Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Dynamic Dataflow Actor Merging with Internal Token Buffering

Master's Thesis

by

Fabian Lukas Dietrich

May 7, 2025

University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Prof. Dr. Schneider
M. Sc. Florian Krebs

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Dynamic Dataflow Actor Merging with Internal Token Buffering“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 7.5.2025

Fabian Lukas Dietrich

Abstract

Dataflow Process Networks (DPNs) work with available tokens. Depending on the number of tokens available, actors can consume the tokens and create new output tokens from them if there are a sufficient number of input tokens. In theoretical models, an infinite number of available actors can be assumed. However, in real systems, there are significantly fewer execution units, such as CPU cores, than actors, resulting in communication overhead that must be managed among a finite number of actors. Therefore, efforts are made to reduce the number of actors. Additionally, it is crucial to establish a process that can be automated. To handle the limitations of finite resources, a strategy for actor merging is applied, whereby multiple actors are combined into larger entities. This introduces new challenges. For instance, merging actors can result in tokens that are generated but remain within the larger "meta-actor," or tokens that are left unconsumed within a single iteration. In such cases, merging is not feasible unless transformations of state variables, additional guard conditions, or feedback loops are applied. The objective of this work is to address these challenges, focusing on the internal token buffering required for actor merging. The actors are modelled in the CAL programming language, and the work logically explains the conditions under which actor merging is achievable and can be comprehensibly proven. Additionally, templates are developed for actor merging that utilize token buffering mechanisms, aiming to standardize and further automate the merging process.

Zusammenfassung

Datenfluss-Prozessnetze (DPNs) arbeiten mit verfügbaren Token. Je nach Anzahl der verfügbaren Token können die Akteure die Token verbrauchen und daraus neue Output-Token erzeugen, wenn eine ausreichende Anzahl von Input-Token vorhanden ist. In theoretischen Modellen kann von einer unendlichen Anzahl von verfügbaren Akteuren ausgegangen werden. In realen Systemen gibt es jedoch deutlich weniger Ausführungseinheiten, wie z. B. CPU-Kerne, als Akteure, was zu einem Kommunikations-Overhead führt, der zwischen einer endlichen Anzahl von Akteuren verwaltet werden muss. Daher wird versucht, die Anzahl der Akteure zu reduzieren. Darüber hinaus ist es von entscheidender Bedeutung, einen Prozess zu etablieren, der automatisiert werden kann. Um mit den begrenzten Ressourcen umzugehen, wird eine Strategie zur Zusammenlegung von Akteuren angewandt, bei der mehrere Akteure zu größeren Einheiten zusammengefasst werden. Dies bringt neue Herausforderungen mit sich. So kann die Zusammenlegung von Akteuren dazu führen, dass Token erzeugt werden, aber im größeren „Meta-Aktor“ verbleiben, oder dass Token innerhalb einer einzigen Iteration nicht verbraucht werden. In solchen Fällen ist eine Verschmelzung nicht durchführbar, es sei denn, es werden Transformationen von Zustandsvariablen, zusätzliche Schutzbedingungen oder Rückkopplungsschleifen angewendet. Das Ziel dieser Arbeit ist es, diese Herausforderungen anzugehen, wobei der Schwerpunkt auf der internen Token-Pufferung liegt, die für die Zusammenführung von Akteuren erforderlich ist.

Contents

1. Introduction

Parallelisation is playing an increasingly important role in the development of ever more powerful computer systems. This is due to the fact that the electrical-physical limits with regard to the miniaturisation of components have already been largely reached. For this reason, hardly any single-core, high-clocked processors are manufactured today. Instead, modern computer architecture is increasingly focussing on processors that are able to work in parallel with several cores or computing units simultaneously. This development has become clear in recent years: processors have evolved from simple single-core designs through quad-core and octa-core processors to hexa-core processors. These multi-core processors usually consist of different computing units, each with different characteristics and strengths. This architectural change makes it possible to further increase the performance of processors without having to accept losses in terms of energy efficiency, which is of considerable interest in today's world. As neither the clock frequency nor the supply voltage need to be increased significantly, power consumption remains constant while performance per watt can be further increased. From a hardware perspective, parallelisation is therefore often not a problem. The real challenge lies in the lack of support from the software, which is often unable to fully utilise the potential of the hardware. Much of today's software is still designed to process tasks sequentially. This sequential processing means that multicore processors cannot be utilised efficiently and their actual performance is not fully exploited. In some cases, this can even lead to a reduction in actual performance compared to single-core processors due to the lower clock frequency and lower supply voltage per core in multicore processors.
Looking at modern programming languages, it is noticeable that they are still manly based on a sequential Model of Computation (MoC) despite the hardware development. An MoC describes when and how operations are executed, how communication between these operations takes place and how concurrent tasks are synchronised.
However, it can be seen that the software development sector is gradually adapting to this change. More and more applications are being developed to utilise multiple cores, for example by defining multiple threads. However, the underlying MoCs are still mainly sequential, which leads to new challenges.
In particular, the use of multiple cores leads to new types of errors. For example, problems can occur when shared memory is not synchronised correctly or when concurrent tasks are managed incorrectly. Such errors are often difficult to identify, fix or even reproduce, as the sequence of competing tasks is not always deterministic. This problem significantly increases the complexity and vulnerability to errors in software development. [29] Against this background,

the exclusive use of sequential MoCs does not appear to be optimal. Fortunately, there are alternative MoCs that address this problem, such as Dataflow Process Networks (DPNs) [31], [33], [40], [7]. DPNs naturally support the use of multiple concurrent tasks and are therefore particularly suitable for modelling distributed parallel systems.
DPNs work with processes that communicate with each other via FIFO buffers. These FIFO buffers are the only means of communication between the processes and decouple them from each other. This decoupling enables the simultaneous execution of the processes. At the same time, frequent sources of error, such as uncontrolled access to shared resources without appropriate synchronisation, are avoided. [29]
A programming language based on this MoC is the CAL Actor Language (CAL) [16]. With CAL, it is possible to specify data flow actors using common language constructs such as if statements, while and foreach loops and functions. This makes it possible to model parallel systems at a higher level of abstraction and at the same time reduce sources of error that could arise due to insufficient synchronisation. In CAL, programmes are described as networks of actors that work independently of each other and only communicate with each other via specified channels. These communication channels are realised by so-called FIFO buffers (First-In, First-Out), which exchange data packets - also known as tokens - between the actors. A major advantage of this approach is that each of these actors can theoretically act independently of the others. This means that each actor in a multicore system could be executed on its own processor core. This would be an ideal use case for modern multi-core processors, as the parallel execution of the actors could achieve a significant increase in performance.
Although this approach seems optimal in theory, there are significant limitations in practice. Most modern multicore processors have a limited number of cores, usually a maximum of 16 cores, even with powerful processors. However, CAL programmes often consist of far more actors than the actual number of processor cores available. For example, if a programme consists of 100 actors but only 16 processor cores are available, then several actors must be assigned to a single core. This inevitably leads to split execution of the actors, which means that the potential of parallelisation is not fully exploited. A sensible assignment of the actors to the cores (scheduling) is therefore a complex challenge. This scheduling requires additional management tasks by the operating system or the runtime environment, which can lead to additional delays.
Another significant problem arises from the way in which the actors communicate with each other. Data is exchanged between the actors via FIFO buffers, in which the tokens are temporarily stored until they can be processed by another actor.
In principle, writing and reading these tokens from the FIFO buffers is nothing more than a memory access that can involve both the main memory (RAM) and the processor's cache. As modern processors have extremely high clock rates and achieve their performance through fast access to data in the cache, slow memory accesses can lead to considerable performance losses .[15]

A simple example illustrates this: When an actor creates a token and writes it to the FIFO buffer, this operation is written to memory. If another actor reads this token from the buffer again, a memory access must take place that may not be cached. This can lead to delays, as memory accesses to the RAM are many times slower than accesses to the cache.
In fact, in extreme cases, such memory accesses can slow down the system by a factor of 100 or more. This means that even if the underlying hardware model is theoretically capable of running multiple actors in parallel, performance is significantly impacted by slow memory accesses.
The problems described above can be addressed by adapting the number of actors in a CAL programme to the actual number of processor cores available in the respective system. Instead of executing each actor on a separate core, which is not possible in real systems with a limited number of cores, a technique is used in which several actors are combined into so-called meta-actors. The concept of merging actors into meta-actors is based on the idea of combining several interacting actors into larger units that can be executed as individual units on a single processor core. This method offers several decisive advantages. Firstly, it enables optimum utilisation of the processor cores. By reducing the number of actors to be processed to a number that corresponds to the available processor cores, the hardware can be utilised in the best possible way. Each meta-actor is assigned to a single core, which maximises parallelisation.
Another advantage is the reduction in memory accesses. As the originally separate actors are now combined within one meta-actor, the majority of communication takes place locally. This means that data exchanged between actors no longer has to be written and read via external FIFO buffers. Instead, processing takes place directly in the internal memory of the meta-actor, which significantly reduces the need for slow memory accesses.
These improvements lead directly to increased performance and runtime. As fewer memory accesses are required, not only does the speed of the calculations increase, but also the energy efficiency of the system. Memory accesses are very energy-intensive compared to pure computing operations. The more efficient utilisation of the hardware means that more computing power per watt is available, which is particularly important for energy-efficient applications.
In this thesis, an algorithm is developed that automates the method of merging actors described in chapter 5. The algorithm is able to merge actors programmed with CAL into a small number of meta-actors. This is achieved through the use of internal token buffering, which means that the tokens that are normally stored in external FIFO buffers are now stored directly in the memory area of the meta-actor. Following the presentation of the algorithm, its functionality is tested using various benchmarks to ensure that the algorithm works correctly (see section 6). Finally, measurement results are presented that illustrate the performance of the algorithm. These measurement results show that the merged network of meta-actors offers a significant speed advantage over the original network of individual actors (see chapter 6.3). The presented algorithm thus represents a promising method for significantly improving the

performance and efficiency of CAL programmes on multicore processors.

1.1. Contribution

This paper presents a novel algorithm that aims to efficiently merge multiple actors of a CAL programme into a single meta-actor. The developed algorithm differs fundamentally from existing approaches as it introduces replacement buffers within the meta-actor to enable communication between the merged actors. These replacement buffers are treated as internal data structures, which results in the previously external FIFO buffers being transferred to a more efficient, internal memory management. In this thesis, the algorithm is described and implemented in detail. Firstly, its correctness is checked in detail to ensure that the functionality of the original CAL programme is not impaired by the merging of the actors. This includes analysing the correct processing and forwarding of data as well as the consistent maintenance of the internal FIFO buffers within the meta-actor. This is followed by a comprehensive evaluation of the algorithm using the well-known benchmark orc-apps, which has established itself as a standard tool for evaluating performance improvements. This shows how the new algorithm can achieve significant runtime improvements. The performance advantages result from the reduction of external memory accesses and the optimised use of multi-core processors by efficiently combining several actors into a single, powerful meta-actor. Finally, the measurement results illustrate the potential of this approach for optimising CAL programs, particularly with regard to more efficient resource utilisation and increased computing power.

1.2. Outline

This thesis examines the optimisation of dataflow networks and their implementation in the CAL programming language. Chapter 2.1 explains the concept of DPNs and compares them with Petri nets to illustrate their modelling capabilities. The next section 2.2 looks at Kahn Process Networks as the basis for dataflow programming and compares them with other dataflow programming languages. Chapter 2.5 introduces the CAL Actor Language, including its language constructs, data types and control structures. The various action types and their syntax are also described in detail. Chapter 4 describes the repeat Actor Merging (RAM) developed by Krebs and Schneider, before chapter 5 covers the main topic of this thesis, Actor Merging with buffering. Here, the algorithm for merging actors is presented, taking into account states, guard conditions and input/output ports, and illustrated using an example. The central topic is actor merging chapter 5), in which actors are merged to optimise data flow networks. The prerequisites for a successful merging are explained, including FSM transformations and the handling of guard conditions. Finally, chapter 6 compares the proposed method with RAM and discusses their advantages and disadvantages. The conclusion in chapter 8 summarises the results and gives an outlook on possible future research work.

2. Preliminaries

2.1. Dataflow Process Networks

Dataflow Process Networks [31], [33], [40], [7] are a model for representing calculations in distributed parallel systems. As a model of computation (MoC), DPNs are characterised by the fact that computations can take place immediately when the required input data is available. This means that no global control or synchronisation is required, which makes them particularly suitable for parallel and distributed systems. Basically, DPNs consist of a finite number of processes (or agents) that are connected to each other via FIFO buffers. These buffers serve as a communication mechanism between processes and enable asynchronous data processing. As the FIFO buffers work independently of the arrival and departure times of the data, communication between the processes is completely decoupled (see figure 2.1). [49] A key feature of DPNs is that each FIFO buffer has exactly one producer and one consumer process. This establishes point-to-point communication in which each process performs its calculations independently of the other processes. This approach makes it possible for processes to be activated as soon as sufficient data is available in their input buffers - a process known as firing.

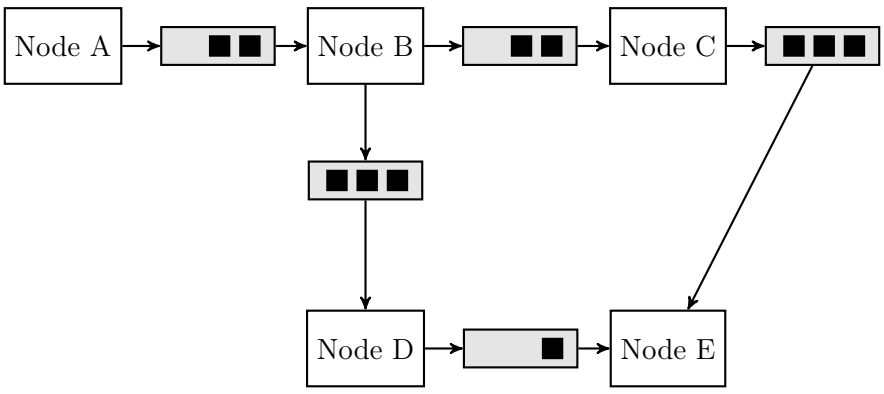


Figure 2.1.: *Example of a DPN with buffers.*

The behaviour of processes is defined by so-called firing rules (see figure 2.1). These determine when a process may fire, based on the availability and suitability of the input data. It is worth noting that DPNs have no restrictions in terms of the programming languages with which the individual processes are implemented. The FIFO buffers are theoretically infinite in size, which means that producer processes can write data at any time, while consumer processes can only read data if it is actually present in the buffer. If a buffer is empty, the consumer process must wait until new data arrives.

One problem that arises with this type of system is how to deal with missing data. As FIFO buffers are considered infinite, limiting the buffer size leads to unwanted synchronisation between processes, which in turn would compromise the asynchronous principle of DPNs.
In implementations, a distinction is often made between two basic variants of data processing:
Input/Data Driven Computation: In this method, the calculation is triggered by newly produced data values. As soon as a new data value arrives in a buffer, it is checked whether the process that is to consume this data can become active. The calculation is therefore triggered directly by the data itself - every data production exerts pressure on the consumer processes.
Output/Demand Driven Computation: Here, the computation is driven by the need for specific output data. When a process requires a data value, this process actively pulls the required data from its input buffer. This approach is also known as a pull-based model, as the demand for data drives the calculation.
It can be shown that these two styles are equivalent [50], [43], [44], [19], [37] and correspond conceptually to the lazy/eager evaluation of functional programmes.

i_1	i_2	b_1
a::A	b::B	a*b

Table 2.1.: *Firing rules example with input channels i_1, i_2, and output channel b_1.*

2.1.1. Synchronous and Asynchronous DPNs

DPNs are typically designed for asynchronous computation, as the processes can be executed independently of each other as soon as the input data is ready. This means that the processes in a DPN do not necessarily have to be executed at fixed times or in a fixed rhythm. In practice, this leads to a high degree of flexibility, but also to more complex requirements in terms of synchronisation and modelling of the processes. Nevertheless, there are special variants of DPNs that enable synchronous behaviour and facilitate their use in certain applications.

Synchronous/Static DPNs

A fundamental extension of DPNs is the concept of synchronous or static DPNs. In this model, the processes work according to a fixed, periodic schedule. Each process is executed several times within this schedule, whereby the number of executions and the buffer sizes for the input and output data can be determined statically. This means that the number of messages exchanged between the processes and the size of the buffers are determined in advance. This is particularly advantageous in embedded systems or real-time applications where deterministic execution is required. Synchronous behaviour leads

to better predictability and easier analysis as the flow of data and computations are periodic and fixed.
In a synchronous dataflow network (SDF), the process of firing (executing a process or function) is performed at fixed times, with each process requiring a fixed number of input values and producing a fixed number of output values in each cycle. The advantage of this model is that it is easier to analyse and implement, especially for applications where a constant rate of data processing is required [46], [34], [39].

Cyclo-Static DPNs

The extension of static DPNs is the Cyclo-Static Dataflow Model (CSDF), which further refines the periodicity of the behaviour of processes. While in static DPNs the processes fire at fixed, periodic times and their functionality remains constant over time, CSDF allows the behaviour of a process to change periodically while it continues to fire in a cyclic pattern. This means that the process can process different inputs or perform different calculations depending on the cycle. This opens up new possibilities for modelling dynamic processes that are not entirely reliant on continuous change. Instead, the behaviour can be repeated within a fixed cycle. For example, a signal processing process that uses different algorithms at different stages could be modelled in a CSDF model by using different parameters or algorithms within a cycle. This type of modelling is useful in many practical applications, such as communication technology or image processing, where the nature of the processing can change over the course of a cycle without disturbing the overall continuity. A major advantage of CSDF is that it provides the flexibility to model the behaviour of processes that can generally be considered 'cyclic', while still allowing static analysis of SDF networks. This means that many of the same planning and optimisation techniques developed for SDFs can also be applied to CSDF networks. For example, CSDF networks can be efficiently planned and optimised using techniques such as static data flow analysis and cyclic reduction, resulting in significant performance gains. Switching to CSDF networks also allows for greater flexibility in the way resources are allocated in a system. An example of this is the ability to use different algorithms or parameters in different cycles, which is beneficial when modelling systems with variable computing power or bandwidth requirements. [46], [34], [39]

2.2. Dataflow Programming Languages

Dataflow Graphs and DPNs serve as foundational Models of Computation for certain programming paradigms. Historically, dataflow programming was primarily used for graphical programming approaches [26] or for programming dataflow architectures. These implementations often relied on pure dataflow graphs, where each operation or function was represented as an independent actor. However, this method proved inefficient for sequential sections of pro-

grams. Mapping these sections onto physical machines incurred significant overhead due to the requirement for frequent communication and scheduling decisions. Specifically, after each actor's execution, tokens needed to be transferred to subsequent functions, and new scheduling decisions were made—even when the execution order was already well-defined. Such inefficiencies were particularly pronounced on von Neumann architectures, which are optimized for sequential execution and support parallelism primarily through limited multithreading. [51], [4], [29]

To address these challenges, dataflow programming evolved from employing fine-grained parallelism with nodes representing individual functions to nodes representing entire processes, as seen in DPNs [4]. By grouping sequential blocks into single actors, this approach reduced the number of actors requiring scheduling and minimized communication overhead. This shift also leveraged the growing parallelism capabilities of von Neumann machines, which can execute increasing numbers of threads in parallel, further improving the efficiency of dataflow programs. [29]

Initially, von Neumann architectures were deemed unsuitable for executing dataflow graphs [26], however advancements in multithreading and coordination mechanisms have rendered the Dataflow MoC more compatible with such systems. This compatibility arises from the separation of computation and coordination, whereby the computational aspects are encapsulated within dataflow actors and specified using a host language [38].Conversely, the scheduling and communication mechanisms are defined using a coordination language. [20]

Modern dataflow programming implementations that utilise DPNs include the CAL Actor Language and its derivative, the Reconfigurable Video Coding CAL Actor Language (RVC-CAL). These languages demonstrate the capacity of dataflow MoCs to efficiently integrate computational and coordination elements, thereby leveraging the strengths of von Neumann architectures. [29]

2.3. Kahn Process Networks

Kahn Process Networks [27], [28], [21], introduced by Gilles Kahn, expanded on the concept of DPNs by establishing conditions for determinacy even in the presence of feedback loops. Kahn demonstrated that determinacy is only guaranteed when process nodes exhibit restricted behaviors, particularly by performing blocking reads and disallowing emptiness checks on FIFOs. If an input buffer is empty, the consumer node must wait until data becomes available. This blocking-read mechanism ensures that processes produce consistent outputs regardless of the execution order, as long as the input data remains the same.
Furthermore, Kahn formulated the semantics of feedback loops as least fixpoints within the denotational semantics. This approach allowed for a more formal and predictable description of networks that include cycles. However, the restriction against emptiness checks and the requirement for blocking reads

place constraints on how these networks can be implemented.

2.4. Different Dataflow Programming Languages

Over the years, various dataflow programming languages [2] have been developed to utilize DPNs as a computational model. Notable examples include:

- Signal (Polychronous) [22] – A language focused on signal processing and embedded systems.
- Lustre [8], Scade [9](Synchronous) – Languages designed for safety-critical applications such as avionics and automotive systems.
- Quatz - A Model-Based Approach to the Synthesis of Hardware-Software Systems [49]
- CAL (Caltrop actor language) [16] – The latest and most versatile language built around DPNs, particularly suited for multimedia processing and hardware design.

Each of these languages employs different strategies to implement dataflow principles. For example, some languages focus on synchronous execution models where nodes fire based on a global clock or periodic schedule, while others allow asynchronous, event-driven processing.
The CAL language, which is the focus of this work, has proven particularly effective in implementing complex dataflow networks due to its flexible syntax and support for actor-oriented programming. By allowing the definition of actors that communicate exclusively through FIFO buffers, CAL provides a natural framework for describing and optimizing dataflow networks in various application domains.

2.5. CAL Actor Language

The CAL Language was developed in 2003 at the University of California, Berkeley, as part of the Ptolemy II project. Ptolemy II is an open-source framework designed to support theoretical work on actor-based networks, with a particular focus on DPNs. The primary goal of this project was to create a platform for understanding and engaging with computational models in a highly compositional way. [45]
At the heart of the Ptolemy II project is the development of directors, which are responsible for guiding the execution of actors within a model. These directors enable users to create and manipulate networks of computational elements, fostering an intuitive understanding of complex systems. The flexibility of the

framework allows for composition in various forms, making it highly adaptable to different types of computational models.
The CAL Actor Language was specifically designed to facilitate the creation of dataflow actors and entire networks in a straightforward manner. It prioritizes ease of use while also incorporating a robust error-detection system to ensure reliability. The language has a minimal semantic core, meaning its syntax and operational semantics are relatively compact compared to other programming languages. This simplicity does not compromise its functionality, making it a powerful tool for both theoretical exploration and practical application in actor-based systems.
The development of CAL was guided by two fundamental principles: the ease of use and the establishment of a robust error detection system. By focusing on these principles, the language offers both a user-friendly interface and the necessary safeguards to detect errors early, thus ensuring the creation of reliable and efficient actor networks.
In summary, the CAL Language plays a critical role in the development and experimentation with dataflow-based computational models, providing an accessible yet powerful tool for both researchers and practitioners in the field of actor-based computation. [16]

2.6. Calculation of Maximum Buffer Size

The maximum number of tokens in a buffer within an actor network is a critical factor, particularly when the network is modeled as a Petri net [42], [1], [41], [48], [35], [11]. A Petri net provides a structured approach to analyzing dynamic systems by representing transitions, states, and resources. This modeling technique enables the visualization of complex processes and helps understand how states and resources evolve when transitions are triggered.

In such a network:

- Places represent buffers or states where tokens (data) are stored.
- Transitions signify actions performed by actors, controlling the flow of resources.
- Edges (arcs) define how tokens move between places and transitions.

To determine the maximum number of tokens in a buffer, it is essential to analyze the network's behavior under different conditions and transitions. This section presents a systematic approach for calculating the maximum buffer size in an actor network using the incidence matrix of a Petri net, which provides a structured method to capture interactions between places and transitions, enabling a precise analysis of token flow and buffer occupancy. [6, 12] Since deterministic process networks (DPNs) can also be represented as Petri nets, they share characteristics with marked graphs [10], [25], [35], which in turn intersect with synchronized, static DPNs, highlighting the connection between these models. [3]

2.6.1. Incidence Matrix

The incidence matrix [13, 14] is a key mathematical tool in the analysis of Dataflow Process Networks (DPNs). It describes how actors interact with data buffers by specifying how many tokens are produced or consumed when a process node executes. Understanding this matrix is crucial for predicting system behavior, especially in terms of cyclic operations and buffer constraints.

Definition of the Incidence Matrix

The incidence matrix $I = (a_{i,j})$ of a DPN with n nodes and m buffers is defined as follows:

$$a_{i,j} := \begin{cases} \mathrm{np}(z_i) - \mathrm{nc}(z_i), & \text{if } f_j = \mathrm{prod}(z_i) = \mathrm{cons}(z_i) \\ \mathrm{np}(z_i), & \text{if } f_j = \mathrm{prod}(z_i) \neq \mathrm{cons}(z_i) \\ -\mathrm{nc}(z_i), & \text{if } f_j = \mathrm{cons}(z_i) \neq \mathrm{prod}(z_i) \\ 0, & \text{otherwise} \end{cases}$$

where:

- **np**(z_i): Number of tokens produced in buffer z_i.
- **nc**(z_i): Number of tokens consumed from buffer z_i.
- **prod**(z_i): The node that produces tokens in buffer z_i.
- **cons**(z_i): The node that consumes tokens from buffer z_i.

This matrix captures how token quantities in buffers change when nodes execute, enabling precise modeling of data dependencies in the system. [49]

Properties of the Incidence Matrix

The incidence matrix of a Dataflow network possesses several key properties that are crucial for understanding system behavior:

- **Balanced Token Flow**: Each row contains exactly one positive and one negative value, corresponding to the producer and consumer of a buffer.
- **Single Nonzero Entries**: If a node both produces and consumes a buffer, only one nonzero value appears in the row. This scenario can cause scheduling issues in periodic execution.
- **Rank Constraint** [13, 14]: The rank of the incidence matrix for a network with n nodes and m buffers is either n or $n-1$. Cyclic networks typically exhibit rank $n-1$.
- **Stability Analysis via** $I \cdot x = 0$: Nontrivial solutions $x = \lambda \cdot r$ provide insight into system stability and feasible execution schedules.

Key Theorems

Two fundamental theorems guide the analysis of the incidence matrix:

- **Theorem 1: Rank of the Incidence Matrix**
  The incidence matrix of a network with n nodes and m buffers has rank n or $n-1$.
- **Theorem 2: Existence of a Positive Solution**
  If the rank of the incidence matrix is $n-1$, then a positive vector $x > 0$ exists satisfying $I \cdot x = 0$. This rate vector describes the token flow between buffers while ensuring system stability.

[32]

Maximum Buffer Size Calculation

Beyond stability analysis, the incidence matrix helps determine the maximum buffer size required for safe execution. The calculation follows these steps:

1. **Identifying Positive Invariants**: The **S-invariant** Ensures token levels remain within finite bounds.
2. **Computing Maximum Tokens**: Solving $I \cdot x = 0$ identifies the upper limit of tokens in each buffer under different operating conditions.
3. **Ensuring Stability**: A well-designed periodic execution plan prevents infinite token accumulation and guarantees smooth operation.

[12]

2.6.2. Illustrative Example

To demonstrate these principles, consider a simple Petri net with:

- **Places (Buffers):** $P = \{p_1, p_2\}$
- **Transitions (Actions):** $T = \{t_1, t_2\}$
- **Incidence Matrix:**
$$C = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$$
- **Initial Marking:** $M_0 = (1, 0)$

This incidence matrix describes how tokens move between places as transitions fire. By analyzing the reachable markings, we can determine the maximum number of tokens in each buffer. [12]

3. Introduction to RVC-CAL

This chapter gives an introduction to dataflow programming with RVC-CAL, focusing on its integration within the ORCC framework. The initial sections explore fundamental concepts, including data types (see section 3.1), statements (see section 3.4), procedures, and functions (see section 3.3). Subsequently, the chapter delves into the definition of actions (see section 3.6) and proceeds to explain the implementation of libraries comprising functions, procedures, and constants, which can be imported into actors. Following this, the chapter outlines the process of defining actors (see section 3.7). Finally, the last section provides a concise overview of the specification of complete dataflow networks using a XML based language (see section 3.11). The introduction emphasises common use cases for the discussed language constructs, as demonstrated in ORCC examples. However, it does not encompass the full functionality of RVC-CAL. For a comprehensive specification of the language, readers are referred to [16].
The following notation is employed: keywords are represented in boldface, literals are enclosed in single quotes (’), optional elements are denoted by square brackets ([ ]), and components that must occur at least once but may appear multiple times are enclosed in braces(). When such components are repeated, they are separated by commas, or in the case of statements, by semicolons or the end keyword. [16],[29]

3.1. Data Types

The CAL-Language is optionally typed and has a minimal type system. This means that every new identifier can have a new type. Nevertheless, there are some basic types in CAL. In the following section, there is a short explanation about how to use data types in the CAL-Language.
In programming languages, there are two basic types. The first is the variable type, which is used to declare variables. The second is the object type, which is used for typing runtime objects. [16]

3.2. Required Types

In the CAL environment, it is imperative that each variable is assigned a specific type, which remains constant throughout its entire scope. Variables can be classified according to their relationship with other variables, thereby assuming the role of a subtype or a supertype of another variable. Within the CAL framework, there exist three fundamental methods through which types

can be expressed.

$$
\begin{aligned}
\text{Type} &\rightarrow \text{ID} \\
&\quad | \ \text{ID}\ '['\text{TypePars}']' \\
&\quad | \ \text{ID}'('['\text{TypeAttr}\{','\ \text{TypeAttr}\}']')' \\
&\quad |' \ ['[\text{Types}] \rightarrow \text{Type}']' \\
&\quad |' \ ['[\text{Types}] \rightarrow']' \\
\text{TypeAttr} &\rightarrow \text{ID}' :' \ \text{Type} \\
&\quad | \ \text{ID}' =' \ \text{Expression}
\end{aligned}
$$

Examples:

```
// Define a basic type alias
type IntList = List[Int];

// A more complex parametric type
type MapList = Map[String, List[Int]];

// Using type attributes (named parameters in a type)
type Point2D = Point(x: Int, y: Int);

// A function type (list of Ints to a String)
type Formatter = [List[Int] -> String];

// A function type with no output (like a procedure)
type Notifier = [String, Int ->];

// Declaring variables with fixed types
var ids: IntList;
var userScores: MapList;
var location: Point2D;
var toString: Formatter;
var alert: Notifier;
```

A type that is merely an identifier is known to refer to a type parameter (if listed in the actor's type parameters) or to represent a named non-parametric type, for example String or Integer.A parametric type is defined as taking other types as parameters, written as T[T1, ..., Tn], and is also referred to as a type constructor. It is noteworthy that numerous built-in types adhere to this paradigm; for instance, List[Integer] for a list of integers. [16]

In the context of CAL, it is important to note that certain types are implicitly created by special language constructs, typically expressions. The following is a partial enumeration of these essential types, which is intended to serve as a

comprehensive overview of the fundamental data types, their characteristics, and the supported operations.

Null:	A data type containing only the value `null`.
Boolean:	A data type that includes the truth values `true` and `false`.
Int:	A numeric type (e.g., integer or natural number).
List[T]:	Defines finite lists of elements of type T.

Additionally, CAL includes built-in data types for functional and procedural closures. The handling of many fundamental types – especially those representing numeric values – and their associated operations is delegated to the environment. The language itself only provides built-in support for these types through literals produced during lexical scanning. The environment is responsible for interpreting these literals and assigning them the appropriate types. [16]

3.3. Procedures, Functions and Native Functions

Closures are commonly used to define functions or procedures that are assigned a fixed name within a particular scope—often within the actor itself. One way to accomplish this is by using the standard variable declaration syntax. For example, you might write:

```
plusthree = lambda(x) : x + 3 end
```

Because this pattern appears so frequently, CAL also offers a more familiar syntax for defining functions and procedures. The previous example can be equivalently expressed as:

```
function plusthree (x) : x + 3 end
```

The general formats for these constructs are as follows:

Function Declarations:

```
FuncDecl → function ID '(' [FormalPars] ')'
           [var VarDecls ':']
           Expression
           end
```

Procedure Declarations:

```
ProcDecl → procedure ID '(' [FormalPars] ')'
           [var VarDecls (begin|do)]
           { Statement }
           end
```

In both cases, the declared variable is immutable; that is, it cannot be reassigned or modified. [16]

3.4. Statements in CAL

In CAL, statements represent the fundamental building blocks of execution. They define the flow of control and modify the state of variables within an actor. Statements typically introduce side effects by updating variable values or controlling execution sequences.

A statement is executed within a given scope, and its effects are reflected through changes in the environment. CAL provides several types of statements, each serving a distinct role in computation:

- **Assignment Statements:** Used to assign new values to variables.
- **Block Statements:** Group multiple statements together to form a structured execution block.
- **Conditional Statements (If Statements):** Execute different branches of code based on boolean conditions.
- **Loop Statements (While and Foreach Statements):** Repeat a block of code while a condition holds or iterate over collections.

These statements enable CAL to express computations in a structured manner, facilitating the development of reactive and dataflow-oriented applications .[16]

3.4.1. Assignment

In CAL, assigning a new value to a variable is the primary way to modify the state of an actor. The general syntax for an assignment statement is:

Assignment Statement:

AssignmentStmt → `ID [Index | FieldRef] := Expression ;`

where an assignment can involve either an index or a field reference:

- **Indexing:** An indexed assignment modifies a specific element within a collection using the syntax:

  Index → `'[' [Expressions] ']'`

- **Field References:** A field assignment modifies an attribute of an object using:

  FieldRef → `'.' ID`

Assignments involving field references or indices are collectively referred to as *mutations*, as they modify the internal state of an object or collection. [16]

3.4.2. Block Statements

In CAL, block statements serve as syntactic sugar for a specific case of the call statement. They are primarily used to introduce a local scope and declare local variables. The general syntax is:

Block Statement:

BlockStmt → `begin [var VarDecls do] { Statement } end`

A block statement of the form:

```
begin var decls do stmts end
```

is functionally equivalent to defining and immediately invoking an anonymous procedure:

```
procedure () var decls do stmts end () ;
```

This conversion shows that block instructions bundle local variables and statements in an ordered area. [16]

3.4.3. If Statements

The `if`-statement is the simplest control-flow construct in CAL. Its syntax is defined as:

IfStmt → `if Expression then { Statement }`
`[else { Statement }] end`

The behavior is straightforward:

- If the `Expression` evaluates to `true`, the statements within the `then` block are executed.
- If an `else` block is present, its statements are executed only if the `Expression` evaluates to `false`.

The condition (`Expression`) must be of type `Boolean`. [16],[29]

3.4.4. While Statements

Iteration constructs in CAL allow the repeated execution of a sequence of statements. The `while` statement executes its body as long as the specified Boolean expression evaluates to `true`. Its syntax is:

WhileStmt → `while Expression [var Var Decls] do`
`[Statements] (end | endwhile)`

The execution flow follows these rules:

- The loop condition (`Expression`) is evaluated before each iteration.
- If the condition is `true`, the statements inside the loop body are executed.
- If the condition is `false`, execution continues after the loop.

Additionally, it is considered an error if a `while` loop does not terminate, meaning it must have a condition that eventually evaluates to `false`. [16], [29]

3.4.5. Foreach Statements

The `foreach` construct in CAL allows iteration over collections by successively binding variables to elements within the collection and executing a sequence of statements for each such binding. The general syntax is:

Foreach Statement:

```
Foreach Stmt → Foreach Generator { ',' Foreach Generator }
               [var Var Decls] do [Statements] (end | endforeach)
```

A `foreach` generator is defined as:

```
Foreach Generator → foreach [Type] ID { ',' ID } in Expression
                    [',' Expressions]
```

This construct operates similarly to comprehensions with generators However, unlike comprehensions, which build a collection iteratively, a `foreach` statement executes a sequence of statements for each complete binding of its generator variables. [16], [29]

3.5. Expressions and their Types

Expressions yield a value and do not produce side effects, meaning they neither alter the actor's state nor modify or assign other variables. The significance of an expression is determined only by the value it evaluates to.

If an expression completes its computation successfully, it results in a value with a specific object type. This value is influenced by the context in which the expression is evaluated. Since environments can vary, the resulting objects of an expression may belong to different object types. However, based on the type system properties outlined in section 3.1, it can determine an upper bound for the possible types of an expression. This is done by considering the declared types of its free identifiers and the types of any literals it contains. The computed objects of the expression must always conform to the set of these upper bound types.

Below is an overview of the different types of expressions and their syntax in CAL:

An expression consists of a primary expression, optionally combined with operators:

Expression → PrimaryExpression{Operator PrimaryExpression}
PrimaryExpression → [Operator]SingleExpression
{'('[Expressions]')' | '['Expressions']' | '.'ID}
SingleExpression → [old]ID
| ExpressionLiteral
| '('Expressions')'
| IfExpression
| LambdaExpression
| ProcExpression
| LetExpression
| TypeAssertionExpr

This structure ensures that expressions remain predictable and side-effect-free, making them fundamental building blocks in CAL. [16]

3.6. Actions

In CAL, an action represents a potentially large or even infinite set of transitions within the actor transition system. A CAL actor can define multiple actions or none at all. Each action specifies:

- Input tokens,
- Output tokens,
- State changes,
- Additional firing conditions,
- Time delay.

At any given state, an actor can execute one or more transitions, or none at all. These transitions correspond to actions defined in the actor description. The selection among available actions depends on the actor's execution context, though constraints can be imposed by the actor itself. [16]

3.6.1. Action Syntax

The general syntax of an action definition is:

Action Definition:

Action → [ActionTag `':'`] `action` ActionHead
[`do` Statements] (`end` | `endaction`)

Action Tag:

ActionTag → `ID {'.' ID}`

Action Head:

ActionHead → `InputPatterns '==>' OutputExpressions`
[`guard Expressions`]
[`var VarDecls`]
[`delay Expression`]

Action tags, which are qualified identifiers (sequences of IDs separated by dots), can be used to identify and organize actions in schedules and priority orders. These tags do not need to be unique and may refer to multiple actions. [16]

3.6.2. Input and Output Patterns

Actions define how inputs and outputs are associated with ports. Associations can be positional or named, but not both simultaneously. Given an actor's port signature:

```
Input1, Input2 ==> ...
```

An input pattern can be written as:

```
[a], [b, c]
```

This binds `a` to the first token from `Input1`, and `b` and `c` to the first two tokens from `Input2`. Alternatively, a named association looks like:

```
Input2:  [c]
```

However, mixed associations such as:

```
[d] Input2:[e]
```

are not allowed. This rule applies equally to input patterns and output expressions. [16]

3.6.3. Action Semantics

An action describes how inputs, outputs, and state transitions are related, but its interpretation depends on the underlying model of computation, not on the actor itself. Therefore, an action should be understood as a declarative specification rather than an imperative operation. For a small example of an action see fig.3.1. [16]

```
action operand_1:[ x ], operand_2:[ y ] ==> result:[ out ]
var
    int o
do
    o = 1;
    o = o + x ;
    o = o + y;
end
```

Figure 3.1.: *Example Action that takes two inputs x, y, adds them to an output value o that is one and then outputs it*

3.7. Actors

In CAL, the concept of actors is fundamental to the modelling and organisation of parallel computations. An actor is an abstraction that represents an independent computing unit that responds to incoming data, processes it and generates outputs. The actor model provides a clear and scalable approach to building parallel and distributed systems by encapsulating both state and behaviour within independent computing units. Actors in CAL can be viewed as the building blocks of a system that interact with each other via messages or data transfer. Each actor can exhibit different behaviours depending on its current state and incoming messages. This chapter discusses the definition, structure, behaviour and scheduling of actors within CAL and provides a comprehensive overview of their role in system modelling.

An actor in CAL is an independent computing unit that is able to interact with its environment and other actors by receiving inputs and generating outputs. It can switch between different states, whereby each state is linked to specific actions. In each state, the actor performs specific actions based on the incoming data, possibly changing its internal state. In addition, the actor reacts to events or conditions that can trigger transitions between states. Communication with other actors takes place via messages or data transfers, which are typically realised via the actor's input and output points.
The behaviour of an actor is determined by its actions, which are conditional and depend on the state of the actor and the incoming messages. State transitions are controlled by FSM (Finite State Machine) principles, whereby each actor has different states and triggers that control the execution sequence. [16]

3.7.1. Structure of an Actor in CAL

The basic structure of an actor in CAL consists of several components:
Ports: These are the communication endpoints via which the actor receives

input data and generates output data. Each actor has specific input and output points that are typically tied to the actor's interaction with other parts of the system. [16] Input ports: These ports receive data from other actors or external sources.
Output ports: These ports send data to other actors or destinations.
States (see section 3.10): An actor can have multiple states, and the behaviour in each state is determined by the actions defined for that state. States in CAL are similar to those in a state machine model, with each state representing a specific execution phase.
Actions (see section 3.6): Each actor can perform actions in response to certain conditions. These actions are usually tied to the current state of the actor and may depend on the input data. Actions can be simple calculations or complex data manipulations.
Guards (see section3.8): A guard condition is an expression that must be fulfilled for a transition to occur. Guard conditions are used to enforce constraints or control when an actor can change its state.

The behaviour of an actor in CAL is controlled by a combination of its current state, the input data and the guard conditions. When an actor receives data at an input point, it processes it and performs actions based on its state. The actor then updates its state, possibly switching to a new state and sending outputs to other actors via its output points.
Actors can exhibit different behaviours depending on their internal state, and these behaviours can be programmed with conditional statements and actions. In fig.3.2 there is simple example of the behaviour of an actor:

```
actor A () int(size=14) Input1, int(size=14) Input2 ==>
    int(size=14)     Output:

    add: action Input1: [a], Input2: [b] ==> Output: [a + b]
    guard
    a > 0
    do
        println("Action Add is adding " + a + " and " + b);
    end

    sub: action Input1: [a], Input2: [b] ==> Output: [a - b]
    guard
    a < 0
    do
        println("Action Sub is substracing " + a + " and " + b);
    end

    zero : action Input1: [a], Input2: [b] ==> Output: [b]
    do
        println("Action zero return b);
    end

    priority:
    add > zero;
    sub > zero;
    end
end
```

Figure 3.2.: *Example Actor performs three actions: an addition, a subtraction and a return of b, where the actions are executed based on the input data Input1 and Input2, with the guards and priorities controlling the behavior.*

3.7.2. Scheduling and Execution of actors

The execution of actors in CAL is mainly controlled by an event-driven scheduling model, which is often referred to as actor scheduling. In this model, an actor is only executed when it receives new data or when its state transitions are triggered by certain events. Actors in CAL have no inherent execution order; instead, they are scheduled based on their readiness to fire (i.e. the availability of input data and fulfilled transition conditions). In systems with multiple actors, the scheduler ensures that actors are executed at the right time based on event and data dependencies.
In systems with high concurrency, actors can be executed simultaneously and

the scheduler manages the execution order to avoid conflicts and optimise resource usage. The priority of an actor can also be taken into account during scheduling to decide which actors are executed first, especially in time-critical applications. [16]

3.7.3. Communication between Actors

In CAL, actors communicate by sending and receiving messages via their input and output points. Communication between actors is asynchronous, which means that an actor can send a message to another actor without waiting for an immediate response. The receiving actor processes the message when it is ready to do so.
An actor can have multiple input and output points, and the connections between actors are defined by these ports. The communication channels between actors are established by connections that indicate which actor's output point is connected to another actor's input point. These connections enable complex interactions between actors in a distributed system. [16]

3.8. Guard Conditions

Guard conditions in CAL are a central concept for controlling the execution of actors and the communication between them. They determine when an actor becomes active in an actor network and enable detailed control over when and how processes are executed. This is particularly important in parallelised systems in which several actors work simultaneously and can influence each other. In CAL, there can be so-called guard conditions for each actor and each action, which act as conditions that are checked before an actor can fire (i.e. perform an action). These conditions ensure that an actor is only activated if all prerequisites for its execution are fulfilled, which leads to synchronisation and the avoidance of inconsistencies. [16]

3.8.1. Principle of Guard Conditions

The guard condition is essentially a logical condition that is linked to the input data of an actor fig.3.3. An actor in CAL has input channels from which it receives data. A guard condition checks whether the correct data is available on these channels in order to activate the action. As soon as the guard condition is evaluated as true, the actor can run the specific action, fire and execute its functionality.
The guard condition is expressed in CAL by a logical formula or condition, which can be defined either directly in the description of the actor or as part of the communication relationships between actors. The conditions can be simple comparisons or involve complex logical operations, depending on the requirements of the system. [16]

```
action i_1:[ a, b ] ==> result:[ a + b ]
guard
    a > 0
end
```

Figure 3.3.: *Example Action with a guard Condition that add a, b if a > 0.*

3.9. Priorities

In CAL, priority management is an essential concept, especially in systems where multiple actors are executing simultaneously and their execution order may be critical to system performance or correctness. Priorities allow finer control over which actors are allowed to fire first, based on certain conditions or requirements of the system.
In CAL, actor priority is typically used to determine the order of execution, especially in scenarios where multiple actors are ready to perform their operations simultaneously. Priority mechanisms can significantly affect the performance and efficiency of a system, especially in real-time systems or in systems with strict latency requirements. [16]

3.9.1. Principle of Priority

In CAL, the priority of an actor is generally determined by an explicit priority attribute or sequence of rules that controls the order in which actors fire within an actor network fig.3.4. These priorities are particularly relevant when there are multiple competing actors that need to access their input data and produce their output data simultaneously. [16]

3.9.2. Priority and Firing of Actors

In CAL, the firing of an actor is linked to the availability of input data and the fulfilment of guard conditions. The priority of an actor determines which actors are executed first if these conditions are met at the same time. If several actors could fire at the same time, the priority system ensures that the 'most important' actor is executed first.
An example of a priority system could be that in a signal processing system, certain actors that rely on real-time data are given a higher priority than other, less time-critical actors. These actors must then fire first in order to avoid delays in the system. [16]

```
priority
    write > read;
end
```

Figure 3.4.: *Example of the priority of an actor with two actions, write and read, where write has a higher priority than read.*

3.10. FSM Scheduling

In CAL, the scheduling of actors is often combined with the FSM technique to organise and control the execution of processes. State machines are a powerful model for describing systems that are in a finite number of states and perform transitions between these states based on certain events or conditions. In FSM scheduling in CAL, the system is organised so that actors in different states perform different tasks, and the scheduling determines how and when actors transition from one state to another. [16]

3.10.1. Principle of FSM Scheduling

FSM scheduling in CAL is based on the idea that actors pass through different states, depending on the input data and the transition conditions. Each state of the actor can be associated with a specific action or set of actions. These actions are executed in a defined time frame, which is determined by the transitions of the state machine. The change between the states takes place through so-called 'transitions', which are triggered by events or conditions. [16]
In the context of CAL, actors can be programmed to work as state machines. For example, an actor could start in an 'idle' state in which it runs for the first input data. As soon as the first input data is read, the actor switches to a 'processing' state in which it processes the next data. When the process is complete, the actor switches to a 'Done' state, which indicates that processing is complete with the second data and run with the next one fig.3.5. [16]

3.10.2. FSM Scheduling and its role in the system

FSM scheduling enables detailed control of the execution of actors in CAL. It helps to determine which actors are active at which time and contributes to the synchronisation of the entire system. Especially in real-time systems or systems with multiple parallel processes, it is important that actors are activated efficiently and at the right time. FSM scheduling ensures that the actors assume the right states at the right time, which optimises the overall performance of the system. [16].

```
schedule fsm idle :
    idle ( init ) --> processing;
    processing (run) --> done;
    done (end) --> idle;
end
```

Figure 3.5.: *Example of FSM scheduling of an actor with 3 states and the initial state idle and a different action for each state.*

3.11. XML

An XML-based data flow network describes a structured processing chain in which different instances perform certain arithmetic operations or signal processing steps. Each instance is defined by a unique ID and an associated class that determines its behaviour. Communication between the instances takes place via defined connections that send and receive data at specific ports. There are one or more data sources that feed information into the network, while the data is then forwarded from instance to instance and transformed in various processing steps. At the end of the process, there are one or more instances that output or forward the processed data. The hierarchical structure of such a network enables a high degree of modularity, as individual instances can be developed and exchanged independently of each other. An exemplary representation of such a network in XML can be found in the appendix of the thesis B. [16]

4. Repeat Actor Merging

In this chapter, the earlier developed RAM, developed by Krebs and Schneider [30], is explained. In the first part, it is demonstrated that CAL features are supported; subsequently, the section shows which CAL features are not included in the Actor Merging. The next part shows which Merging Criteria must be fulfilled for Actor Merging to be possible. Once the requirements are clear, the different steps of Actor Merging are explained. The process commences with the Merging Approach, encompassing Path Enumeration, Dataflow Configuration Merged, followed by a determination of merging feasibility. Subsequently, the Generation of FSM and Priority is initiated, culminating in the creation of the merged actions, actor, and the newly generated network.

It is imperative that no further verification of inputs (DPNs in the CAL Language) is conducted during the merge process. It is imperative to ensure that both the input cal-files and the input network exhibit the correct syntax of the Cal language.

There are more information about Actor-merging with Cal in the following Paper [30].

4.1. Supported CAL Features

It is important to note that not all features of the CAL language are supported by actor merging. The data types, statements and expressions explained in the previous chapter are all supported by Actor Merging. The supported functions of the CAL language include Actors and Actions, including the Init Action, FSM Scheduling, Priority Scheduling, Guard Conditions, State Variables, Functions and Procedures, Import without Renaming, List Type and List Comprehension as well as all basic data types and operations. [30]

4.2. Unsupported CAL Features

Compared to [16], the CAL language has many more features that are ignored by actor merging. In particular, functions such as choose, set, collection, map, time and delay are not supported; state variables should be used instead. Multichannel access is also not supported. Regular Expression Schedules are also not supported, and Priority Scheduling and FSM should be used instead. Other ignored functions include Old , Let, Mutable, Type Inference for Actors and Lambda. In addition, import with renaming is not supported, and the token rates of channels must be constant per action, whereby they can depend on imported constants, but not on other variables. [30]

4.3. Merging Criteria

In order to generate a merged actor that functions in a manner equivalent to the origin unmerged actors and are also represented in correct CAL-Code, there are four distinct criteria that must be fulfilled.

Criteria 1 "*The merging of actors must not introduce additional buffering of tokens within the new actor across different firings.*"[30]

The utilisation of buffers within the new actor gives rise to issues that ought to be circumvented. Primarily, the dimensions of these buffers cannot be assured, which can result in outcomes that are difficult to predict. Moreover, the scheduler must be apprised of the buffering of tokens, as even the buffered tokens exert influence on the order and timing of actions. Consequently, the buffering of tokens within the actor can lead to the merged actor not functioning correctly and should therefore be eschewed.

Instead, it is recommended that tokens be buffered within the communication channels between actors. It should be noted that these communication channels can only be realised between different actors; they cannot be established within an actor. Self-edges are excluded in this context, as they give rise to additional complications, including a greater number of possible configurations, which in turn complicates the process of merging. However, there exist some rare cases in which the buffering of tokens in state variables, such as control tokens, can be safely accomplished.

Criteria 2 "*The token rates of the merged actors must match each other, multiples are allowed.*"[30]

In accordance with the preceding criterion, the buffering of tokens is not permitted. Consequently, in instances where the token rates are incompatible, the implementation of buffering becomes imperative. In such scenarios, the introduction of loops becomes essential to compensate for the discrepancy in token rates and to guarantee the timely processing of all requisite tokens.

Criteria 3 "*The number of required tokens consumed from each input channel and the number of tokens produced for each output channel for the execution of the composite actor must be known when firing.*"[30]
It is evident that CAL and other data flow representations do not permit conditional consumption of additional tokens during the execution of actions. The scheduler is responsible for determining which action is to be executed, and this action is characterised by a fixed consumption and production rate. In scenarios where the requirement for additional tokens exceeds the original intention, it is imperative that the guard conditions or the FSM are specified in such a manner that the scheduler can accurately map the dynamics.

Criteria 4 "*If the initial actors can produce tokens for edges leaving the subnet to be merged for a given number of input tokens, the new "meta" actor should be able to do the same (minimum schedulability).*"[30]

In the event that the actors contained within the subnetwork that is to be amalgamated are capable of producing tokens for a specific quantity of input tokens within the input channels of the aforementioned subnetwork, yet the newly introduced meta-actor is unable to do so, there is a reduction in the predictability of the meta-actor in comparison to that of the preceding actors. This may consequently result in the requirement for other actors to utilise tokens in order to continue processing, yet the new actor is not capable of producing them.This phenomenon could be attributed to a schedule that necessitates the execution of multiple actors on multiple occasions.

However, if an actor is only executed once on each path through the subnet, this criterion is met.

4.4. Dataflow Configuration

A data flow configuration is defined as a depiction of the relationship between the input tokens consumed and the output tokens generated. It consists of (actor, action) pairs that are executed either unconditionally or under certain conditions to generate the output tokens based on the consumed input tokens and the internal state of the actor. [30]

Explanation:
Such a configuration can be imagined as a possible path through a subnet or as a possible flowchart of the actors contained therein. The actors consume a specific quantity of tokens and subsequently produce a predetermined amount of tokens. These configurations form the basis for the scheduler's capacity to plan actions. Consequently, they must define from which channels tokens are read, the quantity of tokens produced, and the conditions under which the configuration is considered valid. These conditions may include, but are not limited to, the values of the consumed tokens, the internal state of the FSM, guard conditions, or defined priorities between configurations. [30]

It is noteworthy that a single configuration can encompass multiple actions of an actor, provided these actions consume and produce an equivalent number of tokens from the same channels. This facilitates parallel execution, though not sequential processing within the same actor.

4.5. Actor Transformations

The following transformation rules pertain to the planning characteristics assigned to individual actors, yet they exert no influence on the time sequence of the actions of the newly created meta-actor. Consequently, all criteria that neither influence the number of tokens generated or consumed nor the channels involved can be transformed according to the following rules. [30]

4.5.1. Transformation 1 (FSM transformation):

A FSM is transformed by a structure of if ... then ... else ... endif statements that covers all states. Additionally, an internal variable is introduced that stores the current state.The transition between states occurs by assigning the variable a new value prior to or subsequent to the execution of the corresponding conditional code block. [30]

4.5.2. Transformation 2 (guard condition transformation):

In the event that multiple actions of an actor are situated within a unified data flow configuration and are distinguished from each other by guard conditions (guards), these conditions are converted into if...then...else...endif blocks. This process verifies the guard conditions and executes the corresponding operations. [30]

4.5.3. Transformation 3 (Priority transformation I):

In the event that multiple actions of an actor within a data flow configuration are arranged according to a specific priority, the planning conditions are evaluated according to this priority sequence. [30]

4.5.4. Transformation 4 (Priority transformation II):

In the event of multiple actions of an actor being present within a data flow configuration, and provided that priorities have been defined for these actions, and if a planning condition of one of these actions is already contained in the planning condition of another, the action with the stricter condition can be removed from the configuration. [30]

The purpose of these transformation methods is to minimise, or ideally avoid entirely, the need to create a complex and extensive state machine that takes into account all the planning characteristics of all the actions involved. In scenarios where an actor comprises multiple actions and an FSM is defined, this can result in a substantially large state machine, which can be challenging to generate automatically .Conversely, the conversion to if...then...else...endif structures enables the formulation of planning conditions at a more abstract level, focusing exclusively on the data flow actors in their entirety, such as the number of tokens consumed or generated, without addressing the internal processing of these tokens. [30]

4.6. Merging Approach

The focus is on Actor Merging, where the networks and actors must first be read and converted into a suitable representation. However, this part is simplified and omitted here in order to clearly illustrate the merging methods.

4.6.1. Path Ennumeration

The first step of the merging is to determine all valid flow configurations for the subnet to be fused. This is done by first determining the initial state of all actors involved. If no FSM is defined for an actor, an empty state is used instead, which means that no action can be planned. The next step is to identify all actors that consume tokens from the subnet or produce tokens themselves, and to check whether they contain actions that are independent of internal input tokens.
A list of these actions is created for each actor, with an empty action added as a placeholder if more than one actor is involved. These lists are combined into so-called starting points for path counting. From each of these starting points, a path is generated through the subnet, checking that the execution state of the actor matches the current state and that there are sufficient tokens available in the given configuration. In certain scenarios, the empty action may also be selected for an actor if no tokens are provided by the other actors.
After this step, each configuration contains at least one action per actor. If an actor has multiple static actions that can be executed in parallel, these actions are added to a configuration as long as there is no other action with a higher priority. This step also uses the FSM transformation, and once the configuration is complete, the state of the actors is evolved based on the planned action. In the event that this state combination has not yet been processed, starting points are created again and the count continues.
However, it should be noted that the resulting configurations may contain configurations that are irrelevant for the further process, such as configurations with empty actions or those that contain paths that produce tokens for internal channels from which nothing is consumed. In such cases, these superfluous configurations can be deleted. [30]

4.6.2. Dataflow Configuration Merging

The next step deals with the merging of configurations. In this step, configurations are combined so that parallel paths are covered by a single action, as is the case with a fork-and-join structure, for example. Two configurations can be merged if the number of tokens consumed and produced for each input and output channel is equal, and the path has a clear fork and join. It is important that the fork does not match the actor that reads the tokens from the input channels. In such cases, the CAL scheduler determines whether merging is required. This recognition and merging of configurations serves to further reduce the number of actions to be generated and optimises the merging process. [30]

4.6.3. Check of the Merging Criteria

In the next phase, the defined merging criteria are checked and the frequency of the actions performed in each configuration is calculated. It is noteworthy that empty dummy actions are assigned an execution count of zero. Furthermore, it is imperative that the token rates between consumers and producers are consistent for each configuration. This principle also pertains to merged configurations, as the overlapping parts of the configurations must be congruent. The minimum criterion for plannability is also verified. When checking the third criterion, it is also ensured that guard conditions can be propagated or converted according to the guard transformation. If this is not possible, it cannot be predicted on which path the tokens will flow through the network and it remains unclear how many tokens will be consumed or produced. [30]

4.6.4. FSM and Priority Generation

In this step, a new FSM is formulated. For each distinct state combination present within the data flow configurations, a new state is defined. The state transitions are then generated based on these combinations. The priorities of the actions are also redefined, with these priorities being based on the original priority definitions of the actors. It is important to note that only priorities between actions that can be planned in the same state are generated. Initially, a vector is created for each configuration that assigns each action according to its priority, with the highest priority being assigned the value '1'. Actions without a defined priority are given a high dummy value. The configurations that can be planned in the same state are then sorted according to these vectors. Repeated sorting of the configurations for the individual actors creates a new tree for the priorities. [30]

4.6.5. Actor Generation

Actor generation is a relatively straightforward step. The ports of the original actors at the boundaries of the merged subnet are adopted, and in the event of any name conflicts, they are renamed and the corresponding data structures are adapted accordingly. The FSM and the priorities are adopted from the original actors, and in addition, the state variables, imports, procedures and functions of the actors are integrated into the new actor. [30]

4.6.6. Action Generation

In the context of action generation, the input port accesses and guards of the input actions are combined to form the new action, in conjunction with the output port accesses of the output actions. The generation of an action is achieved through the iteration of all actors in topological order, with the corresponding action code being appended to the new action. The input port accesses are converted into code that reads the tokens produced and stores them in variables that align with the requirements of the action code. Conversely, the output port accesses are converted into code that stores the tokens

produced for each channel in a variable. In instances where multiple actions of the same actor can be planned in parallel, the relevant transformation rules are applied. When executing multiple repeated actions, a loop is created using the loop variable to access the corresponding tokens. When forking a data flow configuration, the code up to the join is enclosed in an if statement, with the code being added to the merged configuration as an else block. [30]

4.6.7. Network Generation

In the final step, a new XML representation of the data flow network is created with the new actor. Actors that are not involved in the merger are adopted unchanged, and connections between these actors and the merged actors are replaced by connections to the corresponding ports of the new actor, whereby the attributes of the original connections are adopted. Parameter definitions of the actors involved are used for the new actor and adjusted if necessary to avoid naming conflicts. [30]

5. Actor Merging with Buffering

RAM has several restrictions that significantly limit the choice of merging networks. In particular, the strict requirement that networks must not contain cicles or feedback loops is a significant limitation. This restriction is problematic, as such structures frequently occur in many real applications. Networks that require recursive calculations or continuous data processing, for example, typically rely on feedback loops. These restrictions therefore significantly limit the applicability of the RAM algorithm. [30]
To solve this problem, a new approach was developed in this thesis that allows networks with cycles and feedback loops to be fused effectively. The decisive advantage of this new approach over RAM is that the buffers between the actors are not eliminated, but remain as changeable internal structures in the merged actor. These internal buffers are implemented as modifiable lists within the meta-actor, which ensures that the original network structure is maintained without compromising the functionality of the buffers.
This approach makes it possible to correctly map feedback loops and cycles in the merged network. Instead of completely eliminating communication between actors, it is merely moved from external FIFO buffers to internal lists. This allows the original data flow behaviour to be retained while at the same time significantly reducing the number of memory accesses.
This new algorithm makes it possible to merge a far greater variety of networks than was possible with RAM. In particular, networks containing recursive processes or feedback data flows can now be merged with less restrictions. The ability to effectively merge such structures represents a significant advance in the optimisation of CAL programs and opens up new possibilities for their application on multicore processors.

5.1. Supported and Unsupported Features

Similar to RAM [30], the algorithm presented here does not support all the functions and features that the CAL Language theoretically provides. Instead, the focus is on the implementation of basic core functions that are sufficient for a functional proof of concept. This deliberate restriction was made because this work is primarily a proof-of-concept, which is intended to show that the basic functions of CAL can be successfully merged with the new algorithm.
However, the algorithm was developed in such a way that it is built on a solid, extensible foundation. By having all the essential basic elements in place, it is easily possible to add additional features to the algorithm in the future if required.
The currently supported functions of the algorithm include:

- Actors and actions including the initialisation action (init action).
- FSM for sequence control.
- Priority-based scheduling (Priority Scheduling).
- Guard conditions for controlling the flow of actions.
- State variables for storing data within an actor.
- Functions and procedures to define reusable code blocks.
- Import without renaming to integrate external definitions.
- List types and list comprehensions for handling data structures.
- All basic types and operations required for a basic implementation.

If you compare this implementation with the original complete CAL specification as described in [16], it becomes clear that some features have been deliberately omitted. These unsupported features include in particular:

- Choose, Set, Collection and Map for flexible data processing.
- Time and Delay, where state variables are used instead for modelling.
- Multichannel access, in particular the use of at and all for accessing channels is not supported.
- Regular expression schedules, whereby priority control and FSM can be used as an alternative.
- Old constructs, which should be replaced by temporary variables.
- Let constructs, which are not taken into account in the current implementation.

- Mutable data, whereby the assignment operators = and := are supported and are normally sufficient.

- Type inference for actors, which plays no role in the implementation.

- Lambda functions, whereby functions or procedures should be used in their place.

- Import with renaming.

- Token rates of channels, which must be constant per action; they can depend on imported constants, but not on other variables.

5.2. Merging Criteria

Criterion 1

Definition: Internal buffering may only be realised as an explicitly declared, restricted state variable. It must be guaranteed that the buffer does not grow in an uncontrolled manner, i.e. that it always has an upper limit.

Reason: This prevents the actor from unpredictably accumulating more tokens than it can process in one firing round due to internal buffering. At the same time, the scheduler must be able to know the current state of the buffer exactly. In real systems, there is also only a limited amount of memory available. It is therefore imperative that there are no unbounded buffers in internal buffering in order to prevent buffer overflow.

Example: Consider an actor that processes data packets in a real-time system. The actor has an internal buffer with a maximum size of 10 packets. If more than 10 packets are received before processing can occur, the actor will reject additional packets, preventing an uncontrolled growth of the buffer. This guarantees that the buffer size remains within predefined limits, and the scheduler knows exactly when the buffer is full, thus avoiding potential overflow.

Criterion 2

Definition: For each firing process, it must be known exactly how many tokens will be consumed from the inputs, transferred to the buffer and produced at the outputs, taking the internal buffer into account. The internal buffer logic must therefore be designed in such a way that it acts deterministically

and transparently as part of the scheduling process.

Reason: This enables the scheduler to reliably decide which action fires, as all token movements - including the change in the buffer state - are predictable.

Example: Imagine an actor in a video streaming application where each input token represents a frame of video data. The actor's internal buffer temporarily holds up to 5 frames while it processes them one by one. The scheduler knows that for every fire of the actor, exactly 1 input frame is consumed, and 1 output frame is produced, while the buffer retains 4 frames between each firing. This predictable behavior ensures that the scheduler can efficiently manage the flow of video data.

Criterion 3

Definition: For a given number of input tokens - including the effects of the buffer - the new meta-actor with internal buffering must be able to generate at least the same number of output tokens as the original actors intended. The internal buffer logic must not restrict the schedulability, but must maintain or deterministically extend the original production capability.

Reason: This prevents internal buffering from creating unexpected bottlenecks or asynchronies that could disrupt the overall flow in the system.

Example: Suppose a factory production line is modeled with actors that consume raw materials (input tokens) and produce finished products (output tokens). The internal buffer of each actor allows it to store a few raw materials while processing them. Even with this buffer, the actor should still produce the same number of finished products per time unit as it would without the buffer. If the buffer allows the actor to store more materials and process them more efficiently, the actor might even produce more output, but it should never produce fewer products than without the buffer, ensuring consistent production flow and avoiding bottlenecks.

5.3. Theoretical Foundations and Concepts

Having described the criteria that determine the conditions under which the merge process can be initiated in the previous section, a more detailed description of some basic concepts and principles now follows. These concepts are essential for understanding the developed algorithm and play a central role in how it works. The following section therefore explains important theoretical principles and mechanisms that are used in the algorithm itself. These explanations serve to make the subsequent implementation steps comprehensible and to clarify the underlying decision-making processes.

The algorithm assumes that the actors to be merged are connected to each

other within the network. This means that there should be no actors that are completely independent of the others. If this is the case, the algorithm must be applied in several runs in order to process each connected group separation.

Example:

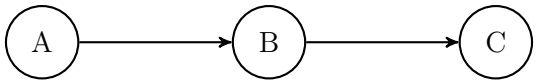


Figure 5.1.: *connecting group 1*

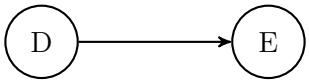


Figure 5.2.: *connecting group 2*

If all five actors (A, B, C, D, E) are to be merged, the algorithm doesn't recognise that there are two separate sub-networks within the set to be merged. To solve the problem, the algorithm is applied separately for each subgroup and a separate meta-actor is created for each.
Once all restrictions have been checked, the individual actors can be merged. To do this, the actors to be merged must first be placed in a topological order. The first step is to determine the start vectors - i.e. the actors that appear first in the network and produce tokens that are required by the other actors to be merged.
An actor is classified as a start vector if it has no incoming edges from other actors to be merged. If an actor only has incoming edges from actors outside the merge set or has no incoming edges at all, it is also defined as a start actor.
Example: As B has an incoming edge from A and A itself has no incoming

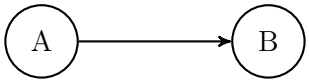


Figure 5.3.: *actors to be merged: A, B*

edge within the merge set, A is selected as the start actor. If several start actors exist, they are randomly inserted one after the other in the sequence.
Example:
In this case, C has two incoming edges ($A \rightarrow C$ and $B \rightarrow C$), while A and B have no incoming edges within the merge set. Therefore, A and B are considered as start actors and are randomly sorted one after the other.

5.3.1. Handling of Cycles

As the network can contain backward edges, cycles can form within the actors to be merged. In such a cycle, each actor has at least one incoming edge from another actor to be merged or refers to itself. If an actor only has one

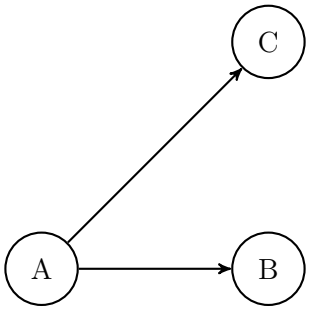


Figure 5.4.: *actors to be merged: A, B, C*

backward edge to itself and no other incoming edges, it can still be treated as a start vector. However, if there is a real cycle, a start actor must be selected at random within the cycle.

Example of a cycle: A circle exists here. As each actor has at least one

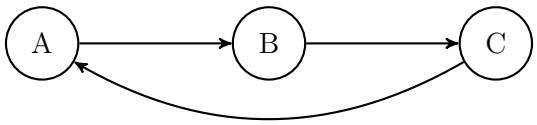


Figure 5.5.: *Example of a cycle:*

incoming edge, no start actor can be determined as before. In this case, the algorithm randomly selects an actor within the cycle as the start vector. A more complex case occurs when some actors initially have no backward edges, but later lead into a cycle. In this case, the algorithm must determine the start actor of the cycle. To do this, we calculate the distance to the last known start vector for each actor in the cycle.

Example:

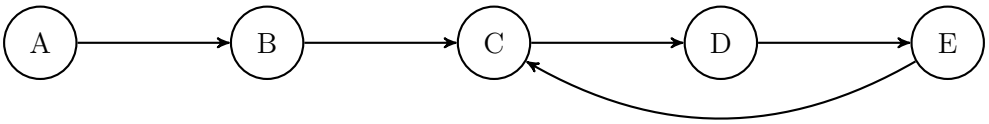


Figure 5.6.: *Example of a cycle with start vector:*

A is recognised as the start actor.
B is added.
C is added.
A cycle exists from here. To determine the sequence within the cycle, we calculate the distance to A for C, D and E. The actors are sorted in ascending order according to their distance. If several start actors exist, the distance to a start actor is determined randomly. As the number of preceding actors is constant, this only affects the cycle itself.
If several cycles exist within the merge set, they must also be prioritised among each other. If two cycles are connected to each other, it must be ensured that

their sequence remains consistent.
We now have the topological order in which the actors are to be merged with each other.

5.3.2. Relationship between Actors

Actors can have different relationships to each other within a network. These different structures influence the algorithm for merging actors, especially when token buffering is used.
There are different basic relationship structures between actors. In the following part some of them are shown.

Linear Sequence (chain)

The simplest relationship between actors is a linear sequence. One actor follows another Actor in a clearly defined sequence.
Example:

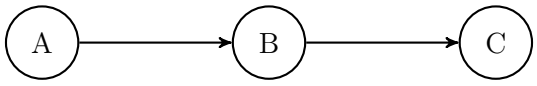


Figure 5.7.: *sequence of three actors*

Here has:
A has an outgoing edge to B.
B an incoming edge from A and an outgoing edge to C.
C has an incoming edge from B.
This structure is the simplest case for merging, as the data flow is unambiguous and the order of processing can be derived directly.

Parallelism (branching, fan-out/split)

Two actors can also be parallel to each other if they have no direct connection but both get tokens from a parent actor.
Example:

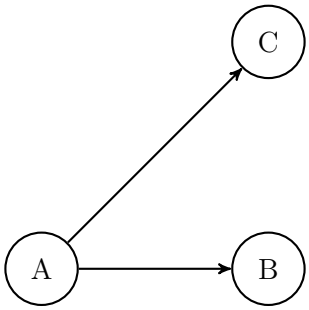


Figure 5.8.: *parallelism of actors*

Here has:
A has two outgoing edges, one to B and one to C.
B and C have no direct connection to each other, only one from A.
This structure is known as a fan-out or split. It requires special handling when merging, as the processing of B and C can run in parallel, which makes it necessary to synchronise the token flow.

Merging (join, fan-in)

Another possibility is that two or more actors have a common successor. In this case, the token flow leads from several sources to a common destination.
Example:

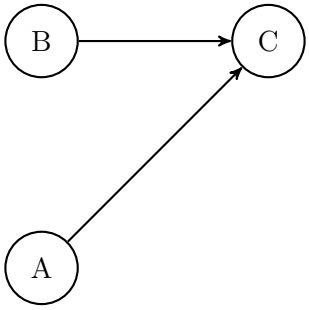


Figure 5.9.: *join of two actors into one*

Here C has two incoming edges: one from A and one from B.
A and B work independently of each other, but their results both flow into C.
This structure is often referred to as a fan-in or join. Unlike the previous case, where a parent had multiple successors, here data from multiple sources is merged. This requires special handling, especially if the inputs arrive asynchronously.

Cycles (Circles) and Backward Edges

In addition to the structures mentioned above, there can also be cycles (circles) in networks in which an actor refers back to itself over several steps or is dependent on other actors in a closed loop.
Example: Here there is a backward edge from C to D, which in turn has a

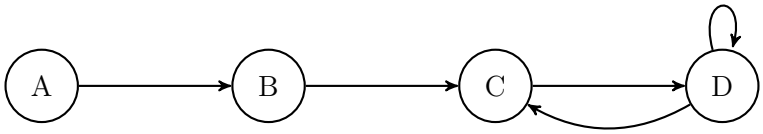


Figure 5.10.: *circle of the actors C and D*

connection to A. This creates a closed cycle.
Such structures pose particular challenges for merging, as they can lead to infinite dependency chains. It is therefore necessary to check whether the cycle can be broken or whether an alternative processing sequence is required.

5.3.3. Internal Buffers

The main difference between the algorithm presented in this paper and the RAM algorithm is the way in which tokens are processed. While RAM algorithm works without dedicated buffering, the approach in this paper uses FIFO buffers to temporarily store the tokens. A central question is which constructs of CAL are best suited to emulate such a buffer, which works according to the First-In-First-Out (FIFO) principle. In this work, FIFO buffers are simulated using lists. The lists, which exist in CAL as a basic data structure, are modified so that they have the desired buffer properties.
In order to use a list as a buffer, some basic aspects must be taken into account. Firstly, the maximum length of the list must be determined in advance in order to avoid memory overflows. It is also necessary to define the data type of the list to ensure that only valid elements are stored. In order for the list to function as a real FIFO buffer, new elements must always be added at the end and older elements removed at the beginning. In addition to the actual list, a variable is introduced that stores the number of saved elements. This variable is initially set to zero, as there are no tokens in the buffer yet. If a new token is saved, it is appended to the list as the last element and the variable for the saved elements is increased by one.
A token is always removed at the front end of the list. As CAL lists do not automatically have FIFO behaviour, all remaining elements must be moved forward by one position after a token has been removed. To do this, the first element of the list is read and returned, whereupon the second element is moved to the first position, the third element is moved to the second position and so on until the entire list has been moved forward by one position. Finally, the variable for the number of stored elements is reduced by one.
In addition to the sequential insertion and removal of individual tokens, there

are also scenarios in which several tokens have to be processed simultaneously. If a number of n new elements are to be inserted into the buffer at the same time, all elements must be checked and added to the list one after the other. It must be ensured that the list has a sufficient size. If there is enough space, all n new elements are appended to the list and the number of stored elements is increased accordingly.
It may also be necessary to remove several tokens at the same time. In this case, the first n elements of the list must be removed and all remaining elements must be moved forwards by n positions. After removal, the variable for the saved elements must be reduced accordingly.
By using lists to simulate a FIFO buffer, the tokens required for merging can be efficiently buffered and processed. Two central operations were considered: the addition of new tokens, which takes place at the end of the list, and the removal of tokens, which takes place at the beginning of the list and requires a shift of the remaining elements. While individual token processing steps are simple, parallel operations can complicate the implementation. In particular, if several elements are removed or added at the same time, care must be taken to ensure that the buffer is managed correctly and that no overflows or underflows occur in the buffer. This approach provides an efficient method for emulating a FIFO principle in the CAL Language so that token buffering can be easily implemented for the merging of actors.

5.4. Transformation

This section presents rules that are applied to the scheduling criteria of actors. However, these transformations do not change the actual order of action execution in the newly created meta-actor. As long as neither the number of tokens produced or consumed nor the affected communication channels are affected, the following transformations can be performed.

Transformation 1: Transformation of the state machine

An actor with a FSM is transformed into a structured sequence of if ... then ... else ... endif statements. An internal variable is used to store the current state. The state transitions are made by assigning a new value to this variable, either immediately before or after the execution of the associated statement block. In this way, the original logic of the state control is retained while the structure is simplified. [30]

Transformation 2: Conversion of guard conditions

If an actor has several actions with different guard conditions within a data flow configuration, these conditions are converted into an if ... then ... else ... endif structure. This ensures that it is checked at runtime which action is

activated. The conditions are evaluated one after the other so that the appropriate action is executed according to the defined logic. This method enables a compact representation without explicit state machines. [30]

Transformation 3: Conversion of priorities

If an actor in a data flow configuration has several actions with predefined priorities, the scheduling conditions are checked in the order of highest to lowest priority. This ensures that the action with the highest priority is always processed first before actions with a lower priority are considered. As a result, the original order of execution is retained and the planned sequence control is correctly mapped. [30]

Transformation 4: Optimisation of the priority check

If several actions of an actor with priorities exist within a data flow configuration and their scheduling conditions overlap, the condition of the stricter action can be omitted. This means that an action whose condition is completely contained in another can be removed without changing the system functionality. This reduces the complexity of planning and enables more efficient processing. [30]

Advantages of Transformations

These transformations avoid the need to create a complex and difficult to manage scheduling state machine that covers all possible conditions. If an actor has multiple actions and also defines an FSM, this could lead to an extremely large and difficult to automate structure.
By using if ... then ... else ... endif structures, the scheduling conditions are represented in a more compact form. This facilitates the modelling of the actor as a unit that consumes and produces a dynamic number of tokens without having to explicitly consider the internal processing of the tokens. This simplification contributes to the efficiency of the system and facilitates both analysis and implementation. [30]

5.5. Process

The process of actor merging is shown in fig.5.11. These steps only show the process itself, not the reading and transformation steps of integrating the different actors and networks into the process. For the sake of clarity, this is omitted here.

5.5.1. Path Enumeration

Before the actors can be merged with each other, the order in which they are processed within the network must be determined. Firstly, the start vectors

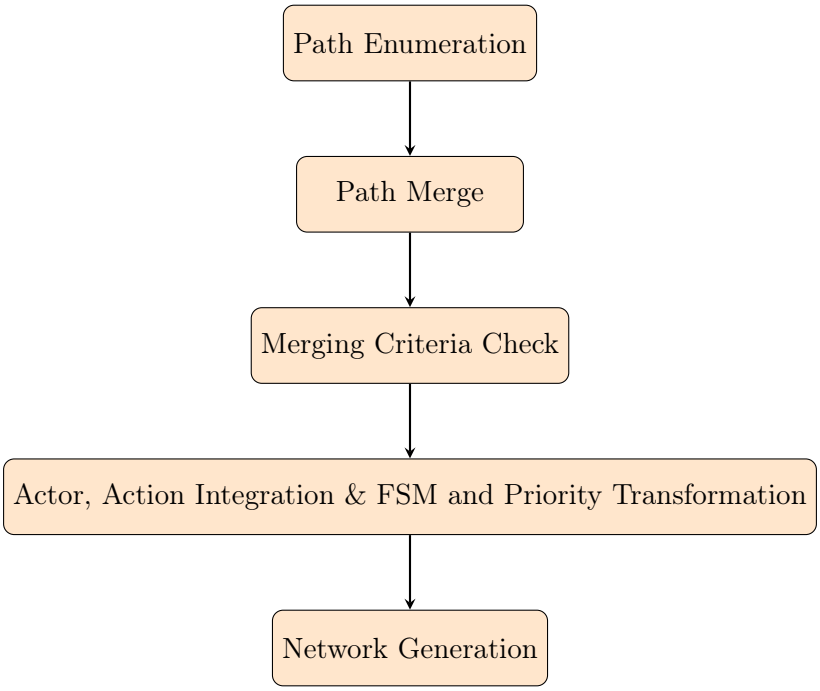


Figure 5.11.: *Process of Merging actors into a meta-actor*

are identified (see section 5.3), i.e. those actors that have no incoming edges. If there are several independent start vectors, these are considered in parallel so that several connected sub-networks can be created. Starting from these start vectors, the network is traversed step by step. The subsequent actors are determined on the basis of the outgoing edges of each actor already visited. If an actor is visited several times, there is either a parallel path or a cycle. If an actor is reached by a second path, this is a join, so it must be ensured that all dependent actors have been processed first before this join. If, on the other hand, an actor is visited again and a reconnection points to itself, this is a cycle. In this case, the second visit is ignored in order to avoid infinite loops in the sorting. Sorting ensures that new actors are always added and that only connected actors are considered together. Since it was specified at the beginning that only connected actors may be merged, the resulting list contains all relevant actors in a topologically ordered sequence after a finite number of steps. It is also taken into account that actors can lie in parallel if several independent paths exist. The handling of parallel structures is particularly important: If there are several parallel paths in the network, it is ensured that all actors are placed in the correct order before a join. Only then is the actor that depends on several predecessors processed. This ensures that all the necessary data is available before an actor that is fed by several paths is executed.

5.5.2. Check Criteria

In this section, the restrictions and criteria presented in the previous section 5.2 are systematically checked to determine whether it is even possible to merge the actors. This check ensures that all the necessary conditions are met before the actual merging can be carried out. If even one of the defined conditions is not met, the merging must be cancelled or alternative adjustments must be made in order to fulfil the requirements. As soon as it has been determined that merging is possible, the actual process of merging the actors begins.

5.5.3. Token Overflow

To enable the merging of actors, the concept of token overflow is introduced. Token overflow is a key figure that indicates for each edge in the network how many tokens the source actor writes to this edge and how many tokens the recipient consumes from the edge.
In addition, the token overflow also takes into account the behaviour of other actors in the network, in particular their token production and consumption. This creates a comprehensive relationship that not only describes the direct connection between two actors, but also their interactions with other actors.
The algorithm calculates the token overflow for all internal edges, i.e. for all connecting edges between actors that are to be merged into a single meta-actor. This step is important in order to identify where internal buffers are required in the network.
If the calculated token overflow for an edge is greater than zero, this means that tokens must be temporarily stored at this point. In such cases, an internal buffer is inserted between the affected actors to ensure that the data can be forwarded and processed correctly.
The token overflow is calculated based on a defined sequence of the actors to be merged. Each actor is considered in turn according to this sorted sequence. All incoming edges are analysed for each of these actors. If the source actor of an incoming edge has not yet been considered in the sequence, it is an edge that originates from a later actor. Such edges are referred to as backward edges.
Initially, the token overflow is zero for all edges. As backward edges have special properties, the token overflow for these edges is set to the value one (1) by default. Backward edges are used to transfer tokens to previous actors in the network. To ensure that these tokens can be cached correctly and processed later, an internal buffer is mandatory for each backward edge. This buffer ensures that the retransfer of the tokens is possible independently of the processing of other tokens in the system.
In the next step, the amount of tokens produced and consumed is calculated for each incoming edge of an actor. These calculations are based on the respective data rate of the edge, whereby the amount of tokens produced and consumed is written to the edge. If the calculated value is not equal to zero, this is recorded on the edge accordingly; otherwise the value remains set to zero.

If one of the edges involved - whether incoming or outgoing - has a repeat function, this is also included in the calculation. A repeat function means that tokens can be processed or sent multiple times, depending on certain input variables. As these repetitions are dynamic and cannot be clearly calculated in advance, the token overflow is also set to one (1) by default to ensure that possible token returns can be handled properly. In addition, if an internal buffer is recognised on one of the incoming edges, it is also necessary to insert a buffer on all other incoming edges of the same actor. This serves to ensure the consistency and synchronisation of processing and prevents tokens from being lost or processed unevenly.

5.5.4. Path Merge

The original token flow of the network must be retained. This is essential to guarantee that the logic of the original network is not changed. To achieve this, all start actors are first identified. These are either actors without incoming edges or those that are already initialised with enough tokens so that they can fire immediately. These start actors are processed first, as they are the first elements in the network's token flow. Once the start actors have been processed, all subsequent actors with a token overflow of zero are considered. Actors with an overflow of zero fire exactly when their predecessors fire and provide the required tokens. Therefore, they are immediately included in the processing as soon as their predecessors have been executed. The central step of this process is controlled by a while loop that runs as long as at least one actor that receives tokens via an internal buffer fires. This means that the actor has excess tokens after firing and can therefore activate further subsequent actors. Within this loop, all actors that have already been processed are considered again in order to activate their subsequent actors. If a subsequent actor has a token overflow of zero, it is executed immediately after its predecessor, as its firing depends directly on it. Particularly relevant is the case in which an actor with a token buffer exists, followed by an actor with a token overflow of zero. In this case, it is ensured that the subsequent actor is executed exactly when its predecessor fires. This ensures the correct token flow by guaranteeing that tokens are passed on exactly as intended in the original network. Actors with a positive token buffer allow other actors to fire again in a later iteration of the loop if the necessary tokens are still available. This mechanism preserves the token flow of the original network, as all actors are executed with the same dependency on each other as in the original model. The order of execution follows directly from the token flows without the need for additional sorting. This ensures that the merging process does not change the behaviour of the original network, but only creates a more efficient structure for executing the same computations.

5.5.5. Input Parameters

If an actor has input parameters that are determined by the network, it is necessary that these input parameters are also adopted for the newly created

meta-actor. This means that all input parameters of the original actor must be inserted as input parameters of the meta-actor. It is also necessary to adapt the network accordingly in order to integrate these input parameters correctly and to adapt the connections to the corresponding sources or other actors in the network.

5.5.6. Functions and Initialisation

In a network of actors that are merged into a meta-actor, not only the input variables must be adopted, but also the associated initialisation functions and methods. This ensures that the meta-actor maintains the same behaviour and functionality as the individual, merged actors. An important part of this process is the transfer of functions and initialisation logic originally defined in the individual actors.

If an actor in the original network has an initialisation function such as `initialize`, this logic must be transferred to the meta-actor and also defined there. This means that all functions that are defined in the original actor as part of the initialisation process or as regularly executed procedures are replicated in the meta-actor. The meta-actor therefore adopts all relevant methods that are present in the original actors and ensures that they can be executed during the operation of the meta-actor.

For example, an actor could have a native initialisation procedure such as `native_source_init`, which depends on an external source. This function could look like this:

```
@native procedure native_source_init(int ind)
end
```

This function is then integrated into the `initialise` procedure in the meta-actor:

```
initialise ==>
do
    native_source_init(tag);
end
```

This ensures that the specific initialisation logic that was required for the original actor is also executed in the meta-actor. The method `native_source_init` is called for the corresponding tag or index of the actor.

In addition to the initialisation logic, all other functions that are defined within the original actor must also be transferred to the meta-actor. This includes functions that control the logic of the actor during operation, such as data processing or state changes. A typical function could look like this:

```
@native function native_source_produce(int ind) --> int(size=SAMPLE_SZ)
end
```

This function, which produces a certain number of samples, is also defined in the meta-actor and must be called correctly there to ensure that the functionality of the meta-actor matches that of the individual actors.

The integration of the original functions into the meta-actor is based on the structure of the meta-actor, with each function and procedure being transferred to the corresponding sections of the meta-actor. The meta-actor is designed in such a way that it takes over the aggregated tasks of all merged actors and executes their respective logic correctly.

For example, the meta-actor could perform the following tasks

- **Initialisation:** The initialisation functions of each individual actor are executed when the meta-actor is started to ensure that all input variables and states are set correctly.
- **Data production:** Functions that produce data (e.g. `native_source_produce`) are called in the meta-actor and ensure that the data flows from the original actors are generated and passed on correctly.
- **State Transitions:** State changes and transitions defined in the original actor are handled in the meta-actor according to the specifications of the individual actors so that the meta-actor is able to coordinate the various states and transitions.

5.5.7. Input Ports

The first step is to check whether the action under consideration requires input tokens that originate from external sources, i.e. from outside the actors to be merged. If this is the case, the corresponding input ports are integrated into the meta-actor. These ports are provided with variables that enable the meta-actor to process and forward the required input tokens correctly. This ensures that the action can be executed correctly. If the action also requires input tokens from the network of the actors to be merged, it must be checked whether these tokens are directly accessible or must be loaded from a buffer. This decision depends on how the tokens are distributed within the network and whether they are available at the required time in the calculation. If the tokens are not immediately accessible, they must be extracted from a buffer. In the network under consideration, each input connection represents a connection (edge) between two actors. This edge represents the communication relationship between the actors and defines how tokens are exchanged between them. In this context, each edge is used as a key in a so-called token difference map. If the token difference map returns the value zero for a particular edge, this means that the preceding actor produces exactly the amount of tokens that the following actor consumes. Under these conditions, there are no further dependencies between the two actors. This means that the subsequent actor can be executed immediately as soon as the previous actor has completed its calculation. Whether a buffer needs to be implemented depends on the previously calculated (token overflow). If this value is not equal to zero, a

buffer is required to process the data correctly. However, if the value is zero, no additional buffer is required.

Code snippet fig.5.12 loads three and two tokens from two input buffers (b_0, b_1) moves the remaining elements in the list to the correct position using the For loop and then updates the buffer size.

Load tokens from b_0**:**
$i_0\_0 \leftarrow b_0[0]$
$i_0\_1 \leftarrow b_0[1]$
$i_0\_2 \leftarrow b_0[2]$
Load tokens:
for each $i \in 0 \ldots b_0\_b\_s - 3$ **do**
$b_0[i] \leftarrow b_0[i+3]$
end for
update size of the buffer b_0**:**
$b_0\_b\_s \leftarrow b_0\_b\_s - 3$

Load tokens from b_1**:**
$i_1\_0 \leftarrow b_1[0]$
$i_1\_1 \leftarrow b_1[1]$
Load tokens:
for each $i \in 0 \ldots b\_1\_b\_s - 2$ **do**
$b_1[i] \leftarrow b_1[i+2]$
end for
update size of the buffer b_1**:**
$b_1\_b\_s \leftarrow b_1\_b\_s - 2$

Figure 5.12.: *Pseudocode: How to load tokens from buffers*

The availability of the required tokens is checked individually for each input port. If the buffers contain sufficient elements, the meta-actor action can be executed. Otherwise, the action is skipped. If the check is successful, the tokens referenced in the variable names of the input tokens are successively removed from the buffer, whereby the buffer contents are moved accordingly and the number of elements stored in the buffer is reduced by the number of tokens removed. This operation is performed at the start of the action execution for all input ports.

5.5.8. Integrating FSM-Structure

An important step in the algorithm is to check whether the actor that is now to be integrated into the meta-actor has different states. An actor can have several states in which different actions are executed. If the actor actually has different states, a so-called state variable is created for this actor, which saves

the current state of the actor. This state variable makes it possible to track the state of the actor and ensure that each action is only executed in the correct state.

Once the state variable has been created for the actor, the algorithm proceeds to add the actions of the actor.Before an action is integrated into the 'meta-action' of the new meta-actor, however, some preparations must be made. Firstly, the actions of the actor must be sorted according to priority. This means that the algorithm performs a sorting that ensures that actions with a higher priority are executed first, while actions with a lower priority follow.
The next step is to check whether the respective action is also executed in the correct state of the actor. Before the action is integrated into the meta-action of the new meta-actor, the algorithm inserts a query that checks whether the state variable of the actor matches the state in which the action is to be executed. This ensures that each action is only executed if the actor is in the corresponding state.
If several actions can be executed within a certain state, the algorithm will use the priority of the actions to decide which action is inserted first. The action with the highest priority is included in the meta-action first. All other actions are then included in descending order of priority using 'if-else' queries. If two actions have the same priority, they are inserted in the order in which they were originally defined in the actor.
This precise handling of states and priorities ensures that the actions are embedded correctly and in the right order in the meta-actor so that all the logic of the individual actors can continue to be merged consistently and without errors.

5.5.9. Guard Conditions

A key aspect of merging multiple actors is the correct handling of guard conditions, which define certain conditions under which an action may be executed. Each actor to be merged can have guard conditions, which can vary depending on the current state of the actor, as an actor can be in different states and therefore have different guard conditions for different actions depending on the state. If an actor has defined guard conditions for certain actions, these conditions are considered separately in the course of the merging and integrated before the actual code section of the actor, which means that the guard conditions are extracted from the original code of the actor and added as separate check conditions in the new meta-actor to ensure that these conditions are also checked correctly after the merging. Before an action of an actor is actually executed, a multi-stage check process is run through, which starts with checking whether the actor is in the correct state required to perform the action. It then checks whether there are enough input tokens to be able to perform the action, whereby these tokens can be provided both externally and internally in the network. The actual guard condition associated with the action to be checked is then evaluated based on the conditions that were originally

defined in the actor. If this guard condition is fulfilled, the action is executed, but if it is not fulfilled, either the action is skipped or a check is made to see whether an alternative function of the actor exists whose guard condition is fulfilled. Another important point when merging actors is the prioritisation of individual actions, where each action is assigned a predetermined priority, and those with higher priority are evaluated before others. During integration, the actions are linked in such a way that only one action can be executed at a time, which is ensured by a hierarchical linking of the conditions so that the action with the highest priority, whose guard condition is fulfilled, is always executed. If there are no priorities between the actions of an actor, the actions are integrated into the meta-actor in a random order. [29]

5.5.10. Actual Action

Once the processing of the input ports has been completed, the main part of the action, the so-called body, can be taken over. At this point, all the required input tokens are already available in the meta-actor so that they can be used in the action body. The tokens that were determined in the previous steps of the process and, if necessary, loaded from buffers, are now available and can be used without restrictions to continue the calculations of the original action. Firstly, all variables defined in the var block of the original action are transferred to the var block of the meta-actor. This step is necessary to ensure that all variables used in the action are correctly available in the meta-actor. However, if there are variables that are defined differently depending on certain guard conditions (conditions that control the execution of the action), this definition is not transferred directly to the var block.
Instead, these special variables are moved to the Do block of the meta-actor. There, they are assigned to the corresponding guard conditions so that they only receive a different definition if the corresponding conditions are met. This ensures precise and context-dependent control of the variables within the meta-actor.
Once all variables have been correctly adopted and assigned, the actual action body is adopted in full. This means that all calculations and operations that were originally defined in the action to be merged are now executed in the same form and sequence in the meta-actor. The logic of the action is therefore completely retained and the calculations can be carried out in exactly the same way as in the original.
At the end of the action, the calculated results are available in the original output variables of the action to be merged. These output variables contain the final values that were calculated in the original action.

5.5.11. Output Ports

The output of the action is now handled in the same way as the processing of the input ports. An in-depth check is carried out for each output port to determine whether it belongs to one of the actors to be merged. This check

is necessary to determine whether the output port needs to be integrated into the meta-actor.
If this is the case, the output port is transferred to the meta-actor and the corresponding variables associated with this output port are also assigned. These variables are important in order to transfer the data correctly between the actors and to ensure that the action functions correctly.
However, it must be noted that in some cases the meta-actor can perform internal calculations without continuously or always generating an output. This means that tokens are not written to the output port in every calculation step. Rather, the decision as to whether output tokens are produced depends on certain conditions that are defined within the meta-actor.
The number of output tokens generated can also vary depending on the current state of the actor. This state is tracked by a state variable that reflects the current state of the actor. The state directly influences how many tokens are generated and whether tokens are output at all. For example, the actor may not generate any output tokens in a certain state, or it may produce more tokens than expected in another state. This state dependency of the output must be mapped in the meta-actor in order to correctly simulate the behaviour of the actor to be merged and enable consistent further processing.
If an actor has output ports and the number of tokens produced depends on the internal state of the actor, it is possible that an output is not produced in every calculation step. This poses a challenge when merging several actors into a meta-actor, as it must be ensured that the original logic of the token output is correctly adopted. To solve this problem, two central mechanisms are introduced in the meta-actor.
Temporary storage of output tokens as it is not guaranteed that tokens are written to the output port in every calculation step, a general list is created in the meta-actor. This list stores tokens temporarily so that they are not lost if the calculation is continued without a direct output. Storage of the previous output status: In addition, a variable is introduced that stores whether tokens were written to the output port in the previous calculation step. This variable serves as an indicator for the status of the output:
If tokens were output in the last calculation step, the variable is set to true.
If no tokens were issued, the variable remains false. The supplementary action that implements this logic must have a higher priority than the regular actions of the meta-actor. This means that this action is executed before the regular processing of the other action steps. This ensures that the cached tokens are checked and output before other calculations or state changes take place that could also affect output tokens.
This additional action uses the stored variable as a guard condition: If the variable has the value true, this means that tokens were written in the previous calculation step. In this case, the supplementary action ensures that the tokens from the buffer are written to the corresponding output port.
If the variable is false, no output takes place, as no tokens would have been output in this state in the original system either. This mechanism ensures that the meta-actor correctly replicates the original logic of the actors to be

merged. Each output port receives exactly the tokens that it should receive according to the state of the system, regardless of whether the calculation steps continuously produce tokens or not.
This approach is implemented for each output port and for each actor that generates output tokens to ensure consistent and correct transmission of the original token output.

It is also possible that no direct write accesses to the output ports of the action take place. Instead, the edges of the outgoing tokens may refer to subsequent actors that are located within the set of actors to be merged.
In this case, it is necessary to use the token difference map again to check whether the tokens on the corresponding edge can be processed directly in the subsequent actor or whether they must first be stored temporarily.
If the tokens are required immediately in the next actor and the number of tokens produced corresponds exactly to the number of tokens consumed, they can be passed on without intermediate storage. The next actor can process the tokens directly as soon as they have been issued by the previous action.
However, if there is a discrepancy between the number of tokens produced and consumed, intermediate storage is required. In this case, a new buffer is created as a list to temporarily store the tokens. This buffer is assigned to the edge in question and ensures that the tokens are available in the correct order when they are needed.
fig.5.13 shows the pseudocode of how internal tokens are stored in two different buffers (b_1, b_2). Each buffer is assigned the number of tokens to be loaded into it and then increases the current buffer size $(b_1\_b\_s, b_2\_b\_s)$ of the corresponding buffers by the number of added tokens.

Output 1 Generation:
$b_1[b_1\_b\_s] \leftarrow o_0\_0$
$b_1[b_1\_b\_s] \leftarrow o_0\_1$
$b_1[b_1\_b\_s] \leftarrow o_0\_2$
$b_1[b_1\_b\_s] \leftarrow o_0\_3$
$b_1[b_1\_b\_s] \leftarrow o_0\_4$
$b_1[b_1\_b\_s] \leftarrow o_0\_5$
$b_1[b_1\_b\_s] \leftarrow o_0\_6$
$b_1[b_1\_b\_s] \leftarrow o_0\_7$
$b_1\_b\_s \leftarrow b_1\_b\_s + 8$
Output 2 Generation:
$b_2[b_2\_b\_s] \leftarrow o_1\_0$
$b_2[b_2\_b\_s] \leftarrow o_1\_1$
$b_2[b_2\_b\_s] \leftarrow o_1\_2$
$b_2[b_2\_b\_s] \leftarrow o_1\_3$
$b_2[b_2\_b\_s] \leftarrow o_1\_4$
$b_2\_b\_s \leftarrow b_2\_b\_s + 5$

Figure 5.13.: *pseudocode: how to store items in intern buffers*

The maximum length of this buffer is determined by the following two calculations, with the larger value being used: The product of the number of tokens produced on the edge and the repeat factor of the producing side (see section 2.6).
The product of the number of incoming tokens multiplied by two and the repeat factor of the consuming side. In addition, a variable is introduced that stores the current length of the buffer. This variable is set with an initial value of zero. In the next step, the algorithm checks which tokens are written to the original output port. These tokens are appended to the list in the order in which they were created. In the same step, the buffer length variable is updated according to the number of tokens added.

5.5.12. Priorities

The following section explains the concept of priorities, which plays a central role in the execution of the actions of actors in the algorithm. The original actors can perform different actions with different priorities. During the merging process, these priorities are transferred to the so-called 'meta-actor' and sorted in descending order. This means that the action with the highest priority is always inserted first, followed by the other actions in decreasing priority.
In order to execute the actions correctly, an if-else-else query is used to call up the appropriate action of an actor. By ensuring that the actions are sorted correctly, the original functionality of the actor is retained. This means that only the highest prioritised action is executed first, followed by the lower priorities, thereby maintaining the desired order of actions.
A practical example to illustrate this: Suppose an actor has three possible actions - A, B and C, with priorities $A > B$ and $B > C$ (where A is the highest priority). If this actor is inserted into the meta-actor, the actions are arranged in the order A (priority 1), B (priority 2) and C (priority 3). In the code, the various actions would then be checked one after the other, if necessary with guard conditions, and the action with the highest priority that fulfils all guard conditions would be executed fig.5.14.

```
if (A) then
    ...
else if (B) then
    ...
else (C) then
    ...
```

Figure 5.14.: *CAL Code - priority handling*

5.5.13. Network Generation

In this step, a new XML representation of the data flow network is created that integrates the newly introduced actor. This ensures that all relevant changes and adjustments are made in order to correctly map the new structure while maintaining the integrity of the network.
Firstly, the actor instances that are not involved in the merger are transferred unchanged to the new representation. These actors are simply copied so that their existing connections and properties are retained. However, the connections of these actors to the actors involved in the merger must be adapted. Instead of continuing to refer directly to the original actors, they are now redirected to the corresponding ports of the new actor. All attributes of the original connections are retained and transferred to the new connections. This ensures that communication in the network continues to function as expected.
For the actors involved in the merger, all parameter definitions are adopted and used for the new actor. The system checks whether the original parameter identifiers can be transferred to the new network. If this is not possible due to name conflicts or other factors, the parameter identifiers are adapted so that they are unique and conflict-free. These adjustments are necessary to ensure that each property can be uniquely identified, which is crucial for the rest of the network process.
Another important aspect of network generation is the management of port names and parameter names. These must be known and consistent at all times, as they can be adjusted during the process to avoid any naming conflicts. In this step, all port and parameter names are therefore carefully checked and changed if necessary to prevent collisions and ensure uniqueness in the network.
At the end of this step, the updated network is completely generated in the XML representation.

6. Experimental Evaluation

In the previous chapter, the developed algorithm was presented, the purpose of which is to merge actors in networks using a process called token buffering and to create a so-called meta-actor from this. This algorithm makes it possible to efficiently merge several actors into a single actor, which can lead to improved clarity and potential performance improvements, especially in complex networks.
In the following, the functionality of the algorithm is demonstrated using an exemplary network with nine actors. This network is processed step by step by the algorithm, whereby the merging process is described in detail. The result of this process is a single meta-actor that integrates all the functions and connections of the original actors. This demonstration is intended to illustrate the effectiveness and efficiency of the algorithm and its applicability to larger and more complex networks.
The developed algorithm is then tested with regard to its efficiency using a suitable data set. The runtimes determined in the process are compared with the existing RAM algorithm [30] and with the original runtime before the token buffering approach was applied. This comparison serves to quantitatively illustrate the improvements in terms of runtime reduction and resource utilisation and to evaluate the advantage of the newly developed algorithm over established methods.
After analysing the test results on the various benchmarks, the algorithm developed in this thesis is compared with the RAM algorithm. It is shown in which scenarios the respective algorithms show their strengths and under which conditions the application of one algorithm could be more favourable than the other.

6.1. Example

In order to explain the presented algorithm in detail, it is applied to a specific network from the orc-apps [36]. This is a modified version of the IIR-lowlevel.xml network (see fig.6.1), which was developed as a subnetwork of the DigitalFiltering project. This project was created by Jani Boutellier at the Department of Computer Science and Engineering at the University of Oulu, Finland.

The modification of the network includes in particular the addition of two extra nodes. Firstly, an additional duplication node was added, whose task is to duplicate incoming tokens, i.e. to generate two outgoing tokens from one incoming token. This duplication is crucial for the further processing of data

paths in the network and ensures that parallel calculations are possible. Secondly, the network was supplemented by a further right shift node, which is used to manipulate data values by means of shift operations.

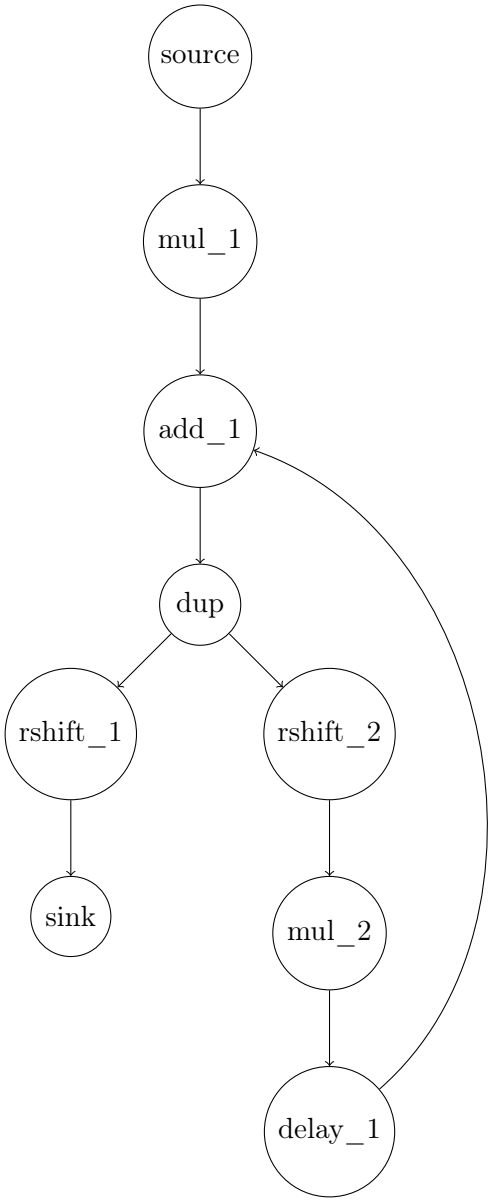


Figure 6.1.: *IIR-network*

A central characteristic of this network is the existence of a circle consisting of the nodes add_1, dup, rshift_2, mul_2 and delay_1. This loop forms a feedback loop that is run through repeatedly and thus enables cyclical data processing. The existence of such cycles poses particular challenges for the algorithm, as not only linear data flows but also recursive data structures must be handled correctly in order to achieve reliable results.

Input Actor	Output Actor	Token Overflow
source	mul_1	0
mul_1	add_1	1
delay_1	add_1	1
add_1	dup	0
dup	rshift_1	0
dup	rshift_2	0
rshift_1	sink	0
rshift_2	mul_2	0
mul_2	delay_1	1

Table 6.1.: *Token-Overflow IIR_lowlevel*

The network is made up of a total of nine individual actors. These actors each perform specific tasks in the data processing process and communicate with each other via directed edges that define the token flow. The algorithm used aims to combine these individual actors into a single meta-actor. This merging process is known as Actor Merging and serves to optimise the network by avoiding redundant calculations and making data flows more efficient. The CAL code of the individual actors and the XML file of the network can be found in the appendix fig.B.1. By applying the algorithm to this network, the aim is to check whether the algorithm is able to correctly calculate the maximum buffer size and optimise network performance, even with complex, cyclical structures.

The algorithm is now applied to the network, whereby the first step is to check whether the criteria for merging according to section 5.2 are fulfilled. These criteria are checked one after the other, and since all conditions are met in our example, the algorithm can continue with the calculation.

In the next step, the maximum buffer size is determined using the incidence matrix. In our network, this size is one. This maximum buffer size is crucial as it is later used to determine the maximum list length of the internal buffers, which in this example also has the value one. Once the maximum buffer size has been successfully calculated, the algorithm determines the token overflow between the individual actors to be merged. This is done for all internal edges of the network tab.6.1.

Now we can start integrating the actors to be merged. Since we are merging the entire network, the newly created meta-actor no longer has any outgoing or incoming edges. As all edges are in the merged network. We now go through the network in the sorted order of the actors and follow the algorithm step by step. The first actor that is added to the meta-actor is the original source actor fig.B.8.
If this actor is integrated into a meta-actor, the entire logic of the actor is transferred to the meta-actor. This includes both the initialisation logic and

the production logic. The meta-actor will be able to execute the `initialize` and `action` procedures for all merged actors to replicate the behaviour of the individual actors and perform the desired calculations correctly.

The native initialisation and production functions are adopted in the meta-actor so that it is able to obtain and process the corresponding data from the external source as intended in the original actor. The actual action of the actor is taken over and integrated into the new meta-action of the meta-actor.
The next step involves analysing the outgoing edges of the actor. The actor has a function with an outgoing edge that refers to the subsequent actor. In this example, this would be the `mul_1` actor. To understand the process, the outgoing edge is considered in more detail. The token overflow is calculated for this edge. The result shows that the token overflow is zero. This means that the number of outgoing tokens matches the number of incoming tokens. It follows that the outgoing token can be stored in a buffer variable in order to be processed directly in the next step. As this is an internal edge, the temporary storage is integrated into the action of the meta-actor fig.6.2.

```
gen_6 := native_source_produce ( tag ) + offset;
```

Figure 6.2.: *CAL Code - merged Actor source part*

The next actor is now integrated into the meta-actor. The next actor in the network, after the correct sorting of the actors to be merged, is the actor mul_1. This actor has both an input edge and an output edge. The mul_1 actor receives a token via its input edge and performs a multiplication with a constant number, which is inserted into the network as a parameter. The result of the calculation is then passed on to the output edge. This parameter, which defines the constant, is also taken from the network and integrated into the meta-actor as an input parameter.
The token overflow is checked for each incoming edge of the actor. In this case, the result is a token overflow of zero, which means that the number of incoming tokens matches the number of outgoing tokens. Therefore, no tokens need to be taken from a buffer. Instead, the previously calculated variable (in this case gen_6) can be used to continue the calculations.
The result of the multiplication is then calculated using the parameter and then the token is passed to the output edge. The output edge has a token overflow greater than zero, which means that a buffer is required to temporarily store the output tokens.
The algorithm now creates a list within the actor that describes the maximum number of buffers required to temporarily store the tokens. A variable that counts the number of elements in the buffer is initially set to zero. As soon as the result is calculated, the output token is added to the buffer and the

number of tokens contained in the buffer is increased to one fig.6.3.

```
x := gen_6;
buffer_mul_1_add_1[buffer_mul_1_add_1_size]
    := x * constant;
buffer_mul_1_add_1_size :=
    buffer_mul_1_add_1_size + 1;
```

Figure 6.3.: *CAL Code - merged Actor mul_1 part*

In the next step, the add_1 actor is integrated into the meta-actor. The add_1 actor has two input edges, both of which are checked in order to calculate the token overflow. The algorithm determines that the token overflow for both input edges is not equal to zero. This means that a buffer is required for each of the two edges to temporarily store the tokens.
The algorithm now checks whether there are enough tokens in the buffers so that the actors can be executed. If there are not enough tokens, the actor would not fire, which would block the calculations in the network. However, if there are enough tokens in the buffer, these tokens are used for the calculations. The add_1 actor loads a token from the corresponding buffer for each of the two input edges and performs the addition. The calculation is added as part of the action of the meta-actor, whereby the loaded tokens are used as input values.
Following these calculations, the result is stored on the output edge of the add_1 actor. As the token overflow for this edge is greater than zero, a buffer is also required here to store the result. The algorithm creates a buffer for the output edge and stores the calculated token there fig. 6.4.

```
if buffer_mul_1_add_1_size - 1 >= 0 &&
        buffer_delay_1_add_1_size - 1 >= 0 then
            x := buffer_mul_1_add_1[0];
            foreach int gen_15 in 0 ..
                buffer_mul_1_add_1_size - 1 do
                    buffer_mul_1_add_1[gen_15] :=
                        buffer_mul_1_add_1[gen_15 + 1];
            end
            buffer_mul_1_add_1_size :=
                buffer_mul_1_add_1_size - 1;
            y := buffer_delay_1_add_1[0];
            foreach int gen_18 in 0 ..
                buffer_delay_1_add_1_size - 1 do
                    buffer_delay_1_add_1[gen_18] :=
                        buffer_delay_1_add_1[gen_18 + 1];
            end
            buffer_delay_1_add_1_size :=
                buffer_delay_1_add_1_size - 1;
           gen_19 := x + y;
           run := true;
```

Figure 6.4.: *CAL Code - merged Actor add_1 part*

For the other actors, the algorithm runs analogue to the three actors already presented. In contrast to the actors discussed so far, delay_1 has several states, which means that the integration process must be extended to include the management of these states. The handling of states is a central component of the correct functionality of the meta-actor, as the current state of the actor directly influences how tokens are loaded and saved.
In order to correctly manage the states of the delay_1 actor in the meta-actor, a special variable is introduced that stores the current state of the delay_1 actor. This state variable is continuously updated during the execution of the meta-actor when the state of the delay_1 Actor is changed due to the executed actions. The state change is based on defined transition rules that determine how the actor reacts to incoming tokens and produces new tokens depending on the state.
The loading of the input and output tokens is also influenced by the current state of the delay_1 actor. The algorithm first checks which state is active and uses this to decide whether and how many tokens need to be loaded from the buffers. This decision is based on the predefined rules that describe how the delay_1 actor behaves in different states. Depending on which state is active, tokens may either be taken directly from the buffers or new tokens may be generated and saved.
As delay_1 has different states, the number of incoming and outgoing tokens

can vary. This requires the provision of buffers for both the inputs and the outputs to ensure that tokens are loaded and saved correctly. The algorithm creates separate buffer variables for this, the size of which is determined based on the maximum number of tokens required during the process.
A significant difference to the previously discussed actors is that with the delay_1 actor, the state variable (state) is adjusted accordingly after a state change has been successfully executed. This means that after each executed action, it must be checked whether a state change has taken place and, if so, the state variable must be updated. This mechanism ensures that the meta-actor correctly maps the internal state of the delay_1 actor at all times and can react to changes fig. 6.5.

```
if (delay_1_state = "s_delay") && count < delay then

    count := count + 1;
    delay_1_state := "s_delay";
    buffer_delay_1_add_1[buffer_delay_1_add_1_size] := value;
    run := true;
else if (delay_1_state = "s_delay") then
    if buffer_mul_2_delay_1_size - 1 >= 0 then
        x := buffer_mul_2_delay_1[0];
        foreach int gen_46 in 0 ..
            buffer_mul_2_delay_1_size - 1 do
                buffer_mul_2_delay_1[gen_46] :=
                    buffer_mul_2_delay_1[gen_46 + 1];
        end
        buffer_mul_2_delay_1_size :=
            buffer_mul_2_delay_1_size - 1;
        delay_1_state := "s_run";
        buffer_delay_1_add_1[buffer_delay_1_add_1_size]
            := x_3;
        buffer_delay_1_add_1_size :=
            buffer_delay_1_add_1_size + 1;
        run := true;
    end
else if (delay_1_state = "s_run") then
    if buffer_mul_2_delay_1_size - 1 >= 0 then
        x := buffer_mul_2_delay_1[0];
        foreach int gen_46 in 0 ..
            buffer_mul_2_delay_1_size - 1 do
                buffer_mul_2_delay_1[gen_46] :=
                    buffer_mul_2_delay_1[gen_46 + 1];
        end
        buffer_mul_2_delay_1_size :=
            buffer_mul_2_delay_1_size - 1;
        delay_1_state := "s_run";
        buffer_delay_1_add_1[buffer_delay_1_add_1_size]
            := x_3;
        buffer_delay_1_add_1_size :=
            buffer_delay_1_add_1_size + 1;
        run := true;
    end
else if (delay_1_state = "s_init") then

    count := 0;
    delay_1_state := "s_delay";
    run := true;
end end end end
```


Figure 6.5.: *CAL Code - merged Actor delay_1 part*

6.2. Speed Improvements

A key approach to optimising the existing algorithm is to reduce the number of queries regarding the current state of an actor. Currently, each state of an actor is defined within the Do part of the actor, which results in the algorithm performing a separate if query for each individual state. This problem is further exacerbated if there are multiple actions within a single state. In such cases, the algorithm must check for each action whether it may be executed in the current state, which leads to a potentially large number of if queries that not only make the code more confusing, but also lead to increased computing times.
To reduce these inefficiencies, the algorithm could be restructured so that the state logic is reintegrated into the FSM scheduling. By controlling the state changes and their associated actions centrally within the FSM scheduling, the number of queries required can be significantly reduced. Such a change would have several potential benefits. Firstly, bringing the state logic back into FSM scheduling avoids the need to manually check each action.
Instead, the state decision is centralised, which reduces the number of if queries. Secondly, the lower computing effort would potentially improve the speed of the algorithm. Thirdly, the simplification of the code through the reduction of if queries contributes to better readability and maintainability. An additional advantage could be that the clear separation between state logic and action execution also simplifies the parallelization of the algorithm.

The code could be further optimised by replacing the For loops currently used to remove an element from the list with a more efficient method. Instead of searching through the entire list each time, two additional variables could be introduced to manage the buffer more efficiently. The first variable would store the position of the first element in the buffer, while the second variable would mark the position of the last element. This system enables faster access to the elements in the buffer, as only the position stored in the first variable is accessed when an element is removed. After access, the first variable is incremented by one to refer to the next element. As soon as this variable reaches the end of the buffer, it is reset to zero in order to realise a circular buffer.
Similarly, elements are added to the buffer using the second variable, which always saves the position of the last inserted element. When a new element is added, the element is saved at the position of the second variable, which is then incremented by one. As soon as this variable reaches the end of the buffer, it is also reset to zero in order to continue filling the buffer.
This procedure significantly reduces memory access, as only the two position variables need to be updated instead of searching through the entire list each time. As extensive memory operations are no longer required, not only is the execution speed significantly increased, but memory management is also simplified.
This reduction in memory accesses leads to a higher efficiency of the algorithm, as the overhead caused by unnecessary loops and memory operations is min-

imised. In addition, this implementation enables faster processing of buffer operations, which can have a positive effect on the overall performance of the system.

6.3. Measurements

For the evaluation of the optimisations developed in this thesis, extensive experiments are carried out with four different, slightly adapted test data sets from the open source project ORC-Apps [36]. The aim of these experiments is to quantify the efficiency of the optimisations in terms of runtime improvement.
The tests run on a Raspberry Pi 3 equipped with a Broadcom BCM2711 SoC. This system is based on a quad-core Cortex-A72 (ARM v8) 64-bit architecture with a clock speed of 1.8 GHz, has 4 GB of RAM and uses Raspberry Pi OS as its operating system. This platform was chosen as it is a representative environment for resource-constrained embedded systems, where efficient code generation and runtime optimisation play a crucial role. [47]
The measurements utilise a compiler developed by Krebs that translates CAL code into C++ [29]. This compiler serves as the starting point for analysing the runtime changes. The standard compiler flags were used throughout the entire evaluation process, with no explicit modifications made to optimize the compilation further. First, the original runtime of the test cases is measured without the optimisations developed in this work. The same code is then executed with the optimised algorithm in order to analyse the effects of the improvements on the overall performance. Additionally, for all test datasets, the channel size was set to 512, ensuring a consistent configuration across all benchmarks. The measured values obtained are documented in the following tabular overview tab. 6.2.

	IIR_Lowlevel	FFT_8	Top_ZigBee	Predistortion
OM	1.884s	27.143s	2.366s	20.298s
ITB	0.873s	5.767s	0.085s	30.249s
RAM	-	16.408s	-	25.435s

Table 6.2.: *Comparison of Network Configurations*

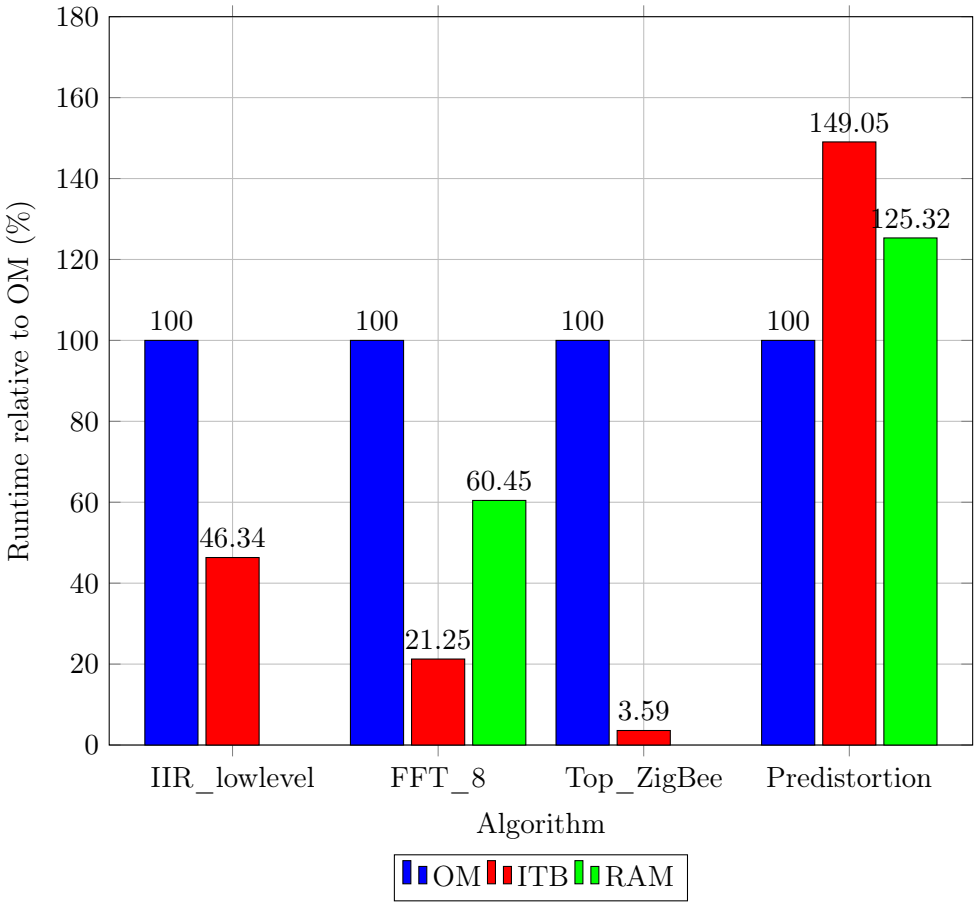


Figure 6.6.: *Comparison of the Runtimes of the various Benchmarks depending on the Algorithm used.*

6.4. Analysis

This section compares the performance of the two algorithms ITB and RAM with the original network of actors that are not merged. These two algorithms are compared with each other based on their performance in four different benchmarks:

- IIR_Lowlevel,
- FFT_8,
- Top_ZigBee,
- Predistortion.

The section focuses in particular on comparing the runtimes in order to identify the strengths and weaknesses of each algorithm. The performance of OM, the original code, is compared with the two optimised variants, ITB and RAM, which merge the original network into a meta-actor. It is investigated how the

runtime of the original network changes in comparison to the merged networks of the two optimised algorithms.

6.5. Algorithm Overview

- **OM (Original Code):** The OM represents the original network, which is operated without specific optimisations and adaptations. It serves as a reference for the other two algorithms. The original code from the orc-apps is only adapted to the extent that the network of actors functions correctly with the specified compiler.
- **ITB (Internal Token Buffering):** ITB (see chapter 5) is a specially developed algorithm that aims to improve performance. This is achieved by implementing internal token buffering and merging several actors into one meta-actor. These optimisations are intended to make the calculations more efficient and significantly reduce the runtime.
- **RAM:** The RAM [30] algorithm is also based on the idea of merging different actors into one meta-actor. However, an important limitation of the RAM algorithm is that it cannot process feedback loops and cycles. Therefore, it is not suitable for all benchmarks. This explains why no measured values are available for some benchmarks such as ZigBee and IIR_Lowlevel.

6.5.1. Comparison of the Algorithms for each Benchmark

The following benchmarks were selected for the analysis: IIR_Lowlevel, FFT_8, Top_ZigBee and Predistortion. The following table 6.2 shows the runtimes of the various algorithms in seconds. The relative percentages are also compared in order to evaluate the performance of the algorithms fig.6.6.

IIR_Lowlevel (Digital Filtering)

- OM: 1.884 seconds
- ITB: 0.873 seconds
- RAM: No measurement available

Analysis: The IIR_Lowlevel benchmark is crucial for the basic initialisation and represents an important reference. The OM algorithm with a runtime of 1.884 seconds is the base value here. At 0.873 seconds, the ITB algorithm achieves a significant improvement of 53.7%. This reduction in runtime suggests that ITB is able to achieve significantly faster execution through internal token buffering and possibly optimised calculations. Unfortunately, no measured values are available for the RAM algorithm because it contains feedback loops and thus does not fulfill the merging criteria. So this benchmark cannot be included in the comparison.

FFT_8 (Fast Fourier Transform)

- OM: 27,143 seconds
- ITB: 5,767 seconds
- RAM: 16,408 seconds

Analysis: The FFT_8 benchmark requires an intensive calculation to perform the fast Fourier transform (FFT). The ITB algorithm performs very well with a runtime of 5.767 seconds and shows an improvement of 78.8% compared to OM (27.143 seconds). This improvement can be explained by the optimised token buffering and parallel processing in ITB, which make it possible to make the FFT calculations more efficient. The RAM algorithm with 16.408 seconds lies between OM and ITB and shows an improvement of 39.5% compared to OM. Although a significant optimisation is achieved compared to the original code, RAM does not come close to the performance of ITB, which indicates the lack of optimisations in the area of parallel data processing and memory access.

Top_ZigBee (ZigBee Network Algorithm)

- OM: 2.366 seconds
- ITB: 0.085 seconds
- RAM: No measurement available

Analysis: The Top_ZigBee benchmark simulates the network operations of a ZigBee system. In this case, ITB shows an outstandingly fast runtime of just 0.085 seconds. This represents an impressive improvement of 96.4% compared to OM, which is 2.366 seconds. This enormous acceleration suggests that ITB is able to handle network communication extremely efficiently through optimised data buffering and parallel processing. As no measured values are available for the RAM algorithm because not all merging criteria are fullfilled. This benchmark cannot be compared further.

Predistortion (Predistortion for Signal Processing)

- OM: 20,298 seconds
- ITB: 30,249 seconds
- RAM: 25,435 seconds

Analysis: The Predistortion benchmark shows an interesting deviation from the results of the other benchmarks. The OM algorithm with a runtime of 20,298 seconds represents the best performance compared to the other two algorithms. ITB takes longer at 30,249 seconds, which indicates the lack of

significant optimisations in the area of token buffering and parallel processing for this particular calculation. The RAM algorithm with 25,435 seconds shows an improvement of 25% compared to ITB and can be considered the most efficient algorithm in this particular use case. These results indicate that the optimisations implemented in ITB are not equally beneficial in all scenarios and may even deliver poorer performance for certain calculations.

6.5.2. Summary and findings

The detailed analysis and the results of the benchmarks highlight the significant differences in performance between the three algorithms tested: OM, ITB and RAM. These differences provide valuable insights into the strengths and weaknesses of each algorithm, particularly in terms of their ability to perform different tasks efficiently. By comparing the algorithms on the IIR_Lowlevel, FFT_8, Top_ZigBee and Predistortion benchmarks, important insights can be gained into the specific optimisations and their impact on runtime.
ITB shows the best overall performance in most benchmarks and represents the most efficient algorithm for a variety of scenarios. Of particular note are the FFT_8 and Top_ZigBee benchmarks, where ITB achieves significant improvements over the other algorithms. The use of token buffering and parallel processing proves to be extremely advantageous. These optimisations enable ITB to complete the tasks significantly faster and more efficiently.
These results indicate that ITB is the ideal choice for scenarios that benefit from optimised data buffering. Especially for tasks that require a high number of calculations and fast data flows, ITB shows outstanding efficiency, which can be crucial in many modern applications.

Although the RAM algorithm also delivers good results in some benchmarks, it cannot match the efficiency of ITB. Especially in the benchmarks FFT_8 and Predistortion, RAM achieves a better performance compared to OM, but does not achieve the same performance as ITB. This is mainly due to the different optimisation approaches that distinguish RAM from ITB in section 6.7.

OM serves as a basic reference and shows the performance of the original code without special optimisations. The analysis confirms that the original code is significantly slower than the two optimised algorithms without additional adjustments and improvements. Particularly in the benchmarks that require intensive calculations and large amounts of data, such as FFT_8 and Top_ZigBee, OM performs significantly worse in terms of runtime. This underlines the need to implement optimisations such as token buffering and parallel processing in order to increase the efficiency and speed of the calculations.
However, the performance of OM is an important starting point to measure the impact of optimisations. The improvement from OM to ITB and RAM shows impressively how powerful the optimisations can be, especially for complex and data-intensive computations.

ITB as the preferred choice, but RAM as a useful alternative Overall, it is clear that ITB is the best choice for most benchmarks. The drastic improvement on most benchmarks, especially FFT_8 and Top_ZigBee, shows that ITB is the best optimised solution for tasks that require high parallelism and fast data processing. Targeted optimisation through token buffering and parallel computation has made ITB a highly efficient algorithm that excels in modern applications that have such requirements. Nevertheless, it is important to emphasise that the choice of the right algorithm depends not only on the performance in the benchmarks, but also on the specific requirements of the application. In scenarios where ITB's optimisations do not deliver the desired results or where specialised calculations are required, RAM can be a valuable alternative. Especially in the case of predistortion, RAM shows that it can provide a more efficient solution than ITB in certain applications. RAM could also be relevant in future applications if its limitations, such as the lack of feedback loops and cycles, are not critical for a particular task. To summarise, ITB emerges as the most efficient solution for the majority of benchmarks. However, RAM could be useful in specific scenarios where ITB's optimisations do not deliver the desired results. Continuous fine-tuning and adaptation of the algorithms to specific requirements could lead to further performance gains in the future and provide the optimal algorithm for an even wider variety of applications.

6.6. Cache Misses and Their Impact on Performance

In this section, the focus is on the impact of cache misses during the execution of the two algorithms (ITB, RAM) on the performance of the benchmarks in comparison to the original code. Cache misses occur when the required data is not found in the cache, forcing the processor to access slower main memory. This increases the latency of data retrieval and can significantly degrade performance, especially for computationally intensive tasks. By measuring cache misses and cache references during the execution of the benchmarks, it can gain insights into the efficiency of each algorithm in terms of memory access and data locality.

Cache misses can be particularly detrimental in high-performance scenarios, as frequent memory accesses can hinder the benefits of parallel processing and optimised calculations. Therefore, understanding and reducing cache misses is crucial for improving runtime performance.

6.6.1. Cache Misses for Each Benchmark

The following cache miss statistics were collected for each of the algorithms across the four benchmarks: IIR_Lowlevel, FFT_8, Top_ZigBee, and Predistortion.

IIR_Lowlevel (Digital Filtering)

For the IIR_Lowlevel benchmark, cache miss statistics for the original algorithm (OM) and the optimised ITB and RAM algorithms are as follows:

- **OM:**
  - Cache misses: 5,254,790
  - Cache references: 2,206,100,829
  - Cache miss rate: 0.24% of all cache references
- **ITB:**
  - Cache misses: 227,784
  - Cache references: 866,529,334
  - Cache miss rate: 0.03% of all cache references

Analysis: The ITB algorithm demonstrates a significant reduction in cache misses compared to OM. With a miss rate of only 0.03% (compared to 0.24% for OM), ITB's optimisations, such as internal token buffering, seem to enhance data locality and reduce memory accesses. This contributes to the overall performance improvement in IIR_Lowlevel, where ITB achieves a 53.7% runtime reduction (see section 6.5.1).

FFT_8 (Fast Fourier Transform)

For the FFT_8 benchmark, the cache miss statistics are:

- **OM:**
  - Cache misses: 2,131,585,702
  - Cache references: 46,318,681,059
  - Cache miss rate: 4.60% of all cache references
- **ITB :**
  - Cache misses: 38,396
  - Cache references: 9,135,176,107
  - Cache miss rate: 0.00% of all cache references
- **RAM:**
  - Cache misses: 36,581
  - Cache references: 36,600,833,578
  - Cache miss rate: 0.00% of all cache references

Analysis: The cache miss rate for OM is significantly higher at 4.60%, compared to the near-zero miss rates for ITB (0.00%) and RAM (0.00%). This suggests that ITB and RAM both perform exceptionally well in terms of memory access efficiency, significantly reducing cache misses compared to the original code. The reduction in cache misses likely contributes to the improved performance of these optimised algorithms, as lower cache miss rates result in faster data retrieval and overall quicker execution times.

Top_ZigBee (ZigBee Network Algorithm)

For the Top_ZigBee benchmark, the cache miss statistics are as follows:

- **OM:**
  - Cache misses: 328,529
  - Cache references: 2,263,803,158
  - Cache miss rate: 0.01% of all cache references
- **ITB:**
  - Cache misses: 23,992
  - Cache references: 50,465,552
  - Cache miss rate: 0.05% of all cache references

Analysis: In the case of Top_ZigBee, the cache miss rate for ITB is slightly higher than for OM, but the overall number of cache misses is still relatively low. Despite this, ITB achieves an impressive performance improvement due to its optimised data buffering and parallel processing. The higher miss rate may reflect the increased complexity of handling network communication, but it is still not the primary factor limiting performance.

Predistortion (Signal Processing)

For the Predistortion benchmark, the cache miss statistics are:

- **OM:**
  - Cache misses: 1,197,713,129
  - Cache references: 58,347,548,118
  - Cache miss rate: 2.05% of all cache references
- **ITB:**
  - Cache misses: 1,624,904,865
  - Cache references: 39,934,061,423
  - Cache miss rate: 4.07% of all cache references
- **RAM:**

- Cache misses: 1,443,346,401
- Cache references: 34,469,610,177
- Cache miss rate: 4.19% of all cache references

Analysis: The Predistortion benchmark reveals an interesting scenario where both ITB and RAM have significantly higher cache miss rates than OM, with ITB having a miss rate of 4.07% and RAM slightly higher at 4.19%. This suggests that the optimisations made in ITB and RAM, which focus on improving the overall algorithmic performance through parallelism and actor merging, may involve complex memory access patterns that increase cache misses. However, the overall performance of ITB and RAM still improves, indicating that other factors, such as parallelism and task decomposition, outweigh the increased memory access costs.

6.6.2. Summary of Cache Miss Analysis

From the analysis of cache misses across the four benchmarks, several important observations can be made:

- In benchmarks such as IIR_Lowlevel and Top_ZigBee, ITB demonstrates a significant reduction in cache misses compared to the OM algorithm. This suggests that ITB's optimisations, such as internal token buffering, result in better memory access patterns and enhanced performance.
- For the FFT_8 benchmark, the original code (OM) exhibits a relatively high cache miss rate of 4.60%, which indicates that memory access is a significant bottleneck. The higher cache miss rate in the original code suggests that data retrieval from memory is slower, which likely contributes to the longer runtime observed in the OM case. In contrast, both ITB and RAM exhibit very low cache miss rates, with values near 0.00%, indicating that memory access is no longer a major limiting factor in these optimised versions. This reduction in cache misses suggests that both optimised algorithms handle memory more efficiently.
- The Predistortion benchmark shows that despite a higher cache miss rate for ITB and RAM, both optimised algorithms still achieve better performance than OM. This highlights the trade-off between optimised computation and increased memory access costs, where the algorithmic improvements outweigh the memory penalties in terms of overall performance.

6.6.3. Further Insights on Cache Efficiency

The analysis of cache misses across the evaluated benchmarks not only sheds light on the memory access behaviour of the different algorithms but also allows for a deeper interpretation regarding their overall cache efficiency. This concept

refers to the effectiveness with which the cache hierarchy is utilised—i.e., how often the processor finds the needed data in faster cache levels without resorting to slower main memory. More formally, cache efficiency can be expressed as the ratio between the number of successful cache accesses (hits) and the total number of memory accesses (cache references), and is thus inversely related to the cache miss rate. [15]

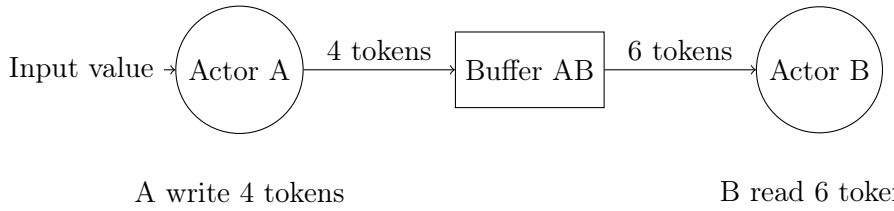
Particularly in benchmarks such as *IIR_ Lowlevel* and *FFT_ 8*, the optimised algorithms ITB and RAM achieve a markedly reduced cache miss rate compared to the original implementation, despite handling a considerable volume of cache references. This observation points to a more favourable memory access pattern, likely characterised by higher spatial and temporal locality. The underlying algorithmic strategies—such as internal token buffering in ITB and actor merge in RAM—appear to structure memory usage in a way that promotes repeated access to nearby or recently used data. As a result, the cache is populated with data that is more likely to be reused shortly thereafter, leading to a more efficient utilisation of the memory hierarchy and ultimately to improved runtime performance.

Moreover, the significant drop in cache miss rate in *FFT_ 8* (from 4.60% in OM to effectively 0.00% in both ITB and RAM) suggests that, in certain computational patterns, the applied optimisations are particularly effective in harmonising memory access behaviour with the underlying cache architecture. This alignment reduces memory access latency and mitigates cache-induced performance bottlenecks, thereby increasing overall throughput in data movement–dominated scenarios. A critical contributing factor to this effect is the improved compatibility of token production and consumption rates between actors after merging. By aligning these token rates, the need for intermediate buffering and excessive synchronisation is significantly reduced, which further streamlines the data flow and enhances pipeline efficiency.

6.7. Comparison ITB and RAM Algorithm

In this section, the differences between RAM algorithm [30] and the algorithm with internal token buffering presented in this paper are explained in detail.
A key difference between the two approaches is the treatment of feedback loops and cyclic structures (cycles) within the network. While RAM cannot completely merge such network topologies, internal token buffering allows these structures to be fully integrated. This is a significant limitation of the RAM algorithm, as networks with circles and feedback loops cannot be merged efficiently.
Another key drawback of RAM is the handling of mismatching token rates. In this case, RAM uses the repeat statement of the CAL actor language to compensate for the different production and consumption rates of the actors. This means that a repeat vector is calculated that determines how often individual actors must be executed in succession to ensure a balance of token rates. However, this method can be problematic, especially if the system does not

continuously feed in new tokens. In such cases, there is a risk of tokens being lost within the network. To illustrate this, consider the following example: An actor A reads an input value, produces four tokens after each processing and passes these on to its output. An actor B consumes six tokens each time and then generates an output value.



In this scenario, RAM would construct the meta-actor in such a way that it is only activated when at least three input values are available. This results from the fact that the first actor A must fire three times to produce a total of twelve tokens so that actor B can then fire twice. However, if the system only provides two input values, the meta-actor created remains inactive as the required token quantities cannot be achieved.
The algorithm with internal token buffering works in a similar way, but differs with regard to the handling of excess tokens. Although there is no output after the first input value has been processed, an output value is generated after the second input value has been read in. The two excess tokens are temporarily stored in an internal buffer and taken into account in the next calculation. [30]
A particularly significant advantage of internal token buffering becomes apparent when the lowest common multiple (LCM) of the production and consumption rates is required as a synchronisation variable. In RAM, this can in the worst case lead to a first output value being written only after $m \cdot n$ if $m = n + 1$. The algorithm with internal token buffering, on the other hand, starts generating output values after just two input values, which significantly increases the efficiency of the calculation and prevents all output values from being generated at the same time.
At first glance, one might assume that RAM is significantly more efficient due to the use of the repeat statement, especially for networks with small repetition factors in the repeat vector. This is because the repetition strategy could theoretically achieve a closer interlocking of the execution sequence of the actors. However, RAM also uses arrays internally to temporarily store values. This means that both read and write accesses to the memory are required - similar to the algorithm with internal token buffering presented in this paper. Accordingly, the use of the repeat statement alone does not result in a fundamental efficiency advantage.
However, a potential advantage of RAM could be that the stored values in the system are kept closer to the processor than in the internal buffers of the token buffering approach. This could have a positive effect on memory access times and thus enable a higher processing speed in certain scenarios.
On the other hand, the type of memory management in RAM may also repre-

sent a significant disadvantage. As the elements in the buffer are continuously shifted, this may require additional computation and memory access operations. However, this aspect could be significantly improved in real implementations through targeted optimisation, as described in section 6.2. [30]

6.7.1. Combination of ITB and RAM

In the previous sections, two fundamentally different approaches to the merging of CAL actors were presented: firstly, the algorithm (see chapter 4), which efficiently generates periodic schedules for static data flow networks, and secondly, the ITB (see chapter 5) approach presented in this paper, which offers significantly greater flexibility in dealing with dynamic or feedback network structures through internal token buffering.
Both methods have specific strengths, but also have limitations that restrict their respective areas of application. RAM is particularly suitable for regular, deterministic networks with static token rates and without cyclic dependencies, while the ITB approach can also be used in more complex, not completely static scenarios due to its dynamic buffer management.
The aim of this section is therefore to develop a concept that combines the advantages of both approaches in a common merge algorithm. By combining the efficiency of RAM scheduling model with the structural adaptability and expressive power of the ITB approach, a hybrid merge mechanism is created that makes it possible to transform even complex and real-world actuator networks into high-performance meta-actors without having to take special restrictions in the network structure into account. The combined algorithm should not only be highly efficient, but also versatile.
The functional principle of this method is described in detail below. The procedure is based on an iterative analysis of the network to be fused and includes both the automatic selection of the most suitable merging procedure and the targeted creation of meta-actors in order to avoid structural obstacles such as feedback loops or rate incompatibilities. This enables a continuous merge process that continues until the entire network is either merged into a meta-actor or both methods no longer allow a valid merging.

The combined merge algorithm follows a step-by-step, iterative procedure in which the strengths of both approaches - the RAM algorithm and the ITB method developed in this thesis - are systematically interlinked. At the beginning of the process, the entire actuator network is analysed in order to identify all actors that can potentially be merged with each other. For each possible group of actors, it is first checked whether the merging criteria of the RAM algorithm are fulfilled.
If all these conditions are met, the merging can take place directly according to RAM efficient procedure. This makes it possible to generate a deterministic, periodic schedule for the resulting subnetwork, which can be executed with minimal memory requirements and high computational efficiency. In many regular or previously well-structured networks, this already leads to considerable optimisation.

However, in more realistic, complex or dynamic systems, it is often observed that certain parts of the network violate one or more of these merging criteria. Typical examples of this are feedback loops, non-uniform or data-dependent token streams and actors with internal state machines that affect their communication behaviour. In these cases, the combined algorithm automatically resorts to ITB, which is characterised by its greater structural flexibility.
The affected sub-networks are considered in isolation and merged as so-called meta-actors. Within these meta-actors, explicitly modelled, limited buffers ensure that non-deterministic or dynamic communication patterns are also correctly mapped. At the same time, the memory requirements remain calculable and manageable thanks to the controlled limitation of the internal buffers.
Once such an ITB-based meta-actor has been created, the entire network is analysed again - now including the new structures. The aim is to check whether the reduction of complexity in the network - for example by removing loops or standardising communication paths - means that the merging criteria for RAM algorithm are now fulfilled after all. If this is the case, the previously problematic subnetwork can now also be merged using the more efficient scheduling approach.

7. Related Work

The literature on DPNs includes various approaches for improving the efficiency and implementation of DPNs. In particular, techniques for combining, merging and optimising actors are examined. In his work 'A Machine Model for Dataflow Actors and its Applications', Janneck presents a machine model for general dataflow programmes that serves as a basis for their efficient implementation. The focus is on finding a balance between expressiveness and efficiency, especially for dynamic dataflow programmes. [24] Another paper by Janneck, 'Actors and their Composition', describes a framework for describing actor compositions and models of computation that is independent of specific modelling languages. The focus is on analysing actor compositions and their properties. [23]
In 'Classification and Transformation of Dynamic Dataflow Programs', Wipliez and Raulet present methods for classifying actors, defining static, cyclic-static and quasi-static categories. Although they present approaches for summarising such actors, no general method for the merging of actors is described [52].
In 'Analysis of SystemC Actor Networks for Efficient Synthesis', Falk examines efficient synthesis techniques for dataflow graphs. In particular, the use of static planning techniques to optimise static parts of a network is emphasised. The approach is extended to graphs with bounded buffers to enable synthesis without dynamic memory allocation .[18]
Ersfolk, in 'Modelling Control Tokens for Composition of CAL Actors', addresses the modelling of control tokens to support the composition and scheduling of dynamic dataflow programs. By analysing control token paths, it is determined whether data dependencies exist or only distribution information is passed on. This enables more efficient planning and composition. [17]
Lee and Parks' fundamental work "Dataflow Process Networks" describes an analytical model for dataflow networks with data-dependent control flow. They define various consistency criteria and develop conditions for scheduling executions with limited memory. In addition to the state enumeration procedure, a clustering technique is also presented to reduce graphs to standard control structures. [33]
In their work 'Actor Merging for Dataflow Process Networks', Boutellier et al. present a method that makes it possible to efficiently merge dynamic dataflow actors. Tokens are buffered internally between actions, with feedback and token buffering being recognised automatically. In the event of inconsistencies between token production and consumption, an internal FIFO buffer is automatically added, which the Orcc compiler optimises where possible. In particular, self-referenced FIFOs with a size of 1 are replaced by state variables, which leads to an acceleration of the software. However, the responsibility for

avoiding overflows or underflows lies with the user. The handling of feedback loops and cycles remains rather superficial and incompletely explained. [5]
In his work, RAM is a method for the merging of CAL actors known as 'repeat Actor merging'. This paper first demonstrates which CAL functions are supported and which are not included in Actor Merging. It then explains the merging criteria that must be met for Actor Merging to be possible. The merging process comprises several steps: First, the merging approach is described, which includes the path enumeration and configuration of the merged data flows, followed by the determination of fusability. The generation of FSMs and priorities is then initiated, which finally leads to the creation of the merged actions, actors and the newly generated network. RAM has the disadvantage that it cannot be used to merge feedback loops and circles. [30].

8. Conclusion and Future Work

As part of this work, a novel algorithm was developed for merging actors that are modelled in the CAL programming language and communicate with each other within DPNs. The aim was to develop a method that allows several actors to be merged into larger units - so-called meta-actors - while retaining the original network functionality. The focus here was particularly on the use of internal buffers, which take on the task of ensuring communication between previously separate actors within this meta-structure. This enables more efficient execution on systems with limited computing resources without compromising the semantic correctness of the overall system.
A key result of this work is the ability of the presented algorithm to successfully merge even networks with complex structures - especially those with feedback loops and cyclic dependencies. Previous approaches often failed due to precisely these challenges, as they were either unable to manage internal states correctly or were unable to reliably map cyclical data flows. The algorithm developed here addresses these limitations with a combination of state analysis, guard conditions and targeted internal token buffering. In addition, merge templates were introduced, which merge frequent constellations in a standardised way and thus enable further automation and standardisation of the merging process.
To validate the approach, the algorithm was tested on a large number of established benchmarks. The results show significant performance gains, especially in comparison to existing methods such as the RAM algorithm. The newly developed ITB approach proved to be particularly effective for benchmarks with high demands on parallelism and data processing speed - such as FFT_8 and Top_ZigBee. There, ITB was able to achieve a significant acceleration of the overall execution through targeted optimisations in the token flow and parallel processing steps.
Despite the superior performance of ITB in most cases, it should be noted that the choice of the optimal merging algorithm should always be made in the context of the respective application. The RAM algorithm proved to be a more efficient alternative in certain scenarios, such as specialised predistortion tasks. This emphasises the relevance of a differentiated consideration of the requirements and the respective network architecture. Overall, this work represents a significant contribution to the optimisation of data flow-oriented systems. The presented algorithm combines flexibility, scalability and correctness in an integrative approach that is both theoretically sound and practically applicable. ITB's ability to merge complex networks including feedback structures opens up new possibilities for the development of powerful DPN applications - both in software-based and hardware-accelerated systems.

One possible outlook for future work is the further generalisation and refinement of the merge templates developed. The aim should be to abstract these templates in such a way that they can be applied to a wider range of different network structures and provide a robust, reusable basis for various application scenarios. Ideally, these templates could be integrated into existing toolchains for automated code generation, for example as a component of compilers or optimisation tools for CAL-based applications.
This would significantly reduce the development effort and further increase the efficiency of generating optimised data flow networks.
Another interesting research path concerns the analysis and optimisation of energy consumption, especially when executing the merging process on embedded systems with limited resources. While the present work has focused on runtime optimisations, future studies could analyse the energy efficiency of the generated meta-actors and identify ways in which targeted actor merging can also reduce energy requirements - which would be particularly crucial in mobile or energy self-sufficient systems. In addition, the extension of the method to heterogeneous system architectures offers a further promising field of development. Systems with GPU or FPGA acceleration typically have very different requirements in terms of data layout, parallelisation and communication strategies.
Adapting the algorithm to such architectures could open up new optimisation potential, as the parallelisation options in particular could be adapted to the respective hardware structure through targeted merging. For example, certain computationally intensive actor combinations could be specifically mapped to specialised FPGA clusters or GPU cores, while less critical paths remain on classic CPU cores.
A particularly exciting and challenging aspect of future work would be the complete automation of the merging strategy in relation to the target architecture of the system. Specifically, the algorithm could be extended to automatically analyse how many computing units (e.g. CPU cores) are available and then merge the actor network in such a way that the resulting meta-actors can be optimally distributed across these resources. The algorithm should not only pay attention to numerical adjustment, but also ensure that the computing load is distributed as evenly as possible between the meta-actors in order to avoid load imbalance and utilise the resources efficiently.
Such an intelligent merge algorithm would therefore have to perform a workload analysis of the original actors, analyse their execution time and communication patterns and perform a merging based on this data that is both hardware-aware and load-balanced. Dynamic profiling or static heuristics could serve as a basis here, for example to prevent a single meta-actor from becoming a bottleneck while others remain largely inactive. Such functionalities would significantly enhance the algorithm and extend its potential applications to systems with widely varying hardware properties - from energy-efficient IoT devices to high-performance clusters.
Overall, this opens up a broad field for further research that encompasses

both theoretical foundations - for example in the formal modelling of optimal merging strategies - and practical aspects such as integration into existing development tools, use on heterogeneous architectures or automatic hardware adaptation. The ITB algorithm developed in this work can thus serve as a solid basis for a whole family of further optimisation methods that pursue the goal of designing data flow-based systems not only correctly and quickly, but also adaptively and intelligently.
To summarise, it can be said that The developed ITB algorithm offers a robust, flexible and high-performance solution to the challenge of actor merging in CAL-based DPNs. It therefore represents a central basis for further optimisations and the automation of complex data flow-based systems.

List of Figures

A. Glossar

DPN	Dataflow Process Network - a general model of computation for distributed computations
FIFO	First In First Out
Actor	An independent entity in a dataflow model that reacts to input data and generates output data.
Action	A component of an actor that consists of a set of conditions and is executed when those conditions are met.
FSM	Finite State Machine – a technique for controlling the execution of an actor based on states.
Guard	A condition that determines whether an action can be executed.
Token	Data objects exchanged between channels in a dataflow network.
CAL	Concurrent Actor Language (CAL) is a programming language that is used to model and simulate parallel systems by enabling communication between independent actors via messages and asynchronous events.
XML	Extensible Markup Language (XML) is a text-based markup language for the structured storage and transfer of data. It enables a hierarchical organisation of information using tags. Here to represent Networks in CAL
Meta Actor	Actor that was merged from several original actors and has the same characteristics
ITB	Internal Token Buffering - Name of the technique to store internal tokens in a Meta Actor
RAM	Repeat Actor Merging - Name of the technique to merge Actors developed by Krebs and Schneider
Buffer	A buffer is a memory area that stores data (tokens) temporarily
Meta Actor	A meta actor is a unit created by merging several CAL actuators that act together as a new, larger execution unit
Meta Action	A meta action describes a summarised, complex action within a super actuator that consists of several originally separate actuator actions.

B. Code from Example

```
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="IIR_lowlevel">
    <Instance id="delay_1">
        <Class name="common.delayi"/>
        <Parameter name="delay">
            <Expr kind="Literal" literal-kind="Integer" value="1"/>
        </Parameter>
        <Parameter name="value">
            <Expr kind="Literal" literal-kind="Integer" value="0"/>
        </Parameter>
    </Instance>
    <Instance id="mul_1">
        <Class name="common.mulc"/>
        <Parameter name="constant">
            <Expr kind="Literal" literal-kind="Integer" value="85"/>
        </Parameter>
    </Instance>
    <Instance id="mul_2">
        <Class name="common.mulc"/>
        <Parameter name="constant">
            <Expr kind="Literal" literal-kind="Integer" value="171"/>
        </Parameter>
    </Instance>
    <Instance id="dup">
        <Class name="common.dup2"/>
    </Instance>
    <Instance id="add_1">
        <Class name="common.add"/>
    </Instance>
    <Instance id="rshift_1">
        <Class name="common.rshiftc"/>
        <Parameter name="constant">
            <Expr kind="Literal" literal-kind="Integer" value="8"/>
        </Parameter>
    </Instance>
```

Figure B.1.: *IIR_lowlevel.xml part 1*

```
    <Instance id="rshift_2">
        <Class name="common.rshiftc"/>
        <Parameter name="constant">
            <Expr kind="Literal" literal-kind="Integer" value="8"/>
        </Parameter>
    </Instance>
    <Instance id="source">
        <Class name="common.source"/>
        <Parameter name="tag">
            <Expr kind="Literal" literal-kind="Integer" value="1"/>
        </Parameter>
        <Parameter name="offset">
            <Expr kind="UnaryOp">
                <Op name="-"/>
                <Expr kind="Literal"
                literal-kind="Integer" value="128"/>
            </Expr>
        </Parameter>
    </Instance>
    <Instance id="sink">
        <Class name="common.sink"/>
        <Parameter name="offset">
            <Expr kind="Literal"
            literal-kind="Integer" value="128"/>
        </Parameter>
    </Instance>
    <Connection dst="mul_1" dst-port="operand_1"
    src="source" src-port="result"/>
    <Connection dst="add_1" dst-port="operand_1"
    src="mul_1" src-port="result"/>
    <Connection dst="dup" dst-port="input" src="add_1"
    src-port="result"/>
    <Connection dst="rshift_1" dst-port="operand_1"
    src="dup" src-port="out1"/>
    <Connection dst="rshift_2" dst-port="operand_1"
    src="dup" src-port="out2"/>
    <Connection dst="sink" dst-port="operand_1"
    src="rshift_1" src-port="result"/>
    <Connection dst="mul_2" dst-port="operand_1"
    src="rshift_2" src-port="result"/>
    <Connection dst="delay_1" dst-port="operand_1"
    src="mul_2" src-port="result"/>
    <Connection dst="add_1" dst-port="operand_2"
    src="delay_1" src-port="result"/>
</XDF>
```

Figure B.2.: *IIR_lowlevel.xml part 2*

```
package common;

import common.constants.*;

actor delayi(int value, int delay) int(size=SAMPLE_SZ) operand_1 ==>
int(size=SAMPLE_SZ) result :

    int count;

    init: action ==>
    do
        count := 0;
    end

    token: action ==> result:[ value ]
    guard
        count < delay
    do
        count := count + 1;
    end

    run: action operand_1:[ x ] ==> result:[ x ]
    end

    schedule fsm s_init:
        s_init  ( init )   --> s_delay;
        s_delay ( token )   --> s_delay;
        s_delay ( run )     --> s_run;
        s_run   ( run )     --> s_run;
    end

    priority
        token > run;
    end

end
```

Figure B.3.: *CAL Code - `delayi` Actor*

```
package common;

import common.constants.*;

actor mulc(int constant) int(size=SAMPLE_SZ) operand_1 ==>
int(size=SAMPLE_SZ) result :

    action operand_1:[ x ] ==> result:[ x * constant ]
    end

end
```

Figure B.4.: *CAL Code - `mulc` Actor*

```
package common;

import common.constants.*;

actor dup2() int(size=SAMPLE_SZ) input ==>
int(size=SAMPLE_SZ) out1, int(size=SAMPLE_SZ) out2:
  action input:[i] ==> out1:[i], out2:[i] end
end
```

Figure B.5.: *CAL Code - `dup2` Actor*

```
package common;

import common.constants.*;

actor rshiftc(int constant) int(size=SAMPLE_SZ) operand_1 ==>
int(size=SAMPLE_SZ) result :

    action operand_1:[ x ] ==> result:[ x >> constant ]
    end

end
```

Figure B.6.: *CAL Code - `rshiftc` Actor*

```
package common;

import common.constants.*;

actor add()
int(size=SAMPLE_SZ) operand_1 , int(size=SAMPLE_SZ) operand_2 ==>
int(size=SAMPLE_SZ) result :

    action operand_1:[ x ], operand_2:[ y ] ==> result:[ x + y ]
    end

end
```

Figure B.7.: *CAL Code - `add` Actor*

```
package common;

import common.constants.*;

actor source(int tag, int offset) ==> int(size=SAMPLE_SZ) result :

    @native procedure native_source_init(int ind)
    end

    @native function native_source_produce(int ind) -->
    int(size=SAMPLE_SZ)
    end

    initialize ==>
    do
        native_source_init(tag);
    end

    action ==> result : [ native_source_produce(tag) + offset ]
    end

end
```

Figure B.8.: *CAL Code - `source` Actor*

```
package common;

import common.constants.*;

actor sink(int offset) int(size=SAMPLE_SZ) operand_1 ==> :

    @native procedure native_sink_init()
    end

    @native procedure native_sink_consume(int(size=SAMPLE_SZ) sample)
    end

    initialize ==>
    do
        native_sink_init();
    end

    action operand_1:[ sample ] ==>
    do
        native_sink_consume(sample + offset);
    end

end
```

Figure B.9.: *CAL Code - `sink` Actor*

```
package common;
import common.constants.*;
actor Actor_f (int constant, int constant_1, int constant_2,
int constant_3, int delay, int offset, int offset_1,
int tag, int value)  ==>  :

@native procedure native_source_init(int ind)
end
@native function native_source_produce(int ind) -->
int(size=SAMPLE_SZ)
end
@native procedure native_sink_init()
end
@native procedure native_sink_consume(int(size=SAMPLE_SZ) sample)
end

int count;
initialize ==>
    do
        native_source_init(tag);
        native_sink_init();
    end

bool run := true;

List(type:int(size=SAMPLE_SZ), size=1) buffer_mul_1_add_1;
int buffer_mul_1_add_1_size := 0;

List(type:int(size=SAMPLE_SZ), size=1) buffer_delay_1_add_1;
int buffer_delay_1_add_1_size := 0;

String delay_1_state := "s_init";

action_1: action  ==>
var
    int(size=SAMPLE_SZ) gen_6,
    int(size=SAMPLE_SZ) x,
    int(size=SAMPLE_SZ) y,
    int(size=SAMPLE_SZ) i,
    int(size=SAMPLE_SZ) x_1,
    int(size=SAMPLE_SZ) gen_30,
    int(size=SAMPLE_SZ) x_2,
    int(size=SAMPLE_SZ) gen_34,
    int(size=SAMPLE_SZ) sample,
    int(size=SAMPLE_SZ) x_3,
```

Figure B.10.: *CAL Code - merged Actor 1*

```
do
    gen_6 := native_source_produce ( tag ) + offset;
    x := gen_6;
    buffer_mul_1_add_1[buffer_mul_1_add_1_size]
        := x * constant;
    buffer_mul_1_add_1_size :=
        buffer_mul_1_add_1_size + 1;

    while (run) do
        run := false;

        if buffer_mul_1_add_1_size - 1 >= 0 &&
        buffer_delay_1_add_1_size - 1 >= 0 then
            x := buffer_mul_1_add_1[0];
            foreach int gen_15 in 0 ..
                buffer_mul_1_add_1_size - 1 do
                    buffer_mul_1_add_1[gen_15] :=
                        buffer_mul_1_add_1[gen_15 + 1];
            end
            buffer_mul_1_add_1_size :=
                buffer_mul_1_add_1_size - 1;
            y := buffer_delay_1_add_1[0];
            foreach int gen_18 in 0 ..
                buffer_delay_1_add_1_size - 1 do
                    buffer_delay_1_add_1[gen_18] :=
                        buffer_delay_1_add_1[gen_18 + 1];
            end
            buffer_delay_1_add_1_size :=
                buffer_delay_1_add_1_size - 1;
            gen_19 := x + y;
            run := true;
        if buffer_add_1_dup_size - 1 >= 0 then
            i := gen_19;
            gen_25 := i;
            gen_26 := i;
            x_1 := gen_25;
            gen_30 := x_1 >> constant_1;
            x_2 := gen_26;
            gen_34 := x_2 >> constant_2;
            sample := gen_30;
            native_sink_consume(sample + offset_1);
            x_3 := gen_34;
```

Figure B.11.: *CAL Code - merged Actor 2*

```
        buffer_mul_2_delay_1[buffer_mul_2_delay_1_size]
            := x * constant;
        buffer_mul_2_delay_1_size :=
            buffer_mul_2_delay_1_size + 1;
    end


    if (delay_1_state = "s_delay") && count < delay then
        count := count + 1;
        delay_1_state := "s_delay";
        buffer_delay_1_add_1[buffer_delay_1_add_1_size] := value;
        buffer_delay_1_add_1_size :=
                buffer_delay_1_add_1_size + 1;
        run := true;
    else if (delay_1_state = "s_delay") then
        if buffer_mul_2_delay_1_size - 1 >= 0 then
            x := buffer_mul_2_delay_1[0];
            foreach int gen_46 in 0 ..
                buffer_mul_2_delay_1_size - 1 do
                    buffer_mul_2_delay_1[gen_46] :=
                        buffer_mul_2_delay_1[gen_46 + 1];
            end
            buffer_mul_2_delay_1_size :=
                buffer_mul_2_delay_1_size - 1;
            delay_1_state := "s_run";
            buffer_delay_1_add_1[buffer_delay_1_add_1_size]
                := x_3;
            buffer_delay_1_add_1_size :=
                buffer_delay_1_add_1_size + 1;
            run := true;
        end
    else if (delay_1_state = "s_run") then
        if buffer_mul_2_delay_1_size - 1 >= 0 then
            x := buffer_mul_2_delay_1[0];
            foreach int gen_46 in 0 ..
                buffer_mul_2_delay_1_size - 1 do
                    buffer_mul_2_delay_1[gen_46] :=
                        buffer_mul_2_delay_1[gen_46 + 1];
            end
```

Figure B.12.: *CAL Code - merged Actor 3*

```
                    buffer_mul_2_delay_1_size :=
                        buffer_mul_2_delay_1_size - 1;
                    delay_1_state := "s_run";
                    buffer_delay_1_add_1[buffer_delay_1_add_1_size]
                        := x_3;
                    buffer_delay_1_add_1_size :=
                        buffer_delay_1_add_1_size + 1;
                    run := true;
                end
            else if (delay_1_state = "s_init") then

                count := 0;
                delay_1_state := "s_delay";
                run := true;
            end end end end
        end
    end
end
```

Figure B.13.: *CAL Code - merged Actor 4*

Bibliography

[1] Wil Aalst. "Putting Petri nets to work in industry". In: *Computers in Industry* 25 (Nov. 1994), pp. 45–54. DOI: 10.1016/0166-3615(94)90031-0.

[2] W. B. Ackerman. "Data Flow Languages". In: *Computer* 15.2 (Feb. 1982), pp. 15–25. ISSN: 0018-9162. DOI: 10.1109/MC.1982.1653938. URL: https://doi.org/10.1109/MC.1982.1653938.

[3] E. A. Ashcroft and R. Jagannathan. "Operator Nets". In: *Proc. of the IFIP TC 10 Working Conference on Fifth Generation Computer Architectures*. Manchester, United Kingdom: North-Holland Publishing Co., 1986, pp. 177–201. ISBN: 0444879870. DOI: 10.5555/20155.20166.

[4] Lubomir F. Bic. "A Process-Oriented Model for Efficient Execution of Dataflow Programs". In: *Journal of Parallel and Distributed Computing* 8 (1990), pp. 42–51. DOI: 10.1016/0743-7315(90)90041-C.

[5] Jani Boutellier et al. "Actor Merging for Dataflow Process Networks". In: *IEEE Transactions on Signal Processing* 63.10 (2015), pp. 2496–2508. DOI: 10.1109/TSP.2015.2411229.

[6] J.T. Buck and E.A. Lee. "Scheduling dynamic dataflow graphs with bounded memory using the token flow model". In: *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 1. 1993, 429–432 vol.1. DOI: 10.1109/ICASSP.1993.319147.

[7] Joseph Buck and Edward Lee. "The Token Flow Model". In: (Dec. 1994).

[8] P. Caspi et al. "LUSTRE: a declarative language for real-time programming". In: POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 178–188. ISBN: 0897912152. DOI: 10.1145/41625.41641. URL: https://doi.org/10.1145/41625.41641.

[9] Jean-Louis Colaço. "An overview of Scade, a synchronous language for safety-critical software (keynote)". In: REBLS 2020. Virtual, USA: Association for Computing Machinery, 2020, p. 1. ISBN: 9781450381888. DOI: 10.1145/3427763.3432350. URL: https://doi.org/10.1145/3427763.3432350.

[10] Frederic Commoner et al. "Marked directed graphs". In: (Oct. 1971). DOI: 10.1016/s0022-0000(71)80013-2.

[11] René David and Hassane Alla. "Discrete, Continuous, and Hybrid Petri Nets". In: *Control Systems Magazine, IEEE* 28 (July 2008), pp. 81–84. DOI: 10.1109/MCS.2008.920445.

[12] RPTU Computer Science Department. *MBES-04: Dataflow and Static Schedules*. https://es.cs.rptu.de/teaching/mbes/slides/MBES-04-Dataflow-StaticSchedules-1.pdf. Accessed: 2025-03-24.

[13] J. Desel. "Basic linear algebraic techniques for place or transition nets". In: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*. Vol. 1491. LNCS. Springer, 1998, pp. 257–308.

[14] J. Desel and W. Reisig. "Place or transition Petri nets". In: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*. Vol. 1491. LNCS. Springer, 1998, pp. 122–173.

[15] M. Dubois, C. Scheurich, and F. Briggs. "Memory access buffering in multiprocessors". In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*. ISCA '86. Tokyo, Japan: IEEE Computer Society Press, 1986, pp. 434–442. ISBN: 081860719X. DOI: 10.5555/17407.17406.

[16] Johan Eker and Jorn W. Janneck. *CAL Language Report: Specification of the CAL Actor Language*. Tech. rep. UCB/ERL M03/48. ERL Technical Memo. University of California at Berkeley, Dec. 2003.

[17] Johan Ersfolk et al. "Modeling control tokens for composition of CAL actors". In: *2013 Conference on Design and Architectures for Signal and Image Processing*. 2013, pp. 71–78.

[18] Joachim Falk et al. "Analysis of SystemC actor networks for efficient synthesis". In: *ACM Trans. Embed. Comput. Syst.* 10.2 (Jan. 2011). ISSN: 1539-9087. DOI: 10.1145/1880050.1880054. URL: https://doi.org/10.1145/1880050.1880054.

[19] D. Friedman and D. Wise. "Cons should not evaluate its arguments". In: *International Colloquium on Automata, Languages and Programming (ICALP)* (1976), pp. 257–284.

[20] David Gelernter and Nicholas Carriero. "Coordination Languages and Their Significance". In: *Communications of the ACM* 35.2 (Feb. 1992), pp. 97–107. DOI: 10.1145/129630.129635.

[21] Thomas T. Hildebrandt, Prakash Panangaden, and Glynn Winskel. "A relational model of non-deterministic dataflow". In: 14.5 (Oct. 2004), pp. 613–649. ISSN: 0960-1295. DOI: 10.1017/S0960129504004293. URL: https://doi.org/10.1017/S0960129504004293.

[22] Bernard Houssais and Irisa Espresso. "The Synchronous Programming Language SIGNAL A Tutorial". In: (May 2012).

[23] Jorn Janneck. "Actors and their Composition". In: *Formal Asp. Comput.* 15 (Dec. 2003), pp. 349–369. DOI: 10.1007/s00165-003-0016-3.

[24] Jorn W. Janneck. "A machine model for dataflow actors and its applications". In: *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*. 2011, pp. 756–760.

[25] Richard Johnsonbaugh and Tadao Murata. "Petri Nets and Marked Graphs–Mathematical Models of Concurrent Computation". In: *American Mathematical Monthly* 89 (Oct. 1982). DOI: 10.2307/2320826.

[26] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in Dataflow Programming Languages". In: *ACM Computing Surveys (CSUR)* 36.1 (Mar. 2004), pp. 1–34. DOI: 10.1145/1013208.1013209.

[27] Gilles Kahn. "The Semantics of Simple Language for Parallel Programming." In: *IFIP Congress*. 1974, pp. 471–475. URL: http://dblp.uni-trier.de/db/conf/ifip/ifip74.html#Kahn74.

[28] Gilles Kahn and David Macqueen. "Coroutines and Networks of Parallel Processes". In: *Information Processing*. Ed. by J.L. Rosenfeld. Stockholm, Sweden: North-Holland, 1977, N/A.

[29] F. Krebs. "A Translation Framework from RVC-CAL Dataflow Programs to OpenCL/SYCL based Implementations". Master. MA thesis. Department of Computer Science, University of Kaiserslautern, Germany, Jan. 2019.

[30] Florian Krebs. *Dataflow Actor Fusion for CAL Actors*. Mar. 2024.

[31] Edward Lee. "Consistency in Dataflow Graphs". In: *IEEE Trans. Parallel Distrib. Syst.* 2 (Apr. 1991), pp. 223–235. DOI: 10.1109/ASAP.1991.238909.

[32] Edward Lee and David Messerschmitt. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing". In: *Computers, IEEE Transactions on* C-36 (Feb. 1987), pp. 24–35. DOI: 10.1109/TC.1987.5009446.

[33] Edward Lee and Thomas Parks. "Dataflow Process Networks". In: *Proceedings of the IEEE* 83 (May 1995), pp. 773–801. DOI: 10.1109/5.381846.

[34] Edward Ashford Lee and David G. Messerschmitt. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing". In: *IEEE Transactions on Computers* C-36.1 (1987), pp. 24–35. DOI: 10.1109/TC.1987.5009446.

[35] T. Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580. DOI: 10.1109/5.24143.

[36] ORCC Community. *ORC Apps – Open RVC-CAL Applications*. Accessed: 2025-03-25. 2025. URL: https://github.com/orcc/orc-apps.

[37] J. Morris P. Henderson. "A lazy evaluator. Principles of Programming Languages (POPL)". In: (1976), pp. 95–103.

[38] T.M. Parks, J.L. Pino, and E.A. Lee. "A comparison of synchronous and cyclo-static dataflow". In: *Asilomar Conference on Signals, Systems and Computers (ACSSC)*. Washington, District of Columbia, USA: IEEE Computer Society, 1995, pp. 204–210.

[39] T.M. Parks, J.L. Pino, and E.A. Lee. "A comparison of synchronous and cycle-static dataflow". In: *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*. Vol. 1. 1995, 204–210 vol.1. DOI: 10.1109/ACSSC.1995.540541.

[40] Thomas Martyn Parks. *Bounded scheduling of process networks*. University of California, Berkeley, 1995.

[41] James L. Peterson. "Petri Nets". In: *ACM Comput. Surv.* 9.3 (Sept. 1977), pp. 223–252. ISSN: 0360-0300. DOI: 10.1145/356698.356702. URL: https://doi.org/10.1145/356698.356702.

[42] C. Petri. "Kommunikation mit Automaten. Ph.D. thesis, Institut für Instrumentelle Mathematik, Universität Bonn, Germany". PhD thesis. 1976.

[43] Keshav Pingali and Arvind. "Efficient demand-driven evaluation. Part 1". In: 7.2 (Apr. 1985), pp. 311–333. ISSN: 0164-0925. DOI: 10.1145/3318.3480. URL: https://doi.org/10.1145/3318.3480.

[44] Keshav Pingali and Arvind. "Efficient demand-driven evaluation. Part 2". In: *ACM Trans. Program. Lang. Syst.* 8.1 (Jan. 1986), pp. 109–139. ISSN: 0164-0925. DOI: 10.1145/5001.5003. URL: https://doi.org/10.1145/5001.5003.

[45] Ptolemy Project. *Ptolemy II: The Ptolemy Project*. Accessed: 2025-01-07 14:30:00. 2025. URL: https://ptolemy.berkeley.edu/ptolemyII/index.htm.

[46] Omair Rafique et al. "Synthesis of Heterogeneous Dataflow Models from Synchronous Specifications". In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2021, pp. 43–48. DOI: 10.1109/COMPSAC51774.2021.00018.

[47] Raspberry Pi Foundation. *Raspberry Pi 4 Model B Specifications*. Accessed: 2025-04-01. 2025. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[48] W. Reisig. "Petri Nets: An Introductions". In: *vol. 4 of EATCS Monographs on Theoretical Computer Science* (1985).

[49] K. Schneider. *The Synchronous Programming Language Quartz*. Internal Report 375. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, Dec. 2009.

[50] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. "Data-Driven and Demand-Driven Computer Architecture". In: *ACM Comput. Surv.* 14.1 (Mar. 1982), pp. 93–143. ISSN: 0360-0300. DOI: 10.1145/356869.356873. URL: https://doi.org/10.1145/356869.356873.

[51] Arthur H. Veen. "Dataflow Machine Architecture". In: *ACM Computing Surveys (CSUR)* 18.4 (Dec. 1986), pp. 365–396. DOI: 10.1145/27633.27634.

[52] Matthieu Wipliez and Mickaël Raulet. "Classification and transformation of dynamic dataflow programs". In: *2010 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2010, pp. 303–310. DOI: 10.1109/DASIP.2010.5706280.

Tools

Note on the Use of AI-based Language Assistance

In the course of writing this Master's thesis, AI-based language assistance tools were used to support the linguistic quality of the text. Specifically, ChatGPT (OpenAI) and DeepL were employed for grammar and spelling checks, stylistic improvements, and translation support. The conceptual development, analysis, and evaluation were carried out independently by the author. The aforementioned tools were used solely to enhance the language quality of the thesis.