Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Bachlor's Thesis

Code Generation for Rust from Dataflow Process Networks

Omar Bihi

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

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

September 10, 2025

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Code Generation for Rust fromDataflow Process Networks“ 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 10.09.2025

Omar Bihi

Abstract

This thesis presents a method for automatically generating Rust code from actor-based dataflow networks written in the RVC-CAL language. Actor-based dataflow programming allows complex systems, such as signal processing or embedded applications, to be modeled as networks of independent computational units called actors. To make these networks executable, many tools were developed that converts RVC-CAL applications into programs that use languages such as C or C++, that support multi-threaded execution. This work integrated the a Rust-backend into one of these code generators.

Two execution approaches were studied. The first uses Tokio with async/await, allowing actors to run asynchronously without blocking each other. The second uses Rayon with Crossbeam channels to run actors in parallel while communicating through fast, lock-free channels. Both approaches were designed to support round-robin and non-preemptive scheduling.

The generated Rust code provides a single interface to manage all actors in the same way, regardless of their specific behavior. The two execution schemes were tested on benchmark networks to measure execution time, performance, and reliability. Results show that the Tokio async approach is flexible and easy to use, while Rayon combined with Crossbeam approach gives higher performance and avoids deadlocks in cyclic networks.

In summary, this thesis demonstrates that it is possible to automatically generate efficient, multi-threaded Rust programs from high-level RVC-CAL networks. It provides insights into the advantages and limitations of language. The proposed framework contributes to the automation of dataflow application deployment and establishes a foundation for future extensions.

Zusammenfassung

Diese Arbeit stellt eine Methode zur automatischen Generierung von Rust-Code aus Actor-based Dataflow Networks vor, die in der RVC-CAL-Sprache geschrieben sind. Die Actor-based Dataflow programmierung ermöglicht es, komplexe Systeme, wie Signalverarbeitung oder eingebettete Anwendungen, als Netzwerke unabhängiger Recheneinheiten, sogenannter Akteure, zu modellieren. Um diese Netzwerke ausführbar zu machen, wurden bisher viele Werkzeuge entwickelt, die RVC-CAL-Anwendungen in Programme in Sprachen wie C oder C++ umwandeln und die Multi-Threading unterstützen. Diese Arbeit integriert einen Rust-Backend in einen dieser Codegeneratoren.

Es wurden zwei Schema untersucht. Der erste verwendet Tokio mit async/await, wodurch Actors asynchron ausgeführt werden können, ohne einander zu blockieren. Der zweite Ansatz verwendet Rayon mit Crossbeam-Kanälen, um Actors parallel auszuführen und gleichzeitig über schnelle, sperrfreie Kanäle zu kommunizieren. Beide Ansätze wurden so gestaltet, dass sie Round-Robin- und Non-Preemptive Scheduling unterstützen.

Der generierte Rust-Code stellt eine einheitliche Schnittstelle bereit, um alle Actors auf die gleiche Weise zu verwalten, unabhängig von ihrem spezifischen Verhalten. Beide Schema wurden auf Benchmark-Netzwerken getestet, um Ausführungszeit, Leistung und Zuverlässigkeit zu messen. Die Ergebnisse zeigen, dass der Tokio-Async-Ansatz flexibel und einfach zu verwenden ist, während die Kombination aus Rayon und Crossbeam höhere Leistung bietet und Deadlocks in zyklischen Netzwerken verhindert.

Zusammenfassend zeigt diese Arbeit, dass es möglich ist, effiziente, multi-threaded Rust-Programme automatisch aus hochrangigen RVC-CAL-Netzwerken zu generieren. Sie liefert Einblicke in die Vorteile und Grenzen der Sprache. Das vorgeschlagene Framework trägt zur Automatisierung der Bereitstellung von Dataflow anwendungen bei und bildet eine Grundlage für zukünftige Erweiterungen.

Contents

Listings

List of Figures

List of Tables

1 Introduction

The increasing complexity of embedded systems and the growing demand for concurrent and parallel applications have made dataflow process networks (DPNs) a popular model of computation for stream-based and signal-processing applications. DPNs offer an intuitive and modular abstraction for describing concurrent systems, where actors communicate asynchronously through channels without shared memory. Their well-defined semantics enable high-level reasoning about system behavior and facilitate formal verification. In this context, the RVC-CAL language (specialized profile of CAL) provides a platform for specifying DPN applications through actors, actions, ports, and firing rules. The language is widely adopted in academic and industrial environments, with toolchains like Orcc supporting simulation and C/C++ code generation.

Beyond research, DPNs and RVC-CAL have proven valuable in practical domains. In digital signal processing, they are used to model streaming pipelines such as filters, transforms, and codecs, where predictable throughput and low latency are essential [1]. In complex heterogeneous systems, dataflow semantics help orchestrate computations across diverse processing units, including CPUs, GPUs, DSPs, and FPGAs, enabling efficient execution on multi-processor systems-on-chip (MPSoCs) [2] [3].

However, the current mainstream backends for RVC-CAL target C and C++, which, despite their performance and maturity, pose several challenges in terms of memory safety, thread management, and integration with modern asynchronous systems. Rust emerges as a compelling alternative due to its strong type system, ownership model, and support for asynchronous and concurrent programming paradigms. While the RVC-CAL language is well-supported by C/C++-based code generators, there is no robust Rust-based backend available for generating efficient and safe actor code. This limits the adoption of RVC-CAL in modern, safety-critical, or performance-sensitive environments that increasingly prefer Rust due to its guarantees against data races and undefined behavior. Moreover, supporting modern concurrency models like async/await, or thread-pooling requires a language that provides compile-time guarantees on resource usage and thread safety — a gap that C++ cannot fill easily.

This thesis aims to bridge this gap by designing and implementing a Rust code generation backend for RVC-CAL applications, capable of simulating and executing actor networks in an asynchronous and safe manner. The key objectives include, designing a Rust code generator that can interpret actor constructs from parsed RVC-CAL specifications , supporting fundamental control structures such as conditionals, loops, and list comprehensions, generating lock-free, bounded FIFO channels with controlled token flow, supporting both rayon-based multi-threaded execution and tokio-based async simulation, and lastly benchmarking the performance against existing C/C++ backends using an actor-based network application.

We start in chapter 2 with an introduction of the theoretical and technical background of dataflow models, RVC-CAL, Rust and the Code-Generator which we going to intergrate the Rust-backends into. In chapter 3 the two Rust schemes - Tokio + async/await and Rayon + Crossbeam - are pressented with the implementation in detail, including channel systems, scheduler, and actor code generation. The evaluation of the generator using test applications and comparesion of its performance will be conducted in chapter 4. The chapter 5 present related frameworks that support code generation targeting languages like C and tools for dataflow programming implemented in Rust. At the end of the thesis the results of the work is summarized and an overview of possible future work is given in chapter 6.

2 Background

2.1 Dataflow Process Networks

Dataflow Process Networks (DPNs) are a formal computational model widely used for describing and executing concurrent systems such as streaming applications, multimedia processing, and signal processing pipelines. They were introduced and formalized by Edward A. Lee and Thomas M. Parks in 1995 as a refinement of the more general Kahn Process Networks (KPNs) [4] [5]. The key idea is to model a system as a network of independent actors that communicate through stream of data.

A DPN can be represented as a directed graph, with nodes represent computational entities (actors) that consume input tokens, perform some processing, and produce output tokens, where edges represent streams that carry data tokens from one actor to another [6]. This graph-based representation makes the flow of data between components explicit. Each actor in the graph executes independently and only interacts with other actors via token exchange, never through shared memory. Actor execution is based on firing rules, where it can only fire if the rules are met.

The communication between actors is achieved through streams that act like FIFO (First-In-First-Out) buffers, which is an ordered sequence of values. The ordering of tokens in each stream is preserved. This is important because the first token sent will be the first token received, ensuring temporal consistency in data processing.

From a practical perspective, streams abstract away the timing of token production and consumption, an actor processes tokens when they are available, and the stream preserves their sequence regardless of the order in which different actors are executed. Additonally they are one-way data paths that allow tokens to flow only from a single producer actor to a single consumer actor.

While KPNs assume unbounded FIFO channels, real systems have finite memory . If token production exceeds consumption for a sustained period, buffers will grow without bound, leading to memory exhaustion. In DPNs, boundedness refers to the property that all channels can operate within a fixed, finite capacity while still preserving determinism and avoiding deadlocks [7] [8]. Determinism means that, given the same sequence of input tokens, the network will always produce the same sequence of output tokens, regardless of the execution order or relative speeds of individual actors. This guarantee holds as long as the semantics of the actors are functional and communication is through unidirectional FIFO channels.

Lee and Parks examined bounded scheduling strategies that allow DPNs to run correctly with limited buffer sizes. Some networks are naturally bounded, while others require careful execution order or additional constraints to ensure bounded execution.

Scheduling is a critical aspect of DPNs, as it determines how and when actors execute within the network. An important specialization of DPNs - Synchronous Dataflow (SDF)- touchs this aspect. In SDF, each actor consumes and produces a fixed number of tokens per firing on each channel. This allows for static scheduling, where the firing order of actors and the minimum buffer sizes can be computed before execution [9].

Another extension is Cyclo-Static Dataflow (CSDF). It builds on SDF by letting the number of tokens produced and used change in a repeating pattern. This makes the model more flexible but still allows scheduling to be done before running the program. Parks, Pino, and Lee compared SDF and CSDF, highlighting that CSDF keeps many of SDF's advantages while supporting more complex and realistic dataflow behaviors [10].

Generally for DPNs, in which actors can consume and generate tokens at changing rates, there needs to be dynamic scheduling. This approach must keep up with the changing availability of tokens so that actors may run as soon as their input data is ready. While dynamic scheduling enables the model to accommodate more flexible, dynamic systems where constant consumption/production rates are not feasible, it increases runtime overhead and often requires runtime deadlock avoidance and buffer management, making it more complex to implement and analyze than static or quasi-static approaches [11].

2.2 The CAL Actor Language

The CAL Actor Language (often referred to simply as CAL) is a domain-specific, high-level programming language designed for modeling and implementing dataflow actors in DPN-based systems [12]. CAL was introduced in the early 2000s as part of the Ptolemy II project at the University of California, Berkeley, to provide a structured way to describe actor-based computations [13]. CAL is designed to be platform-agnostic, meaning the same actor description can be used for simulation, and code generation to multiple target languages such as C. The philosophy behind this design was to focus solely on actor behavior specification, that is to describe how an actor reacts to incoming data tokens, manipulates its internal state, and produces output tokens while leaving all aspects of execution control, resource management, and runtime services to the host environment.

The CAL syntax consists of several key elements [14]:

- Actor Declaration: Each actor in CAL is defined using the keyword `actor`. The definition specifies the name of the actor, input and output ports, internal state variables, and computation actions. The syntax ensures that each actor is a self-contained unit.

- Data types: The CAL type system is host-dependent, meaning that while CAL can define and use types like integers, strings, and lists, their actual implementation and operations come from the execution environment. It supports composite data types such as lists and records, but their implementation depends on the execution platform.
- Input and Output Ports: CAL actors pass data through ports. Input ports are used to accept data tokens, and output ports are used to emit processed data. Actors are connected by means of these ports to form a network based on dataflow, where execution order of the actors is based on data flow.
- State Variables: Each actor can have variables that hold it's internal state across firings. These state variables allow actors to perform stateful computations. They are either assignable, or mutable. Assignable state variables can be set exactly once and then remain constant throughout execution. Mutable state variables can be updated multiple times during the actor's lifetime, allowing the actor to change its internal state as it processes data.
- Actions define how an actor processes incoming data and generates output. An action is executed every time its input patterns match with the tokens in hand. Guards are also a possibility for actions, which are preconditions to be satisfied prior to the invocation of the action.
- Functions and Procedures: Functions are pure, side-effect-free constructs that return a value and are used within expressions. Their syntax includes typed parameters and a return type, using the `function` keyword. Procedures, on the other hand, are impure, may cause side effects, and do not return a value; they are declared with the `procedure` keyword and are called as complete statements, not as part of an expression. Both are defined within actors but are not first-class entities.
- Loops and iteration: CAL supports structured iteration using `foreach` and `while` loops. The foreach loop allows iteration over collections like lists and records, the while loop executes a block of code repeatedly as long as a specified condition holds.
- Priorities define the order in which an actor's actions are considered for execution, resolving conflicts when multiple actions are enabled simultaneously. They help control which action takes precedence.
- State machines (FSM): model an actor's control flow by organizing actions into states with transitions based on conditions. The FSM defines how the actor moves between states, controlling which actions are active and how the actor's behavior evolves over time.

The following Listing 2.1 shows an example of a CAL actor, with two actions and a FSM.

```
actor IncVal () Input1, Input2 ==> Output:

    count := 0;

    A: action Input1: [x] ==> [x]
        do
        count := count + 1;
        x := count;
        end
    end

    B: action Input2: [x] ==> [x] end

    schedule fsm s1:
    s1 (A) --> s2;
    s2 (B) --> s1;
    end
end
```

Listing 2.1: *CAL Actor Example*

2.3 The RVC-CAL Actor Language

RVC-CAL is a specialized profile of CAL defined by ISO/IEC as part of the MPEG Reconfigurable Video Coding (RVC) framework [2] [15]. While CAL in its original form was general-purpose for dataflow actor specification, RVC-CAL was streamlined for the needs of modular, reconfigurable video codec design. Its design draws on the CAL actor language, but adapts and extends it for the constraints and modularity requirements of the MPEG standard.

2.3.1 Open RVC-CAL Compiler

The Open RVC-CAL Compiler (Orcc) is an open-source framework designed to compile and execute applications written in RVC-CAL [16] [17]. It offers a complete toolchain for transforming dataflow network descriptions into executable code for multiple target platforms. Orcc is developed as part of the MPEG RVC standardization effort, but it is general enough to support a wide range of applications beyond video processing.

One of Orcc's key contributions is the ability to take RVC-CAL actor definitions and combine them with XDF (XML Dataflow Format) network descriptions. Together, they define a Dataflow Process Network that can be optimized and mapped to software or hardware targets.

Orcc is not just a compiler but also a development environment. It integrates with Eclipse, providing project management tools, syntax highlighting for RVC-CAL, and a graphical editor for XDF networks. The graphical editor is particularly useful for visualizing the dataflow graph, including the actors, their ports, and the channels between them.

The RVC-CAL applications, from the repository [18], used in this thesis follow the Orcc toolchain conventions, to ensure compatibility with both the standard backends (C/C++) and the newly developed Rust backend.

2.3.2 Network Structure

The XDF format is an XML-based representation of dataflow networks. It builds on the Functional Unit Network Language (FNL), which was originally created to describe dataflow programs in a structured, machine-readable way [2] [19].

An XDF network specifies the set of actor instantiations, the channels that connect actor ports together, and the connections that define the overall structure of the dataflow graph. It supports hierarchical network descriptions, meaning that an actor instantiation can be derived from an actor definition or a child network. These sub-networks act like composite actors that can contain their own actors, channels, and even deeper levels of nested networks.

Connections in XDF are unidirectional FIFOs that preserve the order of tokens, maintaining the deterministic behavior expected from DPNs, where each one defines the source and destination ports.

In the example of Listing 2.2, line 1 declares XML version and encoding, line 2 the start of the XDF network named `Top_Network`. Lines 3–11 define three instances: `Child_Network`, `Source`, and `Sink`. Lines 13–14 specify the connections, where data flows from the `Source` actor to the `Child_Network`, and then from the `Child_Network` to the `Sink`.

At the network level, XDF defines `external ports`, which form the interface between the network and its environment. External input ports provide data from outside the network to internal actors, while external output ports deliver processed data back to the outside world. The Listing 2.3 provides an example of external ports. Lines 1–8 define an input port named `Source`, which accepts integer data with a buffer size of 32. Lines 9–16 define an output port named `Sink`, also handling integers with a buffer of same size.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="Top_Network">
    <Instance id="Child_Network">
        <Class name="CN.Child_Network"/>
    </Instance>
    <Instance id="Source">
        <Class name="CN.Source"/>
    </Instance>
    <Instance id="Sink">
        <Class name="CN.Sink"/>
    </Instance>

    <Connection dst="Child_Network" dst-port="Source"
      src="Source" src-port="Out"/>
    <Connection dst="Sink" dst-port="In"
      src="Child_Network" src-port="Sink"/>
</XDF>
```

Listing 2.2: *Top Network*

```xml
<Port kind="Input" name="Source">
        <Type name="int">
            <Entry kind="Expr" name="size">
                <Expr kind="Literal" literal-kind="Integer"
                   value="32"/>
            </Entry>
        </Type>
</Port>
<Port kind="Output" name="Sink">
        <Type name="int">
            <Entry kind="Expr" name="size">
                <Expr kind="Literal" literal-kind="Integer"
                   value="32"/>
            </Entry>
        </Type>
</Port>
```

Listing 2.3: *External Port*

2.3.3 RVC-CAL Syntax

RVC-CAL inherits the fundamental structure of CAL but introduces syntactic constraints and simplifications tailored for video codec specification and standardization [14] [17]. In this section, we explore the key syntactic elements of RVC-CAL.

Data Type

The biggest difference between CAl and RVC-CAL, is that the later imposes type and size definition, on all variables including lists as well, example in listing. Additionally type parameter **T**, such as generic in C++ or Rust, is prohibited in type expressions such as `List[T]`. Lastly it supports only six data types: bool, float, int, uint, String, List.

Actor Structure and Ports

To illustrate the structure of an RVC-CAL actor more concretely, consider the following example Listing 2.4 of a simple `Add` actor, which defines input ports `Input1` and `Input2` of type int with size of 8, `Output` port of same type and size, and a single action, which fires when tokens are available in both input ports and produces only one to the output.

```
actor Add () int(size=8) Input1, int(size=8) Input2 ==>
   int(size=8) Output:

    A: action Input1: [a], Input2: [b] ==> Output: [a + b]
    end

end
```

Listing 2.4: *Actor Example*

Assignment

There are two assignment operators, = is used for binding values to variables in a way that does not overwrite mutable state, in other words, it introduces a constant within the scope. The := operator is used when updating an existing variable, which overwrites the current value.

Loop and Iteration Statement

Loop statements are used to repeatedly execute a sequence of code, as long as the boolean condition is true. RVC-CAL supports the traditional loop construct `while`.

```
while /*condition*/ do
    /*...*/
end
```

Listing 2.5: *Loop Example*

Iteration statements, which is used for executing a block of code repeatedly. RVC-CAl uses `foreach` to iterate over elements of a list, without explicitly handling indices, and `for` is typically used for numeric iteration over a specified range.

```
foreach value in myList do
    /*...*/
end

for i in 0 .. 9 do
    /*...*/
end
```

Listing 2.6: *Iteration Example*

List Comprehension

List comprehensions provide a concise way to generate and manipulate collections of values. They allow actors to create new lists by applying an expression to each element of an input list, optionally filtering elements using conditions.

Here's a simple example Listing 2.7 of a list comprehension that declares `myList` of 256 elements where each element is of type int size 9 and all elements to 0.

```
List(type: int(size=9), size=256) myList := [ 0 : for int i
    in 0 .. 255 ];
```

Listing 2.7: *List Comprehension Example*

Procedures and Functions

Both constructs help separate reusable computation logic from the main actor behavior, but they differ in purpose and semantics.

A **function** takes input parameters and returns a value. It is side-effect free, meaning that it does not modify the actor state or global variables. Functions are typically used for computations such as arithmetic operations, comparisons, or transformations that always produce the same output for the same input.

On the other hand, a **procedure** encapsulates a sequence of statements that can perform side effects, such as updating actor state variables or calling other procedures. Unlike functions, procedures do not return values directly but instead influence the behavior of the actor.

Listing 2.8 illustrates an example of both constructs. The `square` function computes the square of an integer, whereas the `increment` procedure modifies the actor's internal `counter` variable. Inside action `B`, the procedure updates the counter, and then both the squared input value and the updated counter are produced on the output ports.

```
function square(int(size=8) z) -->  int(size=8) :
    z * z
end

procedure increment()
do
    counter := square(counter);
end


B: action  Input: [x] ==> Output: [y, c]
do
    increment();
    y := square(x);
    c := counter;
end
```

Listing 2.8: *Procedure - Function Example*

Native Functions

RVC-CAL supports `native functions` and `native procedures` to integrate functionality written in external programming languages (e.g., C or Java) directly into an actor's behavior. These constructs are declared in RVC-CAL using the `@native` keyword but implemented externally. The actual implementation must then be provided in the target language during integration.

Listing 2.9 shows an example of such declarations.

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

@native function native_func(int ind) --> int(size=8)
end
```

Listing 2.9: *Native Functions Example*

Priority Order

In RVC-CAL, an actor may define multiple actions that can be triggered when their input availability conditions (guards) are satisfied. If more than one action is enabled at the same time, the priority order determines which action should be executed first.

The priority is specified within the actor definition using the `priority` keyword. Actions are listed in descending order, meaning that actions appearing earlier in the list take precedence over those appearing later.

```
priority
    actionA > actionB > actionC;
end
```

Listing 2.10: *Priority Order Example*

In this example, if both `actionA` and `actionB` are enabled, `actionA` fires first. If `actionA` is not enabled but `actionB` and `actionC` are, then `actionB` is selected.

Finite State Machine

More complex actors can include a finite state machine, defined using the `fsm` keyword. It specifies a set of states and transitions, where each transition is guarded by conditions and linked to an action. After action firing, the FSM moves to the target state specified by that transition.

Listing 2.11 shows that this mechanism allows actors to exhibit structured behavior across multiple modes of operation, such as initialization, execution, and termination.

```
schedule fsm s_init:

    s_init ( initAction ) --> s_run;
    s_run ( runAction ) --> s_done;
    s_done; // No actions terminal state

end
```

Listing 2.11: *FSM Example*

Imports

In RVC-CAL, actors, type definitions, constants, and utility functions can be organized into packages, which allows multiple actors to share common resources.

A package is a named collection of related components grouped under a namespace. The `package` declaration at the top of a file defines its namespace. This means that all components defined in this file belong to that namespace. Other files can then access these components from this namespace using the `import` statement.

RVC-CAL supports two forms of imports. A single import makes a specific entity available and a group import that brings all entities from a sub-namespace into the global environment. For example, the following Listing 2.12 shows a package named `common` that imports all components from the subpackage `common.constants` using the asterisk (`*`), and on line 5 it imports the `square` function from the subpackage `common.utils`.

```
package common;

import common.constants.*;

import common.utils.square;
```

Listing 2.12: *Imports Example*

2.4 Code Generator

The goal of this thesis is to extend an existing Dataflow Code Generator by integrating a Rust backend capable of producing, multi-threaded implementations of Dataflow Process Networks. This generator, originally designed for C/C++ output [20], provides a framework for reading, analyzing, optimizing, and emitting executable code from RVC-CAL actor descriptions and their XDF network definitions. By incorporating the Rust backend, the same high-level model can be translated into Rust.

The underlying architecture of the generator (Figure 2.1) follows a multi-phase pipeline. Each phase transforms the model into a progressively more optimized and lower-level representation, preparing it for final code emission. While the original implementation targets C/C++, the integration of Rust generation fits naturally into this modular structure.

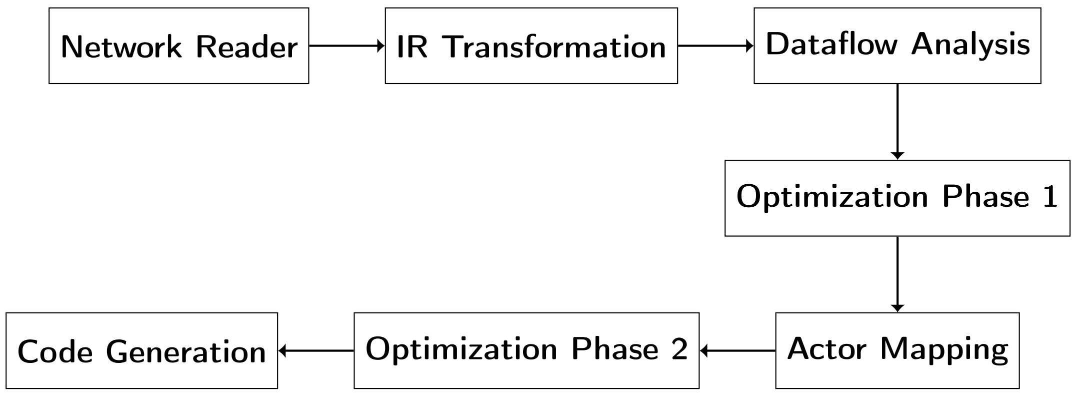

Figure 2.1: *Code Generator Design*

Network Reader

The pipeline begins with the Network Reader, which parses CAL actor definitions and XDF network files. This step handles hierarchical networks, where networks can contain child networks, by flattening them into a single, non-hierarchical representation. During flattening, network instances are replaced with their internal actors, and ports are replaced with explicit channel connections. The result is a fully expanded, standardized network structure that is ready for further processing.

Intermediate Representation Transformation

After reading, the network is transformed into an Intermediate Representation (IR). This IR captures both the topology of the network and the behavior of its actors in a format optimized for analysis and transformation. Preprocessing is applied here to simplify subsequent phases, including normalizing port connections, resolving constants, and preparing guard expressions. The IR serves as the central data structure for all later steps.

Dataflow Analysis

The Dataflow Analysis phase examines the IR to classify actors according to their token consumption and production behavior. It detects sources, sinks, feedback loops, and distinguishes between static and dynamic actors. This classification is crucial for determining how actors will be scheduled and optimized later in the process.

Optimization Phase 1

The first optimization phase is applied before mapping. It focuses on eliminating redundant actors (e.g., pass-through actors) and removing unused channels. This structural simplification reduces computational overhead and ensures that mapping operates on the minimal required graph.

Actor Mapping

This step maps actor instances to processor cores. The mapping process assigns actors to execution units, ensuring that the software generated is multi-threaded. In the absence of an this mapping, actors may require atomic synchronization to avoid concurrent execution on shared state, which can reduce performance.

Optimization Phase 2

Once the mapping is established, a second optimization phase can be applied. Currently left empty but can be set for future use (e.g., actor mergin).

Code Generation

This phase is the final step of the pipeline, transforming the intermediate representation into a complete C/C++ implementation, with actors, channels, and scheduling logic, and producing all necessary files and build configuration to compile and execute the application.

It starts by iterating over the actor variants. Each variant represents a distinct behavioral implementation that may be reused by multiple actor instances. To avoid code duplication, the generator produces one class per variant rather than for each instance. Actor names are derived from their IR identifiers, and generated files are organized into headers. Each generated actor class contains: state variables, actions implementations, and a local scheduler that determines which action to fire next based on token availability and guards.

For channels, which model the FIFO buffers between actors, they are implemented as templates to support various token types, with functions for reading, writing, checking capacity, size bounds, and determining whether the channel is full or empty.

Next is the generation of the main program. It includes instantiating channels, one for each connection in the network, and actor instances with the appropriate input/output channel references, initial parameters and state. This part involves as well executing the global scheduler, which invokes each actor's local scheduler in the proper sequence until completion.

The final step is optional depending on the provided inputs, the generator produces a `CMakeLists.txt` file, listing all generated source files and setting up the required build options.

2.5 The Rust Programming Language

Rust is a relatively young but rapidly maturing systems programming language developed by Mozilla Research and first released in 2010 [21] [22]. Its design goal was to combine the raw performance and control of low-level languages like C and C++ with modern safety guarantees, especially around memory management and concurrency. Since its initial release, Rust has grown into one of the most popular languages for building robust and high-performance software, with a thriving ecosystem and adoption in domains as varied as embedded systems, networking, cryptography, web services, and data processing.

At the heart of Rust's appeal lies its ability to eliminate entire classes of bugs at compile time — including memory corruption, dangling pointers, and data races — while producing native machine code with performance comparable to optimized C. This is achieved through a unique combination of ownership semantics, a strong static type system, zero-cost abstractions, and explicit concurrency control.

2.5.1 Rust's ownership model and borrowing rules

Many programming languages use garbage collector in controlling memory like java, which is a runtime component that has the role of reserving the data in the memory [23] and when the data is no longer in use, it releases this part of the memory. This operation has certain drawbacks because it occurs at runtime in the background, if it wants to clean up the memory it can temporarily pause program execution (also known as freezing that happens for several milliseconds or even seconds). These pauses can slow performance and lead to inefficient outcomes.

In contrast, languages like C++ rely on manual memory management, givings full control over allocation and deallocation through constructs like `new` or modern abstractions such as smart pointers [24]. However this approach makes programs prone to memory leaks, dangling pointers, and other errors .

Rust solves these issues with the concept Ownership. This model works at a compile time to enforce memory safety without the need for a garbage collector. Each value in Rust is "owned" by a variable, and the compiler makes sure that ownership rules are followed. When the owner of a value goes out of scope, the value is automatically dropped. This helps prevent memory leaks and ensures that memory is freed in a predictable way. Ownership rules can be summarized as follows:

- Each value has an owner.
- There can be only one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Rust also allows values to be "borrowed" rather than moved. Borrowing lets code access data without taking ownership, subject to certain constraints. There can be either one mutable reference or multiple immutable references to a value, but not both at the same time. All references must be valid, meaning they must not outlive the data they point to. Rust distinguishes between two types of borrowing:

- Immutable borrowing (&T): allows multiple read-only references.
- Mutable borrowing (&mut T): allows exactly one reference with write access at any given time.

This model helps prevent common bugs like use-after-free, double-free, or data races, all at compile time. In the context of RVC-CAL actor networks, these rules ensure that token buffers, actor state, and communication channels are shared or modified safely without risking undefined behavior.

Additionally, Rust uses lifetimes to track how long references remain valid. Lifetimes make the relationships between variables and references explicit, which is especially helpful in complex scenarios with nested scopes or function calls. While lifetimes add some complexity, they can play an essential role for ensuring correctness in systems that generate or manipulate code involving deeply nested actor interactions.

2.5.2 Concurrency and Thread Safety

Rust was built with concurrency in mind [25]. Its ownership model and type system allow developers to write concurrent code without worrying about common bugs like data races or unpredictable behavior. These safety guarantees are enforced at compile time, rather than at runtime.

When data needs to be shared across threads, Rust requires the use of safe mechanisms such as:

- Channels for sending messages (and transferring ownership of data) between threads.
- Locks like Mutex<T> for safely sharing and modifying data.
- Atomic types for lock-free shared state in specialized cases.

Rust also uses traits like Send and Sync to make sure that only data which is safe to share or move across threads can be used that way. If the data isn't safe, the compiler will reject the program before it

- `Send` indicates that a type's ownership can be transferred safely between threads.
- `Sync` indicates that a type can be safely accessed between threads through immutable references.

While concurrency is about structuring a program so that tasks can be in progress at the same time, parallelism is about running them literally at the same time, often on multiple CPU cores. Rust supports both, and its safety rules apply whether threads run concurrently or in parallel. Libraries like Tokio (for async tasks) and Rayon (for parallel iterators) make it easy to use multiple threads effectively while keeping the program safe.

2.5.3 Rust's Syntax

Rust is designed to be both safe and expressive, and its syntax reflects these goals. While it borrows familiar elements from languages like C++, Java, and Python, Rust introduces new features that support safety and performance. This section provides an overview of Rust's core syntax, using simple examples to illustrate common usage.

Variables and Mutability

By default, variables in Rust are immutable, meaning their values cannot be changed after they are assigned.

```rust
let x = 5;
x = 6;       // This would cause a compile-time error
```

To allow a variable to be changed, the `mut` keyword is used:

```rust
let mut y = 10;
y = y + 1;
```

Data Types

Rust is a statically typed language, so every variable has a type. Types can often be inferred, but they can also be declared explicitly. Rust allows to work with a wide variety of values by categorizing them into different types, each with specific purposes, rules, and capabilities.

Category	Example Types	Purpose
Scalar types	`i32, f64, bool, char`	simple values
Compound types	`(i32, f64), [i32; 3]`	Group multiple values
String types	`String, &str`	Growable String, String Slice
Custom types	`struct, enum`	Define own data structures
Pointer types	`\&T, \&mut T`	References (borrowed data)
Option/Result	`Option<T>, Result<T, E>`	Nullable values, error handling

Table 2.1: *Categories of Rust Data Types*

```rust
let a: i32 = 42;            // 32-bit signed integer
let d: char = "R";          // Character

// Fixed-size collection of elements of the same type
let arr: [i32; 4] = [1, 2, 3, 4];
```

Listing 2.13: *Example of some Data Types declaration*

The following Listing 2.14 illustrates more on the ownership-borrowing rules, through the use of `String` (Mutable and growable, allocated on the heap) and `&str` (Immutable, borrowed view of a string.) types .

```rust
let s1 = String::from("hello");
let s2 = s1; // ownership moves from s1 to s2

println!("{}", s1); // Error: s1 is no longer valid

let message = String::from("Hello, Rust!");
my_function(&message); // message is borrowed, not moved
```

Listing 2.14: *Example of String mutability*

Control Flow

Rust supports all common control flow constructs such as `if`, `else`, `loop`, `while`, and `for`, similar to those found in C and C++. These constructs allow developers to express conditions, iterations, and infinite loops in a familiar way. However, Rust extends these with additional safety and expressiveness. For instance, the `loop` construct is often used in combination with the `break` keyword that can return a value, a feature not present in most traditional languages. Taking an example from the Rust documentation [22], in Listing 2.15 a value is assigned to `result` the `counter` is equal to 10.

```rust
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

Listing 2.15: *Example of match*

The most powerful control flow operator in Rust is the `match` expression, which works similary to `switch` in C++, the difference is that it can match complex patterns. Instead of being limited to constant values, `match` can destructure data types, bind variables, and even apply guards (works as an if condition) to patterns. This makes it an essential tool for working with Rust's data types, such as `enum`, and for writing complex and safe logic.

In Listing 2.16, the `match` expression checks the value of the tuple `point`. Each arm of the `match` corresponds to a different pattern (e.g. the first arm checks for the origin). The final arm _ acts as a catch-all, ensuring that all possible cases are handled. This requirement is one of Rust's safety features, preventing unhandled cases at compile time.

```rust
match point {
    (0, 0) => { /* ... */ }
    (x, 0) => { /* ... */ }
    (0, x) => { /* ... */ }
    _ => { /* fallback */ }
}
```

Listing 2.16: *Example of match*

Functions

Functions in Rust are declared using the `fn` keyword and always include explicit type annotations for parameters and return values. Functions in Rust allow implicit return statements as well, the last expression in a function (written without a trailing semicolon) is automatically returned. Functions can also return tuples or custom data types, which allows multiple values to be returned without the need for out parameters or references.

As shown in Listing 2.17, the `add` function returns the sum of two integers. Because the last expression is not terminated with a semicolon, its value becomes the return value of the function.

```rust
fn add(a: i32, b: i32) -> i32 {
    a + b
}

let result = add(3, 4);
```

Listing 2.17: *Example of Function*

Error Handling

Rust does not use exceptions for error handling. Instead, it uses the `Result` and `Option` types for recoverable errors and `panic!` macro for unrecoverable errors, which immediately terminates the program when a critical error occurs, and returns an error message.

The `Option<T>` type represents values that may or may not exist, and the `Result<T, E>` type represents either success (`Ok`) or failure (`Err`), carrying additional information about the error. Instead of throwing an exception, functions return one of these types, and the caller must explicitly handle both success and failure cases.

The Listing 2.18 demonstrates Rust's approach to safe error handling.

```rust
fn safe_divide(a: f64, b: f64) -> Option<f64> {
    if b == 0.0 {
        None
    } else {
        Some(a / b)
    }
}
fn checked_divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("Division by zero"))
    } else {
        Ok(a / b)
    }
}

fn main() {
    // Using Option
    match safe_divide(10.0, 2.0) {
        Some(result) => println!("Option result: {}",
           result),
        None => println!("Option: Cannot divide by zero"),
    }

    // Using Result
    match checked_divide(10.0, 0.0) {
        Ok(result) => println!("Result value: {}", result),
        Err(e) => println!("Result error: {}", e),
    }
}
```

Listing 2.18: *Error Handling Example*

The safe_divide function returns an Option, providing Some(value) for valid division and None otherwise. In contrast, checked_divide returns a Result, allowing the function to convey specific error messages through Err. The match expressions show how to handle both cases explicitly.

2.5.4 Tokio with Async/Await

Tokio is a popular, asynchronous runtime for Rust that provides the necessary infrastructure to execute async/await code safely on one or more CPU cores [26]. Tokio offers several core capabilities essential for RVC-CAL execution:

- Multi-threaded async runtime: a default work-stealing thread pool schedules and runs asynchronous tasks.
- Synchronization primitives: Tokio includes primitives such as `mpsc` channels, async Mutex, and barriers, that provide safe communication and coordination between actor tasks.
- Blocking thread pool: for actors that perform CPU-intensive or blocking operations, a separate blocking thread pool that preventes synchronous code from stalling the async runtime is provided via `tokio::task::spawn_blocking()`.

The async/await syntax enables writing asynchronous code. Instead of blocking a thread while waiting for an operation to complete, async functions allow tasks to pause and resume without wasting resources. The `async` keyword marks a function as asynchronous, and `await` is used to pause execution until the result is ready. Under the hood, Rust uses futures to represent these pending computations.

When targeting RVC-CAL actor networks, Tokio combined with async/await will allow generated actor code to concurrently run as lightweight, non-blocking tasks that share execution resources without thread-per-actor overhead.

2.5.5 Rayon and Crossbeam

Rayon is a data-parallelism library that focuses on making multi-threaded execution simple for workloads that can be expressed as operations over collections [27]. Instead of manually spawning and managing threads, with Rayon, developers can convert an iterator into a parallel iterator (par_iter(), par_iter_mut(), map(), and for_each()), and the library automatically distribute work across a global thread pool. Rayon's main advantages include:

- Work-stealing scheduler: Threads dynamically balance workloads by "stealing" tasks from other threads' queues, to balance load across available CPU cores.
- Implicit parallelism: The programmer expresses what should run in parallel, not how to distribute it.
- Fork–join model: Parallel tasks are spawned, executed, and then joined back before continuing.

Crossbeam is a library that extends Rust's concurrency capabilities, delivering high-performance and low-latency communication primitives [28]. One of its most important features is the type queue `ArrayQueue`. It avoid traditional locks to reduce contention between threads and has a fixed-size FIFO buffer, which helps prevent uncontrolled memory growth. Another key aspect of Crossbeam is thread-safe communication, where multiple producers-consumers can interact without unsafe manual synchronization, and it uses atomic operations internally to ensure correctness without locking.

Crossbeam supports unbounded channels as well, but only in the module `crossbeam-channel`, not in the `crossbeam::queue` API. In the case RVC-CAL dataflow network bounded channels are preferred, since boundedness is tied to determinism and memory safety in FIFO semantics.

Combining these two libraries, Rayon drives the computation, and Crossbeam moves the data, they form a robust backend for executing RVC-CAL networks in parallel on multicore systems.

2.5.6 Rust's Cargo

Cargo is Rust's official package manager and build system. It is a central part of the development workflow, providing tools to build code, manage dependencies, run tests, and organize projects. Cargo simplifies the process of compiling Rust programs by handling the details of how code is built and linked behind the scenes.

Every Cargo project follows a standard structure, typically consisting of a `src/` directory for source code and a `Cargo.toml` file that defines the project's metadata. This metadata includes the package name, version, dependencies, compiler settings, and more.

One of Cargo's key features is dependency management. Developers can easily include external libraries (called crates) from the online registry crates[29] simply by listing them in the `Cargo.toml`. Cargo automatically downloads, compiles, and links these libraries during the build process. An example is shown in the following listing:

```
[package]
name = "my_project"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = { version = "1", features = ["full"] }
async-trait = "0.1"
```

Listing 2.19: *Cargo.toml Example*

There are different types and categories of dependencies that Cargo can manage other then libraries:

Dev-dependencies used only during development or testing. These crates are not included in the final build if they aren't needed at runtime.

```
[dev-dependencies]
rand = "0.8"  # in tests or benchmarks
```

Build-dependencies used only during the build process, usually for custom build scripts (build.rs).

```
[build-dependencies]
cc = "1.0"  # For compiling C code during build
```

Target-specific dependencies that only apply to a certain OS, architecture, or platform (e.g. only on Windows or only for wasm32).

```
[target.'cfg(windows)'.dependencies]
winapi = "0.3"
```

In the context of RVC-CAL application code generation, Cargo provides a convenient environment for organizing and compiling the generated Rust code. Once the code is generated, it can be placed into a Cargo project, which handles compilation, dependency resolution, and execution.

3 Methodology and Implementation

This chapter begins by explaining how a Rust backend is integrated to the existing dataflow code generator, and explores how Rust can be used to run actor-based Dataflow Process Networks, focusing on RVC-CAL actor networks.

Followed by presenting two distinct execution schemes, each showing a different way to apply Rust's features for this purpose, describing their design, concurrency models, and scheduling strategies. These schemes serve both as practical implementations and as case studies for evaluating different models in Rust.

The chapter concludes with an examination of the structure of the generated Rust code, explaining the role of its main modules and how they work together to manage actors, channels, and scheduling.

3.1 Methodology

The main goal of this work is to extend an existing C/C++-based dataflow code generator with a new backend that produces Rust code. This generator takes RVC-CAL actor definitions and XDF network descriptions, transforms them into an intermediate representation (IR), applies optimizations, and then outputs executable code. While the original backend targeted C and C++, the Rust backend follows a similar process to maintain compatibility and reuse much of the existing parsing and analysis logic.

The integration process began by examining the structure of the existing C/C++ backend. The goal was to understand how actors, channels, and schedulers were represented and generated. The Rust backend was then designed to mirror this structure, ensuring that equivalent functionality existed for actor instantiation, token communication, and scheduling. This approach simplified the transition, as the same high-level dataflow analysis and mapping phases could be reused without major changes.

Actor Generation

The C/C++-backend produces each actor variants as a distinct class. In Rust does not have classes or a built-in concept of class-based inheritance in the traditional object-oriented sense. Instead, it provides alternatives that support many of the same principles as OOP, through a combination of `structs`, `traits`, and `impl` blocks (implementation blocks).

In place of classes, Rust uses structs (short for "structures") to define custom data types. A struct holds related data together, similar to fields in a class.

```rust
struct Actor1 {
    actor_name: String,
    operand_1: Channel<i64>,
    operand_2: Channel<i64>,
    result: Channel<i64>,
}
```

Listing 3.1: *Actor struct*

The struct in Listing 3.1 defines an Actor type with a name and input-output channels. However, on its own, a struct contains only data and no methods. To associate methods with a struct, Rust uses implementation blocks. These are similar to class methods in other languages, allowing methods to be defined on a struct.

```rust
impl Actor1 {
    fn print_action(&self) {
        println!("{} is firing", self.actor_name);
    }
}
```

Listing 3.2: *Implementation block*

In this example Listing 3.2, the print_action method uses self to access the data inside the Actor struct, much like the `this` keyword in C++.

Rust replaces traditional class inheritance with traits, which are similar to interfaces in languages like Java. A trait defines a set of required methods that can be implemented by any type.

```rust
pub trait Actor {
    fn initialize(&mut self);
    fn schedule(&mut self);
    fn is_done(&self) -> bool;
}
```

Listing 3.3: *Actor Trait*

Traits support polymorphism by allowing different types to implement the same interface in their own way (Listing 3.3), without relying on inheritance hierarchies. This allows each actor variant to have its own initialize and local schedule implementation (Listing 3.4).

```
impl Actor for Actor1 {
    fn initialize(&mut self){
        /* ... */
    }
    fn schedule(&mut self){
      /* ... */
  }
}
```

Listing 3.4: *Actor variant impl block*

Additionally, while Rust does not use private, protected, or public modifiers, it offers similar control using the `pub` keyword. Fields and functions are private by default and must be explicitly marked as pub to be accessible from outside the module. This enable to implement a constructor that can be used outside its actor module. A constructor will be implemented as a `new()` function inside of the impl block that returns an instance of the struct (Listing 3.5).

```
impl Actor1 {
    pub fn new(
        name: &str,
        operand_1: Channel<i64>,
        operand_2: Channel<i64>,
        result: Channel<i64>,
        ) -> Self {

            Actor1 {
                actor_name: name.to_string(),
                operand_1: operand_1,
                operand_2: operand_2,
                result: result,
            }
        }
    /* Actor actions */
    }
}
```

Listing 3.5: *Actor Constructor*

Channel Generation

Communication channels between actors are implemented as bounded FIFO channels. These are generated using Rust's libraries (Crossbeam, Tokio mpsc) depending on the chosen execution scheme.

It uses a struct with impl blocks as well for its methods.

Scheduler Generation

The backend produces a global scheduler in main.rs and local schedulers inside each actor. These are adapted from the C/C++ backend's scheduling logic but implemented using Rust's concurrency mechanisms (Tokio's async runtime, Rayon's parallel iterators).

Deeper description on Channel and Scheduler Generation will be at the scheme sections.

Project Structure

The generated Rust project is organized into a modular structure that reflects both Rust's conventional project layout and the requirements of actor-based dataflow execution. This organization will allow the code generator to produce consistent output regardless of the size or complexity of the RVC-CAL network being translated.

```
Cargo.toml            // metadata, dependencies, build config
build.rs              // script integrating native code
native.dir            // dir for native files
src.dir/
- main.rs
- channel.rs          // Channel structs and token handling
- actor.rs            // Actor trait and shared logic
- actor1.rs           // Actor-specific implementation
- actor2.rs
- ....                // Rest of actors
```

Listing 3.6: *Code Organization*

At the top level, the `Cargo.toml` file defines the project's metadata, dependencies, and build configuration. It specifies the external crates used by the generated code, such as tokio, rayon, or crossbeam, depending on the selected execution scheme.

The `build.rs` script provides an integration point for compiling native C code when the generated Rust program depends on external native libraries or performance-critical routines.

For projects requiring low-level native routines, these C source files are placed in a `native/` directory. This separation ensures that platform-specific code remains isolated from the Rust source files, avoiding accidental mixing of build systems or language boundaries.

The generated Rust source code resides in the `src/` directory, following Rust's standard layout. This directory contains the main program entry point, actor definitions, and channel abstractions.

`main.rs` acts as the orchestration hub of the program. It initializes all actors, channels, and schedulers, connects actors according to the parsed network topology, and starts the execution process. The implementation of main.rs differs between Scheme 1 and Scheme 2.

`actor.rs` defines the shared Actor trait and common logic used by all actors. This includes the interface for starting execution, managing ports, and handling termination signals. By keeping this trait in a separate file, the generator ensures that all actors stick to a consistent interface, making it easier to integrate them into different scheduling environments.

`actor1.rs, actor2.rs, ...` contain the concrete implementations of specific actors from the original RVC-CAL specification. Each file declares a Rust struct to hold the actor's state variables and ports, followed by method implementations that encode the actor's finite state machine (FSM) and firing rules. These modules are automatically generated, with naming conventions that correspond to the original actor identifiers in the CAL source.

`channel.rs` implements the communication abstraction used by actors to exchange tokens. The exact implementation depends on the selected scheme: for Tokio, channels are built on top of tokio::sync::mpsc and support asynchronous send/receive operations, while in the second scheme, with Crossbeam they are implemented using a lock-free bounded queue (ArrayQueue) and atomic termination flags.

Build Integration

For systems that combine generated Rust code with existing optimized C code, integration is achieved using Rust's Foreign Function Interface (FFI). FFI allows Rust to call functions defined in external C libraries, enabling interoperability between the two languages.

A key part of this integration is the build script (Listing 3.7), which runs automatically before Rust compilation. The script works in three steps. First, it initializes a compilation context using the `cc` crate, then it discovers all C source files in the `native` directory, and it compiles these files into a static library (`libnative_code.a`) and links it with the Rust binary.

```rust
fn main() {
  let mut build = cc::Build::new();
  let mut found_c = false;
  for entry in fs::read_dir("native").unwrap() {
    let path = entry.unwrap().path();
    if path.extension().unwrap_or_default() == "c" {
      build.file(path);
      found_c = true;
    }
  }
  if found_c { build.compile("native_code"); }
}
```

Listing 3.7: *Build Integration*

The Rust code then declares the C functions using extern "C" blocks, which inform the compiler that the function symbols follow the C ABI and should not undergo Rust name mangling. For example:

```
extern "C" {
    fn source_read_i(buffer: *mut f32, offset: i32, len:
       i32);
}
```

Listing 3.8: *Example Extern C in Rust*

In this project, only `.c` files are included in this FFI integration, excluding C++ sources or other languages. This is because the C ABI is standardized and predictable across compilers, whereas C++ introduces name mangling, exception handling, and compiler-specific variations.

3.2 Scheme 1: Tokio with async/await

Tokio provides an asynchronous runtime for executing tasks without blocking threads. In this scheme, each actor is represented as an asynchronous function (async fn) that awaits input tokens from its inbound channels, processes them according to its actions, and sends results through outbound channels.

Key characteristics of this scheme:

- Concurrency Model: Cooperative multitasking — tasks yield control using .await when waiting for input or performing I/O.
- Channels: Implemented with tokio::sync::mpsc, allowing bounded or unbounded queues with non-blocking send/receive.
- Scheduling: The Tokio runtime maintains a pool of worker threads, multiplexing tasks onto them.
- Termination Handling: Uses Option<T> messages to signal actor shutdown, ensuring upstream and downstream actors terminate cleanly.

Async Channel

In this scheme communication between actors is handled via bounded async channels provided by the tokio::sync::mpsc module (Listing 3.9). Each Channel connects one producer (sender) to one consumer (receiver) with a fixed maximum capacity, ensuring that producers do not overwhelm consumers and that memory usage remains predictable. Each actor uses these channels to send and receive tokens wrapped in an Option<T> which is a clean way to signal end-of-stream without needing a special data value.

A new channel is created using the new_channel() function, which returns a sender and a receiver wrapped in the custom ChannelSender and ChannelReceiver structs. ChannelSender exposes an asynchronous write() method that sends a value into the channel, awaiting until space becomes available or logging a failure if the channel is closed, in practice this should not happen as the size of free slots is checked before writing (with the free() method Line-24). ChannelReceiver provides an asynchronous read() method for retrieving tokens, as well as utility functions (size(), max_size()) to query available capacity and checking channel termination (is_terminated(), is_empty()).

```rust
 1 use tokio::sync::mpsc::{self, Receiver, Sender};
 2 
 3 pub struct ChannelSender<T> {
 4     sender: Sender<T>,
 5     max_size: usize,
 6 }
 7 pub struct ChannelReceiver<T> {
 8     receiver: Receiver<T>,
 9     max_size: usize,
10 }
11 pub fn new_channel<T>(max_size: usize) ->
       (ChannelSender<T>, ChannelReceiver<T>) {
12     let (sender, receiver) = mpsc::channel(max_size);
13     (
14         ChannelSender { sender, max_size },
15         ChannelReceiver { receiver, max_size },
16     )
17 }
18 impl<T> ChannelSender<T> {
19     pub async fn write(&self, value: T) {
20         if let Err(_) = self.sender.send(value).await {
21             println!("Failed to send data to channel!");
22         }
23     }
24     pub fn free(&self) -> usize {
25         self.sender.capacity()
26     }
27 }
28 impl<T> ChannelReceiver<T> {
29     pub async fn read(&mut self) -> Option<T> {
30         self.receiver.recv().await
31     }
32     pub fn size(&self) -> usize {
33         self.receiver.len()
34     }
35     pub fn max_size(&self) -> usize {
36         self.max_size
37     }
```

```
    pub fn is_terminated(&self) -> bool {
        self.receiver.is_closed()
    }
    pub fn is_empty(&self) -> bool {
        self.receiver.len() == 0
    }
}
```

Listing 3.9: *Async Channel*

Actors

Actors in both Schemes are expressed as Rust structs that implement a shared trait, Actor, defined in actor.rs. This trait prescribes three essential methods (Listing 3.10). The initialize(&mut self) method performs any one-time setup needed before execution begins, such as configuring state variables or calling external initialization routines. The schedule(&mut self) method, declared as asynchronous, represents the actor's execution body and is repeatedly invoked until the actor reports completion. The final method, is_done(&self), indicates whether the actor has finished processing and no longer requires scheduling. By adhering to this trait, all actors—regardless of their specific behavior—can be managed by the same scheduling and control logic generated by the backend.

In case of Tokio the schedule(&mut self) method is declared as async because the entire execution model is built on top of Tokio's asynchronous runtime, which expects tasks to be Futures that can yield control back to the runtime while waiting.

```
pub trait Actor {
  fn initialize(&mut self);
  async fn schedule(&mut self);
  fn is_done(&self) -> bool;
}
```

Listing 3.10: *Async Actor Trait*

A representative example of a generated actor can be illustrated in Listing 3.11. This actor generalizes the concept of a stateful dataflow component with input and output channels, an internal state machine, and the ability to call native functions.

The first block `pub struct Actor1{...}` represent everything the actor needs to do its work, from parameters (`initial_sample, actor_name`), internal working variables (`Counter, current_state, done`), to the communication channels it reads from and writes to (`operand_1, result`).

The `impl Actor1{...}` block is where the methods or actions that belong only to this actor are defined. The `new()` function here is a constructor-style function that initializes all fields and returns an actor ready for use. Rust has no special constructor keyword, so we just make a regular public function new that returns Self. The `action()` method represents part of the actor's computation logic. It does some processing (e.g., calling native_function or computations), and writes the result to the output channel. The input channel (operand_1) is been read during local scheduling and the token is passed as a parameter to the action, in this example `x`. The final block `impl Actor for Actor1{...}` is like a public interface implementation so the runtime can schedule and manage the actors.

```rust
use crate::actor::Actor;
use crate::channel::{ChannelReceiver, ChannelSender};
// Native function
extern "C" {
    fn native_function(ind: i32) -> i64;
}
// FSM
pub enum FSM {
    s_init,
    s_run,
}
pub struct Actor1 {
    // Actor Parameters
    initial_sample: i32,
    Counter: i32,
    actor_name: String,
    done_fla: bool,
    // FSM
    current_state: FSM,
    // Input Channels
    operand_1: ChannelReceiver<i64>,
    // Output Channels
    result: Option<ChannelSender<i64>>,
}
impl Actor1 {
    //  Constructor to create a new instance
    pub fn new(
        name: &str,
        initial_sample: i32,
        operand_1: ChannelReceiver<i64>,
        result: ChannelSender<i64>,
    ) -> Self {
        Actor1 {
            Counter: 0,
            actor_name: name.to_string(),
            done_fla: false,
            current_state: FSM::s_init,
            initial_sample: initial_sample,
                operand_1: operand_1,
                result: Some(result),,
            }
        }
        pub async fn start(&mut self) {
            /* ... */
        }
        pub async fn run(&mut self, x: i32) {
            /* ... */
            if let Some(sender) = &self.result {
                sender.write((res).try_into().unwrap()).await;
            }
        }
}
impl Actor for Actor1 {
        fn initialize(&mut self) {
            println!("Initializing {}", self.actor_name);
        }
        async fn schedule(&mut self) {
            /* ... */
        }
        fn is_done(&self) -> bool {
            self.done
        }
}
```

Listing 3.11: *Actor1.rs Tokio*

Initialization in Main

`main.rs` is responsible for creating all channels according to the network topology, instantiating each actor with the appropriate parameters, invoking their initialize() methods and calling the appropriate global scheduling.

```
/*
 * Creates a thread pool (default: number of CPU cores)
 * Async tasks are scheduled across multiple OS threads
*/
#[tokio::main(flavor = "multi_thread")]
async fn main() {
  // Initialize channels
  let ( Actor_1_source1, Add_array_Input1 ) =
     new_channel::<i8>(CHANNEL_SIZE);
  let ( Actor_2_source2, Add_array_Input2 ) =
     new_channel::<i8>(CHANNEL_SIZE);
  let ( Add_array_Output, Actor_3_result ) =
     new_channel::<i8>(CHANNEL_SIZE);

  // Initialize actors
  let mut Actor_1 = Actor1::new("Actor_1", Actor_1_source1);
  Actor_1.initialize();
  let mut Actor_2 = Actor2::new("Actor_2", Actor_2_source2);
  Actor_2.initialize();
  let mut Actor_3 = Actor3::new("Actor_3", Actor_3_result);
  Actor_3.initialize();
  let mut Add_array = Add::new("Add_array",
     Add_array_Input1, Add_array_Input2, Add_array_Output);
  Add_array.initialize();
```

Listing 3.12: *Channels and Actor Instances Initialization*

Scheduling

Scheduling operates on two distinct but complementary levels: local scheduling within each actor and global scheduling at the network level. Together, they ensure that actors execute their logic in the correct order, exchange tokens efficiently, and terminate cleanly once the computation is complete and no token are left in it. Within each actor, local scheduling is implemented in the `schedule()` method defined by the Actor trait Listing 3.13.

This method acts as the actor's private runtime, examines its current FSM state to decide which computational action to perform, checks ports readiness by querying the input channels to determine if enough tokens are available, and if there is enough space to fire an action, executes the action (such as run()) only if all preconditions are satisfied, ensuring compliance with RVC-CAL firing rules, and updates internal state — this might mean changing the FSM state (e.g., from s_init to s_run) or setting the done flag if the actor detects that all inputs are terminated and no more tokens will arrive. The loop is only invoke during Non-Preemptive scheduling to let the actors fire as long as they can.

```
async fn schedule(&mut self) {
 loop {
  FSM::s_run => {
    // if token is available in channel
    if (self.operand_1.size() >= 1) {
        // check action guard
        if (self.Counter  < 10)) {
            if let Some(ref sender) = self.result {
                // check free space in output channel
                if sender.free() >= 1 {
                    match (self.operand_1.read().await) {
                        (Some(operand_1_param)) => {
                            self.run(operand_1_param).await;
                            self.current_state = FSM::s_run;
                    }
                        _ => { // this should never happend as we already check for token availability
                            break;
                        }
                } else {
                    break; // break from current loop
                }
            }
        } else {
            break;
        }
        // if channel is empty and the producer finished his work
    } else if self.operand_1.is_terminated() && self.operand_1.is_empty() {
        self.result = None;
        self.done_flag = true;
        break;
   }
  }
 }
}
```

Listing 3.13: *Scheme 1 Local Scheduler Example*

While local scheduling governs what happens inside an actor, global scheduling coordinates when each actor is given CPU time in the network. The Code generator gives the possibiblity for two scheduling strategies, Non-Preemotive scheduling and Round-Robin Scheduling.

The default scheduling strategy is Non-Preemptive, where actor instances execute as long as they can fire, only after all possible firings are done they return to global scheduling (Listing 3.14). Here, each actor is wrapped inside a Tokio spawn_blocking task, which allows blocking code to run on a dedicated thread pool without freezing the async runtime. Each of these tasks runs the actor's schedule() loop until the actor signals completion.

Once all actor tasks have been spawned, the program waits for their completion using `task1.await.unwrap()`.

`.await` pauses execution until the corresponding task finishes, ensuring that all actors complete their work before the program continues. This prevent premature termination — without it, the main function could return before all actors have processed their remaining tokens. The .unwrap() call then checks the result of the task: if the actor's task panicked or was cancelled, .unwrap() will trigger a panic in main, making the error visible rather than silently ignored.

```
// Non-Preemtive Scheduling
  let task1 = tokio::task::spawn_blocking(move || {
    tokio::runtime::Runtime::new().unwrap().block_on(async {
      while !Actor_1.is_done() {
        Actor_1.schedule().await;
      }
    });
  });
  let task2 = tokio::task::spawn_blocking(move || {
    tokio::runtime::Runtime::new().unwrap().block_on(async {
      while !Actor_2.is_done() {
        Actor_2.schedule().await;
      }
    });
  });

  task1.await.unwrap();
  task2.await.unwrap();
```

Listing 3.14: *Non-Preemtive Scheduling*

The second scheduling approach uses Round-Robin strategy, where each actor gets to fire once, and the scheduler moves on to the next iteration (Listing 3.15). This is achieved using tokio::join!, which allows multiple asynchronous tasks to be executed concurrently and waited on as a group. In each iteration of the loop, every actor is checked: if it is not yet finished, its schedule().await method is called exactly once. The actor then yields control back to the scheduler, regardless of whether it still has work to do. This ensures that all actors are treated fairly and that no single actor monopolizes execution time.

The tokio::task::yield_now().await at the end of the loop explicitly yields back to the Tokio runtime, allowing other tasks (including internal runtime tasks such as actor message passing) to proceed before the next scheduling cycle.

```
loop {
    let (a1_done, a2_done) = tokio::join!(
        async {
            if !actor1.is_done() {
                actor1.schedule().await;
            }
            actor1.is_done()
        },
        async {
            if !actor2.is_done() {
                actor2.schedule().await;
            }
            actor2.is_done()
        },
    );

    if a1_done && a2_done {
        break;
    }

    tokio::task::yield_now().await;
}
```

Listing 3.15: *Round-Robin Scheduling*

3.3 Scheme 2: Rayon with Crossbeam

Scheme 2 implements the generated actor network using a thread-pool, work-stealing execution model provided by the Rayon library and Crossbeams lock-free queues. This approach targets CPU-bound workloads and aims to maximize core utilization by running actors as synchronous tasks on a global thread pool.

Crossbeam Channel

Channels are implemented using the crossbeam::queue::ArrayQueue data structure, which is a bounded, lock-free FIFO queue. The choice of ArrayQueue is motivated by its low overhead and predictable performance under high contention, to be suitable for real-time dataflow execution.

The Channel struct is a thin wrapper around ArrayQueue that also maintains metadata about capacity, termination state, and queue statistics. Wrapping the queue in Arc permits safe shared ownership between multiple actors across threads without copying the buffer itself. The AtomicBool termination flag provides a lightweight, race-free way for producers to announce that no further tokens will be created, and readers can observe this flag and decide to shut down when the queue is empty.

By encapsulating both queue operations and termination logic in a single struct, the Channel type provides a clean abstraction that is easy for the code generator to target. Generated actors only need to depend on this structure, without direct knowledge of Crossbeam's internals.

```
use crossbeam::queue::ArrayQueue;
use std::sync::{atomic::{AtomicBool, Ordering},Arc,};

#[derive(Clone)]
pub struct Channel<T> {
    queue: Arc<ArrayQueue<T>>,
    max_size: usize,
    terminated: Arc<AtomicBool>,
}
```

```
impl<T> Channel<T> {
    pub fn new(max_size: usize) -> Self {
        Channel {
            queue: Arc::new(ArrayQueue::new(max_size)),
            max_size,
            terminated: Arc::new(AtomicBool::new(false)),
        }
    }
    pub fn write(&self, value: T) -> Result<(), T> {
        self.queue.push(value)
    }
    pub fn read(&self) -> Option<T> {
        self.queue.pop()
    }
    pub fn terminate(&self) {
        self.terminated.store(true, Ordering::Release);
    }
    pub fn is_terminated(&self) -> bool {
        self.terminated.load(Ordering::Acquire)
    }
    pub fn size(&self) -> usize {
        self.queue.len()
    }
    pub fn free(&self) -> usize {
        self.max_size - self.size()
    }
    pub fn is_empty(&self) -> bool {
        self.queue.is_empty()
    }
    pub fn is_full(&self) -> bool {
        self.queue.is_full()
    }
    pub fn max_size(&self) -> usize {
        self.max_size
    }
}
```

Listing 3.16: *Crossbeam Channel Struct*

The channels methods can be defined as such:

- `new`: create a new channel with the requested bounded capacity.
- `write`: attempts to push a token into the queue and returns the token back on failure (e.g., if the queue is full).
- `read`: pops a token if available and returns None when empty.

- `terminate`: sets the termination flag so that subsequent readers can see both the flag and prior writes.
- `is_terminated`: returns the termination flag.
- `size`: returns current number of elements in the buffer.
- `free`: return number of free slots.
- `is_empty`: checks if the channel is empty.
- `is_full`: checks if the channel is full.
- `max_size`: returns total channel capacity.

Actor

Unlike with Tokio, Rayon requires Send + Sync, which is important to ensure that actor instances can be shared across threads (or referenced safely) and that the runtime can call actor methods from the Rayon thread pool. Requiring Send + Sync at the trait level forces generated actor types to satisfy Rust's concurrency guarantees at compile time; if an actor contains a non-Send resource, the code will not compile, surfacing concurrency issues early.

```rust
pub trait Actor: Send + Sync {
    fn initialize(&mut self);
    fn schedule(&mut self);
    fn is_done(&self) -> bool;
}
```

Listing 3.17: *Actor trait Rayon*

A generated actor (Listing 3.18) will have the same structure as from scheme1, the difference lies in the type of channels, input token are read from inside the action and not passed as parameters, how channel method are dealt with, and the local scheduling structure.

```rust
use crate::actor::Actor;
use crate::channel::Channel;
/* ... */
pub struct Actor1 {
    /* ... */
    // Input Channels
    operand_1: Channel<i64>,
    // Output Channels
    result: Channel<i64>,
}

impl Actor1 {
    /* ... */
    pub fn run(&mut self) {
        let x = self.operand_1.read().unwrap();
        /* ... */
        self.result.write((res).try_into().unwrap());
    }
}
impl Actor for Actor1 {
     /* ... */
}
```

Listing 3.18: *Actor1.rs Rayon*

Initialization in Main

In Scheme 2 (Rayon + Crossbeam), to be able to pass a channel to more than one actor it has to be cloned. The reason has to do with shared ownership of the same FIFO between multiple actors, and the fact that in Rust, only one owner can hold a value at a time unless it is wraped in a smart pointer like Arc. Because the channel struct derives Clone, calling .clone() does not duplicate the underlying queue; instead, it increments the reference count and makes another handle pointing to the same shared buffer and termination flag. This means that all cloned Channels refer to exactly the same FIFO — if one actor writes to it, another actor with a cloned reference can read from it.

After calling initialize() on all actor instances, the program organizes them into a collection Vec<&mut dyn Actor> that will be passed to the global scheduler.

```
fn main() {
    // Initialize channels
    let Actor_1_source1_Add_array_Input1 =
       Channel::new(CHANNEL_SIZE);
    let Actor_2_source2_Add_array_Input2 =
       Channel::new(CHANNEL_SIZE);
    let Add_array_Output_Actor_3_result =
       Channel::new(CHANNEL_SIZE);

    // Initialize actors
    let mut Actor_1 = Actor1::new("Actor_1",
       Actor_1_source1_Add_array_Input1.clone());
    Actor_1.initialize();
    let mut Actor_2 = Actor2::new("Actor_2",
       Actor_2_source2_Add_array_Input2.clone());
    Actor_2.initialize();
    let mut Actor_3 = Actor3::new("Actor_3",
       Add_array_Output_Actor_3_result.clone());
    Actor_3.initialize();
    let mut Add_array = Add::new(
        "Add_array",
        Actor_1_source1_Add_array_Input1.clone(),
        Actor_2_source2_Add_array_Input2.clone(),
        Add_array_Output_Actor_3_result.clone(),
    );
    Add_array.initialize();

    let mut actors: Vec<&mut dyn Actor> =
        vec![&mut Actor_1, &mut Actor_2, &mut Actor_3, &mut
           Add_array];

    glonal_scheduler(&mut actors);
}
```

Listing 3.19: *main.rs Scheme 2*

Scheduling

As mentioned before, here the local scheduler differs from previous scheme in the semantics of dealing with channels reads and termination conditions.

```
schedule (&mut  self) {
 loop {
  FSM::s_run => {
    if (self.operand_1.size() >= 1) {
     if (self.Counter   < 10)) {
        if (self.result.free() >= 1) {
            self.run();
            self.current_state = FSM::s_run;
        } else {
            break;
        }
     } else {
        break;
      }
    } else if self.operand_1.is_terminated() &&
       self.operand_1.is_empty() {
        self.result.terminate();
        self.done_flag = true;
    } else {
        break;
    }
 }
}
```

Listing 3.20: *Local scheduler Scheme 2*

For the global scheduler, Rayon uses as well an internal thread pool to manage work distribution, so CPU cores are fully utilized, additionally scheduling overhead of constantly creating and destroying threads is avoided.

Rayon allows to use .par_iter_mut(), wich automatically runs schedule() in parallel across available CPU cores, with no manual thread spawning, locking, or task management.

It ensures data race safety via Rust's ownership model. We can only call .par_iter_mut() if the borrow checker confirms no two threads can access the same data mutably at the same time. When the actors are independent, each with their own internal state and references to channels. Rayon can run their schedule() methods in parallel without issues, as long as shared resources (like channels) are thread-safe (which are, via Arc and AtomicBool).

```rust
// Non-preemptive Scheduling
fn glonal_scheduler(actors: &mut [&mut dyn Actor]) {
    loop {
        if actors.iter().all(|actor| actor.is_done()) {
            break;
        }
        // each actor gets to fire as long as it can
        actors.par_iter_mut().for_each(|actor| {
            while !actor.is_done() {
                actor.schedule();
            }
        });
    }
}

// Round-Robin Scheduling
fn glonal_scheduler(actors: &mut [&mut dyn Actor]) {
    loop {
        if actors.iter().all(|a| a.is_done()) {
            break;
        }
        // each actor gets to fire once per round
        actors.par_iter_mut().for_each(|actor| {
            if !actor.is_done() {
                actor.schedule();
            }
        });
    }
}
```

Listing 3.21: *Global Scheduler Scheme 2*

3.4 Rust's Syntax Sensitivity and Type Strictness

An important aspect of implemention in Rust is dealing with the language's strict type system and syntax rules. Unlike more permissive languages such as C or C++, Rust enforces precision in how values are declared, transformed, and used. This strictness plays a central role in preventing bugs, but it also means that small mismatches in types or syntax leads to compilation errors.

For example, Rust differentiates carefully between integer types like i8, i32, and i64. While in C or C++ implicit type promotion often occurs without explicit casting, even if narrowing may introduce subtle bugs since it can lose data silently.

```
// implicit widening
signed char a = -5;
signed int b = a;

// implicit narrowing
signed int a = 300;
signed char b = a;
```

Listing 3.22: *Implicit Conversions in C++*

In this example during widening conversion, signed char (range -128 to 127) is implicitly converted to int (range -2,147,483,648 to 2,147,483,647). No data is lost, because the destination type can represent the full range of char. In the other hand with narrowing conversion, even though it will compile, the behavior may not be what it was expected, because the char type cannot represent all possible values of int. The value 300 does not fit in a signed 8-bit char, so on most compilers b becomes 44 (300 mod 256 = 44)

Rust requires explicit conversions. A variable of type i32 cannot be directly assigned to a variable of type i8 without using methods such as `.try_into()` or `as`. This strictness ensures that narrowing conversions, which could otherwise cause data loss or overflow, are made intentional.

```
let a: i8 = -5;
let b: i32 = a as i32;   // widening

let a: i32 = 300;
let b: i8 = a as i8;     // narrowing, result = 44
```

Listing 3.23: *Implicit Conversions in Rust*

When sending tokens into a channel, the `try_into()` method is used to ensure that the value being written has the correct type. This is necessary because actors may operate with different integer precisions.

```
channel.write( (b).try_into().unwrap() );
```

The `try_into()` method attempts to convert the value into the target type, returning a `Result` that must then be unwrapped or handled. The `unwrap()`

is used to directly extract the converted value, under the assumption that the conversion will succeed within the expected token ranges.

However, using `unwrap()` is risky, if the conversion fails (e.g., when converting an i32 value outside the range of i8), the program will panic at runtime. In practice, this risk is considered acceptable in the generated code because the actor definitions and network semantics typically guarantee that token values stay within valid bounds. This approach intends to avoid cluttering the generated code with unnecessary error-handling logic, while still benefiting from Rust's type safety during development.

Another example is array and vector handling. Rust requires array sizes to be fixed at compile time, written as [T; N] (T is the element type and N is the length), whereas dynamic collections are expressed using Vec<T> type, which is heap-allocated and can grow or shrink at runtime.

```
let arr: [i32; 3] = [1, 2, 3];  // fixed-size of length 3

let mut vec: Vec<i32> = Vec::new();
vec.push(10);
vec.push(20);
```

Listing 3.24: *Rsut Dynamic Collection Example*

Unlike in C/C++, where arrays and pointers can often be mixed freely, Rust enforces a strict separation between [T; N] and Vec<T>. Mixing these two structures without explicit conversion is not allowed. For example, the following will not compile:

```
let arr: [i32; 3] = [1, 2, 3];
let vec: Vec<i32> = arr; //  error: expected Vec<i32>,
    found [i32; 3]
```

Listing 3.25: *Mixing Array and Vector Example*

In the Rust backend, explicit conversions between arrays and collections were not implemented. Instead, Vec<T> collections were preferred for handling cases like list comprehensions that occur in FIFO buffers, they provide more flexibility since their size does not have to be fixed.

For example of this case in RVC-CAL might be such as:

```
action input :[ c_in ] ==> output :[ [n * n : for int n in
    0 .. 31] ] repeat 32
end
```

Listing 3.26: *List Comprehension in FIFO buffer RVC-CAL*

This action reads one token from c_in, generates a list of the squares of numbers 0 to 31, sends it to output, and repeats this process 32 times. In Rust, this is naturally represented using Vec<T>:

```rust
pub fn action1(&mut self) {
    let c_in = self.input.read().unwrap();
    let mut symb_0: Vec<i8> = Vec::new();
    for n in 0..=31 {
        symb_0.push( n * n);
    }
    let mut _2 = 0;
    while _2 < 32 {
        self.output.write((symb_0[_2]).try_into().unwrap());
        _2 += 1;
    }
}
```

Listing 3.27: *Handling List Comprehension in FIFO buffer for Rust*

The same principle applies to ownership and borrowing. A function expecting an immutable reference &T will not accept a mutable one &mut T, and vice versa, without an explicit design decision. These rules may initially appear restrictive but guarantee that generated code is free of undefined behavior.

4 Evaluation

4.1 Benchmarking

To assess the performance of the resuling Rust code, we conducted a series of benchmarks comparing both Rust versions with the original C++. Both backends were used to generate executable code for the same set of RVC-CAL applications, from the ORC-App [18]. The generated code was then compiled and executed under identical conditions to ensure a fair comparison. For rigorous evaluation, execution time was measured across multiple runs (100) to provide a reliable estimate of typical performance. From these runs, we computed the average execution time and standard deviation, which reflect both the central tendency and the variability of the program's performance.

Benchmark Setup

- Hardware: AMD Ryzen 5 5500U with Radeon Graphics (2.10 GHz)
- RAM: 16 GB
- Rust code compiled with release build (cargo build –release)
- C++ code compiled with Optimize Using MinGW -O3 flag
- Benchmark with PowerShell, a skript that uses `(Measure-Command { .\app.exe })` for multiple runs

As of now the Rust backend does not handle type conversion in all cases. For this reason, to be able to perform the benchmarking the files were manually modified. Additonally, the code generator does not currently support multiple reads from the same port, to avoid this, duplicator actors are integrated to the network when needed. These actors take tokens and just resend them through multiple output ports.

4.1.1 ZigBee Multitoken Transmitter

This project contains RVC-CAL descriptions of the IEEE 802.15.4 multitoken transmitter. The network consists of six actor instances: a source that generates payload bits, actors that perform header addition, chip mapping, QPSK modulation, and pulse shaping, connected sequentially, and a sink that collects the processed output and signals completion. The overall dataflow structure is illustrated in the following Figure 4.1:

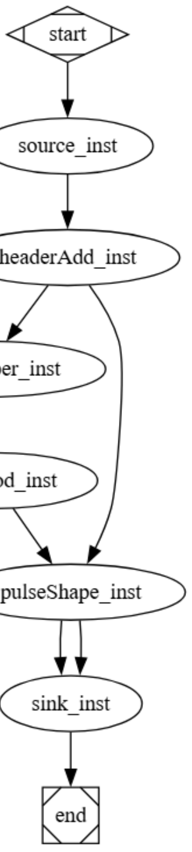

Figure 4.1: *ZigBee Graph*

For the inputs we used the already provided input signals (174 samples) in the project, and a Non-Preemptive strategy for scheduling. To evaluate performance for a larger batch of inputs we created additional input signals with 17,400 tokens. The following table shows the results for this benchmark:

	Rust-Rayon	Rust-Tokio	C++
Avg Time (ms)	2452.95	2924.97	2040.97
Std Dev (ms)	189.47	178.94	98.62

Table 4.1: *ZigBee Multitoken Transmitter Results*

The results indicate that the C++ backend outperforms both Rayon and Tokio. This is mainly due to the scheduling strategies on a predominantly sequential network, where the actor pipeline has limited opportunities for parallel execution. Although Rayon and Tokio both provide efficient concurrency, they are designed to handle arbitrary parallelism. Rayon's work-stealing and Tokio's task polling happen even when there is no parallel work, so their machinery adds overhead that is not needed.

The C++ backend uses a general-purpose multithreaded scheduler. It spawns multiple threads that continuously scan all actors and execute them whenever possible. While this approach adds synchronization overhead, it remains relatively efficient due to the lower runtime costs of raw C++ threads and lock-free channels.

4.1.2 Digital Predistortion DPD

This system implements a digital predistortion block system based on polynomial with FIR filtering. The network includes 9 actor instances: a source that reads samples in batches, Poly instance that generates multiple nonlinear basis signals, five FIR instances applie specific coefficients, adder instance combines the outputs of all FIR branches into the final signals, and the sink instance writes the samples into output files Figure 4.2.

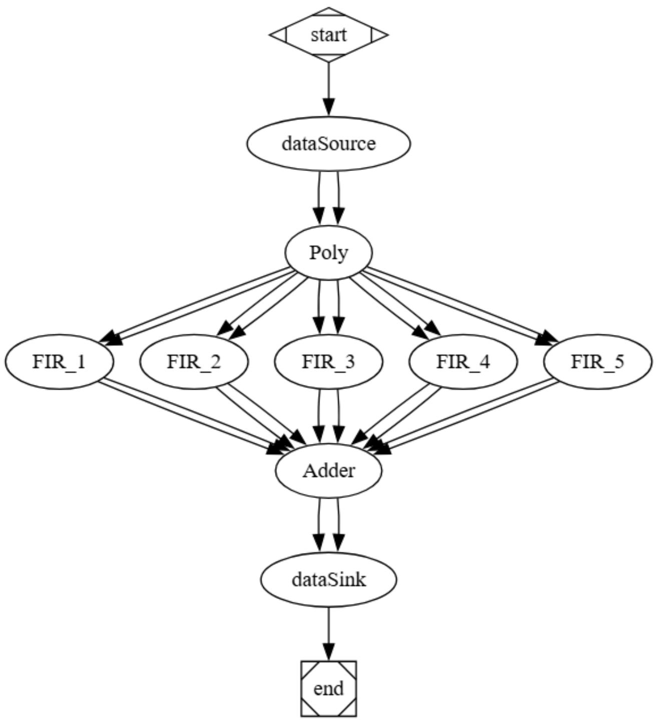

Figure 4.2: *Digital Predistortion DPD Graph*

The provided two input files from this project were concatenated, resulting in a total of 40,000 samples. Same as previously, Non-Preemptive strategy is used for scheduling. The Table 4.2 shows the results.

Unlike previous benchmark the DPD system has five FIR actors operating in parallel. Tokio does not perform as well as Rayon because its model is primarily

	Rust-Rayon	Rust-Tokio	C++
Avg Time (ms)	90.62	128.99	74.12
Std Dev (ms)	34.54	42.33	13.73

Table 4.2: *Digital Predistortion Results*

designed for asynchronous I/O rather than CPU-bound computations, and most DPD actors execute many mathematical operations on large arrays.

Both Rayon and Tokio introduce some overhead due to safety checks, ownership rules, and abstractions for parallelism (like work-stealing queues and bounds checking). Additionally, data conversion between channels and actor inputs/outputs may add extra copying or synchronization that C++ avoids. These points can explain why Rayon execution time is slightly higher than C++.

4.1.3 Digital Filter FIR

This project contains RVC-CAL descriptions of the FIR (Finite Impulse Response) filter. The network contains 16 actor instances, including delays, multipliers, adders, and shift operations, with a source generating input samples and a sink collecting the filtered output (Figure 4.3).

The provided inputs from this project are concatenated (163400 samples).

For FIR Non-Preemptive scheduling is a poor choice in case of Rayon, due to actors monopolizing the worker threads. Rayon's thread pool is based on work stealing, meaning a fixed number of threads are created once and reused. Each thread owns a local queue of tasks, and when it runs out of from, it tries to steal tasks from other threads. When each thread has an idle actor (local scheduler is continuasly invoked but no action is fired and no change of internal state) it cannot run other tasks assigned to it, and the execution of the system does not terminate properly.

The others avoid this hogging problem, where Tokio uses blocking thread pool that can spawn more threads if some threads are blocked. In opposition, C++'s global scheduler runs every thread continuously over all actors.

Round-robin scheduling enforces a switch after each actor fires once, even if it could keep firing, and that makes it more suitable for this case.

	Rust-Rayon	Rust-Tokio	C++
Avg Time (ms)	2079.63	399.79	268.59
Std Dev (ms)	74.67	9.419	39.30

Table 4.3: *Digital Filter FIR Results*

The results from Table 4.3 show that C++ implementation achieves the fastest execution, Rust-Tokio version slightly slower than C++ yet still reasonably efficient, but Rayon performs the worst.

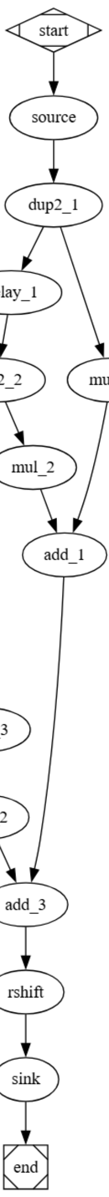

Figure 4.3: *Digital Filtering FIR Graph*

Although Rayon parallelizes actor execution across threads, the fixed thread pool may contribute to the high overhead. To investigate this, we repeated the benchmarking of Rayon with different numbesr of worker threads using `rayon::ThreadPoolBuilder`.

```
ThreadPoolBuilder::new()
        .num_threads(10)
        .build_global()
        .expect("Failed to build global Rayon thread pool");
```

Listing 4.1: *Configure Rayon to use 10 threads Example*

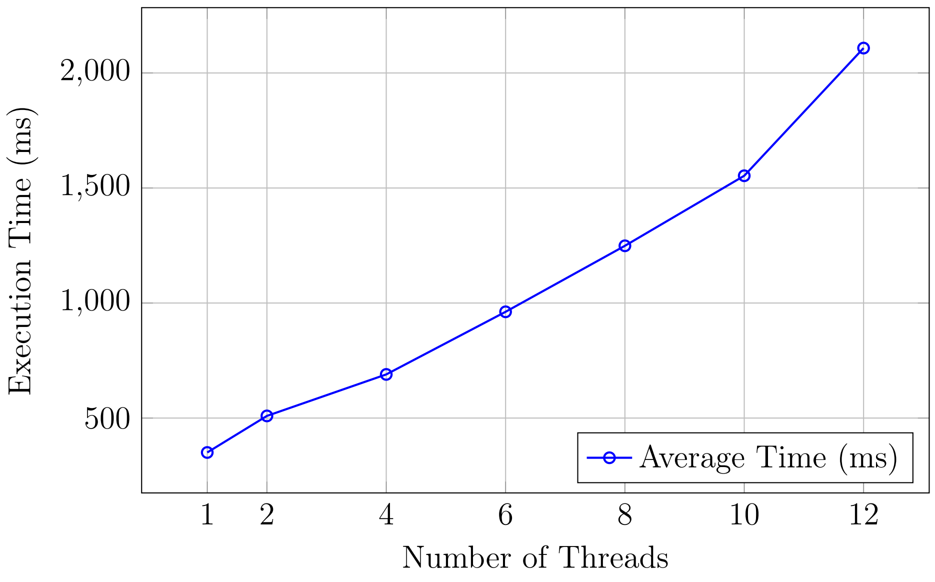

Figure 4.4: *Execution time of Rayon with varying number of threads.*

The Figure 4.4 shows that increasing the number of working threads leads to worse performance. With a single thread, execution time is the lowest (due to the network been mostly sequential), but as more threads are added, the runtime increases steadily. In the first benchmark Rayon used 12 threads by default (AMD Ryzen 5 5500U is a 6-core, 12-thread CPU). This happens because larger number of threads leads to more pooling and work-stealing between the them.

4.1.4 Digital Filter LMS

This project contains RVC-CAL descriptions of the adaptive LMS(Least Mean Squares) filter and uses actors from same packege as FIR. The network consists of 68 edges and 49 actor instances: two sources that generate the inputs, a sink that writes the outputs, and the rest of the instances perform operations such as addition, subtraction, multiplication, and bitwise shifts. The dataflow structure is illustrated at the end of this section (Figure 4.1).

The Round-Robin scheduling strategy was used here as well for the same reasons as previous benchmark and additionally because there is cycles in the network. The original inputs (two files with 16340 token each) were multiplied ten times for a larger batch of inputs. For Rayon we test it with 4-threads as well because C++ backends uses 4-threads by default.

The results in Table 4.4 are similar to the FIR case but with an even larger execution gap between Rayon and the other backends. Reducing Rayon's thread count to 4 improves its performance, yet it still remains slower. This again highlights the overhead of Rayon's work-stealing thread pool when applied to dataflow networks with large sequential dependencies.

	Rust-Rayon 12-threads	Rust-Rayon 4-threads	Rust-Tokio	C++
Avg Time (ms)	17573.85	4254.19	2504.95	1934.50
Std Dev (ms)	471.23	284.60	75.54	464.61

Table 4.4: *Digital Filter LMS Results*

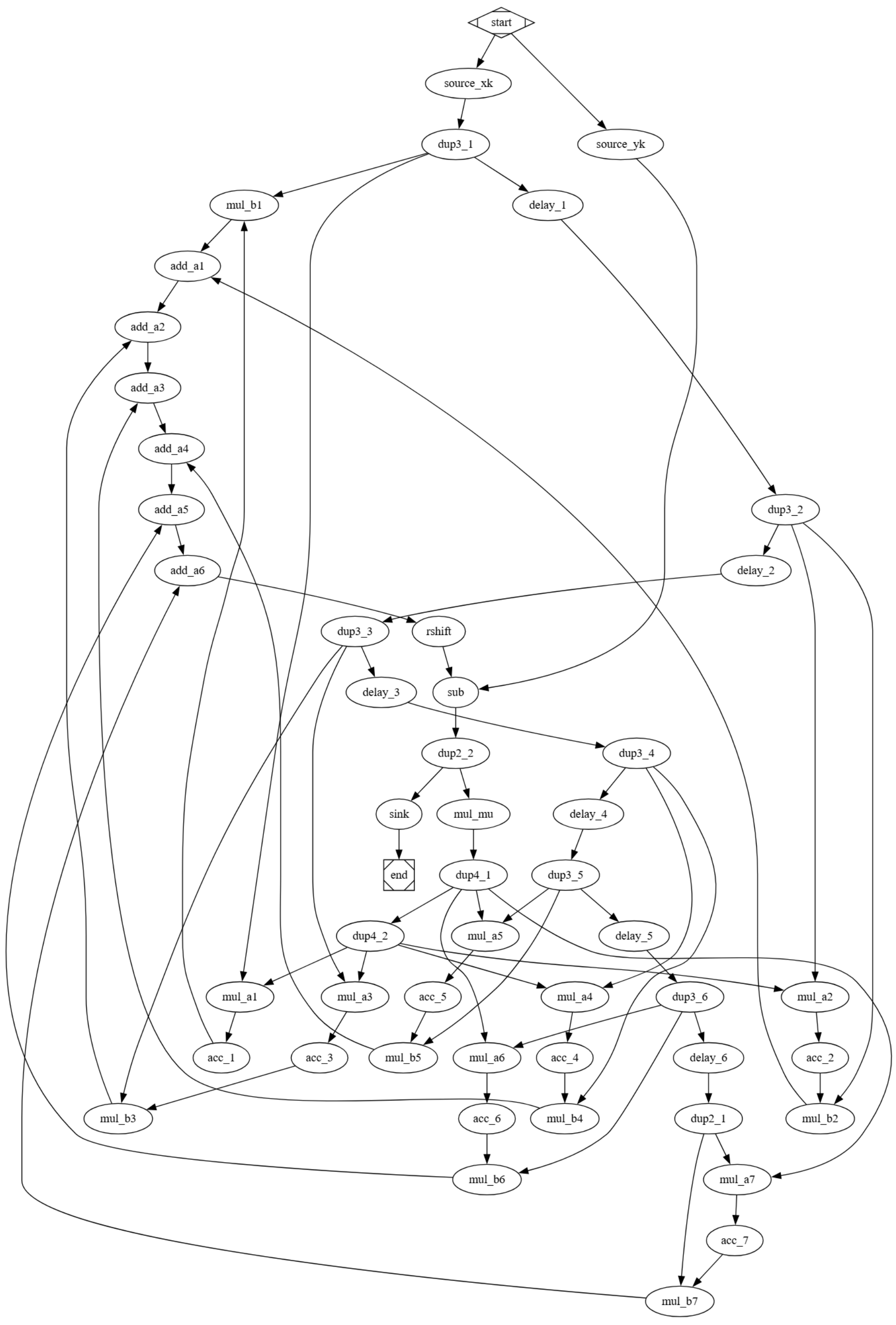

Figure 4.5: *Digital Filtering LMS Graph*

4.1.5 Discussion of Results

The results of the benchmarking show relative performance between Rust backends and C++, which can be explained by the way the languages are designed, how the compilers optimize code, and how each language handles memory and concurrency.

For C++ we used the -O3 option [30], which tells the compiler to perform many aggressive optimizations that include things like combining multiple instructions into one, resulting in removing repeated calculations, predicting loops ahead of time, and rearranging code to make better use of the memory. In Rust, `cargo build -release` passes -C opt-level=3 to LLVM, which almost works as an equivalent to -O3 in C++. This also enables the compiler to apply many optimizations but the key difference is that Rust prioritizes safety, so it includes extra checks to prevent mistakes such as causing data races and dealing with type conversions [31]. These safety checks stay most of the time in the compiled code in contrast, to C++ optimization that allows for more freedom and often removes these checks entirely. This means that, even though both cargo build –release and g++ -O3 enable optimization, C++ code can run slightly faster because it skips certain instructions, while Rust code might run a little slower to ensure it is memory-safe and free of common errors.

Type conversion adds a slight overhead as well. Taking the following example if type conversion is performed on z and c we can distinguish between three cases.

```rust
let x: i32 = (z as _) + (c as _) ;
```

- all 3 variables have the type i32, the conversion makes no difference to the performance. LLVM optimizes away the redundant casts of conversion as if it was not set.

- z or c have different integer type (e.g., i8, u64) it adds a single instruction for conversion.

- z or c have float type (e.g., f32) float to integer conversion, which is more expensive than integer widening/truncation because it involves multiple low-level steps such as rounding the float to the nearest integer.

Using FFI, Rust and C code are able to work together, but it can decrease the performance. Calling C functions from Rust through extern "C" adds a small overhead because Rust must follow the C ABI for argument passing and return values. This cost is usually negligible for operations like reading or writing data blocks, but it can become noticeable in very small, frequent tasks, like calling a C function during every single firing.

However, C and Rust are different languages, therefore not all C operations map directly to Rust syntax or semantics. For example, some C functions may use pointers or memory operations that Rust handles differently. This means it is often needed to adjust the C code so that it aligns with Rust.

Threading and scheduling strategies may further amplify these differences. As mentioned before, with Rayon idle threads steal tasks from busy ones to balance the load, this should help performance but depending on the structure it may instead introduce overhead, as previously seen with predominantly sequential networks.

Tokio, on the other hand, through spawning additional threads when needed, it performs better than Rayon with sequential networks. However because it's model was designed for asynchronous I/O-bound tasks, it can worsen the perforamnce when faced with heavy CPU-bound computations.

Nevertheless, the C++ backend outperformed both Rayon and Tokio across all the benchmarks. This can be attributed to its lightweight multithreaded scheduling, along with the use of raw threads and lock-free channels. These features reduce synchronization costs, enabling efficient execution.

5 Related Work

Several frameworks have been developed to support code generation and dataflow programming. Most of these tools primarily target languages like C, C++ or java, particularly for parallel computing.

PREESM is an open-source framework for rapid prototyping of signal processing applications [32]. It takes synchronous dataflow (PiSDF) models and architecture descriptions as input and generates optimized parallel C or C++ code. Its main strength lies in automatically handling mapping and scheduling on multicore Digital Signal Processors (DSPs) and CPUs, In a similar domain, MAPS designed for modeling actor-based dataflow applications for heterogeneous embedded platforms. It analyzes performance models and emits optimized C code for multicore processors or DSP systems [33] [3].

StreamBlocks is another open-source compiler for heterogeneous dataflow computing. It uses CAL dataflow specifications to generate C++ code for software execution and hardware descriptions for FPGA deployment. [34].

All these tools focus on non-Rust targets. On the Rust side, frameworks like Timely Dataflow provide a parallel and cyclic dataflow runtime written in Rust, designed for low-latency and scalable execution [35]. Timely allows fine-grained progress tracking and scales from single-threaded to cluster environments, but it requires users to implement the graph by manully and does not compile external model specifications.

Suki is an embedded Rust domain-specific language (DSL) [36]. It is designed for distributed and choreographed dataflow, allowing users to explicitly place computations and compile them into zero-overhead binaries. However, the dataflow must still be written in Rust directly, as Suki does not translate external models.

Hydro is a Rust-based framework for building distributed and concurrent programs using a dataflow-inspired approach [37] [38]. It uses an internal intermediate representation called DFIR (Dataflow Intermediate Representation), which allows the framework to represent computation as a graph of interconnected operations. Hydro leverages DFIR to optimize and manage execution within Rust, enabling efficient parallelism and task scheduling. However, Hydro is designed for programs written directly in Rust and does not provide a backend to translate external dataflow specifications.

In summary, although many frameworks exist for dataflow programming, there is currently no tool that can automatically translate external dataflow specifications, such as RVC-CAL, directly into Rust code.

6 Conclusion

In this thesis, we have presented the design and implementation of a Rust backend for a dataflow code generation framework targeting RVC-CAL applications. The main goal was to explore whether Rust could serve as an efficient target language for automatically generated dataflow programs, while preserving concurrency and safety. By extending the existing C++ code generator, we were able to produce Rust code using two different approaches, both capable of executing dataflow networks with multiple actors, channels, and scheduling strategies.

We evaluated the Rust backends through benchmarks against the original C++ implementation. The results showed that both Rust versions performed poorly in cases of networks dominated by sequential actors. This is because their scheduling mechanisms are designed for arbitrary parallelism, which adds overhead by trying to manage parallel execution even when the workload is mostly sequential. Furthermore, Rayon outperformed Tokio on CPU-bound tasks, while Tokio was more effective for I/O workloads.

From these results, we conclude that no single scheduling strategy is optimal for all applications. The best choice depends on the network structure and the type of computations executed by the actors.

Looking ahead, several directions could further improve the this work. First, optimizations in the code generation process could improve runtime performance. Techniques such as actor fusion, token batching, or zero-copy channel communication could reduce overhead and improve throughput, particularly for networks with large numbers of tokens or frequent interactions between actors.

Another future work would be exploring alternative concurrency paradigms that can improve performance for different types of networks. For example Hybrid approaches combining asynchronous tasks and Parallel / CPU-bound Threading (Tokio + Rayon Hybrid) may provide better performance, by combining the strengths of both models. Finally, improving Rust backend by handling type conversions automatically would reduce the need for manual modifications to RVC-CAL files, making the system more accessible to developers.

References

[1] Endri Bezati, Marco Mattavelli, and Jorn W Janneck. “High-level synthesis of dataflow programs for signal processing systems”. In: *2013 8th International Symposium on Image and Signal Processing and Analysis (ISPA)*. IEEE. 2013, pp. 750–754.

[2] Shuvra S Bhattacharyya, Johan Eker, Jörn W Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet. “Overview of the MPEG reconfigurable video coding framework”. In: *Journal of Signal Processing Systems* 63.2 (2011), pp. 251–263.

[3] Jerónimo Castrillón Mazo and Rainer Leupers. *Programming Heterogeneous MPSoCs*. Springer, 2013.

[4] Edward A Lee and Thomas M Parks. “Dataflow process networks”. In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801.

[5] KAHN Gilles. “The semantics of a simple language for parallel programming”. In: *Information processing* 74.471-475 (1974), pp. 15–28.

[6] Joseph Buck and Edward A Lee. “The token flow model”. In: *Data Flow Workshop*. 1992, pp. 267–290.

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

[8] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. “Synthesis of embedded software from synchronous dataflow specifications”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 21.2 (1999), pp. 151–166.

[9] Edward Ashford Lee and David G Messerschmitt. “Static scheduling of synchronous data flow programs for digital signal processing”. In: *IEEE Transactions on computers* 100.1 (2009), pp. 24–35.

[10] Thomas M Parks, José Luis Pino, and Edward 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. IEEE. 1995, pp. 204–210.

[11] Joseph T Buck. “Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams”. In: *Proceedings of 1994 28th Asilomar Conference on Signals, Systems and Computers*. Vol. 1. IEEE. 1994, pp. 508–513.

[12] Johan Eker and Jörn W Janneck. "An introduction to the Caltrop actor language". In: *University of California at Berkeley* (2001).

[13] J Davis II, Christopher Hylands, Bart Kienhuis, Edward A Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, et al. *Heterogeneous concurrent modeling and design in java*. Tech. rep. Technical Memorandum UCB/ERL, 2001.

[14] Johan Eker and J Janneck. *CAL language report: Specification of the CAL actor language*. December, 2003.

[15] Marco Mattavelli, Jorn W Janneck, and Mickaël Raulet. "MPEG reconfigurable video coding". In: *Handbook of signal processing systems*. Springer, 2018, pp. 213–249.

[16] *Open RVC-CAL Compiler*. URL: `http://orcc.sourceforge.net`.

[17] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. "Software code generation for the RVC-CAL language". In: *Journal of Signal Processing Systems* 63.2 (2011), pp. 203–213.

[18] *Open RVC-CAL Applications*. URL: `https://github.com/orcc/orc-apps`.

[19] S Casale Brunet. "Analysis and optimization of dynamic dataflow programs". In: *Diss. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE* (2015).

[20] *Dataflow Code Generator*. URL: `https://github.com/Florian233/Dataflow_Code_Generator/tree/main`.

[21] Nicholas D Matsakis and Felix S Klock. "The rust language". In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. 2014, pp. 103–104.

[22] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[23] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC, 2023.

[24] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[25] Aaron Turon. "Fearless Concurrency with Rust". In: *The Official Rust Blog* (2015). URL: `https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency`.

[26] *Tokio: Asynchronous runtime for Rust*. URL: `https://tokio.rs`.

[27] *Rayon: Data parallelism in Rust*. URL: `https://github.com/rayon-rs/rayon`.

[28] *Crossbeam: Tools for concurrent programming in Rust.* URL: https://crossbeam.rs.

[29] *The Rust community's crate registry.* URL: https://crates.io/.

[30] *Compiler Reference.* URL: https://www.ibm.com/docs/en/xl-c-aix/13.1.2?topic=descriptions-qoptimize.

[31] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. "Towards understanding the runtime performance of rust". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* 2022, pp. 1–6.

[32] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. "Dataflow-based rapid prototyping for multicore DSP systems". In: *Informe técnico PREESM/2014-05TR01, IETR, INSA-Rennes* (2014).

[33] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. "MAPS: an integrated framework for MPSoC application parallelization". In: *Proceedings of the 45th annual Design Automation Conference.* 2008, pp. 754–759.

[34] Endri Bezati, Mahyar Emami, Jörn Janneck, and James Larus. "Streamblocks: A compiler for heterogeneous dataflow computing (technical report)". In: *arXiv preprint arXiv:2107.09333* (2021).

[35] *Timely Dataflow Repository.* URL: https://github.com/TimelyDataflow/timely-dataflow.

[36] Shadaj Laddad, Alvin Cheung, and Joseph M Hellerstein. "Suki: Choreographed Distributed Dataflow in Rust". In: *arXiv preprint arXiv:2406.14733* (2024).

[37] *Hydro Repository.* URL: https://github.com/hydro-project/hydro?utm_source=chatgpt.com.

[38] *Hydro Documentation.* URL: https://hydro.run/docs/hydro/.