



Master Thesis

A formally-verified Network-on-Chip in Coq

Garvit Chhabra

Born on: 29th July 1999 in Rohtak, India
Matriculation number: 5122188

to achieve the academic degree

Master of Science (M.Sc.)

First referee

Prof. Dr.-Ing. Jeronimo Castrillon

Second referee

Dr.-Ing. Sebastian Ertel

Supervisor

Dr.-Ing. Sebastian Ertel

Submitted on: 13th January 2025



Task Description for Master Thesis

For: **Garvit Chhabra**

Degree program: Nanoelectronic Systems

Matriculation number: 5122188

E-mail: garvit.chhabra@mailbox.tu-dresden.de

Topic: **A formally-verified Network-on-Chip in Coq**

At the Barkhausen Institute, we design the next generation of hardware with a special focus on trustworthiness. The evolution of the internet of things makes computer systems an essential part of our daily decision-making process. Such systems will decide when a car can pass another without a collision, a person with diabetes needs an insulin injection and a train can safely enter the station. A failure in all of these situations puts human lives at risk. Research at the Barkhausen Institute focuses on the design and construction of systems that humans can trust, i.e., systems whose decisions are free of unexpected errors. At the very foundation of these system is the hardware.

Computer systems of the future need a formally-verified hardware foundation. The next generation hardware systems need to operate fast but energy-efficient. For that purpose, heterogeneous multi-core systems compose various application-tailored compute units that operate in parallel. To interact with each other, a Network-on-Chip (NoC) connects the different heterogeneous cores. As such, the NoC is the foundation for the next generation hardware systems.

In this thesis, we use formal verification to design a trustworthy NoC. The thesis shall use the theorem prover Coq and the embedded domain-specific language Koika to implement an two-dimensional NoC. The NoC itself shall be parameterizable in at least two aspects: the actual size of the dimensions and the routing itself. The NoC design leaves the attached units entirely abstract such that cores, accelerators and storage can be pluggin creating a highly-heterogeneous hardware platform. Verification focuses on functional correctness, performance characteristics and security properties.


In particular, this Masters Thesis shall include the following tasks:

1. Familiarize with hardware design in Koika.
2. Use Koika to implement a framework that provides the NoC as the foundation of a hardware platforms with (potentially) heterogeneous components.
3. The NoC should be configurable in terms of the actual size of its dimensions and the routing algorithm.
4. Formally-verify properties such as functional correctness, performance guarantees and termination.

Start: 07.08.2024
End: 15.01.2025
1st referee: Prof. Dr.-Ing. Jeronimo Castrillon
2nd referee: Dr.-Ing. Sebastian Ertel
Supervisor: Dr.-Ing. Sebastian Ertel

**Jeronimo
Castrillon Mazo**  Digitally signed by
Jeronimo Castrillon Mazo
Date: 2024.08.02 10:30:50
+02'00'

Prof. Dr.-Ing. Jeronimo Castrillon
(Professor in charge)

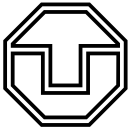
**Thomas
Mikolajick**  Digital unterschrieben
von Thomas Mikolajick
Datum: 2024.08.05
13:36:55 +02'00'

Statement of authorship

I hereby certify that I have authored this document entitled *A formally-verified Network-on-Chip in Coq* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 13th January 2025

Garvit Chhabra



Abstract

This thesis presents a formally-verified Network-on-Chip (NoC) library implemented in Coq, aimed at improving the trustworthiness of communication architectures. Due to the diminishing returns of transistor scaling, SoCs have become the preferred method for building high-performance chips. Inter-core communication is often facilitated by a communication subsystem known as NoC. However, verifying the correctness of NoCs remains a significant challenge because of their large state space and configurable parameters. This work introduces a novel approach that leverages Kôika, an embedded domain-specific language in Coq, to generate NoCs with formal guarantees. The thesis explores the use of metaprogramming and dependent programming to generalize these NoCs, enabling designs of arbitrary sizes with formal correctness proofs. The proposed library aims to reduce verification overhead by proving properties over generic instances, thus eliminating the need to verify every instance of NoC separately. Kôika's formally-verified compiler generates the NoC's Verilog code, which is synthesized to obtain expected hardware resource characteristics.

Contents

Abstract	VII
Symbols and Acronyms	XI
1 Introduction	1
1.1 Motivation	1
1.2 Goal	2
1.3 Structure of the Work	2
2 Background and Foundation	5
2.1 Formal Verification	5
2.2 Network-on-Chip	8
3 Related Work	11
4 Fixed-Size NoC using Kôika	13
4.1 Introduction to Kôika	13
4.2 Simple Pipeline	14
4.2.1 Design	14
4.2.2 Simulation	18
4.3 Advanced Pipeline	20
4.4 Challenges	25
5 From Fixed to Generic NoC	27
5.1 Pipeline NoC Design using Meta-programming	27
5.1.1 Introduction to MetaCoq	27
5.1.2 Generating Inductive types using MetaCoq	29
5.1.3 Generating Definitions using MetaCoq	31
5.1.4 Proving Properties of Generated Code	33
5.2 Dependent Types	36
5.3 Comparison	37

6	Library for Generic NoCs	39
6.1	Pipeline NoC Design using Dependent Programming	39
6.1.1	Defining Inductive Types using Dependent Programming	39
6.1.2	Defining Functions using Dependent Programming	41
6.2	Challenges using Dependent Types in Kôika	45
6.3	Proving Properties of Generic NoC	47
7	Results	49
8	Conclusion and Further Work	53

Symbols and Acronyms

IoT Internet of Things
ABV Assertion-Based Verification
SoC System-on-Chip
IP Intellectual Property
RTL Register-Transfer Level
NoC Network-on-Chip
SVA System Verilog Assertions
EDSL Embedded Domain Specific Language
HDL Hardware Description Language
RHDL Rule-based Hardware Description Language
ORAAT One-Rule-At-A-Time
AST Abstract Syntax Tree
FPGA Field Programmable Gate Arrays

1 Introduction

1.1 Motivation

Due to the rapid development of technology, electronic devices have become ubiquitous. Devices such as processors, cameras, sensors, smart appliances, etc are often interconnected to exchange and process data. This collective network of connected devices is referred to as the Internet of Things (IoT). The evolution of these devices will be pivotal to enhancing operational efficiency, automation, and safety. However, simultaneous with the rise of IoT, the tolerance for bugs will decline. Currently, bugs are usually treated as an inconvenience, but in a world where IoT devices will be found in safety-critical applications, the same bugs may lead to fatal accidents. In contrast to software, patch updates cannot be used to fix bugs in hardware; making verification a crucial part of its design flow.

The goal of verification is to prove that the design meets its specifications. Existing verification techniques comprise a mix of simulation and assertions. Simulation involves testing designs by running numerous input scenarios and checking their outputs. On the other hand, Assertion-Based Verification (ABV) uses embedded assertions to monitor and validate design behavior during simulation or formal analysis. Despite their widespread adoption, both of these techniques are time-consuming and onerous to work with. Needless to say, it was recently reported that more than 50% of median project time is spent on verification [Fos22]. Additionally, these techniques are also not capable of verifying properties like functional correctness and safety, which are key to proving the trustworthiness of complex systems [Ver13].

The limitations of contemporary design and verification techniques also hamper new chip design methodologies. The deceleration of the transistor scaling process, colloquially known as the death of Moore's Law, has encouraged the exploration of other avenues to raise the performance and efficiency of chips. One such avenue is to build chips using the System-on-Chip (SoC) design paradigm. SoCs raise the level of abstraction and integrate a variety of cores, also known as Intellectual Property (IP), on a single chip [RJE03]. These purpose-built specialized cores can achieve better results in their respective tasks, when compared to a general-purpose processor. However, the design of SoCs is inhibited by conventional hardware description languages. The limited programming abstractions and composability provided

by Register-Transfer Level (RTL) languages like Verilog [06] and VHDL [19] clash with the higher-level constructs of the SoC design paradigm. Although, high-level synthesis languages like Vivado HLS [Xil] and Clash [QBa] provide better abstractions and composability, using them often comes at the expense of fine-grained control [Pit+21].

Generally, SoCs use a mesh of routers and wires known as a Network-on-Chip (NoC) to connect the clusters of cores. The correctness of a NoC depends on various facets such as which cores are involved, available routes, and scheduling of packets. Inevitably, this leads to an enormous state space that cannot be verified by techniques like simulation and model checking. Due to their straightforward design, NoCs are also well-suited for configurable implementations, allowing manual adjustment of parameters such as size and routing. However, due to the current state of verification techniques, each configuration needs to be individually verified. This limitation cannot be addressed by System Verilog Assertions (SVA) either, which are considered to be the leading standard of ABV [BP17]. SVAs have similar functionality to C preprocessor macros. Macros operate purely as text substitution tools and do not have an understanding of the actual parameter values used in the design. Due to this limitation, SVAs defined using preprocessor macros cannot dynamically adapt to different configurations during simulation. As a result, SVA can only be verified for one specific parameter value at a time.

Formal verification utilizing theorem provers is a technique that is often overlooked because of the high level of expertise required [BP17]. However, this technique is capable of addressing the limitations of current verification methodologies. Theorem provers can express and verify properties that are generalizable across different parameters of NoC. They can also be used to prove complex properties over the entire possible state space. Also, theorem provers are capable of stating and verifying properties that can be generalized over different parameters of NoC. Prior works have formalized generic NoC properties using the theorem prover ACL2 [SB05], but these implementations could not be synthesized into concrete hardware.

1.2 Goal

This thesis aims to address the concerns discussed in the previous section. The goal of this thesis is to build a library capable of generating formally-verified generic NoCs of different sizes. Reasoning needs to be expanded beyond a concrete design to a size-configurable construction of the NoC. In order to state and verify properties of this type, tools with stronger reasoning capabilities are required. To address this challenge, a novel design and verification approach using Kôika [Bou+20] is adopted. Kôika is an Embedded Domain Specific Language (EDSL) in Coq theorem prover, it provides higher-level abstractions than RTLs while retaining control over generated circuits. The integration with Coq, enables the use of its proving capabilities for verifying the complex properties of a NoC. Another benefit of the theorem prover is that the proven properties can be generalized over the size of NoC. This considerably reduces the verification overhead required, by eliminating the need to individually verify every instance of NoC size possible.

1.3 Structure of the Work

This thesis is organized into eight chapters.

- Chapter 2 provides the necessary background on formal verification and NoCs, laying the foundation for the subsequent chapters.
- Chapter 3 surveys related work in the field of formal verification for NoC, highlighting the existing techniques and their limitations.
- Chapter 4 focuses on fixed-size NoC designs using the hardware description language Kôika. It includes an introduction to Kôika, a simple pipeline design, and its simulation, followed by a discussion of challenges to be addressed to design a generic NoC.
- Chapter 5 explores the transition from fixed-size to generic NoC designs by leveraging meta-programming and dependent programming. It introduces MetaCoq and demonstrates how inductive types and definitions can be generated programmatically, and proving properties of generated code. In addition to this, a brief introduction to dependent types, followed by a comparison between the two methodologies is also presented.
- Chapter 6 presents a library for generic NoC designs, focusing on implementations using dependent programming. This chapter discusses dependently typed inductive types, dependent programming using the Equations plugin, and the associated challenges.
- Chapter 7 presents the results of the research, comparing the synthesis results of the two approaches.
- Chapter 8 concludes the thesis by summarizing the key contributions and suggesting directions for future work.

2 Background and Foundation

In this chapter, a brief overview of key concepts relevant to the thesis is presented, including the state of verification, formal verification and its various approaches, as well as NoCs and their different aspects.

2.1 Formal Verification

According to the IEEE Standard 1012 [12], verification is "the process of providing objective evidence that the system, software, or hardware and its associated products conform to requirements." Thus, verification is a key process in hardware design flow which ensures that the design conforms to the specification. The criticality of the verification phase can be inferred from Fig. 2.1, which shows the percentage of time spent in verification in design projects. It can be inferred from the graph that most design projects spend 50-60% of their project time on

Percentage of ASIC Project Time Spent in Verification

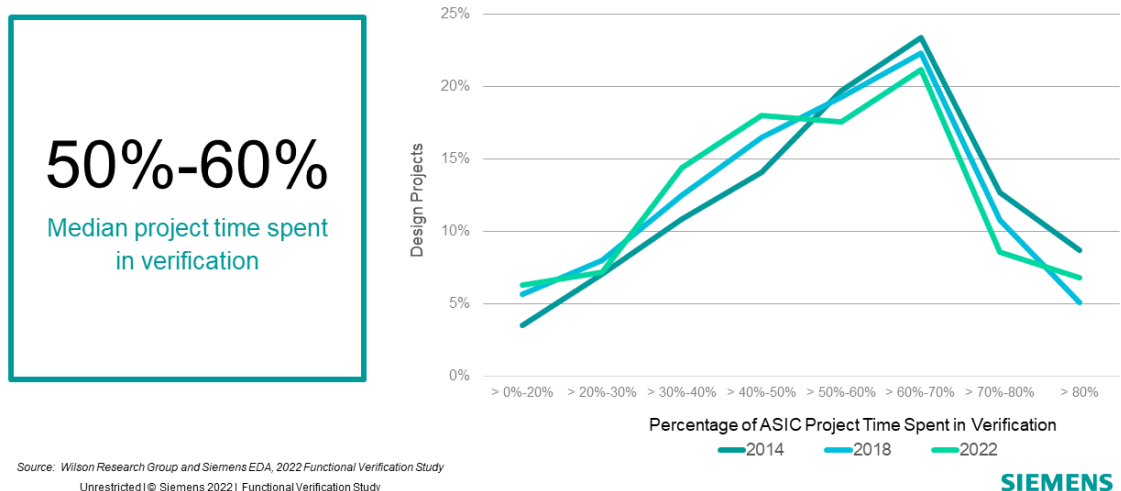


Figure 2.1: Percentage of IC/ASIC project time spent in verification [Fos22]

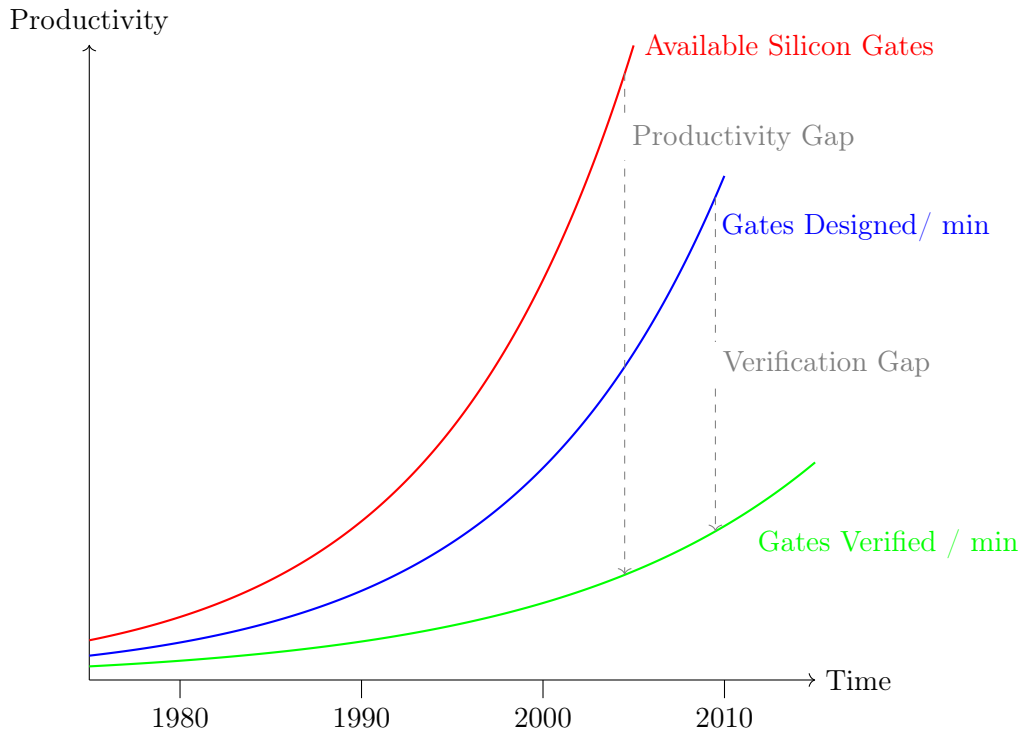


Figure 2.2: Illustration of productivity and verification Gap [Riz02]

verification. Also, the more recent results from 2022 are trending toward the right, suggesting that even more time is being used for verification. Only 5-10% of the projects spend less than 20% of their time in verification. These projects often integrate pre-verified IP. The lack of parameterization and availability of these IPs inhibit their adoption.

Modern-day verification relies heavily on simulation, it is often considered to be the gold standard of verification [BM06]. In simulation, a series of inputs, also known as tests, are run and the output is compared with the expected result. If the output doesn't match the expected result, it means there is a bug in the design or a discrepancy in the test case. The commonly known drawback of this approach is that a lot of effort is needed to cover every possible scenario in building these tests. The effect of this drawback can be seen in Fig. 2.2. The number of available transistors was doubling according to Moore's Law until the aughts, and although the gates designed per minute were not completely utilizing the available silicon, they still scaled better than gates verified per minute because of techniques like IP reuse, SoCs, high-level synthesis, and standardization. In addition to the fact that existing techniques are inefficient, the verification problem is exacerbated because the number of gates is exponentially related to the number of states to be verified. This correlates with the large verification gap, which consequently leads to the enormous productivity gap shown in Fig. 2.2.

Formal verification is a methodology to assess the correctness of a system by formalizing it into a mathematical model. The principles of hardware verification align closely with those of mathematical proving. The description and constraints of the circuit serve as the **premise** of the proof, with the specification being verified, treated as the **proposition** [PMS18]. The advantage of following this mathematical approach is that a proposition if proven holds for all possible cases; in contrast to simulation, where the property is verified for only a handful of cases simulated. This limitation of writing test cases to verify programs has been recognized

Aspect	Model Checking	SAT Solvers	Theorem Proving
Automation	✓	✓	✗
Scalability	✗	✗	✓
Verification Scope	✗	✗	✓

Table 2.1: Comparison of formal verification strategies for hardware

since the 70s, when Dijkstra famously remarked, "program testing can be used to show the presence of bugs, but never to show their absence!" [DW70].

Formal verification is generally conducted through three main strategies:

1. **Model Checking** - Performs exhaustive exploration of a finite-state model.
 - Input - State machines with specifications in temporal logic.
 - Automation - Mostly automated; the state space exploration is systematic and tool-supported.
 - Scalability - Limited by state space explosion; does not handle infinite states well.
 - Verification Scope - Limited; useful for specific properties (e.g., safety, liveness) in finite systems.
 - Tools - SPIN, NuSMV, UPPAAL.
 - Strengths - Systematic and exhaustive; identifies counterexamples for debugging.
 - Weakness - State space explosion; limited to finite models.
2. **SAT Solvers** - Transforming problems into SAT instances and solving them using algorithms.
 - Input - Boolean formulas in Conjunctive Normal Form (CNF).
 - Automation - Fully automated; requires formulating the problem in SAT form.
 - Scalability - Scalable only for finite problems.
 - Verification Scope - Effective only for combinatorial problems, logic synthesis, and specific hardware properties.
 - Tools - MiniSat, Z3, Glucose.
 - Strengths - Efficient for solving Boolean satisfiability.
 - Weakness - Requires translation of the problem into SAT.
3. **Theorem Provers** - A mathematical proof is formalized such that its correctness can be ensured by a computer program.
 - Input - Formal specifications and mathematical properties.
 - Automation - Mostly manual; requires human expertise to formalize proofs.
 - Scalability - Can handle infinite-state and parametric systems
 - Verification Scope - Can verify broader and more abstract properties, including parametric systems.
 - Tools - Coq, Isabelle, HOL.
 - Strengths - Highly expressive and flexible; supports abstract reasoning.
 - Weakness - Non-trivial proofs may require expertise and effort.

A comparison of key aspects of the three strategies is provided in Table 2.1. It can be inferred that, if the manual effort is put into verification; theorem provers can provide better scalability and scope compared to the other two techniques. Additionally, they can also be used to verify

parametric systems (like NoCs). Needless to say, theorem provers were chosen to build the generic NoC library presented in the coming chapters.

2.2 Network-on-Chip

As mentioned in Chapter 1, transistor scaling has hit the point of diminishing returns. The cost of adding additional transistors is much higher with little to no gains in performance. The power density of designs hasn't been constant since the end of Dennard scaling [Esm+11b]. Modern-day chips are built using the SoC design paradigm to tackle this issue. The SoC design paradigm is more efficient, reduces design complexity, and can be used to design heterogeneous architectures. In SoCs, various cores specialized for different tasks are integrated into a single design. Although this addresses the issues with performance and power consumption, enabling communication between these cores still remains a significant challenge.

In a SoC, a variety of prefabricated blocks also known as IPs or cores are integrated. A core can refer to processors, memory controllers, or communication interfaces. Traditionally bus-based architectures were used to communicate across multiple cores, these architectures were favoured because of their simplicity and efficiency in terms of power and silicon cost. These architectures consisted of a simple master-slave connection with a single data and control bus. However, in such architectures, a rise in the number of connected cores can lead to resource contention and drastic reductions in performance [Lee+07]. Thus, a bus interface is not scalable when a huge number of cores are involved [Ver13].

NoCs were proposed as a solution to the scalability problem of bus architectures [Kum+02]. The proposition behind NoCs was to replace the dedicated routing wires between top modules with a communication network made of routers and wires that can route packets between the cores [DT01]. Using NoCs as on-chip interconnect provides many advantages like high-bandwidth and scalability with lower latency and power consumption [BP17].

A NoC leverages the principles and techniques of computer networking to facilitate communication within a chip. The important aspects of a NoC are its topography, switching technique, and routing algorithm. Like computer networks, NoCs can also be built into various 2D and 3D topologies like mesh, torus, ring, star, binary tree, etc. The switching technique and routing algorithm of the NoC determines how packets are handled inside the network. In this thesis, a packet-switched NoC with XY-routing algorithm is implemented. In packet-switched NoCs the data is divided into packets. A packet comprises of two components:

- Header - Contains the data required for transmission, like the destination address.
- Payload - Contains the data that is being transmitted.

In these types of networks, the next hop is determined at every router. Since every packet contains a header, all packets are capable of traveling autonomously from one router to the next.

The routing algorithm of the NoC determines the paths that could be taken by the packets. These algorithms are generally classified into two [Ver13]:

- Deterministic - The same path is taken for a given source-destination pair, e.g. XY Routing.

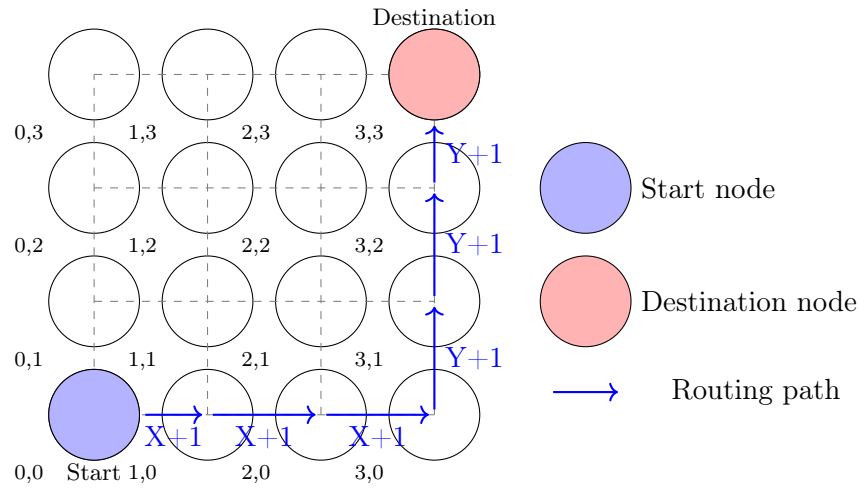


Figure 2.3: A packet-switched mesh NoC with XY routing

- Adaptive - Routes are dynamically determined based on network conditions like congestion or faults, e.g. West-First Routing.

Fig. 2.3 illustrates the movement of a packet from (0,0) to (3,3) in a packet-switched mesh NoC with XY routing. In this algorithm, the packet first traverses in the X-dimension, once the packet has reached the X-dimension of its destination it starts traversing in the Y-direction. Since the NoC is packet-switched, the routers determine the direction of the next hop at each step. Once the packet reaches its destination, the router sends the packet to the destination core.

3 Related Work

The principle verification efforts on embedded communication have been presented in this chapter. In the past, primarily the effort was spent on verifying the cores, thus literature dedicated to the verification of communication architectures is relatively sparse.

Due to their high automation, model checkers have been the preferred method for formal verification of hardware. The initial works in verifying communication architectures were focused on bus protocols.

Roychoudhury et al. [RMK03] used a model checker to debug an RTL-level implementation of AMBA AHB protocol, however, the design was static and had no parameters. Amjad [Amj04] verified the safety properties of AMBA, APB, AHB protocols using a model checker, implemented in the HOL theorem prover. This work also uses a model written with a low level of abstraction for a fixed design.

Works involving model checkers and NoCs often run into problems related to state space explosion. These works attempt to reduce or abstract over the design and are often only able to verify particular aspects of it [ZH14]. Chen et al. [Che+10] implemented a Bidirectional channel Network-on-Chip (BiNoC) model in a state graph manipulator. Using this technique, they were able to prove mutual exclusion and also demonstrated the failure of starvation freedom property in BiNoC. However, they were unable to prove deadlock freedom because of the state space explosion problem. A system-level verification of HERMES NoC was performed by Vinitha et al. [PS11] using the SPIN model checker. The work proves properties over smaller NoC models and suggests that proof-based verification can be used to extend the properties to larger models.

Driven by the advent of SoCs and platform-based design, the trend in hardware design is to raise the level of abstraction and use verified parameterized IPs [Ver13]. However, as discussed in this chapter, most verification techniques use model checkers on low-level RTL with completely static hardware blocks. Thus, developing verified IP that remains configurable for designers poses a significant challenge.

Theorem provers can be used to overcome this challenge. Unlike model checkers, they are capable of reasoning about models without a fixed size. This was shown in the work by Gebremichael et al. [Geb+05], where deadlock-freedom in \mathcal{A} ethereal protocol was proven for an

arbitrary number of master and slave devices. Theorem provers also provide abstract data types which make them suitable for dealing with higher levels of abstraction.

One drawback of the studies discussed until now is that they are dedicated to particular protocols like AHB, \AA ethereal, etc. Schmaltz et al. [SB05] were the first to provide a formalization of communication networks. The main contribution of their work was a generic network-on-chip model, also known as GeNoC, a formal framework encoded in ACL2 that encompasses the essential constituents of communication networks.

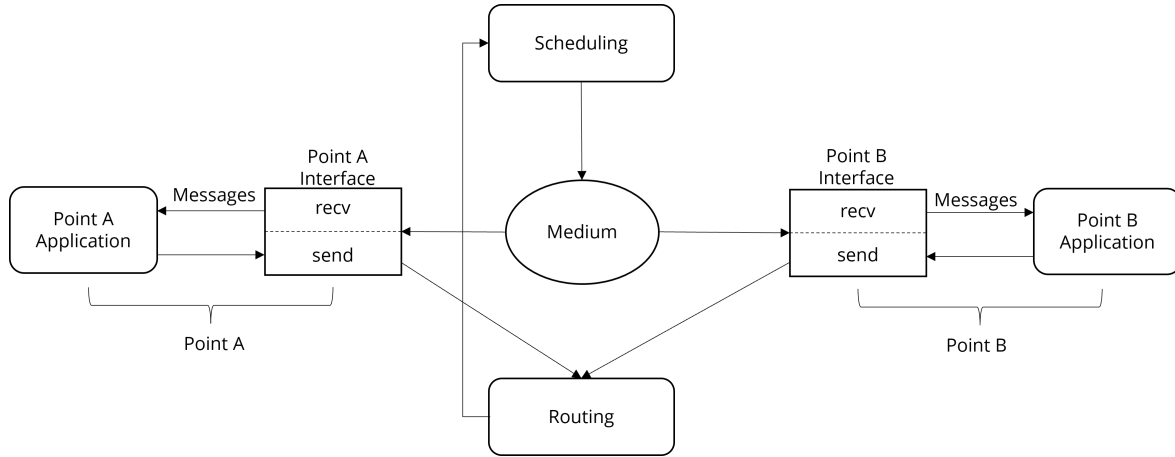


Figure 3.1: GeNoC model

The model of GeNoC is shown in Fig. 3.1. GeNoC not only states properties arbitrary to the size of NoC, but also parameterizes over these key functions:

- Interface - This is built up of two functions: *send* to encapsulate data into packets and *receive* to extract the data from a packet.
- Routing - Computes all valid routes for a pair of source and destination.
- Scheduling - Generates a list of possible communications that can be completed depending upon various conditions like traffic, priority, etc.

GeNoC has been used to instantiate various NoCs like HERMES [Bor+07], Octagon [SB05], and Spidergon [SB05], the latter being a state-of-the-art NoC from industry. Broek et al. [BS09] verified both packet-switched and circuit-switched NoC. Later, Verbeek et al. [VS10] extended the work with additional theorems to prove deadlock freedom and network evacuation.

Formalizations like GeNoC provide valuable insights but fail to establish a link to a concrete NoC implementation that can be synthesized. In contrast, Kôika is used in this thesis because it overcomes this limitation, enabling circuits to be compiled into Verilog and synthesized.

4 Fixed-Size NoC using Kôika

From this chapter onwards, the design of our NoC in Kôika begins. Here a brief introduction to Kôika is provided, followed by the design and simulation of a simple 3-stage pipeline NoC using the language.

4.1 Introduction to Kôika

A hardware program is essentially a cycle-accurate description of register-transfers. It can be said that each register holds the *state* of the circuit. RTLs like Verilog and VHDL allow the expression of state change concurrently. However, if not managed correctly this can lead to undesirable behaviour. If a specific register is manipulated by multiple concurrent blocks in the same cycle, conflicts in the value will be observed and the system will fail to adhere to its specification. Also, as previously mentioned in Chapter 1, these languages also don't provide high-level abstractions and composable semantics. These limitations often lead to design bugs and necessitate huge efforts for the design and verification of modern-day systems.

Kôika, an EDSL in Coq is presented as the solution. Together with Bluespec [Nik04] and Kami [Cho+17], it falls into a category of languages referred to as Rule-based Hardware Description Language (RHDL). These languages have higher-level abstractions and more composable semantics than RTLs with the benefit of better control over generated circuits than high-level languages. In rule-based designs, the manipulation of hardware state elements (i.e. registers) is described by rules. These rules are atomic (executed independently) and follow One-Rule-At-A-Time (ORAAT) semantics. However, executing ORAAT semantics on hardware will severely limit performance. Thus rules are executed concurrently, but the notion of atomic execution is preserved.

Figure 4.1 provides an overview of the Kôika compiler. The compiler here consists of a chain of functions that optimize and lower Kôika programs into Verilog programs. The compiler also defines operational semantics for both Kôika and Verilog. Based on these definitions, the compiler states and proves the theorem which ensures compilation preserves the semantics of the Kôika program. Since Kôika programs are written in Coq, its operational semantics can be used to formally reason about the properties of their execution.

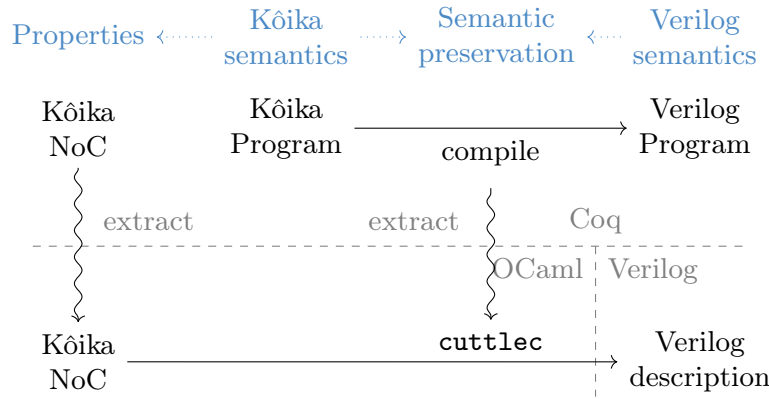


Figure 4.1: The Kôika compile(r) and the Kôika programs are extracted from Coq into OCaml. The formal reasoning is not.

Coq programs cannot contain input/output (I/O) operations, such as reading the Kôika program from a file or writing the compiled Verilog program to a file. For that reason, programs need to be extracted out of Coq, i.e., transpiled, into a language where I/O is allowed. Coq does provide in-built mechanisms for its extraction into OCaml and Haskell. The transformation and compilation to OCaml is itself formally-verified in Coq. To execute the Kôika compiler, it is extracted into OCaml, resulting in the executable `cuttlec`. Additionally, Kôika is extracted into OCaml to provide input for `cuttlec`. Note that all formal reasoning parts are not part of the extraction. Note that the formal reasoning components remain within Coq and are excluded from the extracted code, similar to how test cases are not included in production systems.

4.2 Simple Pipeline

In this section, the design of a 3-stage unidirectional pipeline NoC is presented, to demonstrate Kôika's programming model. In the latter half, the testing of the pipeline using Kôika's semantics is shown.

4.2.1 Design

The pipeline designed in this subsection is shown in Fig. 4.2. The NoC receives packets from `noc_enter` and sends them outside using the `noc_exit` interface. The hardware consists of three routers, two channel registers, and one acknowledge register (not shown in the figure). To start writing code in Kôika, some libraries have to be imported into Coq as shown in Code 4.1

```
1 Require Import Koika.Frontend.
2 Require Import Koika.Std.
```

Code 4.1: Importing Kôika libraries

To demonstrate how a design is described in Kôika, the program has been divided into 4 parts:

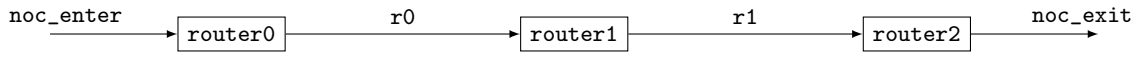


Figure 4.2: A simple 3-stage NoC pipeline.

1. Setting up Registers and I/O First in Code 4.2, the registers are defined as an inductive type. An inductive type allows the creation of new terms of its type from constants and functions. The registers `r0`, `r1`, and `r_ack` are constants that form the terms of the defined inductive type `reg_t`. The registers `r0` and `r1` are used as channel registers, and register `r_ack` is used to write the acknowledge value from `noc_exit`. This is vital to prevent the interface from being optimized out.

```

1 Inductive reg_t :=
2   | r0
3   | r1
4   | r_ack.

```

Code 4.2: Creating registers in Kôika

Next, the type and width of the registers are specified using a **Definition**. Code 4.3 defines a mapping that matches all registers to the type `bits_t` with a width of 4-bits (Line 3-5).

```

1 Definition R reg :=
2   match reg with
3   | r0 => bits_t 4
4   | r1 => bits_t 4
5   | r_ack => bits_t 4
6   end.

```

Code 4.3: Defining types of registers

The register values are initialized using a similar approach. All 3 registers are initialized to 0 as shown in Code 4.4. This could be done by either using the `Bits.of_nat` (Line 3 and 4) function which takes the width of the bits and value to be converted into bits as arguments or using `Bits.zero` (Line 5) which infers the width implicitly.

```

1 Definition r reg : R reg :=
2   match reg with
3   | r0 => Bits.of_nat 4 0
4   | r1 => Bits.of_nat 4 0
5   | r_ack => Bits.zero
6   end.

```

Code 4.4: Initialization of registers

Kôika also provides external functions which can be used to model external IP. The input and output interfaces of the design are also decided by the external functions in the design. The declaration of external functions `noc_enter` and `noc_exit` is shown in Code 4.5. These will be used as the input and output interfaces of the NoC respectively.

```

1 Inductive ext_fn_t :=
2   | noc_enter
3   | noc_exit.

```

Code 4.5: Creating external functions

Similar to `reg_t`, the types (signatures) of the external function are initialized as shown in Code 4.6. Here, both the external functions take an argument of type `bits_t 4` and return an output of the same type.

```

1 Definition Sigma (fn: ext_fn_t) :=
2   match fn with
3   | noc_enter => {$ bits_t 4 ~> bits_t 4 $}
4   | noc_exit => {$ bits_t 4 ~> bits_t 4 $}
5   end.

```

Code 4.6: Defining types of external functions

2. Functionality In the next step, the functionality of the hardware is defined. The routers of the NoC are implemented as actions, i.e. functions that do not have any input. Due to this limitation, actions can only communicate with each other via registers. The type of actions is `uaction reg_t ext_fn_t`. This type represents untyped actions, which modifies the registers, represented by `reg_t`, and interacts with external interfaces represented by `ext_fn_t`.

The first router - `router0` is shown in Code 4.7. In this router, the packets are fed into the NoC using the interface `noc_enter()`. The function `extcall()` is used to invoke external functions (Line 3). Every external call must have an input argument; in this case, `4`d0` is passed. The packet is then sent downstream to `r0` using `write0`; a primitive function in Kôika which is used to modify the state of registers.

```

1 Definition router0 : uaction reg_t ext_fn_t :=
2   {{
3     let msg := (extcall noc_enter(|4`d0|)) in
4     write0(r0, msg)
5   }}.

```

Code 4.7: Leftmost router.

`router1` reads the packet from upstream channel `r0` using the `read0` primitive function (Line 3). Then the packet is written to the downstream channel `r1` (Line 4) as shown in Code 4.8.

```

1 Definition router1 : uaction reg_t ext_fn_t :=
2   {{
3     let msg = read0(r0) in
4     write0(r1, msg)
5   }}

```

Code 4.8: Intermediate router

`router2` reads the message from `r1` (Line 4) and pushes the message out of the NoC using the `noc_exit()` interface (Line 5) as shown in Code 4.9.

```

1 Definition router2 : uaction reg_t ext_fn_t :=
2   {{
3     let msg = read0(r1) in
4     write0(r_ack, extcall noc_exit(msg))
5   }}.

```

Code 4.9: Rightmost router

Kôika also provides additional primitive functions like `read1` and `write1` to relax the restrictions of the ORAAT semantics. These primitives can be used to exchange data between two rules in the same cycle. Their usage is as follows: data written by `write0` in a register can be read by using `read1` on the same register in the ongoing cycle. Any data written by `write1` can only be read in the next cycle. The additional primitives can be used to increase the speed of the NoC being built, but, the goal is to create generalized functions which can be reused to form a generic NoC. Having two types of routers using different primitives will complicate the design. Therefore, for the scope of this thesis, these primitives are not used.

3. Schedule Kôika’s transactional execution model guarantees that no data races occur. Kôika programs are made of rules, every rule is a program that runs once per cycle, as long as it does not conflict with other rules. In Code 4.10 the rules are defined using an inductive type similar to `reg_t` and `ext_fn_t`.

```

1 Inductive rule_t :=
2   | r_router0
3   | r_router1
4   | r_router2.

```

Code 4.10: Defining rules

In Code 4.11, the actions i.e. routers defined in the previous part are mapped to the rules created in Code 4.10.

```

1 Definition to_action rl :=
2   match rl with
3   | r_router0 => router0
4   | r_router1 => router1
5   | r_router2 => router2
6   end.

```

Code 4.11: Mapping rules to routers

The order of execution of rules is orchestrated by the `schedule` defined in Code 4.12. It is of type `scheduler` (Line 1), an inductive type in Kôika which has terms like `done` and `Cons`, the latter being represented by the `|>` notation. In the schedule shown rule `r_router0` is executed

first, followed by rule `r_router1` and then rule `r_router2`. The end of a schedule is marked by `done`.

```
1 Definition schedule : scheduler :=
2   r_router0 |> r_router1 |> r_router2 |> done.
```

Code 4.12: Scheduling the rules.

4. Type checking Lastly, the rules need to be type checked. Code 4.13 demonstrates a definition that checks for type mismatches and returns an error when there are unbounded variables, functions with wrong return types, mismatches in register sizes, etc. This function uses `R` from Code 4.3, `Sigma` from Code 4.6, and the `to_action` mapping from Code 4.11. A similar function `tc_function` is also available to type check functions.

```
1 Definition rules :=
2   tc_rules R Sigma to_action.
```

Code 4.13: Type checking for Kôika rules

4.2.2 Simulation

Simulation of a design in Kôika is based on its semantics. This attribute of Kôika can be used to verify designs. In this subsection, a short demonstration of writing a simple test case in Kôika is shown. The first step here is to import the library for testing which contains additional functions that will be used in testing. This is done by using `Require Import Koika.Testing`.

First, the registers are initialized and external functions are modelled as shown in Code 4.14. In Code 4.14b on Line 3, `noc_enter` returns `x + 5`, with `x` being the input to the external function, which is `|4`d0|` in this case (can be seen in `extcall` of Code 4.7). `noc_exit` is modelled as an identity function, thus register `r_ack` will be written with the same value that is being sent out using the output interface.

```
1 Definition r_test (reg : reg_t) : R reg :=
2   match reg with
3   | r0 => Bits.zero
4   | r1 => Bits.zero
5   | r_ack => Bits.zero
6   end.

(a) Initialize registers
1 Definition sigdenote fn : Sig_denote (Sigma fn) :=
2   match fn with
3   | noc_enter => fun x => x + b (Bits.of_nat 4 5)
4   | noc_exit => fun x => x
5   end.
```

(b) Model external functions

Code 4.14: Setting up registers and external functions for testing

Code 4.15 demonstrates a simple lemma called `test_1` to test the pipeline. To simulate one cycle the `run_schedule` function is used. This function uses the initialized register values (`r_test` in Code 4.14a), type checked rules (`rules` in Code 4.13), model of external function (`sig_denote` in Code 4.14b), and schedule (`schedule` in Code 4.12) (Line 2). The function produces the context which stores the states of the registers. The states in the registers are then updated from `ctxt1` into `r'` (Line 3-9). The values in the registers have to be updated every cycle by fetching new values from the context. Once the schedule has been run the desired number of times, 3 in this case, the values inside specific registers can be accessed at different points of the simulation by using different contexts (Line 13-15). In this case, the goal checks after every cycle if the packet is in the correct register, this is done with the help of the union (`/\`) operator (Line 16-18).

```

1  Lemma test_1:
2    run_schedule r_test rules sigdenote schedule
3    (fun ctxt1 =>
4      let r' := (fun idx =>
5        match idx with
6        | r0 => ctxt1.[r0]
7        | r1 => ctxt1.[r1]
8        | r_ack => ctxt1.[r_ack]
9      end ) in
10   ...
11   run_schedule r'' rules sigdenote schedule
12   (fun ctxt3 =>
13     let bits_out := ctxt3.[r_ack] in
14     let bits_mid := ctxt2.[r1]
15     let bits_in := ctxt1.[r0] in
16     Bits.to_nat bits_out = 5 /\ (* Check r_ack router after 3rd cycle *)
17     Bits.to_nat bits_mid = 5 /\ (* Check r1 router after 2nd cycle *)
18     Bits.to_nat bits_in = 5  (* Check r0 router after 1st cycle *)
19   ))).
```

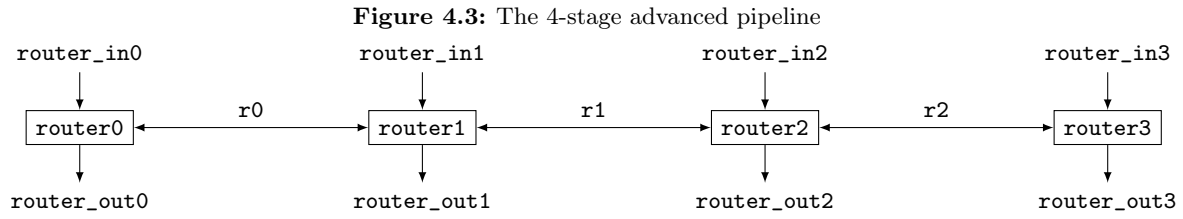
Code 4.15: Writing a test Lemma in Kôika

This test case can be simply computed using a first-order solver `check` as shown in Code 4.16 (Line 2). `Proof` keyword in Coq is used to enter proof mode. Inside proof mode, Coq provides various tactics that can be used for proofs. Similarly, to exit the proof mode the keyword `Defined` is used (Line 3).

```

1  Proof.
2    check.
3  Defined.
```

Code 4.16: Proof



4.3 Advanced Pipeline

In Section 4.2, a simple unidirectional 3-stage pipeline was demonstrated, the design uses dedicated functions for each router, therefore it lacks scalability and is not suitable for generic NoC generation. In this section, more advanced features from Kôika are utilized to achieve the following goals:

1. The NoC routes the packet to the correct destination using the address in the packet.
2. Each router has its own input and output interface, so the packet can enter and leave the network from any router.
3. Support for bidirectional movement of packets.

The updated representation with the new features is shown in Fig. 4.3. Since the NoC is supposed to be packet-switched, each packet should have fields additional to data, containing information about the transport. Thus, packets require a structured data type (like struct in C) to store data pertaining to the packet, such as destination address. Also, since the size of the NoC is configurable, the size of the address field also has to scale accordingly.

The solution to this problem is quite straightforward; Kôika allows us to create structures (or structs) similar to conventional programming languages. A `basic_packet` type is created which will serve as the type of packets.

```

1 Definition addr_sz := log2 (S nocsize).
2
3 Definition basic_packet :=
4   {
5     struct_name := "basic_packet";
6     struct_fields :=
7       [
8         ("new", bits_t 1);
9         ("src" , bits_t addr_sz);
10        ("dest" , bits_t addr_sz);
11        ("data" , bits_t data_sz)
12      ]
13   }.

```

Code 4.17: Defining `basic_packet` type

On Line 1, the size of the address field (`addr_sz`) is initialized in terms of the size of the NoC. Thus `basic_packet` can be used for any NoC regardless of its size. The following fields are present in `basic_packet`:

1. **new** - Records the status of the packet, if the packet hasn't been processed by the router it is set to 1 otherwise 0 (Line 8).
2. **src** - Contains the address of the router from which the packet is coming from (Line 9).
3. **dest** - Contains the address of the destination router where the packet has to be routed to (Line 10).
4. **data** - Contains data being transferred in the packet (Line 11).

In the simple design, every router had a dedicated action, but for a generic NoC this is not feasible. Another challenge in the design is that every channel register is shared by two routers, thus every router has a non-mutual set of registers.

The solution to the first problem is quite intuitive. It is evident that all intermediate routers of the NoC perform the same operation i.e. move elements from one channel register to another. This observation enables us to categorize the design of a pipeline-based NoC into three distinct types of router functions:

1. Start router (leftmost)
2. Middle router (intermediate)
3. End Router (rightmost)

Since each router works with different registers the router actions need to be polymorphic over the registers, otherwise, each router action would have to be manually created. The solution is to define **send** and **receive** functions that are polymorphic over the registers, as shown in Code 4.18.

```

1 Definition r_send (reg_name: reg_t) : UInternalFunction reg_t ext_fn_t :=
2   {{
3     fun r_send (value: bits_t sz) : unit_t =>
4       write0(reg_name, value)
5   }}.

                                     (a) send function
1 Definition r_receive (reg_name: reg_t) : UInternalFunction reg_t ext_fn_t :=
2   {{
3     fun r_receive () : bits_t sz =>
4       read0(reg_name)
5   }}.

                                     (b) receive function

```

Code 4.18: Polymorphic **send** and **receive** functions

Now, as the **send** and **receive** functions are polymorphic, these same functions can be used for any register. These functions will be used for the router actions of the design. The functionality of the middle router is shown in Code 4.19.

```

1 Definition _router_middle
2   (r_addr_p: nat)
3   (r0_send r1_send r0_receive r1_receive r_ack: UInternalFunction reg_t ext_fn_t)
4   (router_in router_out : ext_fn_t)
5   : uaction reg_t ext_fn_t :=

```

(a) Function Header for middle router function

Code 4.19: Action for middle router

In Code 4.19a the function header for the polymorphic middle router action is defined. The parameters of this type of router are:

- **r_addr_p**: Each router should know its own address to compute the direction in which incoming packets have to be sent. Since the router needs to be polymorphic, the address cannot be part of the router function, thus it was chosen to be an argument to the function.
- **r0_send**, **r1_send**, **r0_receive**, **r1_receive**, and **r_ack**: These represent the polymorphic **send** and **receive** functions binded with their respective registers. **r0_send** and **r1_send** are **send** functions with the registers on the left and right side of the routers as arguments respectively. Similarly, **r0_receive** and **r1_receive** are the **receive** functions. **r_ack** function is used to pass the acknowledge signal for the router's output signal.
- **router_in** and **router_out**: To provide an input and output interface for each router the external functions are also parameterized.

```

6 let r_addr := #(Bits.of_nat addr_sz r_addr_p) in
7 let zero := #(Bits.of_nat sz 0) in
8 let nocsize := #(Bits.of_nat addr_sz nocsize) in
9 let m_input := (extcall router_in(#bz)) in
10 let msg := unpack(struct_t basic_packet, m_input) in
11 let new_data := get(msg, new) in
12 let dest := get(msg, dest) in
13 let src_p := get(msg, src) in

```

(b) Handling input from **router_in**

Code 4.19: Action for middle router

In Code 4.19b, first three variables are bound in Line 6-8: the address of the router (**r_addr**), the value 0 (**zero**) which is used to make external function calls, and the size of the NoC (**nocsize**) which is used to check invalid packets. In this design, the router prioritizes inputs from its own interface over the inputs from the channel. Hence, first, it checks the interface by invoking **router_in** on Line 9. Kôika also provides a number of functions to work with structs. Here **unpack** is used to retrieve the message as a **basic_packet** (Line 10). Using the **get** function one can retrieve the data from a specific field in a struct type (Line 11-13).

```

14  if (new_data && src_p == r_addr && dest != r_addr && dest < nocsiz) then
15      if dest > r_addr then
16          r1_send(m_input)
17      else
18          r0_send(m_input)

```

(c) Sending packet towards its destination

Code 4.19: Action for middle router

In Code 4.19c, the `if` statement in Kôika is used to check the validity of the packet (Line 14). The packet is valid if:

- `new_data != 0`: The data is new and hasn't been processed already.
- `src_p == r_addr`: The packet coming from the input interface should have the address of the router as its source.
- `dest != r_addr`: The router input should not send a packet that has the same router's address as its destination.
- `dest < nocsiz`: The destination cannot be an address greater than size of the NoC.

If the packet is valid, it is sent towards its destination; if the destination address (`dest`) is greater than the router address (`r_addr`) it is sent right using `r1_send`, otherwise left using `r0_send` (Line 15-18).

```

19  else
20      let m0 := r0_receive() in
21      let m1 := r1_receive() in
22      let msg := unpack(struct_t basic_packet, m0) in
23      let new_data := get(basic_packet, msg, new) in
24      let src_p := get(basic_packet, msg, src) in

```

(d) Handling input from `r0_receive`

Code 4.19: Action for middle router

Next in Code 4.19d, the packets arriving from the channel registers are handled. Both packets are read using their respective `receive` functions (Line 20-21). This is done before routing, to avoid the contents in one register from being overwritten by the contents of the other register. In more advanced implementations, buffers and scheduling policies can be used to circumvent this issue. The packet from `r0_receive` is given a higher priority, therefore it is unpacked first (Line 22). Similar to Code 4.19b, the necessary fields from the packet are extracted (Line 23-24).

```

25  if src_p != r_addr && new_data then (
26    r0_send(pack(subst(basic_packet, msg, new, 0b~0)));
27    let dest := get(basic_packet, msg, dest) in
28    if dest > r_addr then
29      r1_send(pack(subst(msg, src, r_addr)))
30    else
31      if dest < r_addr then
32        r0_send(pack(subst(msg, src, r_addr)))
33      else
34        r_ack(extcall router_out(m0));

```

(e) Handling input from `r0_receive`**Code 4.19:** Action for middle router

In Code 4.19e, the validity of the packet is checked (Line 25). The `new` bit in the register is reset, to ensure that the packet is not processed again in the next cycle (Line 26). Then the packet is sent towards its destination by comparing the address of the router with the destination address in the packet (Line 28-32). If the destination address and router address are equal the data is sent to the core by calling the `router_out` external function (Line 34). The acknowledge signal from the output is written to the router's respective state register using the polymorphic function `r_ack`.

The functionality for forwarding the packet read from `r1_receive` remains similar to the one defined in Code 4.19d and 4.19e; thus it has been omitted here. Also, the actions for the start and end router will be cut-down versions of the middle router, capable of communicating with only one channel register; these also have not been presented for the sake of brevity.

```

35  Definition router_middle (n:nat) (r1 r2 r_ack: reg_t) (e1 e2: ext_fn_t):
36    uaction reg_t ext_fn_t :=
37      _router_middle n (r_send r1) (r_send r2) (r_receive r1)
38      (r_receive r2) (r_send r_ack) e1 e2.

```

(a) Wrapper function for `_router_middle` action

Next in Code 4.20a, a function `router_middle` is created to wrap up the polymorphic `send` and `receive` functions with their corresponding registers. This is not necessary for the functioning of the action but it keeps the header more readable.

```

39  Definition to_action (rl: rule_name_t) :=
40    match rl with
41    | router_0 => router_start 0 r0 r_ack0 router_in0 router_out0
42    | router_1 => router_middle 1 r0 r1 r_ack1 router_in1 router_out1
43    | router_2 => router_middle 2 r1 r2 r_ack2 router_in2 router_out2
44    | router_3 => router_end 3 r2 r_ack3 router_in3 router_out3
45  end.

```

(b) `to_action` definition for advanced pipeline**Code 4.20:** Mapping actions to polymorphic routers

Lastly, Code 4.20b shows the `to_action` function which is responsible for mapping the rules to actions. Here, the benefit of using polymorphic functions can be seen as separate actions are not required for each rule; `router_middle` is configured with its registers during the function application. Therefore, any number of intermediate rules can be mapped to the same polymorphic `router_middle` action by changing its arguments.

4.4 Challenges

In the previous sections, the design of a fixed-size pipeline NoC using Kôika has been demonstrated. This design methodology is inadequate to generate or reason about generic NoCs. In its current form, the inductive types i.e. `reg_t`, `ext_fn_t`, and `rule_name_t` need to have a fixed number of elements which depends on the size of the NoC. This requirement comes from both Kôika and Coq as they require inductive types to be well-founded and finite.

An inductive type with a runtime-dependent number of constructors (shown in Code 4.21) will be disallowed by Coq, since it needs a fixed, syntactic definition of constructors to ensure complete pattern matching and the termination of all programs.

```

1 Inductive reg_t (n : nat) : Type :=
2   | r_0 : reg_t n
3   | r_1 : reg_t n
4   ...
5   | r_n : reg_t n

```

Code 4.21: Inductive datatype with runtime-dependent number of constructors

A recursive inductive type is an alternative that can be used to circumvent this. However, Kôika programs cannot use dynamic or recursive types. Since Kôika needs an upper bound to be established for every program for successful compilation to Verilog. Consequently, Kôika ensures every inductive type in the program is finite.

A similar challenge is faced when it comes to defining the schedule and the rule to action mapping function. These functions depend on the inductive types and hence they cannot be defined before the size of the NoC is fixed.

The challenge is not only restricted to the design of the generic NoC but also to its verification. Currently, the available `run_schedule` performs the computation of the rules in the schedule. It can only be used with a design of fixed-size with a fixed number of rules. Since the goal is to reason about a parameterized NoC, `run_schedule` won't be sufficient.

In summary, to create a library for generic NoC generation and to reason about NoCs of variable sizes, the simple inductive types used up to this point are not sufficient. In Chapter 5, the possible solutions to these problems are discussed. Considering the challenges, the next chapters will continue to focus on the pipeline topology, to avoid further complicating the design.

5 From Fixed to Generic NoC

As discussed in Section 4.4, the first challenge to overcome in building a library for generic NoC generation is to be able to generate inductive types with a finite but arbitrary number of elements. In this chapter, two solutions using different programming paradigms are presented and compared.

5.1 Pipeline NoC Design using Meta-programming

The first approach involves meta-programming. Meta-programming enables the user to write programs in a meta-language that can produce or manipulate programs written in an object language. In this section, the implementation of a generic pipeline using meta-programming is demonstrated.

5.1.1 Introduction to MetaCoq

MetaCoq [Soz+20] is a framework and library for meta-programming, reflection, and reasoning about Coq itself. It allows us to write Coq terms as an Abstract Syntax Tree (AST). The syntax of Coq as entered by the user is referred to as the concrete syntax. The representation of Coq terms, types, and proofs as data structures (ASTs) that can be manipulated within Coq itself is called the reified syntax. MetaCoq also includes mechanisms to convert one form of representation to another. The *quote* operation is used to convert concrete to reified syntax and the *unquote* operation performs its inverse i.e. from reified syntax to concrete syntax. This is diagrammatically illustrated in Fig. 5.1.

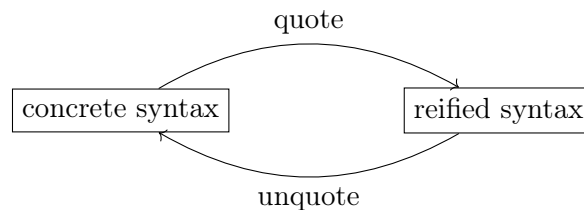


Figure 5.1: Conversion of syntax in MetaCoq

A brief demonstration utilizing these mechanisms is provided in this subsection.

```

1 Inductive nat : Set :=
2   | 0 : nat
3   | S : nat -> nat.
4
5 Definition one := S 0.

```

Code 5.1: Defining natural numbers in Coq

Code 5.1 shows the representation of natural numbers. In Coq, natural numbers are defined inductively using 0 (zero) and S (successor), i.e. every natural number can be defined using 0 or the successor of another natural number (Line 1-3). So the natural number `one` is defined as the application of successor on 0 (`S 0`) in the syntax of Coq (Line 5).

```

1 MetaCoq Quote Definition reified_one := one.
2 Print reified_one.
3   (* tApp
4     (tConstruct
5       {/
6         inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
7         inductive_ind := 0
8       /} 1 []) (* Reified S *)
9   [tConstruct
10    {/
11      inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
12      inductive_ind := 0
13    /} 0 []). (* Reified 0 *)
14   *)

```

Code 5.2: Quoting in MetaCoq

Code 5.2 demonstrates the `MetaCoq Quote` command. This command is used to convert the concrete syntax of `one` to reified syntax (Line 1). The reified syntax is defined as the term `reified_one`. The breakdown of the reified representation on Line 3-14 is as follows:

- The term `one` is the application of function `S` on `0`. The function `tApp` is used to represent function application in AST (Line 3).
- `S` is represented as the second constructor of `nat` using `tConstruct` (Line 4) with index 1 (Line 8). Here, the inductive type `nat` is identified by its module name and path (Line 6 and 11).
- `0` uses a similar representation but since it is the first constructor in `nat` it is indexed with 0 (Line 13).


```

1 MetaCoq Unquote Definition concrete_one := reified_one.
2 Print one.
3   (* concrete_one = 1
4     : nat
5     *)

```

Code 5.3: Unquoting in MetaCoq

The reified syntax of any term can be converted back to its concrete syntax by using `MetaCoq Unquote`. This is shown on Line 1 of Code 5.3.

Another useful command in MetaCoq is `MetaCoq Run`, which allows the execution of Template Monad computations. Template Monad refers to a programming abstraction used in MetaCoq which is used to perform computations that affect the Coq environment.

```

1 MetaCoq Run (tmMkDefinition "foo" reified_one).
2 (*Performs the same operation as: Definition foo := 1. *)

```

Code 5.4: Creating definitions in Coq using `MetaCoq Run`

In Code 5.4, the `reified_one` generated in Code 5.2 is used to define the natural number 1. The function `tmMkDefinition id t_r` is a Template Monad operation that converts the reified syntax (`t_r`) to a concrete one and registers the definition (named using the identifier `id`) into the Coq environment. In Code 5.4, `MetaCoq Run` is used to run the computation and register `foo` in the Coq environment programmatically. This programmatic generation can be used to dynamically create inductive types and definitions.

5.1.2 Generating Inductive types using MetaCoq

As shown in the previous section, using MetaCoq Coq terms can be written in their reified syntax form and then be added to the environment. This approach can be used for inductive types as well. Their syntax can be parameterized and when the size of the NoC is available, the concrete terms can be instantiated.

This is done by creating a function that generates the reified syntax of an inductive type. To generate an inductive type, the `tmMkInductive` Template Monad operation is used. This operation takes an `mutual_inductive_body` as input. The header of the function generating the input for `tmMkInductive` is shown in Code 5.5a.

```

1 Definition generate_ind_syntax
2   (ind_name : string) (prefix : string) (no_cstr : nat)
3   : mutual_inductive_body :=

```

(a) Header for function to generate inductive types**Code 5.5:** Generating inductive types using MetaCoq

The function is parameterized with three arguments, this allows the same function to be used to create all inductive types required for a Kôika program. The parameters are as follows:

1. `ind_name` : Identifier of the inductive type
2. `prefix` : Prefix for the identifiers of the constructors
3. `no_constr` : Number of constructors in the inductive type

```

4  { |
5  ind_finite := Finite;
6  ind_npars := 0;
7  ind_params := [];
8  ind_bodies :=
9    [{ |
10     ind_name := ind_name;
11     ind_indices := [];
12     ind_sort := sType (Universe.make' Level.lzero);
13     ind_type := tSort (sType (Universe.make' Level.lzero));
14     ind_kelim := IntoAny;
15     ind_ctors := generate_constructors prefix no_cstr;
16     ind_projs := [];
17     ind_relevance := Relevant
18   } ] ];
19 ind_universes := Monomorphic_ctx;
20 ind_variance := None
21 } }.

```

(b) `mutual_inductive_body`**Code 5.5:** Generating inductive types using MetaCoq

In Code 5.5b, the record `mutual_inductive_body` is shown. In Coq, records are enclosed using `{|...|}` notation. This record stores information pertaining to the inductive types (Line 4-21). The values for the fields in the record were obtained by quoting a concrete inductive type. Most of these fields are left unchanged, only the name (`ind_name`) and constructors (`ind_ctors`) on Line 9 and 14 have been parameterized to make the function polymorphic.

```

1  Fixpoint generate_constructors (prefix: string) (no_constr : nat)
2  : list constructor_body :=
3    match no_constr with
4    | 0 => []
5    | S n' =>
6      let name := prefix ++ string_of_nat (S n') in
7      (* eg prefix = reg, to create constructors reg1, reg2, ..etc *)
8      let cstr := { | cstr_name := name;
9                    cstr_args := [];
10                   cstr_indices := [];
11                   cstr_type := tRel 0;
12                   cstr_arity := 0 | } in
13      cstr :: generate_constructors prefix n'
14  end.

```

(c) Generating constructors for inductive types

Code 5.5: Generating inductive types using MetaCoq

Another function `generate_constructors` (shown in Code 5.5c) is used to generate the constructors of the inductive type. This function is recursive over the number of constructors (`no_constr`). If the number of constructors is 0 an empty list is returned (Line 4). In the recursive case, a name is generated for the constructor by appending the given `prefix` with the current constructor number (`S n'`). The function then generates the current constructor (`cstr`) and prepends it, using the `::` notation, to the list being generated by recursively calling `generate_constructors` with `n'` (one less constructor to generate). This continues until `no_constr` reaches 0, at which point the recursion stops, and the complete list of constructors is returned.

Once the size of the NoC has been defined, the number of registers, rules, and external functions can be calculated. Then the inductive types can be registered into the Coq environment using `MetaCoq Run` as shown in Code 5.5d. The function `generate_ind_syntax` is used to provide the input to `tmMkInductive``.

```

1 MetaCoq Run (
2   tmMkInductive' (generate_ind_syntax "reg_t" regprefix regno) ;;
3   tmMkInductive' (generate_ind_syntax "rule_name_t" ruleprefix nocsizesize);;
4   tmMkInductive' (generate_ind_syntax "ext_fn_t" extfnprefix interfacesize)
5 ).

```

(d) Registering the inductive types

Code 5.5: Generating inductive types using MetaCoq

5.1.3 Generating Definitions using MetaCoq

MetaCoq can also be used to generate function definitions. This subsection demonstrates an approach to write the syntax for the function to map rules to actions and the schedule. As seen in Code 5.2, the reified syntax of Coq terms can often be quite verbose. Therefore, from this point onward, the syntax of functions will be presented in a more compact form, displaying only the essential terms required to construct the AST of the function.

In Code 5.6 a parameterized concrete version of the rules to action mapping function is shown. It is provided to have a reference of the structure which is being built using AST.

```

1 Definition to_action := fun rl : rule_name_t =>
2   match rl with
3   | router_1 => routestartfn 0 r1 r4 extfn_1 extfn_2
4   | router_2 => routecenterfn 1 r1 r2 r5 extfn_3 extfn_4
5   | ...
6   | router_n => routeendfn (n-1) r(n-1) r(2n-1) extfn_(2n-1) extfn_(2n)
7   end.

```

Code 5.6: Desired function for mapping rules to actions

It can be seen that the function consists of a `match` statement, which is responsible for mapping the rules to their respective actions. The representation of the reified syntax for the function is shown in Code 5.7.

```

1  tLambda "rl"                                (* Function binder: "rl" for rule *)
2    (tInd "rule_name_t")                      (* Input type: rule_name_t *)
3    (tCase                                    (* Case analysis on "rule_name_t" *)
4      "rule_name_t"                          (* case_info: rule_name_t *)
5      (tApp "uaction"                        (* predicate term: uaction with type arguments *)
6        ["reg_t"; "ext_fn_t"])
7      (tRel 0)                               (* term: Match on rule (relational variable 0) *)
8      [                                       (* list (branch term) *)
9        tBranch [] (tApp "routestartfn" [...]),
10       tBranch [] (tApp "routecenterfn" [...]),
11       tBranch [] (tApp "routeendfn" [...])
12     ]
13  )

```

Code 5.7: Compact representation of the AST for rule to action mapping

This compact version can be summarized as follows:

- The mapping from rules to action can be represented as a Lambda (λ) function represented by `tLambda` in MetaCoq (Line 1) with terms of type `rule_name_t` as its input (Line 2).
- The match statement is represented using `tCase`, which has the following syntax:
`tCase: case_info -> predicate term -> term -> list (branch term) -> term.`
Each of these terms is shown in the comments in Code 5.7.
- `case_info` points to the inductive type on which the match is being performed, in this case, `rule_name_t` (Line 3-4).
- `predicate term` stores information regarding the type that is returned by the match, here it is `uaction reg_t ext_fn_t` (Line 5-6).
- `tRel 0` refers to the innermost variable which is the term on which the match will be performed (Line 7).
- The syntax for all branches of the match is presented as a list, with each branch defined using `tBranch` (Line 8-12).

Similar to Code 5.6, a N-rule schedule has been shown as a reference in Code 5.8.

```

1  Definition schedule : scheduler :=
2    router_1 |> router_2 |> ... |> router_n |> done.

```

Code 5.8: N-rule schedule

As stated in Subsection 4.2.1, the schedule consists of the rules in the design strung together using the `Cons (|>)` operator, with its end marked by `done`. Both `done` and `Cons` are constructors of inductive type `scheduler`. Since the schedule contains all the rules in the design, its syntax can be generated by performing a recursion on all constructors of `rule_name_t`. Therefore as shown in Code 5.9, the syntax is defined using a `Fixpoint` recursion.

```

1  Fixpoint generate_scheduler (n: nat) : term :=
2    match n with
3    | 0 => tApp (tConstruct "scheduler" 0 [])
4            [tConst "pos_t" []; tInd "rule_name_t" []]
5    | S n' => tApp (tConstruct "scheduler" 1 [])
6                [tConst "pos_t" [];
7                  tInd "rule_name_t" [];
8                  tConstruct "rule_name_t" n' [];
9                  generate_scheduler n']
10   end.

```

Code 5.9: Syntax for schedule

In the base case of the recursion, the syntax for a schedule with zero rules is handled. In this case, the schedule will only include `done`. `tApp` (Line 3) is used here to apply the constructor. The syntax of `tApp` is defined as follows - `tApp : term -> list term -> term` i.e. `tApp` takes a `term` that points to the constructor (Line 4) and a list of arguments (Line 5) to which the said constructor is applied. Since `done` is the first constructor of the inductive type `scheduler`, it is indexed by 0 on Line 4. The argument list for constructors of type `scheduler` always contains constant `post_t` and inductive type `rule_name_t`.

In the recursive case (Line 5-9), a rule is appended to the schedule. As `Cons` is the second constructor, the index changes from 0 (Line 3) to 1 (Line 5). The `Cons` constructor concatenates a rule and a schedule, hence two additional terms can be seen in the arguments list. The term on Line 8 represents the rule indexed at `n'` in `rule_name_t` and `generate_scheduler n'` on Line 9 is the recursive call which is creating a schedule to which the rule is prepended.

As shown previously, definitions are registered using `tmMkDefinition`, another Template Monad operation that takes the identifier and syntax of the functions as its arguments. Similar to Code 5.5d, in Code 5.10 the syntax generated using the functions is registered into the Coq environment using `MetaCoq Run` and the required Template Monad operation.

```

11 MetaCoq Run ( tmMkDefinition "to_action"%bs match_syn).
12 MetaCoq Run ( tmMkDefinition "schedule"%bs scheduler_synatx).

```

Code 5.10: Registering schedule and rules-to-action mapping into the Coq Environment

5.1.4 Proving Properties of Generated Code

One of the goals of this thesis is to implement a methodology to prove properties generalized over all sizes of NoC possible. To do so, the generation of the NoC should be wrapped in a `Definition` or `Module`. Since `MetaCoq Run` commands are vernacular commands in Coq, they cannot be wrapped inside a `Definition`. Thus, the only option here is to wrap the generation code in a `Module`. As an example, a `Module` that generates rules using a parameter is shown in Code 5.11.

```

1 Module Type NOC_data.
2   Parameter nocsize: nat.
3 End NOC_data.
4
5 Module NOC (ND:NOC_data).
6   MetaCoq Run (
7     tmMkInductive' (quoteind "rule_name_t" ruleprefix ND.nocsize)
8   ).

```

Code 5.11: Wrapping rule generation in `Module`

Unfortunately, the following error is observed when trying to step through the code:

Error: Not Supported raised at unquote_list

Since MetaCoq operates within the same foundational rules and constraints as Coq, it does not inherently provide a way to circumvent Coq's requirement for inductive types to have a fixed number of constructors. Therefore, our NoC generation code cannot be wrapped into a `Module` either. In this subsection, a solution that allows to prove properties about generated code is presented.

```

1 Lemma XYZ : ∀ n,
2   {t & Σ ;;; [ ] |- t : tApp <% P %> (snd (generate_ind_syntax n)::nil) }.

```

Code 5.12: Lemma to reason about generated code

In Code 5.12, a lemma that can be used to reason about generated code is shown. Here, `{t & ...}` is used to describe a dependent pair type. It states that a term `t` exists, such that it satisfies the proposition or property stated in the second part of the term after `&`. In this case, the lemma asserts the existence of a term `t` for each `n` that satisfies a particular typing judgement.

The typing judgement is represented by the $\Sigma ;;; \Gamma \vdash t : T$ notation. Its components are as follows:

- Σ - Represents the global environment. It contains definitions, inductive types, constants, and other global declarations.
- Γ - Represents the local context (or typing context). It is a list of variable declarations or assumptions available when typing the term `t`. In this case, it is an empty list.
- \vdash - Represents the typing judgment. It asserts that the term on the left (`t`) is well-typed and conforms to the type on the right (`T`), given the global and local contexts.
- $t : T$ - This states that the term `t` has type `T` within the given contexts.

```

1 Definition P : Prop := forall n : nat, plus 0 n = n.
2 Definition ind_decls : global_decl := InductiveDecl nat_inductive.
3 Definition global_declaraton_nat : kername × global_decl :=
4   (kername_of_string "gd_nat", ind_decls).
5 Definition Sigma_env : global_env :=
6   add_global_decl (fst P_quoted) global_declaraton_nat.
7 Definition universes : universes_decl := Monomorphic_ctx.
8 Definition Σ : global_env_ext := (Sigma_env, universes).
9 MetaCoq Quote Recursively Definition nat_quoted := nat.

```

Code 5.13: Creating Σ

This approach to proving properties of generated code is demonstrated by a simple example. Here, a lemma is built to prove the left identity of the addition of natural numbers i.e. $\forall n, 0 + n = n$. In Code 5.13, the left identity of addition is defined (Line 1). Then the global environment Σ (Line 2-6) is created; this must contain all the inductive types present in the property. To prove this property, the inductive type `nat` is sufficient (Line 2). The AST of `nat` is defined in `nat_inductive`. Some additional definitions need to be created, to obtain the Σ required for the lemma shown, these are created in Line 2-8. The `MetaCoq Quote Recursively` command on Line 9 is used to quote the entire environment, i.e. it quotes all declarations that will be needed to type check the term.

The new lemma to prove the left identity of addition using meta-programming is shown in Code 5.14. The property `P` is quoted (converted to AST) using the following notation `<% %>`. The term `snd` is a projection function in Coq which extracts the second element of a pair. This function is used to extract the term from the entire environment, which was obtained by using `MetaCoq Quote Recursively`.

```

1 Lemma XYZ : ∀ n,
2   {t & Σ ;;; [ ] |- t : tApp <% P %> (snd nat_quoted :: nil) }.

```

Code 5.14: Lemma to prove left identity using typing judgement

The proof term generated from this lemma is shown in Code 5.15. Although this proof is provable, it is tedious. To satisfy the typing judgement the proof term has to be manually built. This requires a significant effort even for a simple property like the left identity.

```

1 t0 : term,
2   Σ ;;; [ ] |- t0
3   : tApp (tConst (MPdot (MPfile ["metacoq_proof"]) "Test", "P") [ ])
4     [snd nat_quoted]

```

Code 5.15: Proof term

In contrast to this, Code 5.16, demonstrates how this property can be easily proven with the tactics available in Coq.

```

1 Lemma left_identity_add : forall n : nat, 0 + n = n.
2 Proof.
3   intros n.
4   reflexivity.
5 Qed.

```

Code 5.16: Proving left identity using tactics in Coq

Tactics allow for step-wise proof construction and automation while maintaining readability. This ease of proving is lost when the proof terms have to be built manually to prove properties about generated code. This is a major drawback of implementing a generic NoC library using MetaCoq.

5.2 Dependent Types

As shown in Section 5.1, even though MetaCoq can be used to generate NoCs of variable sizes, proving properties for all sizes of NoCs is a major undertaking and requires the construction of a proof term without tactics. Another solution to scale from a fixed-size NoC to the generic NoC is to use dependent programming. Dependent programming is a paradigm of programming where the types of functions or data can depend on specific values.

The concept of dependent programming is tightly connected with dependent types, which are types that can be parameterized by values. In traditional type system, types are fixed and independent of values, e.g. in Coq a list of natural numbers has type - `list nat`. In dependent programming, types can depend on their value, e.g. a vector, which is a list of fixed length has the type - `vector nat n`. Here, the type is dependent on the value `n` which is the length of the list. A type like this can be used to verify the absence of out-of-bounds accesses. Therefore, dependent types can be used to ensure the correctness of programs.

Another example of dependent types is the `Fin.t` type. It represents finite sets of a given size. It is commonly used to model indices into lists or arrays of a fixed length. Similar to a vector, its type depends on a natural number, which specifies the size of the finite set. For example, `Fin.t n` represents a type of finite values constrained to the range $[0, n-1]$. This can also be seen in the definition of the type Code 5.17. In the definition, `nat -> Type` signifies that `Fin.t` is a dependent type that produces a separate type for each value of natural number `n`.

```

1 Inductive Fin.t : nat -> Type :=
2   | F1 : forall {n}, Fin.t (S n)          (* Zero-based first index *)
3   | FS : forall {n}, Fin.t n -> Fin.t (S n). (* Successor index *)

```

Code 5.17: Inductive type `Fin.t`

The constructors build values of type `Fin.t n`, where:

- `F1` represents the first element (index 0) of a set of size `S n` (successor of `n`). Having the successor in the definition ensures that the size of the set is always greater than 0.
- `FS` (successor) constructs the next index by incrementing a given `Fin.t n`.

If `Fin.t n` is used, it is guaranteed to Coq that any index into a data structure of size `n` is within bounds during compile time. Its use allows us to define types and functions where the size is a parameter, while still ensuring type-safety and correctness. In addition to this, it also facilitates structural induction and dependent pattern matching.

5.3 Comparison

Aspect	Meta-programming	Dependent Programming
Design	✓	●
Proving Properties	✗	✓

Table 5.1: Comparison of Metaprogramming and Dependent Programming

In this subsection, a brief comparison is presented between the two techniques discussed in the previous sections: meta-programming and dependent programming.

In the meta-programming approach, all the aspects of the NoC design that vary with the size are converted into parametric ASTs. Once the size of the NoC is fixed the ASTs could be converted into Coq terms. Generally, the method to obtain the AST comprises of the following steps: write a static implementation, quote it to obtain its static AST, find the repeating patterns in the AST, and parametrize these patterns using functions and recursion. Thus, writing Kôika programs using ASTs can be a tedious task. Incorporating small changes in the design might need a lot more time and effort since the modifications need to be made in elaborate ASTs. However, the code generated using MetaCoq conforms with the requirements of Kôika, i.e. it produces inductive types with simple constructors. Consequently, these programs can be automatically type checked and compiled to Verilog with the existing functions in Kôika.

In contrast to that, in the dependent programming approach, the inductive types are defined as dependent types. Due to this, the functions utilizing these data types i.e. the rules to action mapping function and the schedule need to be defined with dependent pattern matching. In addition to this, Kôika in its current state cannot ensure whether the newly defined dependent types are finite or not. As a consequence, it must be proven manually that the dependent types being used are indeed finite. Even though writing Kôika programs using dependent types provides an extra level of abstraction, it will also require more manual effort to get Kôika to accept the program.

Still, as demonstrated in Subsection 5.1.4, there are challenges to verifying properties of generated code. The lemma needs to be formed by creating typing judgements and global environments, and the proof terms need to be created manually. This forfeits the convenience of using tactics present in Coq. On the other hand, if the design is implemented in dependent programming, it is plausible to wrap the design in a Coq `Module`. This would allow us to directly write a lemma over all possible dimensions of NoC. Also, this will enable one to approach the proofs with the aid of the available tactics in Coq.

Building on this comparison, the next chapter demonstrates the development of the generic NoC library using dependent programming.

6 Library for Generic NoCs

In this Chapter, the design of the generic NoC using dependent programming is presented. First, the inductive types used in Kôika program are defined as dependent types, and then the functions that operate on these new types are created. After completing the design, the challenges encountered due to this approach are discussed. Lastly, a brief insight is given on a methodology to verify properties across all possible sizes of NoC.

6.1 Pipeline NoC Design using Dependent Programming

The design of the NoC using dependent programming is divided into two parts: creating dependently typed inductive types, and then defining the functions operating on said dependent types.

6.1.1 Defining Inductive Types using Dependent Programming

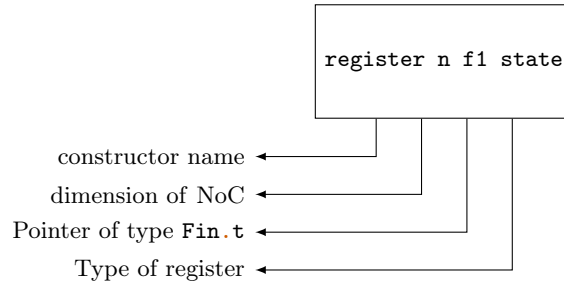
In Section 5.2, the `Fin.t` type is introduced as an example of dependent types. It was mentioned, that it can be used to define types where size is a parameter while ensuring type-safety and correctness. Code 6.1 shows an implementation of the `rule_name_t` inductive type using `Fin.t`. As seen in Section 5.2, for each natural number `n`, there is a corresponding type `rule_name_t n`, implied by the return type `nat -> Type`.

```
1 Inductive rule_name_t : nat -> Type :=  
2   | rule : forall x_dim, Fin.t (S x_dim) -> rule_name_t (S x_dim).
```

Code 6.1: `rule_name_t` using dependent types

The constructor of `rule_name_t` can be broken down as follows:

- Uses `x_dim` (a natural number) to define the type, which also serves as the size of the NoC.
- Accepts an element of type `Fin.t (S x_dim)` (a finite type with `S x_dim` elements, where `S x_dim` is the successor of `x_dim`).

Figure 6.1: Accessing registers in dependent `reg_t`

- Produces an instance of type `rule_name_t (S x_dim)`.

In the current design, every router maps to a dedicated rule/action, but `reg_t` and `ext_fn_t` require constructors almost double that of the routers (the size of the NoC). Rather than simply defining these inductive types with a parameter 2-times that of `x_dim`, let's push the boundaries of dependent types and add another additional parameter (value):

```

1 Inductive ext_com_t : Type :=
2   | input
3   | output.
4
5 Inductive ext_fn_t : nat -> Type :=
6   | ext_fun : forall x_dim, Fin.t (S x_dim) -> ext_com_t -> ext_fn_t (S x_dim).
```

Code 6.2: `ext_fn_t` using dependent types

In Code 6.2, another inductive type `ext_com_t` is defined on Line 1-3. It has two constructors `input` and `output`. Using this new type, another dependency can be added, further dividing the external functions into two types, while still using the same dimension `x_dim` (Line 6).

```

1 Inductive reg_t_type : Type :=
2   | state
3   | downstream.
4
5 Inductive reg_t : nat -> Type :=
6   | register : forall x_dim, Fin.t (S x_dim) -> reg_t_type -> reg_t (S x_dim).
```

Code 6.3: `reg_t` using dependent types

A similar approach is adopted for the registers in Code 6.3. Here, registers are divided into two - `state` which will be used to store the outputs from router, and `downstream` which serve as channel registers. These two types, demonstrate the amount of flexibility possible by using dependent types. In Fig. 6.1, the dissection of the term `register n f1 state` is shown. Terms like these will be used to access elements of the inductive type in the functions.

6.1.2 Defining Functions using Dependent Programming

Coq's standard library provides limited support for dependent pattern matching, making it less convenient to work with dependent types. To remedy this, the Equations [Soz10] plugin is used. Equations is a plugin for Coq which provides additional syntax to define programs by dependent pattern matching and allows for more complex recursion schemes. Equations is used to define the design's rules to actions mapping function (Code 6.4), and its schedule (Code 6.5).

```

1 Equations to_action {x_dim':nat} (rl : rule_name_t (S x_dim'))
2 {x_dim_max : nat} (H: S x_dim' <=& S x_dim_max)
3   : action (tau:= unit_t) (R (x_dim:=x_dim_max)) (Sigma (x_dim:=x_dim_max)) :=

```

(a) Function header

Code 6.4: Mapping rules to actions using Equations

In Code 6.4a the header for a function to map the rules to their respective actions is shown. The functions are now defined using the `Equations` vernacular command instead of `Definition`. Compared to previous implementations this function needs a few additional arguments. The arguments in curly braces `{x_dim'}` and `{x_dim_max : nat}` are implicit arguments and can be inferred by Coq. Even the definition of `Fin.t` uses these to implicitly resolve `n`. However, due to the complexity of this program, Coq fails to resolve these values. In situations like these, the argument can be explicitly provided using `@`. This usage can be seen in Code 6.4b on Line 11. Also on the same line, `?0` is used, this annotation is used to mark the pattern as inaccessible. An inaccessible pattern explicitly tells Coq that this value is already known and does not need to be matched again. Marking certain patterns as inaccessible helps Coq understand that the function or proof is *complete*. Functions in Coq need to be complete, in the sense that they handle all possible cases of an inductive type. This applies to dependent types as well, and using Equations ensures this.

Compared to the previous functions, the dependently typed mapping needs another argument - `H`, a proof term to ensure that the index of the rule being matched (`S x_dim'`) is less than or equal to the size of the NoC (`S x_dim_max`). The polymorphic routers (actions) defined in Section 4.3 can be used here as well by modifying their types to the new dependent form. However, currently, Kôika's automated type checking cannot type check dependent types. This was resolved by rewriting the routers (actions) using Typed Parsing. Typed actions have the type: `action tau R Sigma`, compared to their untyped counterparts which were of type `uaction reg_t ext_fn_t`. Here, `R` and `Sigma` are the initialized registers and external functions, the procedure to define them was shown in Code 4.2 and 4.6 respectively.

Since the function needs to be complete all possible cases need to be handled. The cases handled in the `to_action` function are as follows:

Case 1: Mapping Single Router in 1-stage Pipeline

```

4      to_action (rule 0 (@F1 ?(0))) le_n :=
5          routestartfn
6          0
7          (register 0 F1 state)
8          (register 0 F1 downstream)
9          (ext_fun 0 F1 input)
10         (ext_fun 0 F1 output);

```

(b) Only single router

Code 6.4: Mapping rules to actions using Equations

This case handles the simplest scenario, where $x_dim_max = 0$, i.e. there is only one router in the pipeline. F1 is the first and only element in `Fin.t 1`. The proof `le_n` is used to prove the hypothesis $1 \leq 1$.

Case 2: Mapping First Router in a Larger Pipeline

```

11     to_action (rule 0 (@F1 ?(0))) (x_dim_max:=n) H :=
12         let f1 := widen_fin H (@F1 0) in
13         routestartfn 0
14         (register n f1 state)
15         (register n f1 downstream)
16         (ext_fun n f1 input)
17         (ext_fun n f1 output);

```

(c) First router

Code 6.4: Mapping rules to actions using Equations

This case handles the start router in a pipeline generalized to a maximum size $x_dim_max = n$. For this case the finite type F1 is of type `Fin.t (S x_dim')`, yet it must be embedded into a larger set of type `Fin.t (S x_dim_max)`. Since x_dim' is smaller than x_dim_max , an additional helper function `widen_fin` is used to embed the type into a larger set (Line 12).

Case 3: Mapping Last Router in a Larger Pipeline

```

18     to_action (rule (S c) (@F1 ?(S c))) (x_dim_max:=n) le_n :=
19         let f1 := @F1 (S c) in
20         let f1' := FS (@F1 c) in (* upstream *)
21         routeendfn (S c)
22         (register (S c) f1 state)
23         (register (S c) f1' downstream) (* upstream *)
24         (ext_fun (S c) f1 input)
25         (ext_fun (S c) f1 output);

```

(d) Last router

Code 6.4: Mapping rules to actions using Equations

This case handles the last router in a pipeline with a generalized size. It is similar to the previous case, but here the upstream register is accessed instead of the downstream by using FS (@F1 c). The downstream register of the last router will never be utilized in this design.

Case 4: Mapping Middle Routers in a Larger Pipeline

```

26 to_action (rule (S c) (@F1 ?(S c))) (x_dim_max:=n) (le_S n H) :=
27   let f1 := widen_fin (le_S _ _ H) (@F1 (S c)) in
28   let f1' := FS (widen_fin H (FS (@F1 c))) in (* upstream *)
29   routecenterfn n
30     (register n f1 state)
31     (register n f1' downstream) (* upstream *)
32     (register n f1 downstream)
33     (ext_fun n f1 input)
34     (ext_fun n f1 output);

```

(e) Middle routers

Code 6.4: Mapping rules to actions using Equations

The middle routers serve as the base case of the recursion. Here both a downstream and an upstream register need to be accessed. Since the `Fin.t` will be of type `Fin.t (S x_dim')`, where `x_dim'` is the index currently being iterated on, both the upstream and downstream have to be widened.

Case 5: Additional Cases

```

35 (* case: the fin space of the rule is smaller than the desired one*)
36 to_action (rule (S c) (FS c')) (x_dim_max:=n) (@le_S ?(S c) n H) :=
37   to_action (rule c c') (le_S _ n (le_t_inj H));
38
39 (* case: the types of the fin space match up *)
40 to_action (rule (S c) (FS c')) (x_dim_max:=?(S c)) H :=
41   to_action (rule c c') (le_t_inj H).

```

(f) Additional cases

Code 6.4: Mapping rules to actions using Equations

The recursion propagates through cases where the rule's dimension is smaller than the pipeline dimension, which is not possible. These cases also need to be handled for the completeness of the function. They are handled using proof terms like `le_t_inj`.

The `schedule` function shown in Code 6.5a has a similar header. It can be observed, the hypothesis (H) is also needed for the dependently typed.

```

1 Equations schedule {x_dim'} {x_dim_max : nat} (H: S x_dim' <= S x_dim_max)
2   : Syntax.scheduler pos_t (rule_name_t (S x_dim_max)) :=

```

(a) schedule header

Code 6.5: schedule using Equations

The implementation of the `schedule` in Code 6.5 deals with cases identical to those seen in the rules to actions mapping function.

Case 1: Scheduling Single Rule in 1-stage Pipeline

```

3  (* Case 1: rule 0 for max_dim = 1 *)
4  schedule (x_dim' := 0) (x_dim_max := 0) (@le_n ?(S 0)) :=
5  rule 0 (@F1 0) |> done;

```

(b) Only single rule

Code 6.5: `schedule` using Equations

The first case handles the single router pipeline and creates a schedule with one rule as shown on Line 5.

Case 2: Scheduling Rule Mapped to First Router in a Larger Pipeline

```

6  (* Case 2: rule 0 for max_dim = n *)
7  schedule (x_dim' := 0) (x_dim_max := (S m)) (@le_S (S (S x)) ?(S m) h) :=
8  rule (S m) (FS (widen_fin h (@F1 0))) |> done;

```

(c) Rule mapped to first router

Code 6.5: `schedule` using Equations

The second case handles the rule for the first router in a large pipeline. In this schedule, the first router is placed at the end of the schedule, just before `done`.

Case 3: Scheduling Rule Mapped to Last Router in a Larger Pipeline

```

9  (* Case 3: rule n for max_dim = n *)
10 schedule (x_dim' := (S n)) (x_dim_max := (S n)) (@le_n ?(S n)) with
11   (schedule (x_dim' := n) (x_dim_max := (S n)) (le_S (S n) (S n) (@le_n (S n)))) => {
12   schedule (x_dim' := (S n)) (x_dim_max := (S n)) (@le_n ?(S n)) t1 :=
13   rule (S n) (@F1 (S n)) |> t1
14   };

```

(d) Rule mapped to last router

Code 6.5: `schedule` using Equations

In the third case, the rule mapped to the last router is placed as the first rule in the schedule. Here, the function constructs the rule for this router and recursively schedules the lower dimensions. This is done by using the `with ... =>:` clause which allows pattern matching to destructure recursive calls and extract an intermediate schedule, which is represented by `t1` here.

Case 4: Scheduling Rules Mapped to Middle Routers in a Larger Pipeline

```

15  (* Case 4: rule n for recursive case *)
16  schedule (x_dim' := (S n)) (x_dim_max := (S m)) (@le_S ?(S m) h) with
17    (schedule (x_dim' := n) (x_dim_max := (S m)) (le_S (S n) (S m) (le_t_inj h))) => {
18    schedule (x_dim' := (S n)) (x_dim_max := (S m)) (@le_S ?(S m) h) t1 :=
19    rule (S m) (FS (widen_fin h (@F1 (S n)))) |> t1
20  };

```

(e) Rule mapped to middle router

Code 6.5: schedule using Equations

An identical approach as the last case is used for the rules mapped to the middle routers.

Case 5: Additional Cases

```

21  (* Case 5: absurd *)
22  schedule (x_dim' := (S n)) (x_dim_max := 0) (le_S x h) := absurd_le_t h.

```

(f) Additional cases

Code 6.5: schedule using Equations

The last case handles an invalid state where `x_dim'` exceeds `x_dim_max`. The `absurd_le_t` tactic is used to prove that the provided hypothesis `h` is invalid.

6.2 Challenges using Dependent Types in Kôika

After completing the design, the next step was to write a lemma to prove the properties of the new dependently typed NoC. However, the following error was observed:

```

Error: Could not find an instance for "FiniteType (reg_t (S nocsiz))"
in environment

```

As discussed in the Sections 4.4 and 5.3, Kôika requires the inductive types used in the program to be finite. Kôika defines a typeclass called `FiniteType`, to formalize the idea of finite types and to make it easier to work with such types generically. In Kôika, if a type `T` is an instance of this typeclass then it is implied that the type is a finite, complete list of elements. The components needed to ensure that a type `T` is an instance of this class can be checked by running `Print FiniteType`. The result of this is shown in Code 6.6.

```

1  Record FiniteType (T : Type) : Type := Build_FiniteType
2  { finite_index : T -> nat;
3    finite_elements : list T;
4    finite_surjective : forall a : T,
5      nth_error finite_elements (finite_index a) = Some a;
6    finite_injective : NoDup (map finite_index finite_elements) }.

```

Code 6.6: FiniteType typeclass

The terms inside the curly braces are known as the *methods* of the typeclass. As stated previously, inductive types that are part of the design (i.e. `reg_t`, `rule_name_t`, and `ext_fn_t`) must be an instance of this typeclass. For the simple inductive definitions seen in the previous sections, instances have already been defined in Kôika, and thus are automatically resolved as an instance of the typeclass. However, due to the intricate structure of the dependent types created in subsection 6.1.1, these could not be automatically resolved as instances of this typeclass. To solve this an instance for each inductive type had to be manually constructed with all the required methods. The purpose and implementation of each method for the dependently typed `reg_t` from Code 6.3 is described as follows:

1. `finite_index : T -> nat`:

This function maps each element of the finite type `T` to an index of type `nat`. The implementation of this method is shown in Code 6.7.

```

1 Equations regt_idx {n} (r: (reg_t (S n))) : nat :=
2   regt_idx (register 0 (FS f) r)      := absurd_fin f;
3   regt_idx (register m F1 state)      := 0;
4   regt_idx (register m F1 downstream) := 1;
5   regt_idx (register (S m) (FS f) r)  := S (S (regt_idx (register m f r))).

```

Code 6.7: `regt_idx`

Here, Equations prove to be essential for dependent pattern matching. First `absurd_fin` covers an impossible case where an element is accessed in an empty list (Line 2). In the base case, the first `state` register is assigned index 0 (Line 3) and the first `downstream` register is assigned index 1 (Line 4). For a register with an incremented index (`FS f`), it assigns an index that is `S (S (regt_idx ...))`, effectively incrementing by 2 for each level of recursion (Line 5).

2. `finite_elements : list T`:

This function provides a `list` that serves as the complete enumeration of all possible elements of the finite type `T`. The implementation of this method for `reg_t` is shown in Code 6.8.

```

1 Equations regt_elems {m} : list (reg_t (S m)) :=
2   regt_elems (m:=0)      :=
3     cons (register 0 F1 state) (cons (register 0 F1 downstream) nil);
4   regt_elems (m:=(S m')) with regt_elems => {
5     | tl := cons (register (S m') (@F1 (S m')) state)
6               (cons (register (S m') (@F1 (S m')) downstream)
7                     (map lift_register tl))
8     }.

```

Code 6.8: `regt_elems`

For the base case `m:=0`, a list with two elements a state and a downstream register (Line 2-3) is generated. As the dependent type was defined with `S x_dim` in Code 6.3, a case with an empty list is not possible. In the recursive case, two registers are prepended at index `S m'` to the recursive call to the smaller `m'`. Destructuring using `with` is used to map (apply to every element in the list) function `lift_register` to the smaller set.

This function provides similar functionality to `widen_fin`, it lifts all registers from a smaller set to the current set.

3. `finite_surjective`:

```
forall a : T, nth_error finite_elements (finite_index a) = Some a:
```

This property guarantees that for every element `a` of type `T`, indexing into the `finite_elements` at the position given by `(finite_index a)` must return `a`. The proof of this property for `reg_t` uses induction on `n` (the size of the register set). The base case is handled by using reflexivity. In the inductive step, dependent elimination is applied to break down the type into its possible constructors. For each case, the inductive hypothesis is applied and a lookup of the index is performed to ensure that the correct result is obtained.

4. `finite_injective : NoDup (map finite_index finite_elements)`:

This property guarantees that every element of `finite_elements` has a unique index, and there are no duplicates in the indices. The proof for `reg_t` again uses induction on the size `n`. In both cases, it is shown that the indices form a unique sequence. The proof for the base case is trivial since it has only two elements. In the inductive step, rewrites are performed on the list using mapping functions `map_cons` and `map_map` to show that the result is equivalent to a sequence `seq(0, (S n) * 2)`. Once that is done, there is a lemma available to prove that a sequence has no duplicates.

The methods for `FiniteType` instance of `ext_fn_t` are identical to the ones shown for `reg_t`. Since `rule_name_t` is dependent only on `Fin.t` its methods are a simpler variation of the methods shown.

Another problem was observed with the `Show` functions of `Kôika`. These functions are used to convert the inductive types into strings so that they can be written to the compiled Verilog code. The cause is similar, the existing `Show` functions cannot support dependent types, thus new `Show` functions were also defined for all the dependent types.

6.3 Proving Properties of Generic NoC

As discussed in Section 5.3, the pivotal reason behind the switch to dependent programming was that properties parametric over all possible sizes of NoC required construction of the proof term in MetaCoq. Once all the challenges from Section 6.1.1 were addressed, the next step was to state the property. In Code 6.9, a simple property that checks whether all state registers have the value 0 after one run of the schedule, has been shown.

```
1 Lemma check_state_registers (n:nat) (H: (S n) <=< (S nocsize)) :
2   run_schedule_compact r_r2l to_action sigdenote schedule
3   (fun ctxt =>
4     let bits_r := ctxt.[register nocsize (Helpers.widen_fin H (@Fin.F1 n)) state] in
5     Bits.to_nat bits_r = 0).
6 Proof.
7 unfold nocsize, run_schedule_compact, create.
8 Admitted.
```

Code 6.9: Property verifying correct initialization of registers

The lemma is accepted by Coq. It is also possible to use Coq tactics like `unfold` with this lemma. However as stated in Section 4.4, the existing `run_schedule` is not capable enough to work with generic designs. Currently, additional functions and tactics like `run_schedule_compact` are being developed for Kôika which will enable us to proceed with this proof. At the time of writing, it is possible to state properties about the generic NoC, however, in its current state Kôika lacks the machinery to reason about such properties.

To summarize this chapter, the pivot to dependent typing does require some additional work like rewriting the actions using Typed Parsing and creating new instances of `FiniteType` and `Show` functions tailored to the types. Still writing and modifying programs using dependent programming is relatively easier when compared to creating designs in meta-programming. Additionally, dependent programming also demonstrates how abstractions from Coq’s rich type system can be incorporated into a Hardware Description Language (HDL). Last but not least, this technique also lays the foundation for proving properties for all configurations of a system using a generic construction.

7 Results

As mentioned previously, Kôika is capable of generating Verilog code. The generated code can be used for RTL simulation, to run on a Field Programmable Gate Arrays (FPGA), or to even run synthesis to obtain a netlist that can be evaluated in terms of hardware costs, namely the area and estimated power consumption. In this evaluation, the results of the synthesis of generated Verilog code from the two discussed NoC design methodologies - meta-programming and dependent programming, have been presented and compared.

The synthesis is done with Globalfoundries 22 nm FD-SOI process under typical conditions (0.8 V, 25 °C). Timing-clean designs are achieved with clock frequencies of up to 4 GHz. This is possible because the current NoC implementations are quite simple and only consist of registers and short paths with logic cells. However, to achieve a high clock frequency, buffers and inverter cells are added to meet the timing constraints. To prevent the design from being packed with buffers, the design was synthesized at 1 GHz. The synthesis was performed on NoCs of sizes 4, 16, 32, 64, and 96 were chosen.

Fig. 7.1 shows the number of cells needed per router for both approaches. In the dependent programming implementation, every router is connected to a downstream register, even the last register. Due to this design decision the dependently programmed NoC uses more sequential elements than its meta-programmed counterpart. Apart from this, the functionality for both approaches remains the same since they are using the same router actions. As explained in Section 4.3, the address field in the NoC pipeline is dependent on its size. So the bit-width of the registers increases with the number of routers. Thus the number of cells consumed is increasing with the number of routers.

Naturally, the area required by the designs follows a similar trend to the no. of cells as shown in Fig. 7.2. As per expectations, more routers consume more area. The area consumed is comparable for both programming techniques up until 64 routers, after which the area for the dependently typed NoC starts to increase sharply. Due to its simple design, the total area consumption is small compared to other state-of-the-art NoC implementations. For reference, a NoC router from [HA22] requires around $22,000 \mu\text{m}^2$ in the same 22 nm technology. However, that implementation has a 128-bit wide data field in the packet whereas the one shown in the thesis is only 4-bit wide.

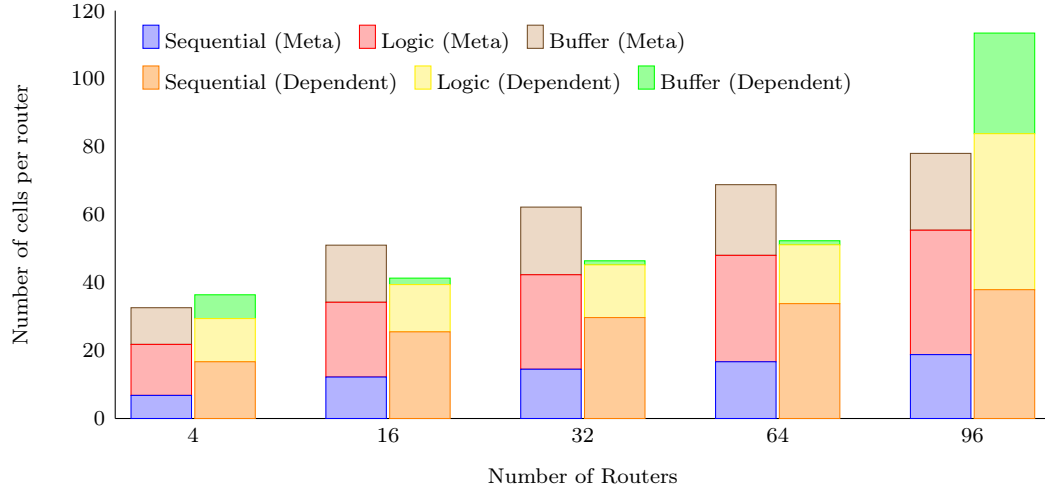


Figure 7.1: Number of cells consumed per router for NoCs designed by Meta-programming and Dependent Programming for different router configurations.

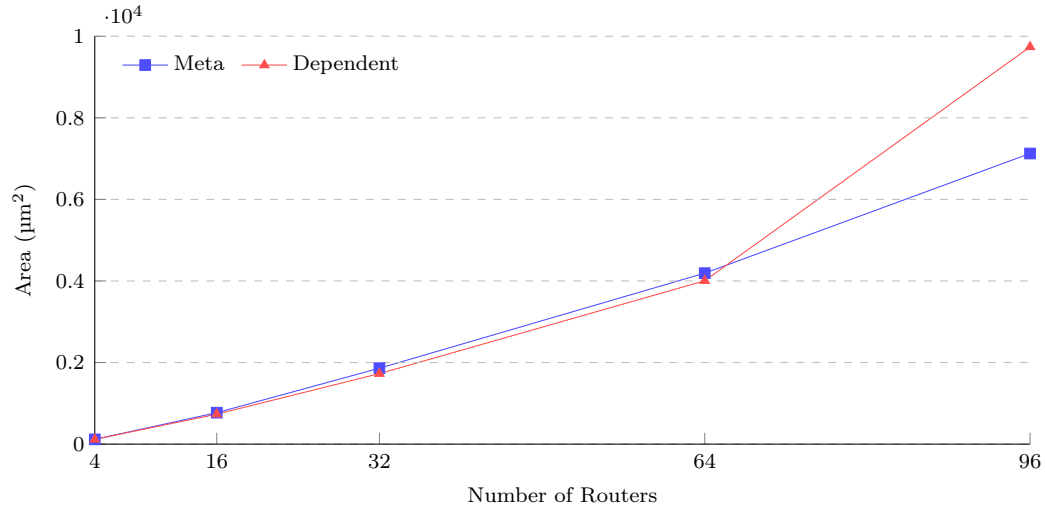


Figure 7.2: Area comparison between NoCs designed by Meta-programming and Dependent Programming for different router configurations.

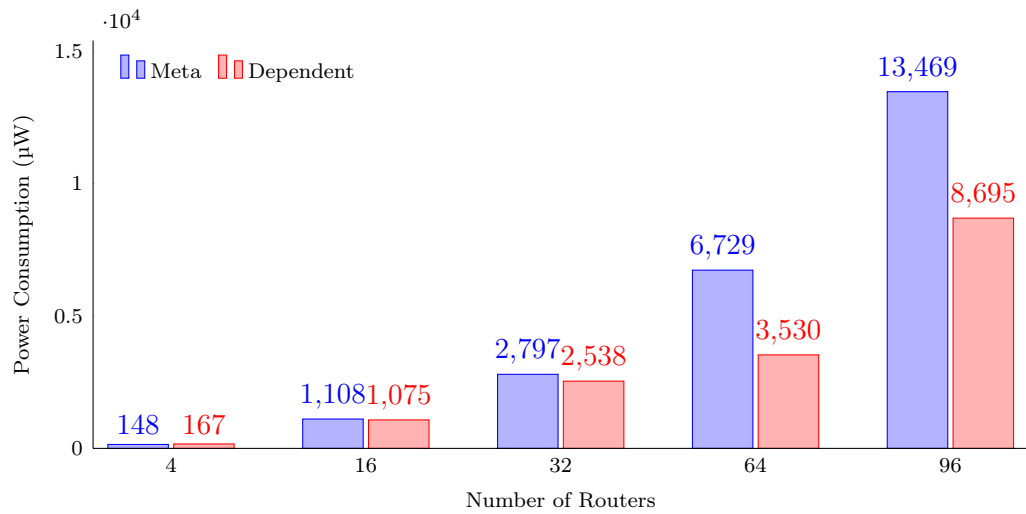


Figure 7.3: Power consumption for NoCs designed by Meta-programming and Dependent Programming for different router configurations.

In terms of power consumption (shown in Fig. 7.3), upto 32 routers, NoCs designed using both the programming techniques are comparable. However, as the number of routers is increased beyond that point, the power consumption of the NoC designed using meta-programming increased drastically. At the time of writing, the reason behind this is not known.

8 Conclusion and Further Work

This thesis presents a comprehensive approach to building a library to generate formally-verified NoCs of various sizes. Kôika, an EDSL in Coq is used for this implementation. This integration allows us to use tools available for formal reasoning and theorem proving in Coq. The research addresses one of the critical challenges in hardware design — verification of properties of NoCs generalized over every, configurable size possible.

The first part of this work focused on fixed-size NoC designs using Kôika. Here a step-by-step guide of how Kôika programs are written is provided. In addition to this, the methodology to simulate static circuits within Coq is shown. This provides a baseline framework for verified hardware using Kôika.

However, verifying fixed-size designs alone is insufficient; owing to the increase in complexity, modern designs require huge amounts of resources. The industry is more inclined towards configurable and verified IP to increase reuse and to reduce the effort required in verifying separate configurations. This was the motivation of the thesis. A significant challenge in designing a generic NoC with parameterizable size is that Kôika programs need to be compiled to Verilog and require the elements of the program to be finite. Coq also doesn't allow dynamic generation of inductive types either as such types could break termination guarantees and type safety. As solutions to this, two methodologies were explored in the thesis to overcome this challenge.

The first attempt was focused on meta-programming using MetaCoq, a project that provides a certified meta-programming environment in Coq. MetaCoq allows the representation of Coq terms as ASTs. These ASTs can be manipulated and parameterized to generate any possible term in Coq. Although, MetaCoq allows the generation of hardware with a user-defined parameter, proving properties generalized over the size of the NoC is challenging. The challenge comes from the fact that the reasoning requires proof terms to be built without using the tactics available in Coq.

The second attempt uses dependent programming, a paradigm that is already a part of Coq. This paradigm uses dependent types, which are types that depend on values of other types. Dependent types were used extensively to encode size constraints directly in the types, ensuring that the generated designs are well-formed and adhere to specified properties. The definition of the functions for this approach is done with Equations which provides better support for

well-founded recursion and dependent pattern matching compared to Coq. Using this approach, a property generalized over all possible sizes of the NoC can be proven using tactics provided in Coq.

Despite its successes, this work also highlights several areas for improvement and future research. The design presented in the thesis is contained within a single module due to certain limitations of Kôika. Once addressed, the NoC could be broken down into its individual components like buffers, arbiters, FIFOs, etc and the properties for each of these components can be proven and be used to prove more complex properties of the NoC. Once the required tools are available in Kôika properties like deadlock freedom, starvation freedom, timing side-channel resilience, etc. can also be proven. While this thesis primarily focused on a simple packet-switching pipeline NoC with XY routing algorithm, more switching schemes, topologies, and adaptive routing schemes could be explored in future research. The NoC could also be designed to have these options configurable by the user. Lastly, while the current framework targets NoC designs, the underlying methodology can be extended. For instance, the techniques developed in this thesis could be applied to the formal verification of other scalable hardware components, such as multipliers, memory controllers, or even entire SoCs.

In conclusion, this thesis contributes to the field of hardware verification by presenting a library to generate generic formally-verified NoC designs. The integration of Kôika and dependent programming in Coq enables the generation of generic NoCs and their corresponding proofs, ensuring correctness across a wide range of NoC sizes. By addressing both fixed-size and generic designs, this work lays a foundation for future research on scalable, formally verified hardware, ultimately advancing the reliability and correctness of modern hardware systems.

Bibliography

- [DW70] E.W. Dijkstra and Technische Hogeschool Eindhoven. Onderafdeling der Wiskunde. *Notes on Structured Programming*. T.H.-report. Technological University, Department of Mathematics, 1970. URL: <https://books.google.de/books?id=e6gMNQAACAAJ>.
- [GN92] C.J. Glass and L.M. Ni. “The Turn Model for Adaptive Routing”. In: *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. 1992, pp. 278–287. DOI: 10.1109/ISCA.1992.753324.
- [DT01] W.J. Dally and B. Towles. “Route packets, not wires: on-chip interconnection networks”. In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 2001, pp. 684–689.
- [Kum+02] Shashi Kumar et al. “A network on chip architecture and design methodology. VLSI, 2002”. In: *Proceedings. IEEE Computer Society Annual Symposium on*. 2002, pp. 105–112.
- [Riz02] L. Rizzatti. *High-Level Synthesis to Design in the New Millennium*. IEEE Tech Forum. Accessed: 06.01.2025. 2002. URL: <https://californiaconsultants.org/wp-content/uploads/2014/05/200205.pdf>.
- [RJE03] K. Richter, M. Jersak, and R. Ernst. “A formal approach to MpSoC performance verification”. In: *Computer* 36.4 (2003), pp. 60–67. DOI: 10.1109/MC.2003.1193230.
- [RMK03] A. Roychoudhury, T. Mitra, and S.R. Karri. “Using formal techniques to debug the AMBA system-on-chip bus protocol”. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. 2003, pp. 828–833. DOI: 10.1109/DATE.2003.1253709.
- [Amj04] Hasan Amjad. *Model checking the AMBA protocol in HOL*. Tech. rep. UCAM-CL-TR-602. University of Cambridge, Computer Laboratory, Sept. 2004. DOI: 10.48456/tr-602. URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-602.pdf>.

- [Nik04] R. Nikhil. “Bluespec System Verilog: efficient, correct RTL from high level specifications”. In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04*. 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818.
- [Geb+05] Biniam Gebremichael et al. “Deadlock Prevention in the Æthereal Protocol”. In: *Correct Hardware Design and Verification Methods*. Ed. by Dominique Borrione and Wolfgang Paul. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 345–348. ISBN: 978-3-540-32030-2.
- [SB05] Julien Schmaltz and Dominique Borrione. “A Generic Network on Chip Model”. In: *Theorem Proving in Higher Order Logics*. Ed. by Joe Hurd and Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 310–325. ISBN: 978-3-540-31820-0.
- [BM06] Tobias Bjerregaard and Shankar Mahadevan. “A survey of research and practices of Network-on-chip”. In: *ACM Comput. Surv.* 38.1 (June 2006), 1–es. ISSN: 0360-0300. DOI: 10.1145/1132952.1132953. URL: <https://doi.org/10.1145/1132952.1132953>.
- [BC06] L. Bononi and N. Concer. “Simulation and analysis of network on chip architectures: ring, spidergon and 2D mesh”. In: *Proceedings of the Design Automation & Test in Europe Conference*. Vol. 2. 2006, 6 pp.-. DOI: 10.1109/DATE.2006.243841.
- [06] “IEEE Standard for Verilog Hardware Description Language”. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.
- [SB06] Julien Schmaltz and Dominique Borrione. “Towards a formal theory of on chip communications in the ACL2 logic”. In: *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*. ACL2 '06. Seattle, Washington, USA: Association for Computing Machinery, 2006, pp. 47–56. ISBN: 0978849302. DOI: 10.1145/1217975.1217985. URL: <https://doi.org/10.1145/1217975.1217985>.
- [Bor+07] Dominique Borrione et al. “A Generic Model for Formally Verifying NoC Communication Architectures: A Case Study”. In: *First International Symposium on Networks-on-Chip (NOCS'07)*. 2007, pp. 127–136. DOI: 10.1109/NOCS.2007.1.
- [Lee+07] Hyung Gyu Lee et al. “On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches”. English (US). In: *ACM Transactions on Design Automation of Electronic Systems* 12.3 (Aug. 2007). ISSN: 1084-4309. DOI: 10.1145/1255456.1255460.
- [Sch+08] Julien Schmaltz et al. “A functional formalization of on chip communications”. In: *Formal Aspects of Computing* (2008). DOI: 10.1007/s00165-007-0049-0.
- [Bor+09] Dominique Borrione et al. “A formal approach to the verification of networks on chip”. In: *EURASIP J. Embedded Syst.* 2009 (Jan. 2009). ISSN: 1687-3955. DOI: 10.1155/2009/548324. URL: <https://doi.org/10.1155/2009/548324>.
- [BS09] Tom van den Broek and Julien Schmaltz. “Towards a formally verified network-on-chip”. In: *2009 Formal Methods in Computer-Aided Design*. 2009, pp. 184–187. DOI: 10.1109/FMCAD.2009.5351124.

- [Che+10] Yean-Ru Chen et al. “Formal modeling and verification for Network-on-chip”. In: *The 2010 International Conference on Green Circuits and Systems*. 2010, pp. 299–304. DOI: 10.1109/ICGCS.2010.5543050.
- [Soz10] Matthieu Sozeau. “Equations: A Dependent Pattern-Matching Compiler”. In: *Interactive Theorem Proving*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 419–434. ISBN: 978-3-642-14052-5.
- [VS10] Freek Verbeek and Julien Schmaltz. “Formal specification of networks-on-chips: deadlock and evacuation”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’10. Dresden, Germany: European Design and Automation Association, 2010, pp. 1701–1706. ISBN: 9783981080162.
- [Esm+11a] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 365–376. ISBN: 9781450304726. DOI: 10.1145/2000064.2000108. URL: <https://doi.org/10.1145/2000064.2000108>.
- [Esm+11b] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 365–376. ISSN: 0163-5964. DOI: 10.1145/2024723.2000108. URL: <https://doi.org/10.1145/2024723.2000108>.
- [PS11] Vinitha Arakkonam Palaniveloo and Arcot Sowmya. “Application of Formal Methods for System-Level Verification of Network on Chip”. In: *2011 IEEE Computer Society Annual Symposium on VLSI*. 2011, pp. 162–169. DOI: 10.1109/ISVLSI.2011.57.
- [12] “IEEE Standard for System and Software Verification and Validation”. In: *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)* (2012), pp. 1–223. DOI: 10.1109/IEEESTD.2012.6204026.
- [Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN: 0262026651.
- [Ver13] Freek Verbeek. *Formal verification of on-chip communication fabrics*. SI: sn, 2013.
- [ZH14] Anam Zaman and Osman Hasan. “Formal verification of circuit-switched Network on chip (NoC) architectures using SPIN”. In: *2014 International Symposium on System-on-Chip (SoC)*. 2014, pp. 1–8. DOI: 10.1109/ISSOC.2014.6972449.
- [BP17] Haseeb Bokhari and Sri Parameswaran. “Network-on-Chip Design”. In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jürgen Teich. Dordrecht: Springer Netherlands, 2017, pp. 461–489. ISBN: 978-94-017-7267-9. DOI: 10.1007/978-94-017-7267-9_16. URL: https://doi.org/10.1007/978-94-017-7267-9_16.
- [Cho+17] Joonwon Choi et al. “Kami: a platform for high-level parametric hardware specification and its modular verification”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110268. URL: <https://doi.org/10.1145/3110268>.
- [PMS18] M. Pandey, B. Murphy, and S. Safarpour. *Finding Your Way Through Formal Verification*. CreateSpace Independent Publishing Platform, 2018. ISBN: 9781986274111. URL: <https://books.google.de/books?id=ANPCswEACAAJ>.

- [Sep+18] Johanna Sepulveda et al. “Towards the formal verification of security properties of a Network-on-Chip router”. In: *2018 IEEE 23rd European Test Symposium (ETS)*. 2018, pp. 1–6. DOI: 10.1109/ETS.2018.8400692.
- [19] “IEEE Standard for VHDL Language Reference Manual”. In: *IEEE Std 1076-2019* (2019), pp. 1–673. DOI: 10.1109/IEEESTD.2019.8938196.
- [Bou+20] Thomas Bourgeat et al. “The essence of Bluespec: a core language for rule-based hardware design”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 243–257. ISBN: 9781450376136. DOI: 10.1145/3385412.3385965. URL: <https://doi.org/10.1145/3385412.3385965>.
- [Soz+20] Matthieu Sozeau et al. “The MetaCoq Project”. In: *J. Autom. Reason.* 64.5 (June 2020), pp. 947–999. ISSN: 0168-7433. DOI: 10.1007/s10817-019-09540-0. URL: <https://doi.org/10.1007/s10817-019-09540-0>.
- [Pit+21] Clément Pit-Claudel et al. “Effective Simulation and Debugging for a High-Level Hardware Language Using Software Compilers”. In: *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual, April 19-23, 2021*. Ed. by Tim Sherwood, Emery Berger, and Christos Kozyrakis. ASPLOS 2021. Association for Computing Machinery, 2021. URL: <https://pit-claudel.fr/clement/papers/cuttlesim-ASPLOS21.pdf>.
- [Fos22] Harry Foster. *The 2022 Wilson Research Group Functional Verification Study*. Siemens Blogs. Accessed: 06.01.2025. 2022. URL: <https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/>.
- [HA22] Sebastian Haas and Nils Asmussen. “A Trusted Communication Unit for Secure Tiled Hardware Architectures”. In: *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2022, pp. 1–4. DOI: 10.1109/ICECS202256217.2022.9971056.
- [QBa] QBayLogic. *Clash: A modern, functional, hardware description language*. <https://clash-lang.org/>. [n.d.]
- [Xil] Xilinx. *Vivado HLS*. <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>. [n.d.]

List of Codes

4.1	Importing Kôika libraries	14
4.2	Creating registers in Kôika	15
4.3	Defining types of registers	15
4.4	Initialization of registers	15
4.5	Creating external functions	16
4.6	Defining types of external functions	16
4.7	Leftmost router.	16
4.8	Intermediate router	16
4.9	Rightmost router	17
4.10	Defining rules	17
4.11	Mapping rules to routers	17
4.12	Scheduling the rules.	18
4.13	Type checking for Kôika rules	18
4.14	Setting up registers and external functions for testing	18
4.15	Writing a test Lemma in Kôika	19
4.16	Proof	19
4.17	Defining <code>basic_packet</code> type	20
4.18	Polymorphic <code>send</code> and <code>receive</code> functions	21
4.19	Action for middle router	22
4.19	Action for middle router	22
4.19	Action for middle router	23
4.19	Action for middle router	23
4.19	Action for middle router	24
4.20	Mapping actions to polymorphic routers	24
4.21	Inductive datatype with runtime-dependent number of constructors	25
5.1	Defining natural numbers in Coq	28
5.2	Quoting in MetaCoq	28
5.3	Unquoting in MetaCoq	29
5.4	Creating definitions in Coq using <code>MetaCoq Run</code>	29
5.5	Generating inductive types using MetaCoq	29
5.5	Generating inductive types using MetaCoq	30
5.5	Generating inductive types using MetaCoq	30

5.5	Generating inductive types using MetaCoq	31
5.6	Desired function for mapping rules to actions	31
5.7	Compact representation of the AST for rule to action mapping	32
5.8	N-rule schedule	32
5.9	Syntax for schedule	33
5.10	Registering schedule and rules-to-action mapping into the Coq Environment .	33
5.11	Wrapping rule generation in <code>Module</code>	34
5.12	Lemma to reason about generated code	34
5.13	Creating Σ	35
5.14	Lemma to prove left identity using typing judgement	35
5.15	Proof term	35
5.16	Proving left identity using tactics in Coq	36
5.17	Inductive type <code>Fin.t</code>	36
6.1	<code>rule_name_t</code> using dependent types	39
6.2	<code>ext_fn_t</code> using dependent types	40
6.3	<code>reg_t</code> using dependent types	40
6.4	Mapping rules to actions using Equations	41
6.4	Mapping rules to actions using Equations	42
6.4	Mapping rules to actions using Equations	42
6.4	Mapping rules to actions using Equations	42
6.4	Mapping rules to actions using Equations	43
6.4	Mapping rules to actions using Equations	43
6.5	<code>schedule</code> using Equations	43
6.5	<code>schedule</code> using Equations	44
6.5	<code>schedule</code> using Equations	44
6.5	<code>schedule</code> using Equations	44
6.5	<code>schedule</code> using Equations	45
6.5	<code>schedule</code> using Equations	45
6.6	FiniteType typeclass	45
6.7	<code>regt_idx</code>	46
6.8	<code>regt_elems</code>	46
6.9	Property verifying correct initialization of registers	47

List of Figures

2.1	Percentage of IC/ASIC project time spent in verification [Fos22]	5
2.2	Illustration of productivity and verification Gap [Riz02]	6
2.3	A packet-switched mesh NoC with XY routing	9
3.1	GeNoC model	12
4.1	The Kôika compile(r) and the Kôika programs are extracted from Coq into OCaml. The formal reasoning is not.	14
4.2	A simple 3-stage NoC pipeline.	15
4.3	The 4-stage advanced pipeline	20
5.1	Conversion of syntax in MetaCoq	27
6.1	Accessing registers in dependent <code>reg_t</code>	40
7.1	Number of cells consumed per router for NoCs designed by Meta-programming and Dependent Programming for different router configurations.	50
7.2	Area comparison between NoCs designed by Meta-programming and Dependent Programming for different router configurations.	50
7.3	Power consumption for NoCs designed by Meta-programming and Dependent Programming for different router configurations.	50

List of Tables

2.1	Comparison of formal verification strategies for hardware	7
5.1	Comparison of Metaprogramming and Dependent Programming	37