycle can be summed up, and in this way, a corresponding synchronous dataflow model is obtained. Based on this model a static schedule can be determined like in synchronous dataflow networks [33]. However, it should be noted that for each firing in the resulting schedule of the synchronous model, a complete cycle in cyclo-static model has to be fired.
2.2 Dataflow Programming Languages

The previously presented Dataflow Graphs and Dataflow Process Networks are used as the underlying MoC for programming languages. First, dataflow programming was often used as a graphical programming approach [34] or to program dataflow architectures. Mainly this approach focused on pure dataflow graphs as described at the beginning of the previous section about Dataflow Process Networks. However, this approach is not beneficial for sequential sections of the program. If sequential sections are mapped to a real machine, they would require many resources and cause communication overhead because every function or operation is a separate actor. After the execution of each actor, the tokens have to be communicated to the next function, and a new scheduling decision has to be made, despite in sequential sections it is clear which actor to execute. Thus, the overhead for the execution of dataflow graphs is rather high, especially on von-Neumann machines that are made for the sequential execution of instructions, supporting parallelism only by a small number of threads [35], [36]. Hence, von-Neumann machines were considered as not suitable for the execution of dataflow graphs [34].

To overcome this issue, the fine-grained parallelism of dataflow graphs with nodes representing functions has changed to complete processes as nodes like in DPNs [36]. This way sequential blocks can be combined to one actor and the resulting larger actors can be executed with the multithreading mechanism of von-Neumann machines more efficiently. Hence, the high scheduling and communication overhead is reduced because much fewer actors have to be scheduled and are communicating. Additionally, von-Neumann-machines increasingly exploit much more parallelism for performance gains and can execute more and more threads in parallel. This further increases the efficiency of the execution of dataflow programs on von-Neumann-machines.

Thus, for von-Neumann-machines, the Dataflow MoC is appropriate as the underlying model that defines how the computations are organised, e.g. the scheduling of the actors, or how the actors are communicating. But the processes have to be specified differently. In this approach, programming is separated into computation and coordination [37]. Dataflow actors specify the computation. The language used to specify them is called host language [1]. The language used to specify the coordination or scheduling and communication is called coordination language.

Two programming languages that are using Dataflow Process Networks as their underlying MoC are the CAL Actor Language and the derivated Reconfigurable Video Coding (RVC)-CAL Actor Language (CAL).
2.2 Dataflow Programming Languages

2.2.1 CAL Actor Language

The CAL Actor Language (CAL) was developed during the Ptolemy II project at the University of California at Berkeley in 2001. CAL is a textual programming language that can be used to specify asynchronous and non-deterministic dataflow actors [2]. The actors have input and output ports to which the communication channels are connected to the actors. But the behaviour of the connection channel between two ports is not part of the description of CAL. Actors can have state variables that can also influence the scheduling of the actions. CAL provides, besides the basic data types, like Number, Character, Boolean and String, a variety of more complex built-in data types, like Set, Map, and List. They are organised as a generic data type and can contain any other valid data type, for example, a list of lists of numbers is possible. But in contrast to many other programming languages, lists in CAL are more like arrays in other languages because the size must be specified at initialisation and cannot change over time.

Actors contain one or more actions. Each time the actor fires, one of these actions is executed. Actions consume tokens, produce tokens and cause state transitions of the actor. In addition to actions, the language also offers functions and procedures. The difference between them is that functions have a return value and procedures not. For the implementation of the functionality of the actions, functions, and procedures, CAL provides the basic statements, like if-then-else, while and foreach.

The specification of CAL also does not define how actors are composed to complete networks or how the scheduling happens. However, the language provides mechanisms to influence the firing of actions, for instance finite state machines, priorities and guards. Finite state machines define states and the actions that can be fired in this state. Every firing of an action causes a state transfer. Priorities define the order in which the actions are tested whether they can be fired or not. Guards define conditions that must be satisfied to fire an action. Each action can have multiple guard conditions.

In summary, can CAL be used to model actors, but CAL does not define the complete semantics of a network composed of the actors. Thus, the underlying MoC is a DPN, but it is not entirely fixed. The communication between the ports and how scheduling decisions are made is not stated. Hence, the final MoC results from the way communication and scheduling are implemented.
2 Background

2.2.2 RVC-CAL

The Reconfigurable Video Coding (RVC)-CAL Actor Language (CAL) is based on the previously presented CAL Actor Language (CAL). RVC-CAL is part of the standardized Reconfigurable Video Coding (RVC) framework of MPEG [3], [4]. The purpose of this framework is to provide a way of specifying video codecs with library components instead of monolithic algorithms. Often the different codecs contain similarities, but there was no standard way of building codecs from standardised library components. For the dynamic configuration and reconfiguration of video codecs and to exploit concurrency, dataflow models are well suited. The ability to specify a video codec with standardised components enables a higher abstraction in the reference specifications compared to C or C++ code. Thus, dataflow models also facilitate modularity and reuse.

RVC-CAL is a subset of CAL. However, in contrast to CAL, all actors are fully typed, and some language constructs are not supported, like Set or Map. These restrictions are made to enable efficient hardware and software generation.

In addition to RVC-CAL, the standard also provides the XML based Functional unit Network Language (FNL), a language to specify complete networks [38]. A network is created by instantiating actors or other networks and connections between the ports of the instances. Connections represent the communication channels. Networks can also contain ports that connect the instances of the network to other instances through a communication channel. A use case for network instances is, for instance, a decoder that is specified by a network and instantiated in another network where the functionality of this decoder is required. In the following RVC-CAL and FNL will be referred as RVC-CAL.

2.3 Open RVC-CAL Compiler

The Open RVC-CAL Compiler (ORCC) is a eclipse based open-source compiler for RVC-CAL code [5]. Additionally, ORCC provides an eclipse plugin for the development of RVC-CAL code, visual graph creation and editing of dataflow networks, simulation of the networks and code generation for various back-ends and a repository of samples like JPEG Encoding and Decoding or digital filters [6]. ORCC can create code for ANSI C with multi-core ability, Java, Low Level Virtual Machine (LLVM) code meant to be used with the JIT Adaptive Decoder Engine (Jade), Verilog and Promela that can be used with the SPIN model-checker. ORCC also offers a TTA back-end for co-design to design a system based on the Transport-Trigger Architecture. However, the C code generation is the only back-end that is working correctly.
For code generation and simulation, besides using the eclipse environment directly, ORCC also provides a headless build that can be used in the command-line [39]. Unfortunately, the code generation process relies on eclipse. Even the headless build uses eclipse as its backend. The code or network that shall be compiled has to be part of an eclipse project.

ORCC generates code with FIFO buffers of fixed length between the actors. For scheduling, two data structures are maintained: global scheduler data structure and multiple local scheduler data structures. The global scheduler data structure contains for each thread one local scheduler data structure. The number of threads is equal to the number of processors, and each thread gets an affinity to one core. The global scheduler data structure is only responsible for holding the local scheduler data structures.

Each actor is mapped to one thread, and a reference to this actor is inserted into the list of the local scheduler data structure of this thread. Alternatively, the mapping can be determined by the user by inserting a mapping file containing the desired mapping. Additionally, a scheduling strategy has to be provided. The code generated by ORCC supports two scheduling strategies: Round Robin (RR) and Data/Demand Driven (DDD). The threads start their execution in a scheduling routine. This scheduling routine uses the local scheduler data structure of the thread to decide based on the scheduling strategy which of the actors in the list to execute next. Then the scheduling routine of the thread calls the scheduling routine of this actor. This scheduling routine determines which action to execute. To make this scheduling decision the scheduling routine has access to the FIFO buffers including size and free space and the state variables of the actor and the finite state machine. Additionally, the scheduling routine makes this decision based on the priorities defined in the actor. If an action is not schedulable due to too less free space in at least one output buffer or no action is schedulable at all the scheduling routine terminates, and the control returns to the local scheduler of the thread, and the next actor is scheduled. The termination of the actor’s scheduling routine due to too less free space avoids a different scheduling in this case because the original MoC contains infinite FIFO buffers that do not have this constraint.

As already mentioned the way actors are communicating and how scheduling is done determines the actual underlying MoC. Hence, this description gave an impression of the underlying MoC of code generated by ORCC.
2 Background

2.4 Open Computing Language

The Open Computing Language (OpenCL) is an open standard for the task and data parallel programming of heterogeneous systems [40]. OpenCL can be used to execute code on CPUs, GPUs, Digital Signal Processors, Cell Processors, FPGAs or embedded systems many times in parallel [7].

For parallel execution, every platform that implements OpenCL can be used. A platform consists of a host and at least one device and provides the OpenCL implementation. A device consists of at least one compute unit, and a compute unit consists of at least one processing element. The processing elements are the execution units for the code. Devices have different restrictions regarding memory and the degree of parallelism.

The host manages the execution on the devices, it selects the devices and controls the execution of kernels on them. A kernel is a function that shall be executed on a OpenCL device. First, the host creates for each argument of the kernel OpenCL buffers and inserts data into the buffers if necessary, e.g. input data. Then the kernel is enqueued for execution. In this step also an index space called NDRange has to be provided. The index space defines how often the kernel is executed. The index space is N-dimensional and consists of a global index space and a local index space. The global index space defines how many kernel instances overall are required. Each kernel instance is called a work-item. The local index space divides the global index space into smaller portions and groups multiple work-items together into one group called work-group. One work-group is mapped to one compute unit of the device, and its work-items are executed concurrently on the processing elements of the compute unit. Each work-item has a unique global and within its work-group a unique local ID. Additionally, each work-group has a unique work-group ID. The global ID or the local ID together with the work-group ID can, for example, be used to index the buffers to find the data the particular work-item shall process.

The work-group size, the maximum number of overall work-items and the maximal dimension of the NDRange are dependent on the device or the platform.

Like the index space, the memory is divided into different regions mostly relating to the execution model with a global space, work-groups, and work-items. There is a Global Memory accessible from every work-item. In this region, primarily buffers for input and output data are located. The Global Memory is the only memory region that can be accessed by the host through OpenCL functions. Additionally, in the Global Memory region, there is a Constant Memory region that stays constant throughout the execution of the kernel. The conceptual OpenCL device architecture also contains a cache for the global and constant memory. Each work-group has
2.5 SYCL

SYCL is a abstraction layer built on top of OpenCL. Hence, SYCL inherits the efficiency and the ability of OpenCL to execute programs on multiple platforms, but it abstracts from the constraint to specify the kernels using OpenCL C. Instead, SYCL enables a Single-source Multiple Compiler-Passes (SMCP) design that makes the integration of the kernel into the host C++ program possible. The ability to specify the kernels embedded into the host program shall simplify the development because kernels no longer need to be written in OpenCL C. Instead, they can be written in the host language, and kernels are not located in separated .cl files. Hence, the development is more comfortable because only one language is used. Maintaining the code is also more convenient due to its much lower complexity. Also, the SMCP
design increases the reusability of the code because it bridges the gap between host and device code, making it easier to reuse the whole module. The third primary benefit from the SMCP design and the ability to embed device code into the host code is efficiency. Efficient algorithms and datatypes, even with C++ template techniques, from standard libraries can be used in the kernels. This design shall enable the compiler to make use of inlining and produce more efficient code.

Another benefit of SYCL is that a lot of the initialisation of OpenCL is hidden from the programmer. Using OpenCL, the programmer has to initialise the execution on the device with dozens of lines of code. SYCL, on the other hand, provides classes that are hiding most of the platform and device selection and kernel initialisation from the programmer and manages all related OpenCL objects. For this purpose, the SYCL framework provides three components: a SYCL C++ Template Library, the SYCL runtime, and SYCL Device Compilers. The SYCL C++ Template Library provides classes that are abstracting from OpenCL objects like command queues, OpenCL memory objects and access to them, platforms and devices and their initialisation. The SYCL runtime is the interface between the host program and the OpenCL implementations. The runtime selects the required OpenCL device and manages its command queue, handles kernels and their scheduling and takes care of memory management and data movements. SYCL Device Compilers translate the kernels that are specified in C++ to OpenCL C to enable the execution on OpenCL devices.

Basically, with SYCL only three lines of code excluding a lambda expression for the kernel are necessary to initialise the SYCL based execution. Listing 2.1 shows an example for vector addition with SYCL.

```cpp
#include <CL/sycl.hpp>
using namespace cl::sycl;
//create arrays for the inputs and outputs
float inputDataA[1024];
float inputDataB[1024];
float outputData[1024];
//...insert data into inputDataA and B
//create SYCL Buffers
buffer<float, 1> inputBufferA(inputDataA, range<1>(1024));
buffer<float, 1> inputBufferB(inputDataB, range<1>(1024));
buffer<float, 1> outputBuffer(outputData, range<1>(1024));
//initialize the device and command queue
default_selector selector;
queue queue(selector);
//specify and execute the kernel
```
queue.submit([&](handler &cgh){
    //get access to the buffers
    auto inputPtrA = inputBufferA.get_access<access::mode::read>(cgh);
    auto inputPtrB = inputBufferB.get_access<access::mode::read>(cgh);
    auto outputPtr = outputBuffer.get_access<access::mode::write>(cgh);
    //define index space
cgh.parallel_for<class vectorAddition>(
        nd_range<3>(range<3>(16, 8, 8), range<3>(4, 2, 2)),
        [=](nd_item<3> workitem) {
            //the actual code of the kernel
            outputPtr[workitem.get_global_linear_id()] = 
                inputPtrA[workitem.get_global_linear_id()] +
                inputPtrB[workitem.get_global_linear_id()];
        });
    //create host accessor to the output buffer to read the data
    auto hostAccessor = outputBuffer.get_access<access::mode::read>();
});

Listing 2.1: Vector Addition in SYCL

First, two arrays containing the input data and one for the output data are created, and for each array, a SYCL buffer is created. Then, a default selector is created. The default selector selects an OpenCL device by default settings, most likely it will be a GPU if one is available. SYCL also provides a cpu selector, a gpu selector or the programmer can define an own selector to select precisely the required device. Unfortunately, SYCL relies on SPIR [41], an intermediate representation for code, that is not supported by NVIDIA. Thus, NVIDIA GPUs are not utilisable by SYCL.

Next, for the selected device a command queue is created. The kernel is defined directly as a lambda in the submit call that enqueues the kernel to the command queue. The name, given in the template argument of parallel_for, in this case, vectorAddition, is the name of the kernel and has to be unique among all kernels defined in the program. Also, there is no need to set the arguments for the kernel. Only accessors to the buffers that are accessed during the execution of the kernel instances have to be created. Constants that are defined outside of the kernel can be used in the kernel without any accessor. After the execution of the kernel, the buffer can be accessed by the host in the same way it is done in the kernel code. If no handler is supplied, a host accessor is created. Alternatively, the buffer can be destroyed. Due to the supplied pointer to the host memory region, the memory is mapped back to this region.

There are two implementations of SYCL: ComputeCpp by Codeplay [42] and the
open-source version triSYCL [43] mainly funded by AMD and Xilinx.
3 Introduction to RVC-CAL

This chapter gives an introduction into dataflow programming with RVC-CAL with special regard to ORCC. The first sections introduce into data types, statements and procedures, and functions. The following section discusses how actions are defined. The next section explains how a library with functions, procedures, and constants that can be imported into actors, is implemented. The following section finally describes how actors are defined. The last section briefly introduces into the specification of complete dataflow networks with FNL.

The following introduction reflects the common use cases of the presented language constructs based on the samples of ORCC, but they are not presenting the complete functionality of RVC-CAL. For a complete specification of the language see [3] and [4].

As notation, the keywords are marked with boldface writing, literals are put in ’ , optional parts are put in square brackets and parts that can occur multiple times but at least once are put in braces. If the parts occur multiple times, they are separated by commas or, in the case of statements, by semi-colons or the end keyword.

3.1 Data Types

In RVC-CAL a data type has to be specified at every parameter, constant or variable declaration. In CAL this is not necessary. The data types of RVC-CAL that are also supported by ORCC are:

- int
- uint
- bool
- half
- float
- String
3 Introduction to RVC-CAL

- List

For each data type an additional size can be specified, e.g. `int(SIZE=8)` specifies an 8-bit integer. For the size specification also constants and expressions of constants can be used, e.g. assuming `a=5` and `b=4` are constants, then `int(SIZE=a*b)` specifies a 20-bit integer. If no size is specified the default is 32. The lists supported by RVC-CAL are not variable length lists like they are known from many other programming languages. Hence, besides the type, also the size of the list has to be specified in a list declaration, e.g. `List(type:int,size=5)` declares a five elementary list of type integer with 32 bits. The type could also be another list to declare multi-dimensional lists. Lists are accessed like arrays with an index and the `[ ]` operator. However, ORCC also supports arrays, although they have the same behaviour than lists.

3.2 Procedures, Functions and Native Functions

Procedures and functions are of the following form:

Function: \[[@native] \text{function \ ID \ '}(\ [\{\text{Parameter}\}] \ ') \ '--- >' \ \text{returnType \ [\var \ \{\text{VariableDeclaration}\} \ ] \ [' : \ \{\text{Statement}\}] \ end}\]

Procedure: \[[@native] \text{procedure \ ID \ '}(\ [\{\text{Parameter}\}] \ ') \ [\var \ \{\text{VariableDeclaration}\}] \ [\begin{\text{begin} \ \{\text{Statement}\}] \ end\]

The difference between procedures and functions is that procedures do not return something.

If a function or procedure declaration is marked with @native, the implementation of this function or procedure is not provided by the RVC-CAL code. Instead, the implementation is provided by a code file written in the language the RVC-CAL code is compiled to, most likely C. This file is only needed during the compilation process of the target language code, but not during the compilation of the RVC-CAL code.

An example of a function is finding the minimum of two integers as shown in listing 3.1.

This function returns `a` if `a` is smaller than `b` and otherwise `b`. But RVC-CAL does not use return statements for this purpose. Instead, the result of the last executed expression is returned.

In this function also two variables are defined, dummyValue and dummyFlag. These are only there to show where variables and constants can be defined. Variables and constants can only be declared in the var block, between the var and the colon. A comma separates different variable and constant declarations. If the function does
function min(int a, int b) ---> int

var
int(size = 8) dummyValue = 5,
bool dummyFlag :
  if(a < b) then
    a
  else
    b
end
end

Listing 3.1: min of two integers in RVC-CAL

not define any variable or constant the var block can be omitted, and the colon directly follows the return type.

If this function shall be carried out by native code, only the native function has to be declared in the RVC-CAL code. Listing 3.2 shows the native function declaration of the min function.

@native function min(int a, int b) ---> int end

Listing 3.2: native function declaration for min

Compared to functions, procedures are defined with the keyword procedure, instead of the keyword function. Additionally, no return type needs to be declared, and instead of the colon, the keyword begin is used.

Functions and procedures whether native or not are called by their name followed by the parameters in parenthesis.

3.3 Statements

Most of the widely used programming language constructs are also part of RVC-CAL. Nevertheless, RVC-CAL keeps this list rather small for efficient software and hardware generation, but large enough to preserve the expressivity.

3.3.1 Assignment

RVC-CAL has two different assignment operators: := and =. The = operator is used to initialise a constant. This is the only way to define a constant. The := operator is used to assign a value to a variable. It is also possible to assign lists to lists or arrays at any time in the program, not just at initialisation. If only one value of a
list or array shall be manipulated, the [] operator can be used to assign a value to the desired location at the given index. An assignment is terminated by a semicolon.

### 3.3.2 If-Statement

The if-statement is the most basic conditional statement. The complete statement looks as follows:

```rvc-cal
if Expression then {Statement} [else {Statement}] end
```

If the Expression evaluates to true, the statement block after the keyword *then* is executed, if the expression evaluates to false the statement block after the else is executed or if no else block is defined, the if-statement is terminated. The else part is only optional. The whole statement is concluded with an *end*.

This statement can also be used inline for conditional assignments as shown in listing 3.3.

```rvc-cal
int min := if a < b then a else b end
```

Listing 3.3: Inline If in RVC-CAL

In this example the minimum of the values a and b is assigned to min.

### 3.3.3 Foreach-Statement

The foreach-Statement is used to iterate over lists. The basis structure of this statement is:

```rvc-cal
foreach {Type ID{, ID} in {Expression} } [var {VariableDeclaration}] do {Statement} end
```

Listing 3.4 shows an example for a foreach-statement.

```rvc-cal
foreach int i in 4 .. 20
do
  //...loop body
end
```

Listing 3.4: Foreach-Statement Example

The loop in this example runs from i=4 to i=20 with the step width of 1 and executes the loop body for each i.
3.3.4 While-Statement

The while-statement is used to execute statements as long as a given boolean expression evaluates to true. The basic structure of the while-statement is as follows:

```vbnet
while Expression [var {VariableDeclaration}] do {Statement} end
```

Listing 3.5 shows an example for a while-statement.

```
while a < 5
var
    int counter
do
    //...loop body
end
```

Listing 3.5: While-Statement Example

In this example, the execution of the loop body is repeated as long the condition \( a < 5 \) evaluates to true.

3.4 Expressions

Expressions can always be evaluated to values of an explicit type. The evaluation is side-effect-free. Hence, the evaluation of an expression is not allowed to modify any variable of the actor or action state. Expressions can consist of the following parts:

- Identifiers of variables and constants
- Indexing of lists
- Literals
- Function and procedure calls
- Arithmetic operators, e.g. +, -, *, /, mod, >>, <<
- Logical operators, e.g. not, and, or
- Comparative operators, e.g. !=, <, >, >=, <=, ==(equal to == in many other programming languages)
- List Comprehension
- Inline If like it is explained in subsection 3.3.2
- brackets
3.4.1 List Comprehension

A list comprehension is used to generate lists. List comprehensions are of the following form:

\[
[\text{Expression} : \{ \text{for Type } \{ \text{ID} \} \text{ in } \{ \text{Expression} \} \} ]
\]

The first expression defines the value that is inserted into the result of this list comprehension. How often the expression is evaluated is determined by the for loops on the right side of the colon. The for loops are equal to the previously discussed foreach loops.

An example for list comprehension is shown in listing 3.6.

```
List(type:int,size=5) temp := [a*2:for int a in 1..5]
```

Listing 3.6: List Comprehension Example

In this example, the list comprehension generates the list \([2,4,6,8,10]\). This list is then assigned to the variable temp.

Strictly speaking, lists that are assigned to variables or constants are also list comprehensions without any dynamic list generation, e.g.\([1,2,3]\). The dynamic list generation is only an optional part of a list comprehension.

3.5 Actions

Actions are the core functionality of an actor. They are consuming tokens from the input buffers and producing new tokens that are inserted into the output buffers through the ports of the actor to which the buffers are attached. The basic structure of an action looks like this:

```
Action : [ID '.' ID ':'] action InputPatterns '=>' OutputExpressions [guard {Expression}] | [var {VariableDeclaration}] | [do {Statement}] end
```

First, an action identifier can be specified. The identifier consists of a tag and a name, separated by a dot. This identifier is only optional. If an action identifier is specified, it is separated from the further action definition by a colon. This identifier is used to specify priorities between actions or action schedules. After the action keyword, the accesses to the buffers are defined:

```
InputPattern : ChannelID ':' ['| ID '] repeat Expression
```

Each InputPattern consists of the ID of the input buffer or channel that is accessed, followed by a colon. Then, in parenthesis, the identifiers for the consumed tokens are defined. The number of identifiers specifies how much tokens the action consumes.
from this buffer. The number of consumed tokens per identifier is defined by the keyword `repeat` followed by an expression that can be evaluated to a number at compile time. Without `repeat` one token per identifier is consumed and assigned to the identifier, otherwise, as many tokens as defined by `repeat` are consumed per identifier and are assigned to the identifier as a list. The consumed tokens are assigned to the identifiers in the order they are consumed, e.g. if the input buffer that is attached to the port `dataIn` contains the following stream: `[1,2,3,4,5]` the InputPattern `dataIn:[a,b] repeat 2` evaluates to `a = [1,2]` and `b = [3,4]`. The InputPatterns are separated by commas. The accesses to the output buffers basically work the same way but are more complex:

OutputExpression : ChannelID ':' '[' Expressions ']' `[repeat` Expression]

Like the InputPattern, the OutputExpression starts with the identifier of the buffer or channel that is accessed, followed by a colon. However, in the parenthesis, there are not only identifiers but complete expressions. The values the expressions are evaluated to are inserted into the output buffer that is attached to the port specified by the ChannelID. Each expression evaluates to as many values as defined by `repeat` or, if no repeat value is specified, each expression produces one value. Since the expressions can be inserted directly in the action head, it might not be necessary to define an action body at all.

After the OutputExpressions, guard conditions can be specified. If an action has guard conditions, this action can only be fired if all conditions evaluate to true. The conditions can refer to actor state variables and the tokens that might be consumed from the buffers. By evaluating guard conditions, the tokens are not consumed, only when the action is actually executed.

Listing 3.7 shows an example for an action with guard conditions.

```
tag : action Buffer1:[a,b], Buffer2:[c] repeat 3 ===> OutputBuffer:[out]
guard c[1] != a
var int out, int temp
do
    temp := a + b;
    out := temp/(c[0]+c[1]+c[2]);
end
```

Listing 3.7: Example Action

This action is only executed if the first token in the buffer attached to port `Buffer1` is not equal to the second token in the buffer attached to port `Buffer2`. After the execution of the action’s body, the value of `out` is inserted into the buffers attached to the port `OutputBuffer`. 

3.5 Actions
A special form of an action is the initialisation action. Instead of the action keyword, the initialize keyword has to be used. This action is fired only once initially when the actor is executed the first time. The initialize action cannot consume tokens, but it can produce tokens.

### 3.6 Import Code Files

Besides actors, RVC-CAL also supports libraries that can be imported into actor definitions. The library code has to be encapsulated by a unit construct:

```plaintext
unit ID ': {Constant|Function|Procedure} end
```

The ID after the unit keyword is the identifier of this unit construct. The identifier has to be equal to the name of the file in which the unit is stored. The construct can contain constants, functions, and procedures.

A unit file is imported into an actor definition by the following statement: import path.fileName.ID;. The path is the path to the file relative to the project folder. The ID specifies which part of the unit file shall be imported, e.g. if a unit construct named math contains a function called min, this function is referred to as import path.math.min;. If the complete content of the unit construct shall be imported .* can be used, e.g. import path.math.*;.

### 3.7 Actors

Actors can contain state variables, constants, functions, procedures, functions and actions. The basic structure of an actor looks like this:

```plaintext
[{{import ID { ‘.’ ID ‘;’} } ] actor ID '{‘ Type ID [‘=’ Expression] ‘} ‘ ] [ {Type ID} ] ‘==> ‘ ] { {Type ID} } ‘: ‘ {VarDecl|Action|FunctionDef|ProcedureDef} [InitializationAction] [PriorityOrder] [ActionSchedule] end
```

Before the actor definition, the imports are listed as described in the previous section. The actor definition starts with the keyword actor followed by the identifier of the actor. Then, in the parenthesis, parameters and their default values can be defined. These parameters are constants. For each instance of an actor, the values of the parameters are specified in the network definition or the default value is taken. However, default values are only optional. After the parameter definition, the input and output ports are defined, separated by the literal ==>. A colon follows the port definition. After the colon, the body of the actor starts. In the body, the actions, functions, state variables, and so on are defined.
Listing 3.8 shows an example of an actor.

```plaintext
actor example_actor (int offset=5,bool flag)
    int data_in ==> int data_out :

    //...state variables,functions,procedures,actions

    priority
        Action1 > Action2;
        Action3 > Action1;
    end

    schedule fsm init_state:
        init_state(Action1) ==> running;
        running(Action1,Action2) ==> done;
        done(Action3,Action1) ==> init_state;
    end
end
```

Listing 3.8: Example Actor

This example shows the definition of the actor example_actor. This actor has two parameters: offset with the default value five and flag, one input port called data_in and one output port called data_out. In the actor’s body, state variables, functions, procedures and actions are defined, indicated by the comment. For this example, it is assumed that the actions Action1, Action2, and Action3 are defined. What comes next in the example is the definition of the priority order and the action schedule. Both constructs are used to influence the scheduling of the actions.

### 3.7.1 Priority Order

The basic structure of a priority order is as follows:

```plaintext
priority {ActionTag{’>’ ActionTag};'}end
```

The priority order defines scheduling priorities between actions, e.g. in the previous example, Action1 has a higher priority than Action2. If more than one action is schedulable and priorities between them are defined, the priorities decide which action is actually scheduled.

This priority order can be understood as the order in which the schedulability of the actions is checked. If an action is schedulable, it can then be scheduled right away without checking any further action.
3.7.2 Action Schedule

A finite state machine defines the action schedule with the firing of actions as state transitions. The basic structure of an action schedule is as follows:

```
schedule fsm InitStateID ':' {StateID '(' ActionTag [',' ActionTag] ')' '−−' >'} StateID ';' } end
```

The identifier between fsm and the colon determines the initial state of the finite state machine. After the colon, all possible state transitions are listed. A state transition consists of the identifier of the current state, followed by the action identifiers whose execution trigger this state transition in parenthesis. On the right side of the arrow, the new state is stated. For instance, in the previous example, the firing of Action1 in the state init_state causes a state transition to the state running. In the state running the actions, Action1 and Action2 can potentially be scheduled. The new state can also be equal to the previous state.

In a state, only the actions are enabled that would cause a state transition. All other actions are disabled in this state.

3.8 Network Specification

With the just introduced actors, complete networks can be defined. An actor can be instantiated multiple times in a network, called actor instance in the following. The networks are specified in XML. A network specification starts with a node tagged with XDF and the network name as an attribute:

```
<XDF name="Network_Name">
  ...
</XDF>
```

This node has child nodes tagged with `Instance`, `Port` and `Connection`.

An instance node looks like shown in listing 3.9. The attribute `id` of the Instance

```
<Instance id="InstanceID">
  <Class name="common.ActorName"/>
  <Parameter name="offset1">
    <Expr kind="UnaryOp">
      <Op name="−"/>
      <Expr kind="Literal" literal="−kind="Integer" value="128"/>
    </Expr>
  </Parameter>
  <Parameter name="offset2">
    <Expr kind="Literal" literal="−kind="Integer" value="5"/>
  </Parameter>
</Instance>
```

Listing 3.9: Instance XML Node Example
node is a unique id for an actor instance or network instance. The actor or network that is instantiated is referred to in the attribute name of the child node Class. The attribute name contains the path to the actor or network relative to the project’s top folder. Instead of slashes, points are used to separate the sub-folders. The nodes tagged with Parameter and their child nodes are used to specify the parameter values of the actor instance. In this example the value -128 is assigned to the parameter offset1 and the value 5 to the parameter offset2.

A Port specification looks like shown in listing 3.10.

```
<Port kind="Output" name="WIDTH">
  <Type name="int">
    <Entry kind="Expr" name="size">
      <Expr kind="Literal" literal="-16"/>
    </Entry>
  </Type>
</Port>
```

Listing 3.10: Port XML Node Example

Ports are used to specify input and output ports of a network. These ports are used when a network is instantiated in another network. Through these ports, instances of the network can communicate with other network or actor instances in the higher layer network. The attribute kind of the Port can either be Input or Output. The node Type and its child nodes specify the data type of the port. The node tagged with Type has an attribute name whose value is the name of the data type. The size of the data type is specified in the attribute value of the child node Expr.

Listing 3.11 shows an example for nodes tagged with Connection.

```
<Connection dst="adder" dst-port="op1" src="or" src-port="result"/>
<Connection dst="serialize" dst-port="in" src="" src-port="bits"/>
```

Listing 3.11: Connection XML Node Example

A node tagged with Connection has the attributes dst and dst-port that state the destination of the connection and the attributes src and src-port that state the source of the connection. The attributes dst and src contain instance IDs. The attributes dst-port and src-port contain port IDs of the actor instances identified by src and dst. If the src or dst attribute is empty, the source or destination of this connection is the network itself. In this case, the src-port or dst-port has to be specified in this network by a Port. To one source port, multiple outgoing connections can be attached.

Code listing 3.12 shows a complete network specification.
This network performs the componentwise addition of two arrays. The actor instances Source_1 and Source_2 produce streams with the content of the two input arrays, the actor instance Add_array adds a token from each input stream and Sink prints the result to the screen.
4 Methodology

The aim of this thesis is to design a framework to generate SYCL or OpenCL based implementations from RVC-CAL actors and FNL network descriptions. This chapter discusses how and in which cases SYCL or OpenCL based code can be generated from actors, how scheduling in the actors and globally is done, and how the communication between the actors works.

Afterwards, the resulting MoC is summarised.

Although the focus of this work is to use SYCL or directly OpenCL for parallelisation, for the parallel execution of actor instances the common multithreading mechanism provided by the programming language C++ or the operating system is used.

The general idea of Dataflow Process Networks (DPNs) is to establish a continuous flow of tokens through the network what enables the parallel execution of ideally all actor instances. However, for parallel execution, it is more beneficial to process high numbers of tokens at the same time, e.g. by executing an action several times in parallel. For this purpose, tokens have to be buffered for some time to create larger chunks. For this reason, not all actor instances will be active at the same time because there is no constant token flow through the network. Only the actor instances that are processing a chunk have to be active. This behaviour can be exploited to fire actions multiple times in parallel without waiting until the tokens in the input buffers exceed a certain threshold to make parallel execution beneficial.

4.1 Generation of SYCL and OpenCL based Code

To generate SYCL or OpenCL based code, it is necessary that the actor fulfils certain criteria that enable the execution of actions multiple times in parallel. Executing actions several times in parallel matches with the data parallel programming approach of OpenCL. OpenCL executes a kernel multiple times in parallel, and each kernel instance is working on different data. Hence, there must be a concrete assignment of tokens to the kernel instances or work-items, e.g. a regular pattern for the memory access. If after each firing there has to be a new scheduling decision with a result
that is not predictable during code generation, it is not possible to assign the tokens to the kernel instances. If for example one action consumes five tokens and another action consumes three tokens from the same buffer, and it is not possible to determine the scheduling order in advance, parallel execution of these two actions is not possible. The tokens that are consumed by the different work-item are dependent on the previous firing sequence, e.g. the second work-item could either consume tokens starting with index 4 or with index 6 depending on whether the first work-item consumed 5 or 3 tokens. Thus, all work-items have to wait until the scheduling decisions of all previous work-items are made. This results in a rather sequential processing of the firings. Hence, if no concrete assignment of tokens to work-items is possible, the actor will not be parallelised in this approach.

There are two possible ways to parallelise actions: make a scheduling decision and execute the scheduled action as often in parallel as possible or let each work-item make its own scheduling decision and execute the associated action. The second option is only possible if all actions consume and produce the same number of tokens. Otherwise, there is no regular memory access pattern, and the scheduling decision of a kernel instance would influence other kernel instances.

This sounds like synchronous dataflow is a prerequisite for the generation of SYCL or OpenCL based code. But this criterion is too strong. Synchronous dataflow requires that all actions of the actor are consuming and producing the same number of tokens. However, only the parallelised sequence of action firings must show synchronous behaviour. But it does not matter whether all or just one action is part of this sequence, and a firing sequence of multiple times the same action always shows synchronous behaviour. If the same action is executed multiple times in parallel, all kernel instances are consuming and producing the same number of tokens. Thus, if the local scheduler can make a scheduling decision and the scheduled action can be repeated as often as enough tokens and free space are available OpenCL or SYCL based code can be generated. In this case, it does not matter if the actor is synchronous because a sequence of firings of the same action shows the same behaviour. However, this is only possible if the firing of an action does not influence the scheduling as it is the case with action schedules or guard conditions. If more tokens become available during the execution, this cannot influence the scheduling because the MoC does not have a timing and, therefore, it can be assumed that the scheduling decision is still valid for the whole sequence of firings of this action.

In RVC-CAL actors can contain a state consisting of state variables or an action schedule defined by a finite state machine. If the scheduling depends on state variables, an action can manipulate the state variables and influence the scheduling. The
worst case regarding predictability is that the state manipulation is dependent on the values of the consumed tokens. Thus, the scheduling cannot be known in advance.

The same is true for action schedules. Action schedules require a scheduling decision after each firing according to the state transition. After the state transition, a different set of actions might be enabled as in the previous state. The scheduling decision is dependent on all previous scheduling decisions because the scheduler has to know the current state. But the current state can only be known after all scheduling decisions of all previous work-items are known. It is assumed here that the work-items execute the action sequence according to their global ID, e.g. the work-item with the ID 1 executes the first action in the sequence and so on. Hence, the scheduling cannot be known at code generation time, too, and no regular memory access pattern that is beneficial for parallel execution can be derived. Thus, if an action schedule is defined in the actor, the actions of this actor cannot be parallelised.

Even if state variables do not influence the scheduling, they introduce another issue for parallelisation. If multiple instances of an action would be executed in parallel and the action manipulates the state variables, a work-item would have to wait until all previous work-items have manipulated the state variables. Only then it can know the state it has to deal with. Hence, the actions are executed rather sequential, and a parallelisation is not beneficial. There might be a case where an action does the writes first and then the rest of its computations. In this case, the actions could be parallelised because the next work-item only has to wait until the last write of a state variable of all previous work-items is done. If a read would happen directly before the write, they could also be merged into one atomic action. So far this would be possible because OpenCL supports atomics.

However, as mentioned in section 2.4, OpenCL uses a relaxed memory model, and memory consistency can only be guaranteed within the same work-group. Hence, OpenCL does not guarantee that atomics are read and written in the order of the global IDs of the work-items. Different work-groups can be scheduled on different compute units. This makes sequential writing of the state variables for a large number of work-item impossible because the global ID is used as the index to access the data or tokens in the input buffers. Thus, if an actor has state variables, the actions of this actor cannot be parallelised.

The last language construct concerning scheduling that has to be considered here are guard conditions. Guard conditions can either refer to state variables or the values of the consumed tokens. Actors with state variables are not parallelised, and therefore state variables need not be considered in this context. Hence, the scheduling of the actions depends on the consumed tokens. If the actions are consuming different
numbers of tokens, a work-item can only know which tokens to consume, if it can determine which tokens the previous work-item consumed. The same is true for the output buffers. If the actions are producing different numbers of tokens, a work-item can only know where to write after the previous work-item has done its scheduling and it is clear where the previous work-item will write.

However, if the actor is synchronous, multiple instances of the actions can be executed in parallel because a regular memory access pattern can be determined. The following subsection discusses this particular case.

Priorities also influence scheduling, but they only change the order in the local scheduler. This does not influence whether actions can be parallelised or not. If only one action is executed multiple times in parallel, the scheduling happens before the start of the parallel execution, and if multiple actions are parallelised in the same kernel, the priorities only influence the ordering of the if cases.

In summary, actors with either action schedules or state variables are not parallelised. If this is fulfilled, an action can be executed multiple times in parallel if it has no guard conditions. In this case, no scheduling decision has to be made in-between the firings because the scheduling only depends on the availability of tokens. This criterion can be checked for multiple firings in advance. Actions with guards can only be parallelised if all actions consume and produce the same number of tokens to ensure that a regular memory access pattern can be determined.

However, there is one exception regarding the action schedules and the parallelizability of actors that contain them. The next but one subsection discusses this exception.

4.1.1 Converting Actors with Actions with Guard Conditions

So far, all parallelizable actions can be executed without making any scheduling decision in the parallelised code. However, actions that have guard conditions are a special case. Neither a static schedule nor a dynamic scheduling decision in advance of the parallel execution is practicable. The first is not possible because, in this case, the scheduling depends on the values of the consumed tokens that are not known during the code generation. The second is not practicable because the scheduler has to go through all elements and create a scheduling sequence. Additionally, the scheduler has to determine for each work-item the indices for each SYCL or OpenCL buffer. This introduces a high overhead, and together with the overhead of OpenCL and the moving of the data, the parallel execution is not beneficial anymore. Hence, the only practical approach is to make the scheduling decisions also in parallel. As a result, in each parallel instance the guard condition has to be checked, and in order
for a regular memory access pattern, the actor has to be synchronous. If the actor is synchronous, each action consumes and produces the same number of tokens. Thus, a regular memory access pattern is maintained regardless of the executed actions. This way, the scheduling decisions that would otherwise be made by the local scheduler can be made in parallel, and the associated action can be executed without affecting other parallel instances.

Figure 4.1 shows how actions with guard conditions of an synchronous actor are parallelized. In this example the two actions, ActionA and ActionB with the corresponding guard conditions guard ConditionA and guard ConditionB are executed in parallel. Each work-item executes the first action whose guard condition evaluates to true.

4.1.2 Converting Actors with a unicyclic FSM

A similar issue arises from action schedules. The problem with action schedules is that after each firing a scheduling decision has to be made. If multiple work-items would execute the actions in parallel and in the order of their global ID, each work-item can only make a scheduling decision if the scheduling decision of the previous work-item is known.

This is only possible if in a particular state only one state transition is possible. In this case, it is possible to determine an execution order of the actions in advance. For example, the action schedule defined by the finite state machine in figure 4.2 starting with state $s_0$ is Action1,Action2,Action3,Action1,....

Figure 4.2: Unicyclic Finite State Machine Example
The actor that contains such a finite state machine is equivalent to cyclo-static dataflow. Concerning cyclo-static dataflow, this actor has three states. State $s_1$ consumes and produces as many tokens as Action2 does and so on. As described in section 2.1, a cyclo-static dataflow model can be reduced to a synchronous dataflow model by merging all states and summing up the numbers of consumed and produced tokens.

In the same way, an actor that has a finite state machine with precisely one cycle can be parallelised. Instead of executing each action in parallel, complete cycles are executed in parallel to achieve the synchronous behaviour.

If complete cycles are executed, a regular memory access pattern can be determined. Hence, it is possible for each work-item to determine which tokens it shall consume and where to write. Figure 4.3 illustrates this for the previous example. In this example, the parallelized cycle starts in state $s_0$. Each work-item executes the complete cycle, as it is shown in the box below each work-item.

![Figure 4.3: Parallelization of Action Schedules](image)

### 4.2 FIFO Buffer Design

The communication channels are First-In, First-Out buffers of fixed length. The maximum or fixed size of the buffers can be specified by the user or requirements of the network or by a default value. The tokens are stored in a linear array that is addressed in a cyclic way.

The buffers support one reader and one writer. However, in RVC-CAL many FIFO buffers can be attached to an output port. Thus, the buffer design also includes ports, both input ports and output ports. Like in RVC-CAL, the ports are connecting the actors to the buffers. An input port reads the data that is demanded by its actor from the attached buffer. To an input port, only one buffer should be attached. Everything else indicates an error. An output port distributes the tokens that are emitted from the actor to all attached buffers. Every attached buffer gets a copy of the token.

Figure 4.4 illustrates the communication between two actors.
4.2 FIFO Buffer Design

The actor passes its output tokens on to the port, and the port forwards the tokens to the FIFO buffer. The FIFO buffer is illustrated by the cells of the array that is used to store the tokens, and the dashed line from the last to the first cell indicates the cyclic addressing.

A buffer shall be readable and writable at the same time because its producer and consumer can be active at the same time. Thus, the buffer must be able to handle a read and a write at the same time without resulting in a corrupted state. For this purpose, a semaphore or lock could be used. Both introduce some overhead for locking, waiting and unlocking and, therefore, reduce the performance. To avoid locks and race conditions, the consumer and producer must write different state variables of the buffer. Race conditions, however, can be avoided easily because a cyclic buffer needs a pointer to the location in the array where to read the next element, called read pointer in the following, and one pointer where to store the next element, called write pointer in the following. The reader only manipulates the read pointer by adding one and calculating modulo the maximum buffer size after a read has happened. The writer only manipulates the write pointer. After writing the new token to the memory location the write pointer is pointing to, the writer increments the write pointer and calculates it modulo the maximum buffer size. An access to the buffer must happen in the following way: First, the element is read or written, then the corresponding pointer is updated. In the case of reading, this order ensures that the element is not overwritten in-between the update of the read pointer and the access to the array. If the order would be the other way around, the writer could write an element to the place where the read is happening concurrently. In the case of writing, the reader could read an element that has not been written yet.

However, there exists one case where this simple accessing scheme breaks: When all fields of the array are filled with valid tokens, the read and write pointer point to the same memory location. In this case, it is not decidable whether the buffer is full or empty. Hence, a flag is used to indicate whether the buffer is full or not. If after

![Figure 4.4: Communication between two Actors](image-url)
an update of the write pointer both pointers are equal this flag is set. Unfortunately, the consumer and producer have to write this flag. But when the reader resets the flag, the writer cannot be active because until the point where the flag is reset the writer sees a full buffer.

For reading the buffer has to provide functions to read one element and to read multiple elements in case a repeat value is specified and the same for writing. Additionally, the consumer must be able to find out how much tokens are stored in the buffer because, as mentioned in subsection 2.2.1, (RVC-)CAL actors can be non-deterministic. Hence, the FIFO buffers do not implement a blocking read because it is circumvented by the consumers anyhow. Due to the limited buffer size and the, therefore, introduced additional scheduling condition, the producer must be able to determine how much free space in the buffer is left. The size and the free space is calculated by the difference of the write and the read pointer modulo the maximum buffer size except the full flag is set. As RVC-CAL allows the scheduling of actions based on the values of the potentially consumed tokens the buffer also has to provide the functionality to read tokens without deleting them or changing the state of the buffer.

The ports reflect this interface to the actors.

### 4.2.1 FIFO Buffer Access in SYCL and OpenCL

Ports also manage the creation of SYCL and OpenCL buffers. These buffers are created on demand. This means, if an actor wants to execute actions in OpenCL, it requests the ports for the SYCL or OpenCL buffers containing the input data or empty buffers in case of output ports. When requesting a SYCL or OpenCL buffer, the actor has to specify how much elements are consumed from this FIFO buffer or produced for this FIFO buffer. According to the given number, new SYCL or OpenCL buffers are created with the required size. SYCL or OpenCL buffers that are created in the input ports are directly filled with the data from the FIFO. The tokens are placed in the SYCL or OpenCL buffers in the order they are consumed. Hence, no modulo operation is required when accessing the SYCL or OpenCL buffers. The read pointer is updated in the same step. Thus, after the SYCL or OpenCL buffer creation, the data is no more in the FIFO buffer. After the SYCL or OpenCL call is finished, the SYCL or OpenCL buffers containing input data are deleted. The content of a SYCL or OpenCL buffer containing output data is first inserted into the associated FIFO buffers that are attached to the port, and then this buffer also is deleted.

It would also be possible to create SYCL or OpenCL buffers that are as large as
the maximum FIFO size and keep them alive throughout the whole execution. In this case, the same memory location could be used for the SYCL or OpenCL buffer on the host and the FIFO.

However, then the regular memory access patterns described at the beginning of this chapter cannot be utilised to the full extent. Nevertheless, regular memory access patterns are very relevant for performance, especially on GPUs [44]. As the most performance gains are expected by also utilising the GPU, this approach makes use of regular memory access patterns. If the data would be copied into a SYCL or OpenCL buffer without aligning it to the start of the OpenCL buffer, a memory access pattern like in figure 4.5 would be achieved due to the cyclic buffer behaviour. But this memory access pattern relies on the modulo operation and an offset and,

![Figure 4.5: Irregular Memory Access Pattern Example](image)

therefore, most GPUs access this memory in an inefficient way [45]. Often multiple memory transactions per work-item are necessary or no vectorization is possible. In the worst case, a work-item has to consume a token placed all right in the buffer, and one placed all left in the buffer. If the tokens, however, are ordered in the way they are consumed, as in figure 4.6, based on the regular memory access pattern an aligned memory access or vectorisation is possible. This results in a much higher performance compared to a memory access with offsets, modulo operations and irregular patterns.

![Figure 4.6: Regular Memory Access Pattern Example](image)

Additionally, since the SYCL or OpenCL buffers are created with the adjusted
size, only data that is required for the computation is transferred to the device. This reduces the communication overhead and increases the performance slightly.

However, this approach also leads to additional memory copying each time a SYCL or OpenCL buffer for input tokens is created and before one for output tokens is destroyed. This memory copying can be done efficiently with memcpy calls. In both cases at maximum two blocks of memory need to be copied. In the worst case the circular buffer is organised in a way that in the first part is the tail of the buffer, then comes an empty part, and then the head of the buffer. In this case, first, the head is copied to the SYCL or OpenCL buffer and then the tail. If output tokens are copied to the FIFO buffer, in the worst case, the valid tokens are in the middle of the array, and the copying works similarly to copying during the creation of a read buffer.

4.3 Scheduling

Scheduling is divided into two parts, global scheduling that decides when which actor instance is scheduled and local scheduling that decides which actions of the scheduled actor instance are executed. The global scheduler is a scheduling routine that is called initially by each thread and when an actor instance terminates. This routine enables each thread to choose a new actor instance that is then executed in this thread. Thus, global scheduling is distributed over all running threads and not a central instance. The only scheduling criterion that is considered by the global scheduler is whether the actor instance is running or not. An actor instance that is running cannot be started again by another thread. The global scheduler, however, does not check whether tokens are available because this is done anyhow by the local scheduler of the called actor instance. The overhead for calling the actor instance’s local scheduler is rather low and would be caused anyhow by the additional if statement and by accessing the FIFO buffers to read their size. The global scheduler runs in an infinite loop over all actor instances in the network. When an actor instance is scheduled and eventually terminates, the global scheduler of this thread continues with the next actor instance in the list and not at the beginning of the list. The list of actor instances is sorted according to the token flow in the network, e.g. a producer will always be scheduled before the consumer of its tokens is scheduled.

When a local scheduling routine is called, it tests the actions that are potentially schedulable in the current state according to their priorities. If no action schedule or finite state machine is defined, all actions are potentially schedulable. Actions with a higher priority are tested first. If no priorities are defined, the actions are
tested in the order in which they are defined in the RVC-CAL code. The first action that is schedulable is executed. To find out whether an action is schedulable, the local scheduler tests if there are enough tokens available to execute this action and whether the guard conditions are fulfilled. If not, the scheduling criterion of the next action is tested. If no other action can be tested, the local scheduler terminates, and the global scheduler selects the next actor instance. If there are enough tokens available and the guard conditions are fulfilled, the local scheduler then tests whether there is enough free space in the output buffers. If there is enough free space in the output buffers, the action is executed. If not, the local scheduler terminates. It would also be possible to test the other actions for schedulability, but this would break the dataflow model of computation because there it is assumed that buffers are of infinite size. Thus, whether free space is available in the output buffers or not should not influence the scheduling because this can never occur in the original model. Hence, the local scheduler terminates to give the consumer of the full buffer the possibility to consume the tokens and create more free space.

After an action has been executed, the local scheduler adjusts the finite state machine state of the actor, if necessary, and selects the next action. This procedure is repeated until no action is schedulable or not enough free space is available in one of the output buffers, as discussed before.

However, if the generated code of an actor is based on OpenCL or SYCL, scheduling works slightly different. If the actions have no guard conditions and no action schedule is defined, the local scheduler tests if enough tokens and free space is available to execute the action once, as described above. Afterwards, for each input FIFO buffer, the number of available tokens is divided by the number of consumed tokens. The same is done for each output FIFO buffer regarding the available free space. The minimum of all of these calculations determines how often the action is executed in parallel.

If the actions have guard conditions and the actor is synchronous the final scheduling decision is made in the SYCL or OpenCL code. To decide whether the kernel shall be enqueued for execution, the local scheduler tests if enough tokens and free space are available. If not the local scheduler terminates. As already explained in subsection 4.1.1 the guard conditions are validated in the kernel. The execution of the actions works as before. The number of parallel instances is calculated, and the SYCL or OpenCL based execution is started.

If an action schedule is defined by a finite state machine that contains exactly one cycle, complete cycles are executed in parallel. To avoid a larger number of kernels and reduce the complexity of code generation, the parallel execution will always start
in the same state. In this case, only one kernel is needed, instead of one for each state where the cycle can start. But not always complete cycles have to be executed, or the previous firings might have left the actor in a wrong state for parallel execution. Hence, first of all, actions are fired without using SYCL or OpenCL until the initial state of the parallelised cycle is reached. At most a complete cycle is executed. The number of parallel instances is calculated by dividing the number of available tokens by the number of consumed tokens in a complete cycle, and the same for produced tokens as before.

The non-deterministic property of RVC-CAL is fulfilled in the sense of the ability of the local scheduler to read the size of the input buffers and make a scheduling decision according to them. But not in the sense that anyone is chosen randomly when several actions are schedulable.

4.4 Model of Computation

According to the CAL specification [2] the Model of Computation (MoC) is defined by the data flow (communication) and the control flow among the actors (scheduling). However, the usage of SYCL or OpenCL in this approach also introduces some way of synchronisation between actor instances in addition to the fixed size FIFO buffers.

The original model was supposed to work asynchronously without any synchronisation between actors. But the processing of large chunks of data or tokens disables the continuous token flow that would otherwise be possible. Instead, it is a flow of big chunks of tokens mainly limited by the size of the buffers. The parallel execution of the actions of an actor instance is only started when the output of the previous SYCL or OpenCL call of this actor instance is written back to the FIFO buffers. Thus, the instances of SYCL or OpenCL based actors synchronise themselves because a big chunk of data is made available instantaneously and the actor instance is inactive otherwise.

Apart from that, if more than one work-group is used, it is not possible without synchronisation because OpenCL cannot guarantee a specific execution order. The work-groups might be executed in any order. The synchronisation can either be done as in this approach or by using one SYCL or OpenCL buffer in-between two adjacent actor instances. The latter also requires an offset and modulo operations and OpenCL synchronisation methods to share the FIFO state between the consumer and the producer.

However, large chunks of data are required to utilise parallelism. Initially, they are produced because the processing of tokens is much faster than the starting of the
new threads and their global scheduling. Thus, the initial producer of the tokens by, for example, reading them from a file most likely fills its output buffers before the consumers of these tokens starts running. Then the actor instances can synchronise on the fixed buffer size. Additionally, instances of a SYCL or OpenCL based actors are only scheduled the next time after the previous SYCL or OpenCL call is finished. In this time also tokens are buffered. Hence, wait conditions in local schedulers of SYCL or OpenCL based actor implementations would not work properly because waiting for more tokens to arrive would take more time than the execution with fewer tokens would take.

The actor instances are executed by a fixed number of threads that are launched after the initialisation of the program. Due to the large chunks that are processed only a limited number of actor instances are active at the same time. Thus, a small number of threads is sufficient for the execution of the actor instances. Ideally, the number of threads is equal to the number of parallel threads that are supported by the hardware. In this case, this approach does not require any context switching because no central global scheduler launches new threads.

The communication and scheduling were discussed in detail in the previous sections.
5 Development of the Code Generator

Based on the methodology discussed in the previous chapter this chapter provides details concerning the implementation of the code generator and the overall design of the generated C++ code. The first section discusses the parsing of network files. The next section describes how imports are treated by the code generator, especially how calls to native functions and the integration of the corresponding source files are handled. Then the actual code generation is discussed. This section describes the design of the generated code and discusses some issues regarding code generation, especially when generating SYCL based code. The following two sections cover the code generation of the buffer and port class and the main function including the global scheduler. The last section discusses the conversion of RVC-CAL data types to C++ data types.

5.1 Parsing the Network Files

The networks are specified with the Functional unit Network Language (FNL), as discussed in section 3.8. FNL is an XML dialect and, therefore, an XML parser can easily parse the network files. As already mentioned in the RVC-CAL introduction the top node, representing the complete network, can have different kinds of child nodes: Instance, Connection, Port, and Decl. Decl was not introduced in the RVC-CAL introduction because it is not relevant for the overall functionality of the language. Decl is used to declare constants. The identifier of the declaration is stored in the attribute name, the value is stored in the attribute value of a child node called Expr. These declarations can, for example, be used to specify parameter values.

During code generation, the network is stored as a set of connections (dstID, port, srcID, port) and actor instances (ID, path, parameters). But a network can also contain other networks. However, in FNL there is no differentiation between instances referring to actors or networks. Whether it is a network or an actor has
to be tested by reading the file ending of the file referred by the class attribute. Networks have a .xdf file ending, and the file ending of actors is .cal. These network instances are not considered in this representation, just like ports. When a network instance is found, the corresponding network is also parsed. The actor instances in the subnetwork get the ID of the network instance node in the higher layer network as a prefix. The same is done with the IDs in the connections. For example, if the network instance has the ID tx_inst in the higher layer network, every ID in the instantiated network gets the prefix tx_inst. Adding the network instance ID as a prefix is necessary because the same IDs could be used in both networks, then the actor instances cannot be identified correctly, and the connections cannot be established appropriately. Connections from or to a network port also get this prefix. Additionally, they are marked for further processing. These connections can easily be detected because connections to or from ports do not have a src or dst ID. For this purpose, the connection representation contains two flags, one to indicate whether the source is a port and the other one to indicate whether the destination is a port.

However, connections from or to a network port have to be removed because ports are not present in the representation that is used to generate code for the network. For this purpose, connections are merged. Two connections A and B are merged if the flag that indicates that the source is a network port is set in connection A and the flag that indicates that the destination port is a network port is set in connection B. Furthermore, the source ID and port of connection A must be equal to the destination ID and port of connection B. The resulting connection contains the source ID and port of connection B and the destination ID and port of connection A and the corresponding network port flags. This procedure is repeated until no network port flag is set in any connection.

Figure 5.1 shows an example how two connections are merged. Part 5.1a of the figure shows the two connections that are merged. These two connections can be
merged because the source of the left one is the port named input of the network instance tx_inst. The src flag indicates that tx_inst is a network instance. The same applies to the destination of the connection on the right-hand side of figure 5.1a. Part 5.1b of the figure shows the result of the merging. The result is the destination of the connection on the left-hand side and the source of the connection on the right-hand side of part 5.1a, along with their corresponding flags.

The result is a completely flat model for which code can be generated more easily than for a model with sub-models.

### 5.2 Import Handling

RVC-CAL provides the ability to define functions, procedures, and constants in a separate unit. These units can be imported into other units or actors. However, RVC-CAL has the ability to only import parts of the unit by specifying explicitly the function, procedure or constant that shall be imported. Unfortunately, C++ does not have this ability. Thus, converting the unit files into a header file is not appropriate. The programmer might implement a function that is also defined in an imported unit file, but not imported. If the complete file were imported anyway, this would lead to conflicts in the generated C++ code because now there are two definitions of the same function although there is no conflict in the original RVC-CAL code. Thus, only the requested parts of the unit files are copied into the generated class.

#### 5.2.1 Native Function Calls

Besides the unit files that are part of the RVC-CAL code, there is also native code written in C. RVC-CAL does not manage the imports for the native code files. They have to be set manually by the programmer in the generated code. For this purpose, into every file that is generated from an RVC-CAL actor that calls native functions, an extern ”C” block is inserted at the beginning. In this block, the C headers containing the corresponding function declarations have to be included.

However, the native code that originates from ORCC projects does not have headers. Thus, the code generator provides a rather experimental functionality for generating a header and a source file from a C source file. The header file is generated by removing the variables and constants, except those with the extern keyword, and the function bodies. The source files are generated by removing extern variables and the includes and adding the include statement of its header. However, this functionality has not the ability to know whether the included files are ORCC related and
therefore can be removed or not. Hence, it is advisable to check the generated code. 
The code generator also inserts all generated headers into every extern ”C” block.

Since the reference point is ORCC and ORCC projects are used as benchmarks, 
the corresponding native code also has to be used. Thus, the generated code has to 
provide partly the same environment than ORCC. The native code often relies on 
data stored in a struct called options_t. Hence, the generated code also provides an 
options header that contains variables like the path to input and output files. Add-
ditionally, the main contains code to parse the corresponding command line inputs. 
This options header is included in every header that is generated from native code. 
The remaining part of the ORCC environment is not taken over and, therefore, the 
 corresponding include-statements in the native code have to be removed.

5.3 Code Generation for Actors

As described in the section about the parsing of the network, for each actor instance 
a data structure consisting of its ID, the path to the actor, and its parameters is 
created. For each actor, but not for each instance, a C++ class with the complete 
functionality of the RVC-CAL code is generated. However, parameterised RVC-CAL 
actor instances are an exception. It would be possible to parameterise each instance 
differently by the constructor, but then they are no constant class members. This 
would make the generation of SYCL based code more complicated, as discussed in 
subsection 5.3.2. Instead, for each parametrisation, a new class is generated. The 
name of the class is the name of the actor followed by a Dollar sign followed by the 
instance ID. The Dollar sign is used as a separator because this symbol is forbidden 
in CAL but not in C++. Thus, the concatenation of two identifiers separated by a 
Dollar sign cannot introduce naming conflicts. The Dollar sign as a separator is also 
used when action tags and action names are concatenated to get the C++ function 
name.

5.3.1 Structure of a generated Actor

The general structure of a generated C++ class is as shown in listing 5.1.

```cpp
#pragma once
#include "Port.hpp"

class Actor_name {
private:
    //Imports
```
//Constant Actor Parameters

//Variables, Constants, Functions

class states{
  //states
} states state = states::initial_state;

//Ports

//Actions

public:
  Actor_name(Ports):Port initializer list{
    initialize();
  }

void schedule(){
  for(;;){
    //...scheduling decision making
  }
}

Listing 5.1: The General Structure of a generated C++ Actor

The content of the class is arranged in the order the code is generated. For this reason, buffering of RVC-CAL code is necessary because the parts are not in the same order than in an RVC-CAL actor, e.g. in RVC-CAL, the ports are actually at the beginning of the actor definition. But the ports can only be converted to C++ when their data types can be determined. As already mentioned, the data types in RVC-CAL can have different sizes. This might result in a different type in C++, e.g. an int of size 8 is a char in C++. However, the sizes can be defined by constants that are defined in the actor’s body, even for the data types of the ports and actor parameters. Thus, the port definitions and actor parameters are buffered and are converted when all constants, actor state variables and functions or procedures have been converted.

Action definitions are buffered, too. Not because the size of the data types might not be known at this point but rather because it is not clear whether OpenCL/SYCL
based code or pure C++ code has to be generated. As discussed in the previous chapter, this depends on state variables, the finite state machine and guard conditions. But the finite state machine is often defined at the end of the actor. Hence, the conversion of the actions to C++ code has to be delayed.

5.3.2 Code Generation for SYCL

If SYCL based code is generated there are only small differences compared to pure C++ code generation. The index space and the SYCL buffer accessors must be created before the kernel is executed. The kernel is similar to a standard C++ function generated from an action. However, the accesses to ports have to be replaced by accesses to SYCL buffers by SYCL accessors. To simplify code generation, the ports manage the SYCL buffers.

In contrast to the pure C++ code generation, for each SYCL buffer, an index by which this particular work-item accesses the SYCL buffer is created before any other code is executed. The index is calculated by multiplying the global index of the work-item with the number of tokens that are consumed from or produced for this particular buffer. Every time a SYCL buffer is accessed with its corresponding index, this index is incremented by one afterwards. Then the inputs are assigned to variables as specified in the InputPattern of the action, see section 3.5. After that, the code generated from the action body is executed. At last, the results are written to the output SYCL buffers as specified in the OutputPattern of the action. The incrementing of the indexes ensures that the tokens are read in the correct order and that the output tokens are written to the correct location in the buffers. The different parts a kernel consists of is illustrated in figure 5.2. This structure is the same for both SYCL and OpenCL based code.

The SYCL queue to which the kernels are enqueued is created in the main file as a static variable. This variable is declared static because their destructor is called when the program exits. Due to the reason that DPNs do not terminate, the program is usually terminated by exit calls in the native code. Using a static variable guarantees that in this case all SYCL objects are cleaned up properly and the SYCL runtime does not complain about it. A reference to the queue is handed over to the local scheduler of an actor instance that is based on SYCL each time the local scheduler is called. The local scheduler then passes the reference on to the function that carries out the action.

After the kernel has been enqueued, the event that indicates the execution status of the SYCL call is returned to the global scheduler. The global scheduler will not schedule this actor before the SYCL call is finished. But before scheduling the actor
instance again, the global scheduler first notifies the ports of the actor instance about the finished SYCL call. Then the ports can read back the produced output, insert it into the FIFO buffers and destroy the SYCL buffers. This way meanwhile the SYCL call is executed this thread can execute another actor instance. This is possible because in SYCL the enqueueing of kernels is not blocking. If it were blocking, the enqueue call would return after the execution of the kernel is finished.

However, although SYCL claims to simplify the access to OpenCL by embedding the code into the host code, SYCL has some shortcomings that make the generation of SYCL based code more complicated again. (At least for the current ComputeCpp implementation 1.0.0) Constants defined outside of the kernel can be used in the kernel without creating a SYCL buffer for them. But not if the constants are defined in a class. The same is true for functions that are called in a SYCL kernel. Thus, the functions and constants must be defined outside of the class. Usually, they are defined in the source file. This would not cause any conflicts like multiple declarations of the same function or constant. In this case, the kernel has to be defined in the source file as well. However, SYCL does not support this either. If the implementation is split up into a header and a source with the kernel being defined in the source file enqueueing of the kernel seems to take forever. The function never returns. Thus, the functions and procedures and the implementations of the class member functions
are also defined in the header. To avoid conflicts, a namespace with the same name than the class is put around the code generated from this actor.

As SYCL seems to be rather slow due to the overhead of its runtime, for experimental use the code generator is also able to generate code that uses a small number of work-groups with one work-item in each. This is mainly meant to target the CPU because it often only has a small number of cores. Every work-group only contains one work-item to distribute the computations among the cores without context switching. This occasionally improves the performance of SYCL.

5.3.3 Code Generation for OpenCL

The generation of code that utilises OpenCL is more complicated because the kernels cannot be embedded into the host code. Instead, they have to be written in OpenCL C. Additionally, the initialisation and kernel creation, in this case, has to be done by the generated code. Thus, much more code has to be generated. Fortunately, most of the code can be generated statically like functions to setup OpenCL, create kernels, and execute them.

In contrast to the SYCL based version, in the OpenCL based version, each actor instance that utilises OpenCL has its own OpenCL context and command queue. This is done to avoid passing on references of all OpenCL specific objects through the whole program and does not influence the performance. In SYCL they are managed by the runtime and, therefore, hidden from the programmer. In this version, the constructor creates all OpenCL specific objects of an actor instance. The OpenCL based version uses the same approach for asynchronous execution of the OpenCL kernels than the SYCL based version. The host code of both is also similar. The only difference is that, besides the programming overhead to assign OpenCL buffers to the kernel arguments and the initialisation of OpenCL, OpenCL C code has to be generated. Since the language constructs of RVC-CAL are nearly equal to the basic language constructs of C and they are also part of C++, only little of the code generator has to be adapted to the new requirement. Nearly no language constructs of C++ that are not part of C are used in the conversion of functions, procedures, actions and constants. The main difference is that the actions have to be converted to kernels by adding the keyword `kernel` and the parameters, one for each accessed buffer. The functions can remain the same because they can also be used in OpenCL C. But the generated kernels, functions and constants have to be copied to a separate OpenCL C file.

Unfortunately, constants that are defined in the actor scope or after the conversion to C++ in class scope are after they are copied to an OpenCL C file in file scope.
However, this is not allowed in OpenCL C. Thus constants have to be copied into
the kernels and functions that use them.

### 5.3.4 Local Scheduler Generation

Local scheduling works in the following way: First, the local scheduler traps into
the corresponding code block that is responsible for the scheduling in the current
state. Then the local scheduler tests for each action that can be scheduled in this
state whether its guard conditions are fulfilled and the required tokens are available.
It does so according to their priorities. For the first action whose guard conditions
are fulfilled and the required tokens are available, the local scheduler tests if there is
enough space in the output FIFO buffers to execute this action. If so, this action is
executed, and the state is transitioned according to the finite state machine. Then
the local scheduler proceeds at the beginning. If not enough free space is available,
the local scheduler terminates. This procedure is repeated until no schedulable action
is found. In this case, the local scheduler also terminates.

Thus, for the generation of the local scheduler, in particular, the priority construct
and the finite state machine are essential. Priority definitions basically look like
this: \textit{Action1} > \textit{Action2} > \textit{Action3}. This expression is converted into tuples of
higher priority and lower priority pairs, e.g. (\textit{Action1},\textit{Action2}), (\textit{Action1},\textit{Action3})
and (\textit{Action2},\textit{Action3}). The transitive closure is also inserted there because it makes
sorting of a list of action IDs according to their priorities much easier. The relation
between two actions can then be determined by iterating through all tuples.

Regarding the action schedules, besides the enum for the states and the current
state variable, for each action firing that causes a state transition, the code generator
creates a tuple (current state, action ID, next state). With these tuples, all actions
that are executable in a particular state can be determined, and the local scheduler
can set the new state accordingly.

The basic structure of a local scheduler looks like in figure 5.2.

```plaintext
void schedule(){
    for(;;){
        if(state == states::state1){
            if(port1.get_size() > 1 && guard condition){
                if(port2.get_free_space() > 3){
                    action1();
                    state = states::state2;
                } else{
                    break;
                }
            } else{
                break;
            }
        }
    }
}
```
This example shows how local scheduling works when neither SYCL nor OpenCL is used. First, the local scheduler checks the current state, then all actions that can be scheduled in this state are checked in the order of their priority. The order of two actions that have no priority relation stays according to the order they are defined in the RVC-CAL actor. If no action schedule is defined, the first step is omitted.

For SYCL or OpenCL based implementations, the infinite for-loop is omitted because the actions are fired in parallel in one step. Thus, no further scheduling decision is required. Additionally, this ensures that the SYCL or OpenCL call is finished and the tokens are copied into the FIFO buffers in the right order before the next SYCL or OpenCL call is made. Otherwise multiple SYCL or OpenCL buffers have to be managed by the ports.

### 5.4 Port and FIFO Buffer Code Generation

Ports and FIFO buffers are the two parts of the generated code that are responsible for the communication between actors. However, actors only deal with ports. The FIFO buffer in-between two ports is not visible for the actor. The actor reads tokens from the port and transfers output tokens to the port. The port only reflects the interface of the FIFO buffer for reading, writing and viewing. Viewing means reading without erasing the element from the buffer. This is required to check guard conditions. When a port is created, it gets a list of FIFO buffer references of all FIFO buffers that are attached to it. It is assumed that to a port that is meant for reading only one FIFO buffer is attached. Multiple FIFO buffers can only be attached to output ports. In this case, the port distributes a copy of the inserted tokens to each FIFO buffer. Additionally, the port is responsible for the creation
5.5 Global Scheduler Generation

The global scheduler uses the round-robin method to schedule the actor instances. Every time a local scheduler terminates the control flow of this thread returns to the global scheduling routine. This routine executes the next actor instance in the list that is not running. Since multiple threads are executing actor instances, the flag that indicates whether an actor instance is running has to be atomic. Otherwise, more than one thread could execute the same actor instance. If the implementation of this actor is based on SYCL or OpenCL, an additional execution criterion is that the SYCL or OpenCL call of the previous run of the actor instance has to be finished. In the case that the implementation is SYCL based, the global scheduling routine also triggers the deletion of the SYCL buffers and the copying of the data from the SYCL buffers to the output FIFO buffers. In the OpenCL based version, this is done by the local scheduler. As discussed in the methodology the list of the actor

and deletion of SYCL and OpenCL buffers and to copy data between the FIFO buffers and the SYCL and OpenCL buffers. For this purpose, ports have access to the token storage of their attached FIFO buffers. This way, for performance reasons, memcpy can be used to copy the data, instead of using the FIFO buffer interface. By default, the code creates the OpenCL buffers with the memory flag `CL_MEM_ALLOC_HOST_PTR` because this is the safest and most portable way to create OpenCL buffers. With this memory flag, the OpenCL runtime allocates the memory as it is appropriate for the device. SYCL buffers are created without passing in a pointer to already allocated memory. Thus, the SYCL runtime will internally allocate the required memory.

There are no dedicated implementations of FIFO buffers between two actors that both are SYCL or OpenCL based or between an actor that is SYCL or OpenCL based and one that is not. The SYCL and OpenCL buffers are only created on demand. Thus, while generating code, no differentiation between different versions of FIFO buffer implementations has to be made. This makes code generation a lot easier.

The core of the FIFO buffers is a cyclically addressed array, as described in section 4.2. However, this property makes the calculation of the size more complicated. The size is calculated by the following formula: $size = (N + (state.write_index - state.read_index) \% N) \% N$. $N$ denotes the maximum size of the buffer. This formula looks this complicated because in C++ the sign of the result of the modulo operation is dependent on the implementation. Nevertheless, this way, it always works.
instances that are processed in a round-robin fashion by the global scheduler is sorted according to their positions in the network. The sorting is done the following way: First, a list of all actor instance IDs is taken, while iterating through all connections, from this list every ID is removed that is the destination of a connection. After this step, the list only contains IDs of actor instances that are initially producing tokens. Then all IDs of actor instances are inserted to this list that are consuming those tokens. In the last step, all IDs of actor instances that are consuming tokens of the ones that are already on the list are inserted to this list. The last step is repeated until all actor instance IDs are in the list. The list is then ordered basically in the way the tokens are flowing through the network.

5.6 Conversion of Data Types

As RVC-CAL provides the ability to specify the size of data types, it is necessary to translate them to C++ data types. Just using the specified data type and ignore its size is not appropriate because the specified size might be higher than the size of the corresponding C++ data type or the programmer might use overflow or underflow behaviour on purpose. However, the size of RVC-CAL data types can be any number, but C++ only provides data types with a size that is a multiple of a byte. Thus, an RVC-CAL data type is converted to the next larger C++ data type, e.g. an int of size nine is converted to a short and an int of size 8 to a char. Unfortunately, this slightly changes the expressiveness, but the use of bit fields might decrease the performance.
6 A Comprehensive Example

This chapter demonstrates the functionality of the code generator using the ZigBee project from the ORCC samples repository. For this purpose, RVC-CAL code is compared with the C++ code that is generated from the former. However, this project does neither contain an actor with parallelizable actions with guard conditions nor an actor with a unicyclic finite state machine. For this reason, this chapter shows an additional example for both cases.

6.1 Conversion of a Complete Network

The network of the ZigBee Multitoken sample is shown in figure 6.1 and figure 6.2. Figure 6.1 shows the so called top network. This is a network that has no ports and, therefore, is not meant for further instantiation. This network connects the actual ZigBee Multitoken implementation with a source that reads input data from a file and a sink that writes the output to a file again. Figure 6.2 shows the network...

![Diagram of the top network](image1)

Figure 6.1: Top Network of ZigBee Multitoken

![Diagram of the network](image2)

Figure 6.2: Network of ZigBee Multitoken
that actually implements the ZigBee Multitoken transmitter. SYCL or OpenCL can accelerate the actor instances chipMapper_inst and qpskMod_inst. However, they cannot accelerate the actor instances source_inst and sink_inst because both call native functions to access files and contain state variables, the actor instances headerAdd_inst and pulseShape_inst contain finite state machines and state variables.

Listing 6.1 shows an extract from the global scheduling routine for this network.

```c++
if(!source_inst_running.test_and_set()){
    source_inst->schedule();
    source_inst_running.clear();
}
if(!tx_inst_headerAdd_inst_running.test_and_set()){
    tx_inst_headerAdd_inst->schedule();
    tx_inst_headerAdd_inst_running.clear();
}
if(tx_inst_chipMapper_inst_status.get_info<info::command::execution_status>()
   == info::command::status::complete
   && !tx_inst_chipMapper_inst_running.test_and_set())
{
    if(update_buffer.tx_inst_chipMapper_inst){
        tx_inst_chipMapper_inst_data$FIFO.sycl_read_done();
        tx_inst_chipMapper_inst_chip$FIFO.sycl_write_done();
        update_buffer.tx_inst_chipMapper_inst = false;
    }
    tx_inst_chipMapper_inst_status = tx_inst_chipMapper_inst->schedule(queue1,
                      update_buffer.tx_inst_chipMapper_inst);
    tx_inst_chipMapper_inst_running.clear();
}
```

Listing 6.1: Extract from the generated Global Scheduler for ZigBee Multitoken

This listing shows how the scheduling decision regarding the actor instances source_inst, tx_inst_headerAdd_inst, and tx_inst_chipMapper_inst are made. For each actor instance, an atomic flag is created that indicates whether the actor is running or not. The identifiers of the atomic flag variables are their corresponding actor instance ID and the postfix _running. Furthermore, the actor instance ID tx_inst_headerAdd_inst shows how the IDs are built-up. The prefix tx_inst is the ID of the ZigBee network instance in the Top network. Thus, all actor instance IDs in the instantiated network get this prefix. The remaining part is their actual ID.

In this example, the actor chipMapper is converted to SYCL or OpenCL based code. For this reason, the scheduling decision regarding the instance of this actor is also made based on a variable with the postfix _status. This variable is a SYCL event that indicates whether the last SYCL call by this actor instance is completed. If it is completed, the actor instance can be scheduled again. But before it is scheduled again the SYCL buffers of the previous SYCL call are destroyed, and
the output data is written back to the corresponding FIFO buffers by calling the functions sycl_read_done and sycl_write_done of the ports that are managing these SYCL buffers. The variable with the prefix update_buffer keeps track of whether the buffers must be updated before the actor instance is scheduled again. In the OpenCL based version, the global scheduler also keeps track of the execution status of the OpenCL calls, but the local schedulers initialise the updating of the buffers. A local scheduler of an OpenCL based actor is shown later.

An actor in this example that contains basically every part an actor can be composed of is the headerAdd actor. Code listing 6.2 shows the code of this actor.

```plaintext
import common tx.constants.*;
actor headerAdd()
uint(size=PL_DATA_SZ) pl_bits
  ==>
uint(size=DATA_SZ) data, uint(size=PAYLOAD_LEN_SZ) len :

  uint(size=PAYLOAD_LEN_SZ) octet_count;
  uint(size=PAYLOAD_LEN_SZ) octet_index;
  int HEADER_LEN = 5;
  List(type: int(size=DATA_SZ), size=HEADER_LEN) Header = [0, 0, 0, 0, 167];

  initialize ==>
  do
    octet_count := 0;
  end

  get_data_len: action pl_bits: [ bits_in ] ==> len: [ octet_count ]
  do
    octet_index := 0;
    octet_count := bits_in + HEADER_LEN + 1;
  end

  send_header: action ==> data: [ data_out ]
  guard
    octet_index < HEADER_LEN
  var
    uint(size=DATA_SZ) data_out
  do
    data_out := Header[octet_index];
    octet_index := octet_index + 1;
  end

  send_length: action ==> data: [ octet_count − HEADER_LEN − 1 ]
  do
    octet_index := octet_index + 1;
  end

  send_payload_octet: action pl_bits: [ bits_in ] ==> data: [ bits_in ]
```

73
This actor adds the ZigBee header to the packet and then forwards the data as long until the packet size is reached. Then it does a reset and starts again. The actor contains an action schedule and state variables. Hence, no SYCL or OpenCL based implementation can be generated for this actor.

However, this actor also contains priorities and an initialisation action. Thus, with this example, most of the basic functionality of the code generator can be demonstrated. Listing 6.3 shows the C++ code that is generated from this actor.

```cpp
class headerAdd{
private:
  //Imports ...
  //Functions and Variables:
  unsigned short octet_count;
  unsigned short octet_index;
  const int HEADER_LEN = 5;
  const char Header[5] = { 0, 0, 0, 0, 167 };
  enum class states{
    s_header, s_idle, s_payload
  };
  states state = states::s_idle;
  //Input FIFOs:
```

Listing 6.2: HeaderAdd Actor of ZigBee
Port< unsigned char , 100000>& pl_bits$FIFO;
//Output FIFOs:
Port< unsigned char , 100000>& data$FIFO;
Port< unsigned short , 100000>& len$FIFO;
//Actions:
void initialize$(){
    octet count = 0;
}
void get_data_len${}{
    auto bits_in = pl_bits$FIFO.get_element();
    octet index = 0;
    octet count = bits_in+HEADER_LEN+1;
    len$FIFO.put_element( octet count );
}
void send_header${}{
    unsigned char data_out;
    data_out = Header[octet_index];
    octet_index = octet_index+1;
    data$FIFO.put_element( data_out );
}
void send_length${}{
    octet_index = octet_index+1;
    data$FIFO.put_element( octet_count−HEADER_LEN−1 );
}
void send_payload_octet${}{
    auto bits_in = pl_bits$FIFO.get_element();
    octet_index = octet_index+1;
    data$FIFO.put_element( bits_in );
}
void done$I${}
This code reflects the general structure of the generated code as presented in the previous chapter, subsection 5.3.1. In the class, first the imported code, that is omitted here, is placed, then the functions, variables, and the enum for all states, next, the port declarations and the actions. The functions that are generated from the actions show quite clearly how they are organised. First, the input tokens are read and stored in variables with the identifiers that are defined in the RVC-CAL code. This part is followed by the code that is generated from the body of the action and then the code that is generated from the expressions in the output part. The results of the expressions in the OutputPattern is put into an output FIFO buffer through an output port.

Up to here, everything is a private member of the class to avoid accidental bypassing of the local scheduler. The public interface of the class is restricted to the constructor, that in this case also calls the initialisation function generated from the initialisation action and the local scheduler.

As described in the methodology, chapter 4, and the development of the code generator, chapter 5, the local scheduler first checks the state of the actor instance. Then the local scheduler determines for each action that can be executed in the current state whether the guard conditions are fulfilled. This is done in the order of their priorities. Here the local scheduler first checks whether send_header can be executed before send_length because the priorities in the RVC-CAL actor specified that send_header has a higher priority. In the extract from the local scheduler, no action is shown that consumes tokens. Otherwise, the local scheduler would also check whether enough tokens are available together with the guard conditions in the first step. If this is fulfilled, in the next step it is checked whether enough free space in the output buffers is available. If no action can be executed or there is not enough space in an output buffer, the local scheduler terminates by breaking the infinite for-loop. To save space the rest of the local scheduler is omitted. Nevertheless, the general structure of the local scheduler is visible.

So far the functionality of the code generator only has been demonstrated using an example that cannot be converted to SYCL or OpenCL based code. Thus, in the
6.1 Conversion of a Complete Network

following the qpskMod actor is converted to SYCL and OpenCL based code. Code listing 6.4 shows the code of this actor.

```plaintext
actor qpskMod() uint(size=32) chip ==> int(size=8) symb :

    function q7_map( int(size=32) bit ) ==> int(size=8) :
        255 * bit - 128
    end

    action chip: [ c_in ]
    ==> symb: [ q7_map((c_in >> n) & 1) : for int n in 0 .. 31 ] repeat 32
end

Listing 6.4: Extract from the qpskMod Actor of ZigBee

Besides the property that this actor can be converted to SYCL or OpenCL based code, it also used to demonstrate how functions are converted by the example of the q7_map function. The action shown here has no body. Instead the complete functionality is defined in the head of the action by using a list comprehension.

Listing 6.5 shows an extract from the SYCL based C++ code that is generated from this actor.

```cpp
auto q7_map(int bit) -> char {
    return 255*bit-128;
}
c::sycl::event qpskMod::action$2(c::sycl::queue& $queue){
    int number$instances = std::min({chip$FIFO.get_size()/ 1 , symb$FIFO.get_free_space()/ 32});
    auto $ev = $queue.submit([&](cl::sycl::handler& $cgh){
        auto chip$FIFO$ptr = (*chip$FIFO.get_read_buffer(number$instances*1)).get_access<cl::sycl::access::mode::read>($cgh);
        auto symb$FIFO$ptr = (*symb$FIFO.get_write_buffer(number$instances*32)).get_access<cl::sycl::access::mode::write>($cgh);
        $cgh.parallel_for<class action$2$qpskMod>(
            c::sycl::range<1>(number$instances), [=](c::sycl::item<1>$it){
            int chip$FIFO$index = $it.get_linear_id() * 1;
            int symb$FIFO$index = $it.get_linear_id() * 32;
            auto c_in = chip$FIFO$ptr[chip$FIFO$index++];
            for(int n = 0; n <= 31;n++){
                symb$FIFO$ptr[symb$FIFO$index++] = q7_map((c_in >> n)&1);
            }
        });
    });
    return $ev;
}
```
This code listing shows the functions q7_map and action$2, that is generated from the action of the qpskMod actor, and the local scheduler of the generated actor class. The action is called action$2 because this naming scheme is used when no action identifier is specified in the RVC-CAL code. In this case action$ and a consecutive number is used as the function name.

The q7_map function looks similar to the RVC-CAL definition of this function. Only the keyword function is replaced by auto, −− > is replaced by − > and the C++ function body is encapsulated by braces. When OpenCL C code is generated the function head looks like this: char q7_map(int bit) Additionally, the return keyword is required in C++ whereas it does not exist in RVC-CAL.

In the function generated from the action, first, it is calculated how much instances of the action can be executed in parallel based on the number of available tokens and the free space in the FIFO buffers. For each input FIFO buffer, the number of available tokens is divided by the number of tokens consumed by a single action instance from this FIFO buffer. The same is done for each output FIFO buffer regarding the free space and the number of produced tokens. The minimum of all of them is the number of parallel instances. Then the SYCL call is set up. The ports are requested for the SYCL buffers and for each a SYCL accessor is created. Then the kernel is instantiated in the lambda expression of the parallel for call that also creates the index space according to the number of instances that was calculated before. The index space only contains the global worksize, but not the local worksize. Thus, the SYCL runtime will use NULL, and the OpenCL runtime will pick an appropriate size. In the kernel, before executing any operation, for each SYCL accessor an index is created. This index is incremented each time this buffer is accessed. The index depends on the global id of this instance and how much tokens each instance consumes from or produces for the buffer for which the index is created. These two numbers are multiplied to get the index with which this instance can access the buffer. This way each instance reads and writes the correct memory locations as
6.1 Conversion of a Complete Network
described in section 4.2. Additionally, there is no need to calculate the index every
time the corresponding buffer is accessed. This makes code generation much simpler.
The remaining part of the kernel is code that is converted from RVC-CAL to C++. This function returns a SYCL event that can be used to determine the execution status of the SYCL call.

The next function in the code listing is the local scheduler. The local scheduler only checks if there are enough tokens and free space in the FIFO buffers for one execution of the action, if yes it sets update_fifo to true. This boolean is a reference to the boolean of the global scheduler that keeps track of whether the SYCL buffers must be deleted and the output copied to the usual buffers. Then the local scheduler executes the action and returns the corresponding SYCL event. If the action is not executed a new SYCL event is returned that by default has the execution status complete and update_fifo remains false.

Code listing 6.6 shows the host code of the OpenCL based version of this actor.

The object opencl_arguments contains every object that is required by OpenCL for this particular class instance. Along with other OpenCL related functions, this struct is provided by a library that is also generated by the code generator. As in the SYCL based version, the function that is generated from the action is named action$2. This function takes a pointer to a cl_event as a parameter. This pointer is used to give the global scheduler access to the execution status of this OpenCL call just like in the SYCL based version. In the function, also the number of parallel instances is calculated. Then the globalWorksize of the opencl_arguments object is set to the previously calculated number of parallel instances, and the dimension is set to 1 because the index space is one-dimensional and the localWorkSize is set to NULL to let the OpenCL runtime determine appropriate work-group sizes. Then the OpenCL buffers that are created by the ports are set as the arguments of the kernel. Finally, the ExecuteKernel function of the library is called. This function uses the parameters stored in the opencl_arguments object, the pointer to the cl_event and the index of the kernel to enqueue the OpenCL kernel. The index of the kernel is an index to an array in the opencl_arguments object where all the kernels of this class instance are stored. There can exist multiple because an actor can have multiple actions. The kernels are created in the constructor. The constructor is also responsible for the initialisation of OpenCL. For this purpose, it calls SetupOpenCL and CreateAndBuildProgram from the library generated by the code generator. SetupOpenCL creates an OpenCL context with a device of the given type and vendor, in this example an NVIDIA GPU. CreateAndBuildProgram builds the program object for the given file containing all the kernels and functions for this actor or class. In contrast to OpenCL, in the SYCL
opencl_arguments ocl;

```cpp
void action$2(cl_event *event){
    int number$instances = std::min({chip$FIFO.get_size() / 1,
                                      symb$FIFO.get_free_space() / 32});
    ocl.globalWorkSize[0] = number$instances;
    ocl.work_Dim = 1;
    ocl.localWorkSize[0] = NULL;
    clSetKernelArg(ocl.kernels[0] , 0, sizeof(cl_mem),
                   (void *)chip$FIFO.get_read_buffer(number$instances * 1,ocl));
    clSetKernelArg(ocl.kernels[0] , 1, sizeof(cl_mem),
                   (void *)symb$FIFO.get_write_buffer(number$instances * 32,ocl));
    ExecuteKernel(&ocl,0,event);
}
```

```cpp
qpskMod( Port< unsigned int , 100000>&_chip$FIFO,
         Port< char , 100000>&_symb$FIFO )
    : chip$FIFO(_chip$FIFO),symb$FIFO(_symb$FIFO)
{
    SetupOpenCL(&ocl, CL_DEVICE_TYPE_GPU, "NVIDIA");
    CreateAndBuildProgram(&ocl, "qpskMod.cl");
    cl_int err;
    for(int i = 0;i < 1;++i){
        ocl.kernels[0] = clCreateKernel(ocl.program, "action$2", &err);
    }
}
```

```cpp
void schedule(cl_event *event){
    chip$FIFO.opencl_read_done();
    symb$FIFO.opencl_write_done(ocl);
    if(chip$FIFO.get_size() >= 1){
        if(symb$FIFO.get_free_space() >= 32) {
            return action$2(event);
        }
    }
}
```

Listing 6.6: Extract from the C++/OpenCL Code generated from the qpskMod Actor
6.2 Conversion of an Actor with Actions with Guard Conditions

Based version, the local scheduler first checks whether the OpenCL buffers have to be destroyed and output data has to be copied back by calling the corresponding functions of the ports. This is done in local scheduler instead of the global scheduler because to access an OpenCL buffer the corresponding context object is required. Hence, to keep the data as local as possible it is done here. SYCL manages these object internally, and therefore SYCL buffers can be destroyed without them.

Code listing 6.7 shows the kernel that belongs to this actor.

Listing 6.7: Generated OpenCL Kernel for the qpskMod Actor

This kernel basically looks like the kernel in the SYCL version. However, the function is in OpenCL C syntax and not in C++ syntax.

6.2 Conversion of an Actor with Actions with Guard Conditions

The ZigBee project does not contain an actor that is synchronous, has no finite state machine and no state variables, and guard conditions. Hence, the RoundAndPack actor (CALHStone/src/dfadd/common) from the CALHStone project is used to demonstrate how the code generator generates OpenCL based code for a synchronous actor with guard conditions. This section only demonstrates this by the OpenCL kernel because the SYCL kernel looks similar and showing both does not add more value. Listing 6.8 shows the RVC-CAL source code of this actor.

For the sake of simplicity parts of the code that are not relevant for this purpose are omitted, e.g. the import statements and the body of the actions.

This actor executes the action named process if the first token in the FIFO buffer
actor RoundAndPack()
uint ZSign_In, uint ZExp_In, uint(size = 64) ZSig_In, uint Round_In
==>
uint ZSign_Out, uint ZExp_Out, uint(size = 64) ZSig_Out, uint Round_Out, uint Flag :

  process: action ZSign_In:[zSign], ZExp_In:[zExp], ZSig_In:[zSig], Round_In:[r]
            ==> ZSign_Out:[zSign_out], ZExp_Out:[zExp_out], ZSig_Out:[zSig_out],
                Flag:[flag], Round_Out:[round]
  guard
    r != 0
  var
    ...
  do
    ...
  end

bypass: action ZSign_In:[zSign], ZExp_In:[zExp], ZSig_In:[zSig], Round_In:[r]
        ==> ZSign_Out:[zSign], ZExp_Out:[zExp], ZSig_Out:[zSig],
            Flag:[0], Round_Out:[0]
  guard
    r = 0
  end
end

Listing 6.8: RoundAndPack Actor

attached to the port Round_In has a value that is not equal to 0. Otherwise, the
action named bypass is executed. Both actions consume the same amount of tokens
from the corresponding FIFO buffers.

   Code listing 6.9 shows the generated OpenCL kernel from this actor.

_kernel void action_RoundAndPack(_global unsigned int * ZSign_In$FIFO$ptr
    , _global unsigned int * ZExp_In$FIFO$ptr
    , _global unsigned int * ZSig_In$FIFO$ptr
    , _global unsigned int * Round_In$FIFO$ptr
    , _global unsigned int * ZSign_Out$FIFO$ptr
    , _global unsigned int * ZExp_Out$FIFO$ptr
    , _global unsigned long * ZSig_Out$FIFO$ptr
    , _global unsigned int * Flag$FIFO$ptr
    , _global unsigned int * Round_Out$FIFO$ptr )
{
  int ZSign_In$FIFO$index = get_global_id(0)* 1;
  int ZExp_In$FIFO$index = get_global_id(0)* 1;
  int ZSig_In$FIFO$index = get_global_id(0)* 1;
  int Round_In$FIFO$index = get_global_id(0)* 1;
  int ZSign_Out$FIFO$index = get_global_id(0)* 1;
  int ZExp_Out$FIFO$index = get_global_id(0)* 1;
  int ZSig_Out$FIFO$index = get_global_id(0)* 1;
}
6.2 Conversion of an Actor with Actions with Guard Conditions

```c
int Flag$FIFO$index = get_global_id(0)*1;
int Round_Out$FIFO$index = get_global_id(0)*1;

if(Round_In$FIFO$ptr[Round_In$FIFO$index + 0] != 0) {
    unsigned int zSign = ZSign_In$FIFO$ptr[ZSign_In$FIFO$index++];
    unsigned int zExp = ZExp_In$FIFO$ptr[ZExp_In$FIFO$index++];
    unsigned long zSig = ZSig_In$FIFO$ptr[ZSig_In$FIFO$index++];
    unsigned int r = Round_In$FIFO$ptr[Round_In$FIFO$index++];
    ...
    ZSign_Out$FIFO$ptr[ZSign_Out$FIFO$index++] = zSign_out;
    ZExp_Out$FIFO$ptr[ZExp_Out$FIFO$index++] = zExp_out;
    ZSig_Out$FIFO$ptr[ZSig_Out$FIFO$index++] = zSig_out;
    Flag$FIFO$ptr[Flag$FIFO$index++] = flag;
    Round_Out$FIFO$ptr[Round_Out$FIFO$index++] = round;
}
else if(Round_In$FIFO$ptr[Round_In$FIFO$index + 0] == 0) {
    unsigned int zSign = ZSign_In$FIFO$ptr[ZSign_In$FIFO$index++];
    unsigned int zExp = ZExp_In$FIFO$ptr[ZExp_In$FIFO$index++];
    unsigned long zSig = ZSig_In$FIFO$ptr[ZSig_In$FIFO$index++];
    unsigned int r = Round_In$FIFO$ptr[Round_In$FIFO$index++];
    ZSign_Out$FIFO$ptr[ZSign_Out$FIFO$index++] = zSign;
    ZExp_Out$FIFO$ptr[ZExp_Out$FIFO$index++] = zExp;
    ZSig_Out$FIFO$ptr[ZSig_Out$FIFO$index++] = zSig;
    Flag$FIFO$ptr[Flag$FIFO$index++] = 0;
    Round_Out$FIFO$ptr[Round_Out$FIFO$index++] = 0;
}
```

Listing 6.9: Generated OpenCL Kernel for the RoundAndPack Actor

This kernel is named action concatenated with the actor name because this one kernel contains all actions of the actor. Besides the initial index definitions, the kernel contains concatenated if-statements that check the guard conditions. If the token that is consumed from Round_In by a kernel instance is not equal to zero the body of the first if-statement that is generated from the action process is executed, otherwise the body of the second if-statement that is generated from the action bypass. This way, one kernel is created for a synchronous actor without a finite state machine. If the actor is not synchronous, it would for each kernel instance be hard to determine where to read and write the OpenCL buffers.

In the SYCL based version, the kernel looks the same but it is embedded into the host code, and the SYCL library is used instead of the OpenCL library.
### 6.3 Conversion of an Actor with a Finite State Machine

Unfortunately, none of the actors in the ORCC samples repository has a unicyclic finite state machine. Thus, this is demonstrated by a test actor that is derived from the headerAdd actor. This actor has no meaningful functionality. Code listing 6.10 shows the code of the test actor.

```plaintext
actor test_actor() uint(size=8) pl_bits ==> uint(size=8) data, uint(size=8) len :
    List(type: int(size=12), size=5) Header = [0, 0, 0, 0, 167];
    get_data_len: action pl_bits:[ bits_in ] ==> len: [ octet_count ]
        var uint(size=8) octet_count
        do octet_count := bits_in + Header[3] + 1;
        end
    send_length: action ==> data:[ octet_count − Header[0] − 1 ]
        var uint(size=8) octet_count
        do octet_count := 5;
        end
    done: action ==> 
        do
        end
    schedule fsm s_idle :
        s_idle ( get_data_len ) ---> s_header;
        s_header ( send_length ) ---> s_payload;
        s_payload ( done ) ---> s_idle;
        end
```

Listing 6.10: A Test Actor with a unicyclic Finite State Machine

This actor has three actions that are intended to be executed in the order get_data_len, send_length, and done. Additionally, it contains a constant list named Header.

Listing 6.11 shows an extract of the SYCL based code that is generated from this actor.

```plaintext
const short Header[5] = { 0, 0, 0, 0, 167};
cl::sycl::event test_actor::FSM$Cycle$test_actor(cl::sycl::queue& $queue){
    int number$instances = std::min({pl_bits$FIFO.get_size()/ 1
        , data$FIFO.get_free_space()/ 1, len$FIFO.get_free_space()/ 1});
```

84
6.3 Conversion of an Actor with a Finite State Machine

```cpp
auto $ev = queue.submit([&](cl::sycl::handler& $cgh)
{
  auto pl_bits$FIFO$ptr =
  (*pl_bits$FIFO.get_read_buffer(number$instances*1))
  .get_access<cl::sycl::access::mode::read>($cgh);
  auto data$FIFO$ptr =
  (*data$FIFO.get_write_buffer(number$instances*1))
  .get_access<cl::sycl::access::mode::write>($cgh);
  auto len$FIFO$ptr =
  (*len$FIFO.get_write_buffer(number$instances*1))
  .get_access<cl::sycl::access::mode::write>($cgh);
  $cgh.parallel_for<class FSM_Cyle_test_actor>(
    cl::sycl::range<1>(number$instances), [=](cl::sycl::item<1> $it)
    {
      int pl_bits$FIFO$index = $it.get_linear_id * 1;
      int data$FIFO$index = $it.get_linear_id * 1;
      int len$FIFO$index = $it.get_linear_id * 1;
      {
        unsigned char octet_count;
        octet_count = 5;
      }
      {
        auto bits_in = pl_bits$FIFO$ptr[pl_bits$FIFO$index++];
        unsigned char octet_count;
        octet_count = bits_in + Header[0] + 1;
        len$FIFO$ptr[len$FIFO$index++] = octet_count;
      }
    });
  return $ev;
});

cl::sycl::event test_actor::schedule(cl::sycl::queue& queue, bool& update_fifo){
  if(state == states::s_header){
    if(true){
      if( data$FIFO.get_free_space() >= 1 ) {
        send_length$();
        state = states::s_payload;
      }
    }
  }
  if(state == states::s_idle){
    if(pl_bits$FIFO.get_size() >= 1){
      if( len$FIFO.get_free_space() >= 1 ) {
        get_data_len$();
        state = states::s_header;
      }
    }
  }
  if(state == states::s_payload){
    if(true){
```
done$();
    state = states::s_idle;
}
}

if (state = states::s_header && pl_bits$FIFOget.size() >= 1
    && data$FIFOget_free_space() >= 1 && len$FIFOget_free_space() >= 1 )
{
    update_fifo = true;
    return FSM$Cycle$test_actor(queue);
}
return cl::sycl::event();

Listing 6.11: C++/SYCL Code generated from the Test Actor

This code listing shows the function FSM$Cycle$test_actor and the local scheduler. The function FSM$Cycle$test_actor executes one complete cycle of the action schedule. For the calculation of the number of parallel instances, for each input FIFO buffer, the number of consumed tokens in one complete cycle are added up during the code generation. The number of available tokens is then divided by this number to get the number of parallel instances that are possible concerning this particular FIFO buffer. The same is done for the output FIFO buffers regarding the free space. The minimum of all of them is the number of parallel instances. The SYCL call looks like before, but each code block generated from the different actions gets a distinct scope. Distinct scopes are necessary because the variables of the different actions could have the same identifiers. This would result in multiple definitions of the same variable, and the compiler would complain about it.

Additionally, besides the extract shown here, each action is converted to an additional function that does not use OpenCL or SYCL.

The local scheduler first goes through the whole cycle once before executing the function that executes complete cycles in parallel. This is necessary because it might not always be possible to execute a complete cycle. If not enough tokens are available to execute complete cycles the token flow must also continue. Otherwise, this actor could cause a deadlock. In the worst case, one complete cycle is executed in sequence before executing many cycles in parallel, but this is an acceptable performance decrease. On the other hand, complete cycles are only executed starting at one particular state of the finite state machine. Thus, it must be possible to reach this state if the previous execution left the actor in a different state. In the OpenCL based version this works in the same way.
7 Evaluation

Since the primary objective of this thesis is to produce code with a higher performance than ORCC by exploiting parallelism with SYCL and OpenCL, this chapter compares the performance of the generated code with the performance of the code generated by ORCC. For this purpose, four projects from the ORCC samples repository are used as benchmarks. The benchmarks are executed on the following hardware:

- Intel i5-7200U CPU
- Intel HD Graphics 620 integrated Graphics Processor (iGPU)
- NVIDIA GTX 950M GPU
- 8GB RAM

The code is compiled with Microsoft Visual C++ 2017 with the following compiler flags:

- O2 for maximum optimization favouring speed
- Ob2 to inline everything that’s possible
- Ot to favour fast code
- GL to optimize the whole program

The following software environment is used:

- ComputeCpp Version 1.0 conformant to SYCL 1.2.1
- Intel OpenCL SDK Version 7.0.0.2519
- Windows 10 Pro Version 1803 Build 17134.407
For benchmarking the performance of the code generated by ORCC is compared with the code generated by the code generator developed during this thesis. With the code generator a pure C++, a SYCL based version, and an OpenCL based version are generated. The SYCL based version is evaluated with the CPU and iGPU as targets for the SYCL calls and the OpenCL based version with the CPU, iGPU and GPU as targets. The SYCL based version cannot be evaluated on the GPU because SYCL requires SPIR, but NVIDIA does not support this. The pure C++ version is the reference point of this evaluation. With this reference point, it can be evaluated if the parallelisation is beneficial at all and if ORCC produces code that results in a higher performance.

Although the generated code creates the OpenCL buffers with the memory flag CL_MEM_ALLOC_HOST_PTR, this memory flag is only used in the first benchmark. For performance reasons, the OpenCL parameters are adjusted to achieve a higher performance. Therefore, the last three benchmarks create the OpenCL buffers with CL_MEM_USE_HOST_PTR and a pointer to memory that has just been allocated, and that is page and cache line aligned, is passed in as Host Pointer. When creating an input buffer, the data is copied to the newly allocated memory before creating the OpenCL buffer.

### 7.1 Evaluation of Image Processing

Table 7.1 shows the results for the Image Processing or RIP project, especially the network ImageProcessing/src/segmentationtextureAnalysis/LBP81_Top.xdf. As input, the given reference input (ImageProcessing/lib/input_signals/LBP81/input_0.in) file is concatenated multiple times until it has 5242881 lines. With this input 12 million output samples are produced.

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>ORCC CPU</th>
<th>OpenCL CPU</th>
<th>OpenCL iGPU</th>
<th>OpenCL GPU</th>
<th>SYCL CPU</th>
<th>SYCL iGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ms]</td>
<td>19205</td>
<td>8841</td>
<td>14157</td>
<td>23751</td>
<td>27331</td>
<td>720000</td>
<td>&gt;720000</td>
</tr>
<tr>
<td>Relative Performance [%]</td>
<td>100</td>
<td>217</td>
<td>135</td>
<td>80</td>
<td>70</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.1: Results for Image Processing
7.2 Evaluation of a Digital Filter

The network contains 56 different actor instances. Only the execution of actions of 31 of them is parallelised with OpenCL or SYCL. However, all parallelizable actors or actions are performing only simple operations. Most of the OpenCL kernels only read two values from the global memory, multiply or add them up and write the result back to the global memory. This is especially bad for the graphics processing units. When using OpenCL both the GPU and the iGPU perform worse than the pure C++ version, the GPU even worse than the iGPU. Even though the GPU has, in theory, the higher performance, it cannot utilise its full potential because the transfer of the data causes more overhead than on the iGPU. The calculations are rather negligible. Only the CPU shows an improved performance. The CPU seems to be more suitable to execute many kernel instances with few calculations and memory access as the significant factor. The global memory of the CPU is the RAM. Thus no further transfer of the data to the global memory of the device is necessary, and the CPU performs these kinds of kernels much faster than the graphics processing units.

However, when using kernels with only a few calculations, the performance gain is limited. Most of the times the overhead for data transfer and copying and the overhead of OpenCL is higher than the potential performance increase due to parallel processing of the data.

The performance of SYCL, either on the CPU or iGPU, is far off from the performance of all other versions. Unfortunately, the SYCL runtime introduces a significant overhead. Primarily when the application utilises all cores, the SYCL runtime slows down the whole execution. The SYCL runtime basically utilises one complete core. Since OpenCL is the underlying layer of SYCL and the versions based on OpenCL perform much better than the versions based on SYCL, it must be the SYCL runtime that is causing this large overhead.

7.2 Evaluation of a Digital Filter

Table 7.2 shows the results for the FIR filter from the Digital Filtering project. The network is located in DigitalFiltering/src/FIR/FIR_lowlevel.xdf

As input the given reference input (DigitalFiltering/lib/input_signals/fir/input_0.in) is concatenated multiple times until it contains 10130802 lines.

The network contains 13 actor instances. Eight of them can be parallelised with OpenCL or SYCL. The kernels are rather simple in this case, too. In four of the kernels a number is read from the global memory, multiplied by constant and then written back to the global memory, one kernel does bit-shifting, and the rest is just
7 Evaluation

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>ORCC CPU</th>
<th>OpenCL CPU</th>
<th>OpenCL iGPU</th>
<th>OpenCL GPU</th>
<th>SYCL CPU</th>
<th>SYCL iGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ms]</td>
<td>6458</td>
<td>5961</td>
<td>5921</td>
<td>7589</td>
<td>7321</td>
<td>106032</td>
<td>102350</td>
</tr>
<tr>
<td>Relative Performance [%]</td>
<td>100</td>
<td>108</td>
<td>109</td>
<td>85</td>
<td>88</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.2: Results for the FIR Filter

Thus, this benchmark produces roughly the same results than the previous one. However, in this case, the OpenCL version that targets the CPU is roughly equally fast than the ORCC version. The difference in the performance of the pure C++ version and ORCC is not as severe as before. One reason for this could be that in this benchmark much fewer actors are involved and, therefore, less communication through FIFO buffers is necessary. Hence, the communication between the actors could be implemented more efficiently in ORCC. Another reason could be that the ORCC design can handle a large number of actor instances more efficiently.

As in the previous benchmark, the performance of the version with SYCL is still bad.

7.3 Evaluation of a ZigBee Multitoken Transmitter

Table 7.3 shows the results for the ZigBee multitoken benchmark. The network can be found at this location: ZigBee/src/multitoken_tx/Top_ZigBee_tx.xdf As input the given reference input (ZigBee/lib/input_signals/tx_stream.in) is concatenated multi-

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>ORCC CPU</th>
<th>OpenCL CPU</th>
<th>OpenCL iGPU</th>
<th>OpenCL GPU</th>
<th>SYCL CPU</th>
<th>SYCL iGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ms]</td>
<td>4900</td>
<td>5405</td>
<td>4753</td>
<td>5149</td>
<td>4839</td>
<td>14244</td>
<td>14543</td>
</tr>
<tr>
<td>Relative Performance [%]</td>
<td>100</td>
<td>90</td>
<td>103</td>
<td>95</td>
<td>101</td>
<td>34</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 7.3: Results for ZigBee Multitoken
ple times until the file contains 40369 lines.

The network contains only 6 actor instances. The execution of two of them is parallelised with OpenCL or SYCL. However, like in the previous benchmarks the kernels are rather small and perform simple operations, either a bitwise-and and a shift operation or a 32 times a multiplication, subtraction, bitwise-and and a shift. Nevertheless, the kernels in this benchmark, even though they are small, they are much larger than the kernels in the previous benchmarks. The execution of less but larger kernels already favours more the graphics processing units because they exploit much more parallelism. But they are still not large enough to significantly improve the performance compared to the pure C++ version. However, the version based on OpenCL targeting the GPU performs slightly better than the pure C++ version just like the OpenCL based version targeting the CPU.

This benchmark is the only one in which the versions based on SYCL perform to a certain extent good compared to the previous benchmarks. But it is still much slower than the pure C++ version, the version based on OpenCL or ORCC.

### 7.4 Evaluation of a JPEG Decoder

Table 7.4 shows the results for the JPEG Decoder benchmark. The network is located in /src/jpeg/decoder/Top_Decoder.xdf. In contrast to the other benchmarks, in this case, the performance is not measured by the execution time of the program. Instead, the produced frames per second without a graphics output are taken as a performance measure.

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>ORCC</th>
<th>OpenCL CPU</th>
<th>OpenCL iGPU</th>
<th>OpenCL GPU</th>
<th>SYCL CPU</th>
<th>SYCL iGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frames per second</td>
<td>856</td>
<td>1490</td>
<td>1070</td>
<td>810</td>
<td>901</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>Relative Performance [%]</td>
<td>100</td>
<td>174</td>
<td>125</td>
<td>95</td>
<td>105</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 7.4: Results for the JPEG Decoder

The input for benchmarking is the JPEG encoded Foreman video. This is a popular test case for JPEG decoders. When the decoder reaches the end of the file, it instantaneously starts again at the beginning. Thus, a high number of frames can be decoded without using a large JPEG file. While benchmarking, the decoded
frames are not displayed. Displaying would also take some execution time that could influence the result of the benchmark, especially when using the GPU as the target for the OpenCL calls.

The network that implements this JPEG Decoder contains six actor instances. OpenCL or SYCL can accelerate the execution of only one of them. In contrast to the kernels of the previous benchmarks, this kernel, however, requires a lot more computation. The GPU, in particular, benefits from this. The gain of executing larger kernels many times in parallel is higher than the loss due to the overhead of OpenCL and the data transfers and copying. Also, the performance of the CPU is remarkable. Since the network is that small and has a rather linear structure, most of the times only one actor is working. Thus, OpenCL targeting the CPU most of the time does not interfere with the execution of other actor instances.

However, ORCC performs clearly best. ORCC establishes a constant token flow through the network. This seems to be beneficial for small and linear networks. The approach of ORCC utilises all cores, resulting in a more parallel and faster execution and, therefore, a higher performance. The processing of big chunks of tokens is not beneficial in this case because it does not utilise all available resources.
8 Conclusion and Future Work

The benchmarks have shown that at least OpenCL based implementations perform reasonably well compared to pure C++ implementations. Using the CPU to carry out the OpenCL calls is even beneficial for a large number of rather small kernels. Overall the CPU performs well in this scenario of applying a simple function on a large data set. Hence, a higher utilisation of the CPU is achievable with OpenCL. However, most of the actors are too small to achieve a performance improvement by the parallelisation with OpenCL targeting the GPU. Only in the JPEG Decoder example, the GPU was significantly faster than the pure C++ version. When executing kernels on the GPU, the data first has to be transferred to the device. The data transfer causes additional overhead. If the kernel is rather small and only performs a single multiplication, the performance is suffering more under the overhead and the communication than the performance gain by the parallel execution. This is observable in the benchmarks: With a shrinking number of simple kernels, the performance of the OpenCL version targeting the GPU is improving. But only in the JPEG Decoder benchmark with one larger kernel, the performance benefits from targeting the GPU.

However, all these benchmarks are worst cases for exploitation of parallelism by OpenCL. These benchmarks were developed to be compiled with ORCC. Despite that, the performance of the generated code is comprehensibly good.

But this framework provides a way of converting RVC-CAL code to SYCL or OpenCL based C++ code that is independent of eclipse. ORCC on the other hand strongly relies on eclipse and cannot be used without it. To use ORCC the actors and networks have to be organised in an eclipse project. Hence, this framework provides a good alternative to ORCC for generating C++ code from RVC-CAL code.

Additionally, this framework provides a possibility to compare the performance of SYCL and OpenCL. Unfortunately, SYCL did not perform well during benchmarking. The SYCL runtime is causing much overhead that makes the SYCL based implementations perform comparatively poor. Some experiments with SYCL showed that SYCL performs reasonably well if there are no other threads active in this process that are competing for runtime with the SYCL library. However, the concept
of SYCL to simplify the development of formerly OpenCL code by embedding the kernels into the host is a pretty good idea. The design and implementation of the framework were much easier for the SYCL version. Nevertheless, despite the high expectations that have resulted from the promotion of SYCL, the performance of the used SYCL implementation is not promising. Using OpenCL directly still seems to be the better option concerning performance. With some additional libraries, most of the programming overhead can also be hidden from the programmer.

However, the ORCC samples introduce some performance decreasing factors, especially for parallelisation, a lot of the actors perform only a small number of operations. As already mentioned OpenCL especially when targeting the GPU performs better with larger kernels. Thus, it might be beneficial to merge sequential actors like the chipMapper and the qpskMod actor in ZigBee to one larger actor to achieve a higher performance. Merging actors creates a larger kernel that might perform better on the GPU and eliminates the communication via OpenCL buffers and FIFO buffers on the host and reduces the OpenCL overhead. This might also be beneficial for actors that are not mapped to OpenCL. For actors with only a small number of operations, the communication is much more expensive concerning performance than the potential gain by parallelism, even for non-linear parts of the network. If non-linear parts of the network are merged, this would result in a duplication of code, but also potentially in a higher performance. This, however, depends on how much actor instances the network consists of and the hardware on which the generated code shall be executed. If there are still enough actor instances to utilise all cores, this can improve the performance.

Merging linear parts of the network can also be beneficial for the approach of this thesis because there is no constant token flow through the network. Large chunks of tokens are required for the parallel execution of actions. Thus, it is likely that only some of the actor instances in the linear part are active at the same time. Merging some of them would reduce the number of actors and, therefore, the communication and scheduling overhead without reducing the utilisation of the hardware.

Another improvement can be to map actor instances to a specific thread and use for each thread a different scheduler. Basically, the list of actor instances that is sorted according to the token flow through the network can be taken, and for n threads, every n-th actor instance in the list is mapped to the same thread. This partition can reduce the scheduling overhead of the global scheduler slightly, and no atomic flags would be needed. To use the load balancing of the operating system, the number of threads must be higher than the number of available cores. As ORCC does this in a similar way and performs significantly better for large networks, this
could improve the performance in these cases.

Scheduling could be further improved by identifying synchronous parts of the network and determining a static schedule for this part. Ideally, the complete network is synchronous but, based on the ORCC samples, this seems to be a rare case. With the static schedule of this part of the network, it would be possible to use the output OpenCL buffer of the producer as the input OpenCL buffer of the consumer of those tokens. In the approach of this thesis, the OpenCL buffer that keeps the output is destroyed, the tokens are copied to the buffer on the host, and a new OpenCL buffer is created as the input of the consumer. This is done to maintain a regular memory access pattern. With a static schedule, the regular memory access pattern could also be maintained without creating a new buffer because it can be determined how often the actor must execute actions until the consumer can consume all the produced tokens.

However, the approach of this framework focuses on general purpose computers with a CPU and a GPU, and it is a rather host-based approach that uses OpenCL only to accelerate the execution of some actors. However, OpenCL also deals with task level parallelism. With the Task Level Parallelism approach, every actor instance can be mapped to OpenCL. Actions that can be carried out in parallel can then be executed by different processing elements of the compute unit on which the action instance is executed. Whether actions or complete cycles of action schedules are parallelisable can be determined in the same way as discussed in this thesis. However, OpenCL is not meant to execute kernels that are running infinitely long. Thus, there has to be a global scheduler in the host that checks if there are enough tokens available and starts a kernel for this kernel instance. In this approach, all tokens would be stored in OpenCL buffers, and if they are accessed circularly, this could destroy the regular memory access pattern.
Bibliography


Bibliography


Bibliography


List of Abbreviations

**OpenCL** Open Computing Language

**CAL** CAL Actor Language

**RVC** Reconfigurable Video Coding

**DPN** Dataflow Process Network

**ORCC** Open RVC-CAL Compiler

**MoC** Model of Computation

**FNL** Functional unit Network Language