Remark on the master's dissertation and the oral presentation

This master's dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master's dissertation are not included in this text.

Acknowledgements

I would like to express my deepest gratitude to Prof. dr. ir. Wim Bogaerts and Prof. dr. ir. Dirk Stroobandt for their time, guidance, patience, and trust in applying for an *FWO* proposal to extend this Master Thesis. Through their advice and guidance, I have gained a breadth of knowledge and understanding that I have done my best to share in this thesis. It is with great pleasure that I write this document to share these findings and insights with them and others within the scientific community.

I would also like to give my most heartfelt thanks to the best friend one could ever ask for: ir. Thomas Heuschling, for his patience, friendship, guidance and all of the amazing moments we spent throughout our studies. I would also like to thank him for his help in proofreading this thesis and his advice on the PHÔS programming language. I also would like to thank Alexandre Bourbeillon for his help and advice for the creation of the formal grammar of the PHÔS programming language and being a great friend for over a decade.

I must also thank the incredible people that helped me proofread and improve my thesis: Daniel Csillag and Mossa Merhi Reimert for their time, advice and support. And Léo Masson for his help on programmatic visualisation of hexagonal lattices and his advice regarding typesetting.

Finally, my parents, Evelyne Dekens and Baudouin d'Herbais de Thun, were also there for me every step of the way and I deeply thank them for their support and listening to my endless rambling about photonics and programming.

– Sébastien d'Herbais de Thun

Wavre BE, 16th of June 2023

Permission of use on loan

The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In the case of any other use, the copyright terms have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation.

– Sébastien d'Herbais de Thun

Wavre BE, 16th of June 2023

Abstract

In this thesis, a novel way of programmatically designing photonic circuits is introduced, using a new programming language called PHÔS. This thesis' primary goal is to research which paradigms, techniques, and languages are best suited for the programmatic description of photonic circuits, with a special emphasis on programmable photonics as it is being researched at Ghent University. This involves an in-depth analysis of existing programming languages and paradigms, followed by a careful analysis of the functional requirements of photonic circuit design. This analysis highlights the need for a new language dedicated to photonic circuit design that is able to concisely and effectively express photonic circuits.

The design of this language is then shown, with all of the steps for its implementation carefully detailed. Parts of this language are implemented in a prototype compiler. One of its components, the constraint-solver, was the primary focus of this development effort, which has shown to be capable of simulating many photonic circuits based on simple constraints and operations.

Finally, meaningful demonstrations of the capabilities of the language and the constraint-solver are shown.

Keywords

Programmable photonic, photonic circuit design, programming language, photonic circuit simulation.

Table of contents

Glossary

2X2 TUNABLE COUPLER	*A tunable coupler with two inputs and two outputs*	5, 7
ADC	*Analog to Digital Converter*	8, 9
ADT	*Algebraic Data Type*	XIII, 64, 65, 72, 78, 90
API	*Application Programming Interface*	17, 20, 21, 22, 23, 25, 29, 32, 39, 40, 43, 62, 82, 110
ASIC	*Application Specific Integrated Circuit*	22, 36
AST	*Abstract Syntax Tree*	XI, XIII, XIV, 78, 83, 85, 86, 87, 88, 89, 90, 98, 135
AWGN	*Additive White Gaussian Noise*	116
BC	*Bytecode*	95, 96, 97, 98, 102
BER	*Bit Error Rate*	114, 116
CFG	*Control Flow Graph*	XI, 91, 92, 94, 95
CPLD	*Complex Programmable Logic Device*	3
CPU	*Central Processing Unit*	33, 116
CST	*Concrete Syntax Tree*	86
DAC	*Digital to Analog Converter*	8, 9
DRY	*Don't Repeat Yourself*	24, 55
DSL	*Domain Specific Language*	13, 16, 28
DSP	*Digital Signal Processor*	
ECS	*Entity-Component-System*	
EDA	*Electronic Design Automation*	54, 57
EVM	*Error Vector Magnitude*	114, 115
FFI	*Foreign Function Interface : a way to call functions from other languages*	106
FIR	*Finite Impulse Response*	7
FPGA	*Field Programmable Gate Array*	1, 3, 7, 8, 12, 13, 22, 28, 29, 32, 33, 39, 56, 104
FPPGA	*Field Programmable Photonic Gate Array*	3
FSR	*Free Spectral Range*	52
GADT	*Generalized Algebraic Data Type*	90
GPL-3.0	*GNU General Public License version 3.0*	26
GPU	*Graphics Processing Unit – also commonly used for highly parallel computing*	33
HAL	*Hardware Abstraction Layer*	42, 43, 51, 54, 55, 57, 60, 77, 99, 100, 104, 105, 106, 109
HDL	*Hardware Description Language*	13, 16, 17, 22, 23, 28, 29, 32, 33, 34, 36, 38, 53, 69
HIR	*High-level Intermediate Representation*	89, 90, 98
HLS	*High Level Synthesis*	27, 28, 29, 33, 34, 35, 36
HPC	*High-Performance Computing*	15
HTTP	*Hypertext Transfer Protocol – the protocol used for web navigation*	21
IC	*Integrated Circuit*	9, 10, 32

LIST OF FIGURES

List of tables

List of listings

Accessibility in this document

Short overview of accessibility and readability features of this document.

Accessibility

This document was designed with accessibility to colorblind and visually impaired people in mind. The colors used are generally chosen to have good contrast for colorblind individuals, as can be seen here. Most colored elements are accompanied by an icon, usually from the unicode standard to make it screen reader friendly. Additionally, all images have alternate text descriptions.

Navigation

All elements and references are clickable for ease of navigation in the document. Additionally, all figures, table, glossary entries, and references are clickable and will take you to the appropriate section. External links, those being links that lead to a website, are highlighted in blue and underlined.

Info boxes

For improved readability and breaking up of the monotony of the text, this document uses info boxes. These boxes are used to highlight important information, such as definitions, remarks, conclusion and important hypotheses. Below you will find a full list of the different types of info boxes used in this document.

DEFINITION: This is a **definition**, the word or phrase in bold is the term being defined. Most are adapted from the literature and are used for important elements that are not common knowledge for photonic engineers.

Definitions usually have a source in the footer

This is an info box, it contains a remark that is tangential or useful for the understanding of the document, but not essential.

Note: This is a note, additional information that is not essential for the understanding of the document, but is useful for the reader.

This is a question box, it contains an important research question or hypothesis that is being investigated in the document.

The footer contains a link to where the question is answered.

This is a conclusion or summary box, it contains a summary with key information that are needed for subsequent sections. Additionally, this is where answers to questions and hypothesis are given.

1. INTRODUCTION

Photonic integrated circuits have become a significant industry in the past few years. However, their design is still difficult and slow, requiring a lot of expertise and time. This thesis provides a novel way of designing photonic circuits using code and a new way of simulating them using a faster simulation paradigm. This should allow for the rapid prototyping and iteration of photonic circuits. And the co-simulation of photonic circuits with other system components. This thesis will focus on its applicability to the field of programmable photonics, especially recirculating programmable photonics. However, the concepts and ideas presented in this work apply to the broader field of photonic circuit design.

To enable rapid development and prototyping, several teams around the world, including at the PRG (*Photonics Research Group*) , have been researching ways of creating so-called photonic processors: devices that are generic enough to be reconfigured to meet a variety of use cases. These devices are generally based on the concept of recirculating meshes, which allow the designer to create all kinds of different circuits by simply changing the configuration of the mesh [3]. As such, these devices are often referred to as programmable PICs (*Photonic Integrated Circuit*) . These devices were used as inspirations and will be used as demonstration platforms for the work and mockups presented in this thesis.

This thesis merges aspects from several disciplines, including photonics and computer science. As such, it will first give some background information about photonics that is required to understand this thesis, followed by an in-depth analysis of the computer science concepts and paradigms discussed in this thesis. This analysis will allow for a comparison of existing languages, paradigms, and techniques and will lead to the development of a novel solution for photonic circuit design. Then, a discussion will be presented on the translation of the user's intent and the requirements that must be met to translate this intent. Finally, a discussion of the language created as part of this thesis, *PHÔS* , will be presented, as well as a set of examples showing its effectiveness and simulation capabilities.

1.1 MOTIVATION

There is a need to develop appropriate abstractions and tools for the design of photonic circuits, especially as it pertains to the programming of so-called photonic FPGAs (*Field Programmable Gate Array*) [4] or photonic processors. As with all other types of circuit design, such as digital electronic circuits, analog electronic circuits, and RF circuits, appropriate abstractions can allow the designer to focus on the functionality of their design rather than the implementation [5]. One may draw parallels between the abstraction levels used when designing circuits and the abstractions used when designing software. Most notably, the distinction made in the software-engineering world between imperative and declarative programming. The former is concerned with the "how" of the program, while the latter is focused on the "what" of the program [6].

At a sufficiently high level of abstraction, the designer is no longer focusing on the implementation details (imperative) of their design but rather on the functionality and behavioural expectations of their design (declarative) [6]. In turn, this allows the designer to focus on what truly matters to them: the functionality of their design.

Much of the design work on photonic circuits is currently done at a low-level of abstraction, close to the component-level [7]. This lack of abstraction leads to several issues for broader access to the fields of photonic circuit design. Firstly, it requires expertise and understanding of the photonic components, their physics, and the sometimes complex relationship between all of their design parameters. Secondly, designing and simulating a photonic circuit requires a lot of time and effort. Physical simulation of photonic circuits is slow [7, 8], which has led to efforts to simulate them using *SPICE* (*Simulation Program with Integrated Circuit Emphasis*) [9]. Finally, the design and implementation of photonic circuits are generally expensive, requiring taping out of the design and working with a foundry for fabrication. Therefore, the low-level nature of current methods increases the cost and the time to market for the product [3]. Due to this, there is considerable interest in constructing new abstractions, simulation methods, and design tools for photonic circuits, especially for rapid prototyping and iteration. This master's thesis aims to find new ways in which the user can easily design their photonic circuit and program them onto those programmable PICs [3].

Additionally, photonic circuits are often not the only component in a system. They are often used in conjunction with other technologies, such as analog electronics, used in driving the photonic components, digital electronics, to provide control and feedback loops and RF (*Radio Frequency*) to benefit from photonics' high bandwidth and high-speed capabilities [10]. Therefore, it is of interest to the user to co-simulate [7, 11] their photonic circuits with the other components of their systems. This problem is partly addressed using *SPICE* simulation [9]. However, *SPICE* tools are often lacking, especially regarding digital co-simulation, making the process difficult [12], relying instead on ill-suited alternatives such as *Verilog-A* (*A continuous-time subset of* Verilog-AMS *(*Verilog for Analog and Mixed Signal*)*) .

This work will offer a comprehensive solution to these problems by introducing a new way of designing photonic circuits using code, a novel way of simulating them, and a complete workflow for designing and programming them. Finally, an extension of the simulation paradigm will be introduced, allowing for the co-simulation of the designs with digital electronics, which could, in time, be extended to analog electronics.

1.1.a Research questions

The main goal of this work is to design a system to program photonic circuits. It entails:

1. How can the user express their intent?
    - Which programming languages and paradigms are best suited?
    - What workflows are best suited?
    - How can the user test and verify their design?
2. How to translate that intent into a PIC configuration?
    - What does a compiler need to do?
    - How to support future hardware platforms?
    - What are the unit operations that the hardware platform must support?

2. Programmable photonics

As previously mentioned in Section 1.1, the primary goal of this thesis is to find which paradigms and languages are best suited for the programming of photonic FPGAs . However, before discussing these topics in detail, it is necessary to start discussing the basics of photonic processors. This chapter will discuss photonic processors, their niche, and how they work. From this, the chapter will discuss the different types of photonic processors and how they differ. Finally, this chapter will conclude with the first and most important assumption made in all subsequent design decisions.

> This document uses the names photonic FPGA and photonic processor interchangeably. They both refer to the same thing: a programmable photonic device. The difference is that the former predates the latter in its use. Sometimes, they are also called FPPGA (*Field Programmable Photonic Gate Array*) [13].

2.1 Photonic processors

A photonic FPGA or photonic processor is the optical analogue to the traditional digital FPGA . It comprises gates connected using waveguides, which can be programmed to perform some function [14]. However, whereas traditional FPGAs use electrical current to carry information, photonic processors use light confined within waveguides to perform analog processing tasks.

However, it is interesting to note that, just like traditional FPGAs , some devices are more general forms of programmable PICs (*Photonic Integrated Circuit*) [4] than others, just like CPLDs (*Complex Programmable Logic Device*) are less general forms of FPGAs . As any PIC with configurable elements could be considered a programmable PIC , it is reasonable to construct a hierarchy of programmability, where the most general device is the photonic processor, which is of interest for this document, going down to the simplest tunable structures.

Therefore, looking at Figure 1, one can build four large categories of PICs based on their programmability. The first one (a) is not programmable at all; they require no tunable elements and are, therefore, the simplest. The second category (b) contains circuits that have tunable elements but fixed functions; the tunable element could be a tunable coupler or phase shifter, and allows the designer to tweak the properties of their circuit during use for purposes such as calibration, temperature compensation, or more generally, altering the behaviour of the circuit. The third kind of PIC is the feedforward architecture (c), which means that the light is expected to travel in a specific direction; it is composed of gates, generally containing tunable couplers and phase shifters. External devices such as high-speed modulators, amplifiers, and other elements can be added. Finally, the most generic kind of programmable PIC is the recirculating mesh (d), which, while also composed of tunable couplers and phase shifters, allows the light to travel in either direction, allowing for more general circuits to be built as explored in Section 2.1.b.

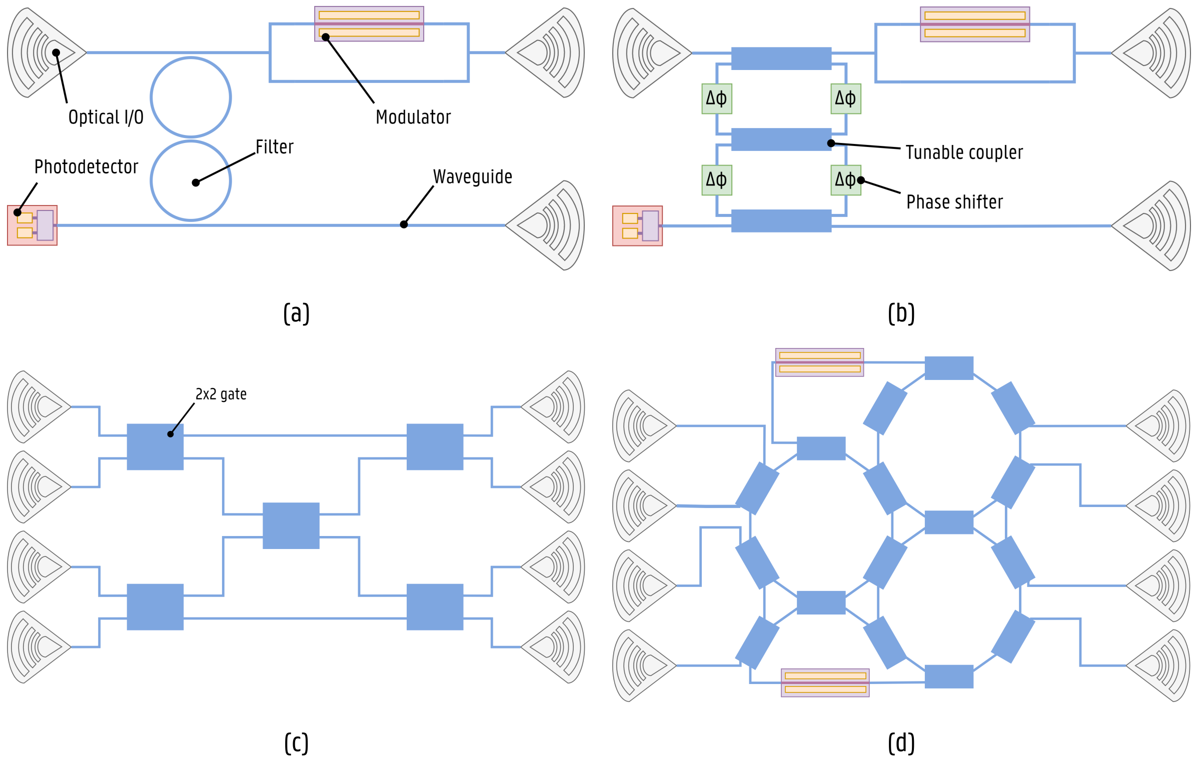


FIGURE 1 Shows a hierarchy of programmable PICs (*Photonic Integrated Circuit*) , starting at the non-programmable single function PIC (a), moving then to the tunable PIC (b), the feedforward architecture (c) and finally to the photonic processor (d).

In this work, the focus will be on the fourth kind of tunability, the most generic. However, the work can also apply to photonic circuit design in general and is not limited to photonic processors. As discussed in Section 2.1.b, the recirculating mesh is the most general kind of programmable PIC but also the most difficult to represent with a logic flow of operation because the light can travel in either direction. Therefore, the following question may be asked:

> **?** At a sufficiently high level of abstraction, can a photonic component be considered equivalent to a feedforward component?
>
> This will be answered in Section 2.2.a.

This question, which will be the driving factor behind this first section, will be answered in Section 2.2.a. However, before answering this question, it is necessary first to discuss the different types of photonic processors and how they differ. The answer to that question will also show that the solution suggested in this thesis also applies to feedforward systems. As this thesis is not focused on creating a photonic processor, but on the programming of said processors, building techniques will not be explored in detail.

2.1.a Components

As previously mentioned, a photonic gate consists of several components. This section will therefore discuss the different components that can be found in a photonic processor and how they work, as well as some of the more advanced components that can also be included as part of a photonic processor.

Waveguides

The most basic photonic component that is used in PICs is the waveguide. It is a structure that confines light within a specific area, allowing it to travel, following a pre-determined path from one place on the chip to another. Waveguides are, ideally, low loss, meaning that as small of a fraction of the light as possible is lost as it travels through the waveguide. They can also be made with low dispersion allowing the light to travel at the same speed regardless of its wavelength. This last point allows modulated signals to be transmitted without distortion, which is essential for high-speed communication.

Tunable 2x2 couplers

A 2x2 tunable coupler (*A tunable coupler with two inputs and two outputs*) is a structure that allows two waveguides to interact pre-determinedly. It is composed of two waveguides whose coupling, the amount of light "going" from one waveguide to the other, can be controlled. There are numerous ways of implementing couplers. In Figure 2, an overview of the different modes of operation of a 2x2 tunable coupler are given, along with a basic diagram of a 2x2 tunable coupler (a). It shows that depending on user input, an optical coupler can be in one of three modes; the first one (b) is the bar mode, where there is little to no coupling between the waveguides, the second one (c) is the cross mode, where the light is mainly coupled from one waveguide to the other, and the third one (d) is the partial mode, where the light is partially coupled from one waveguide to the other based on proportions given by the user.

The first mode (b) allows light to travel without interacting, allowing for tight routing of light in a photonic mesh. The second mode is also useful for routing, allowing signals to cross with little interference. The final state allows the user to combine two optical signals based on predefined proportions. This is useful for applications such as filtering for ring resonators or splitting.

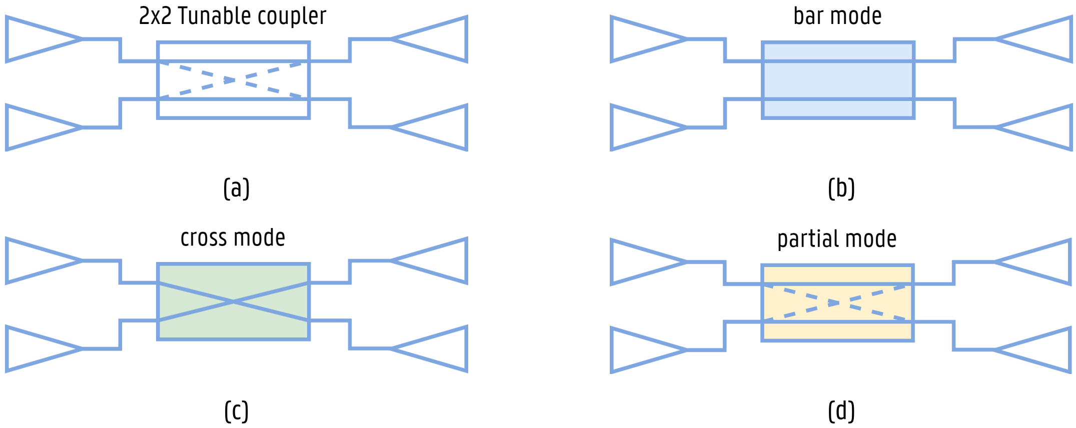


Figure 2 2x2 tunable coupler (a) and its different states: in "bar" mode (b), in "cross" mode (c), and in "partial" mode (d). The blue triangles are optical inputs and outputs.

Note: The colour scheme shown in Figure 2 for the different modes is kept throughout this document when showing photonic gates and their modes.

There are many construction techniques for building 2x2 tunable couplers , each with its own advantages and disadvantages. The most common ones are the Mach-Zehnder interferometers with two phase shifters. However, other techniques involve using MEMS (*Microelectromechanical Systems*) or liquid crystals [4, 14, 15].

Detectors

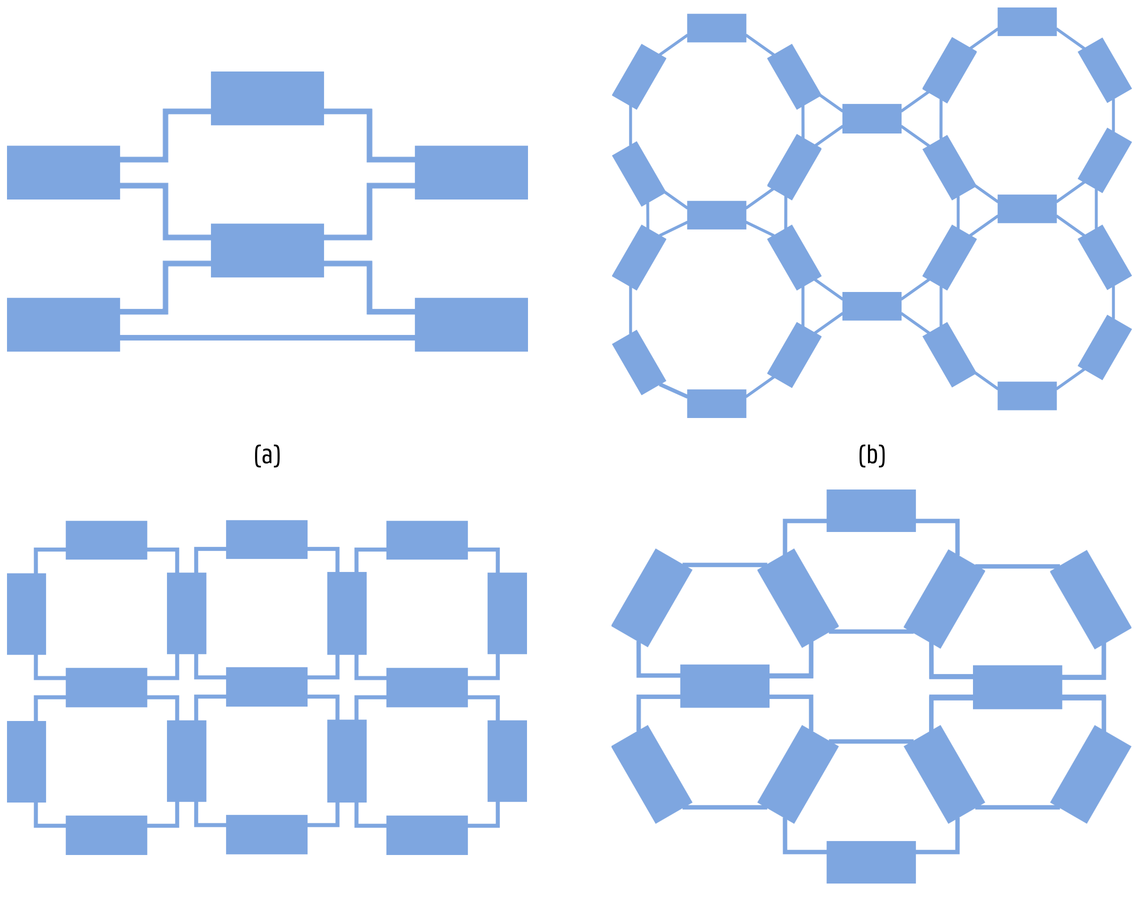
Detectors are used to turn optical signals into electrical signals. In photonic processors, there are commonly two kinds: low-speed detectors used to measure optical power at several points inside the processor and high-speed detectors used to demodulate high-speed signals.

Modulators

Contrary to detectors, modulators turn electrical signals into optical signals. They do not do this by producing the optical signal but by modulating the phase or the amplitude of an existing optical signal.

2.1.b Meshes

Four main kinds of meshes can be built for programmable photonics, shown in Figure 3: the feedforward mesh (a) and three kinds of recirculating mesh: hexagonal (b), rectangular (c), and triangular meshes (d). It is also possible to create meshes not made of a single kind of cell, but these will not be discussed in this thesis. This section will discuss the major differences between the feedforward and the recirculating architectures. In the case of this thesis, hexagonal meshes are the primary focus. This is because they are the most capable kind of meshes [4].



(a) (b)

(c) (d)

Figure 3 | The four kinds of programmable meshes: feedforward (a), hexagonal (b), rectangular (c), and triangular (d).

All of the architectures rely on the same components [4], those being 2x2 tunable couplers (*A tunable coupler with two inputs and two outputs*) , optical phase shifters and optical waveguides. These elements are combined in all-optical gates, which can be configured to achieve the user's intent. Additionally, to provide more functionality, the meshed gates can be connected to other devices, such as high-speed modulators, amplifiers, and detectors [4, 14, 15].

The primary difference between the feedforward architecture and the recirculating architecture is the ability of the designer to make light travel both ways in one waveguide. As is known [16], light can travel in two directions in a waveguide with little to no interactions. This means that, without additional waveguides or hardware complexity, a photonic circuit can support two guiding modes, one in each direction. This property can be used for more efficient routing and creating more complex and varied structures [4].

As it has been shown in [15], recirculating meshes can create more advanced structures, such as IIR (*Infinite Impulse Response*) elements, whereas feedforward architectures are limited to FIR (*Finite Impulse Response*) components. This is due to the inability of the feedforward architecture to express feedback loops, limiting them to FIR components, whereas recirculating meshes allow the creation of feedback loops and IIR components. Indeed, in a feedforward mesh, the typical structure being built is the Mach-Zehnder interferometer. In contrast, in a recirculating mesh, one may build structures such as ring resonators, which are inherently IIR components. Recirculating meshes can also express structures such as MZI (*Mach-Zehnder Interferometer*) and can represent both IIR and FIR components.

The recirculating mesh is more capable as it allows feedback loops and IIR components, whereas the feedforward mesh is limited to FIR components. Additionally, recirculating meshes allow light to travel in both directions in a single waveguide, allowing for more efficient routing and complex structures.

2.1.c Potential use cases of photonic processors

There are many use cases for photonic processors, some of which will be shown in this thesis as examples in Section 6. However, this section will first discuss areas of particular interest for photonic processors. Photonic processors' first and primary advantage is that they can replace the need to develop custom PICs , which is extremely expensive. They can also be used during the development of said PICs as a platform for prototyping. Another one of their advantages, which broadens their reach, is the ability to reprogram the processor in the field, just like a traditional FPGA . In the following paragraphs, several broad areas of interest will be discussed, and examples of applications in those areas will be mentioned. Those areas are telecommunication, optical computation, RF processing, and sensing applications.

Telecommunications The telecommunications industry is one of the largest existing users of photonic technologies, most notably optical fibers. Therefore, it should come as no surprise that this is one of the applicative areas of particular interest for programmable photonics. They can be used by service providers close to their customers for multiplexing, transceivers, and resource allocations, such as in fiber-to-the-home deployments [4].

Optical computation In some cases, such as machine learning, it has been shown that processing can be accelerated and energy efficiency improved by using photonic processing. As photonic processors are reprogrammable by nature, they could be used to accelerate workloads in data centers and edge computing scenarios. Companies like *LightMatter* are already making strides in optical computing accelerators for machine learning applications [17].

RF PROCESSING RF processing refers to the concept of processing radio signals using photonic technologies by modulating the RF signals of interest onto optical signals and then benefiting from the low-energy, very high bandwidth of photonic components to efficiently process signals in the analog domain. This has interest for RADAR applications, but also mm-Wave communications and 5G [18].

SENSING Sensing can take many forms, such as LiDAR, which is used in self-driving cars, or even in the medical field, such as in the case of sensing for the detection of cancerous cells, where a photonic processor could be used to process the signals produced by a sensor [4, 19]. In recent years, fiber sensing has been used in many applications, such as aviation, oil, gas, and more [20, 21]. The advantages of photonic processors for these use cases allow the reduction of weight, overall system complexity, and design cost. Therefore, photonic processors are interesting for sensing applications, as they may significantly reduce system design costs.

2.1.d Embedding a photonic processor in a larger system

In this thesis, the focus will mainly be on the design of circuits for photonic processors. However, one must keep in mind that photonic processors are not standalone systems but rather components of larger, more complex systems. Complex systems are rarely limited to a single domain. Indeed, they are often composed of multiple technologies, such as digital electronics, analog electronics, photonic circuits, and real-time processing. Therefore, it is important to understand how photonic processors can be embedded into these larger systems and how they can be integrated into existing systems. The act of integrating multiple technologies is often called codesign. Photonic processors are already a form of codesign since they are composed of both photonic and electronic components.

INTEGRATION WITH ELECTRONICS As previously mentioned, photonic processors integrate two types of electro-optic interfaces: modulators and detectors. These components can be used to interface the insides of the photonic processor with a larger electronic scheme. Due to their nature, these components can be used for both digital and analog electronics. Indeed, photonic processors can be used as analog signal processors and in mixed-signal systems. This is particularly interesting for RF signal processing, as photonic processors can offer high bandwidth, high-speed, and low energy consumption signal processing, all of which are difficult to achieve in analog electronics.

INTEGRATION WITH SOFTWARE While this thesis will not discuss integration with electronics at length, integration with the software will. As discussed in Section 2.4, there is an interest in integrating software control over the functionality and behaviour of photonic processors. Due to their nature, photonic processors cannot make decisions, but they can be interfaced with software that is able to process data and make decisions. Additionally, the designer may use software to create feedback loops to control their photonic circuit from the software.

In Figure 4, one can see how a photonic processor may be integrated with analog electronics, digital electronics, and embedded software by using DACs (*Digital to Analog Converter*) and ADCs (*Analog to Digital Converter*) to interface with the photonic processor, an FPGA for high-speed digital processing, and an embedded processor for control. However, while it does not show how optical inputs and outputs may be used, it provides a high-level overview of how a photonic processor and some of its internal components may be interfaced with in a bigger system.

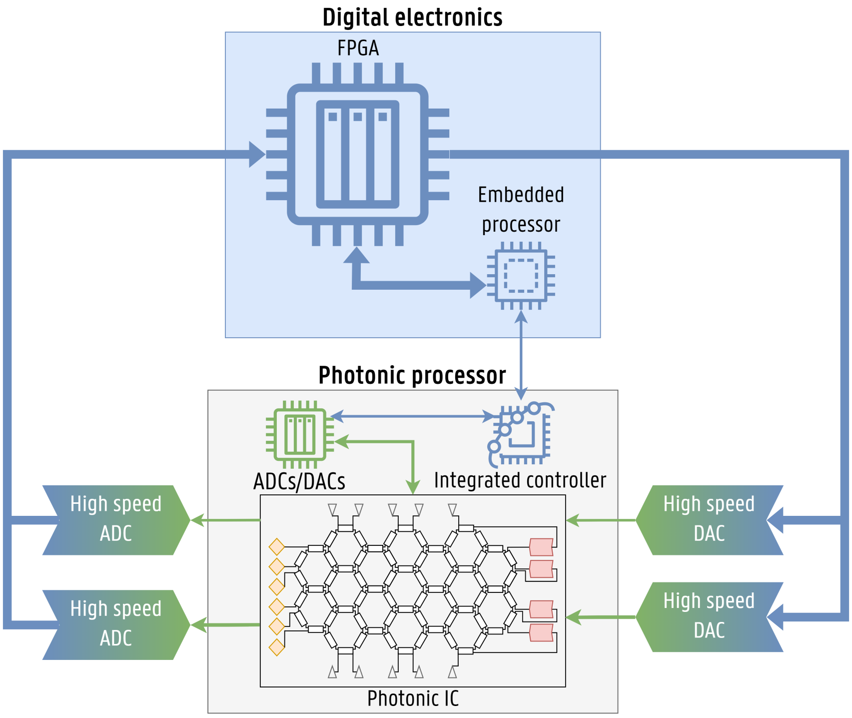


Figure 4 Figure showing the integration of a photonic processor with electronics and software. It shows how a photonic processor is composed of the photonic IC (*Integrated Circuit*) , ADCs , and DACs to interface with its integrated controller. It then shows how the overall photonic processor may be integrated with digital electronics and software running on an embedded processor. Blue elements represent digital electronics, while green elements represent analog electronics.

2.2 Circuit representation

When creating tools for circuit design, it is important to carefully consider how the circuit will be represented, as both the users of the tool, and its developers, will need to interact with this model for many steps in the design process, such as for simulation, optimisation, synthesis, and validation. As far as this thesis is concerned, it will use one of the most common photonic circuit representations, the netlist. In addition, the guided mode in each direction of the waveguides will be represented as a separate net.

Definition: A **netlist** is a list of components and their connections used to represent a photonic circuit. A **net** is a connection between two components, which represents a waveguide.

Adapted from [22].

Netlists are common abstractions in electronics to represent a list of components and their connections. However, in the case of photonics, the definition is altered and made more limited: in electronics, a net can be connected to many components. However, in the case of a photonic circuit, the port of a component is only ever connected to the port of another component, never more. This is because, in this thesis, splitters are considered components in and of themselves; therefore, splitting the signal is equivalent to adding a component to the circuit.

Bidirectional ports, such as the ones on the edge of a photonic processor, are represented as two separate logical ports, one for each direction: incoming and outgoing light. This representation is acceptable because light can travel in both directions in photonic waveguides with little to no interaction. Therefore, one can model these modes as being two separate signals [23]. This model is beneficial in the case of recirculating photonic meshes, as it helps distinguish the direction of the light in the circuit, making it easier to efficiently reuse photonic gates for bidirectional signal routing.

2.2.a Feedforward approximation

As mentioned in Section 2.1, a type of programmable photonic IC is the feedforward processor, which assumes that light travels from an input port to an output port in a single direction. However, this thesis is focused on the more general kind of processor that uses recirculating meshes. Therefore, one may wonder whether it is possible to model components using a feedforward approximation. Indeed, this section will discuss the axiom that any photonic circuit can be represented as a feedforward circuit, given a sufficiently high level of abstraction.

From theory, it is known that light can travel in both directions of a waveguide with little to no interactions [23]. Additionally, the scattering matrix defining the circuit is symmetric for reciprocal and time-invariant components. Conveniently, it so happens that most passive, or pseudo-passive[1] photonic components are reciprocal and time-invariant. Therefore, most photonic components can always be represented under this formalism. For components that are not reciprocal, one can split the component into two components, one for each direction, and model them as a set of separate components. Finally, components that are not time-invariant, such as modulators, can be modelled as time-invariant components as long as the variation in time is slow enough, compared to the oscillation period of the light, such that it can be considered constant. Finally, for components such as isolators, one can consider them a three-port device with two inputs, one being sunk, the other not, and one output.

Some components may be difficult to accurately model in this formalism. For example, SOAs (*Semiconductor Optical Amplifier*) can be challenging to express in this formalism since their gain is spread over both modes. However, this can be solved by modelling the SOA as a unidirectional component. This removes the ability to model the SOA as a bidirectional component, but it is a reasonable approximation for most cases. Additionally, if the user needs to model the SOA as a bidirectional component, they could model it as two components whose exact response depends on the other component.

Intuitively, one can think of these abstracted models as black boxes, where the contents do not matter as long as the expected functionality is present. For example, a ring resonator can be modelled as a black box with two inputs and two outputs, where the input and output ports are labelled as $a_{\mathrm{in}}, b_{\mathrm{in}}$ and $a_{\mathrm{out}}, b_{\mathrm{out}}$ respectively. This is shown in Figure 5. In this model, one can use the properties of a ring resonator to model the relations between these ports.

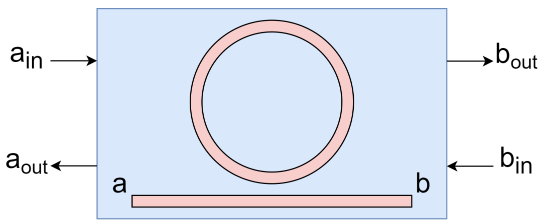


Figure 5 A black box representation of a ring resonator. The input and output ports are labelled as $a_{\mathrm{in}}, b_{\mathrm{in}}$ and $a_{\mathrm{out}}, b_{\mathrm{out}}$ respectively.,

[1] Components that are actively powered, but slow varying enough to be considered passive.

> It has been shown that, given a sufficient level of abstraction, any bidirectional photonic circuit can be represented by an equivalent, higher-level, feedforward circuit. This result is crucial for formulating the requirements for the programming interface of such a photonic processor.

2.3 Non-idealities

Like all other photonic components, photonic processors are impacted by non-idealities, temperature dependence, and manufacturing variabilities. These impact the well-functioning of the circuit programmed within the processor. Therefore, one must consider this impact when designing the circuit, or programming the design, in the case of a photonic processor. However, photonic processors should mitigate these variations as much as possible, as they are high-level design platforms, to allow for a more efficient design process. This section will discuss these non-idealities, and solutions to automatically mitigate their impact will be proposed.

Temperature dependence Most photonic components exhibit a temperature dependence, which can be exploited to build structures such as phase shifters. However, undesired temperature changes can impact the circuit in unexpected ways. One traditional way to mitigate temperature changes is to maintain the device at a constant temperature using external means, such as Pelletier elements. However, this limits the potential use cases of the device, as such means tend to be bulky and power-hungry.

Wavelength dependence In addition to the temperature dependence mentioned above, photonic components also exhibit a strong wavelength dependence. Part of this dependence is desired, such as in the case of wavelength filters. Nevertheless, in all other cases, the user expects that the device behaves with a flat frequency response. Therefore, photonic processors must also provide mitigations for this dependence.

Manufacturing variability Variabilities in the manufacturing process of the PIC cause the third kind of non-idealities; they can introduce all kinds of non-idealities, such as higher losses, reflections, and more. The imperfection of the manufacturing process causes these variabilities; therefore, the resulting devices are not identical. This means that the user's design will work differently from one chip to the next. However, one of the goals of photonic processors is to act as a high-level design platform, which means that the user should not have to worry about these variabilities.

2.4 Initial design requirements

This section will discuss the basic design requirements for a programming interface to photonic processors. These requirements form the base of the design of the programming interface and are, therefore, crucial to the design of the interface. More requirements and details for each will be provided in Section 4.

High-level of abstraction & modularity Ideally, the programming interface of photonic processors should be high-level enough that it is entirely abstracted from the underlying hardware, making it easier to design circuits. Furthermore, the interface should be modular, allowing users to build complex circuits from smaller, simpler building blocks. This is a crucial requirement, as it allows the user to build complex circuits from incrementally more complex building blocks, allowing them to build more advanced circuits without understanding the underlying hardware.

Platform independence The code should work across devices with little to no adjustment. This would avoid the fracturing of the ecosystem that can be observed in FPGAs , as well as reduce the burden of the user to port their code to different devices.

Tunability and reconfigurability Ideally, the user would want to tune their design while it is running, allowing them to build feedback loops of their control and adjust their design's behaviour on the fly; this functionality is called tunability. Furthermore, the user might want to completely replace parts of their device's functionality without reprogramming the entire device. This is called reconfigurability. These two features work hand-in-hand, providing a powerful tool for the user to build their designs and fully exploit the photonic processor's field-programmable nature.

Solutions to non-idealities One can categorise the previously mentioned non-idealities into time-invariant non-idealities and time-variant ones. The former does not change over time, such as manufacturing variabilities. It, therefore, can be compensated by using calibration tables that are uniquely generated for each chip or batches of chips. Time variant non-idealities, however, require an active approach, such as feedback loops, which the design solution for these photonic processors must provide for the user. In order to build these feedback loops, measurements must be taken of the signals of interest. These measurements are taken inside the photonic processor using photodetectors built into the chip. These measurements allow the integrated control system to adjust components, such as gain sections or phase shifters, to compensate for the non-idealities.

3. Programming of photonic processors

The primary objective of this chapter is to explore the different aspects of programming photonic processors. This chapter will start by looking at different traditional programming ecosystems and how different languages approach common problems when programming. Then, the existing software solutions and their limitations will be analysed. Finally, an analysis of relevant programming paradigms will be done. This chapter's secondary objective is to familiarise the reader with the concepts and terminology used in the rest of the thesis. This chapter will also introduce the reader to different programming paradigms relevant to the research at hand, as well as programming language concepts and components. As this chapter also serves as an introduction to programming language concepts, it is written in a more general way, exploring components of programming ecosystems – in Section 3.4 – before looking at specificities relevant to the programming of photonic processors.

3.1 Programming languages as a tool

> **Definition**: **Imperativeness** refers to whether the program specifies the expected results of the computation (declarative) or the steps needed to perform this computation (imperative). These differences may be understood as the difference between *what* the program should do and *how* it should do it.
>
> Adapted from [6]

Programming languages, in the most traditional sense, are tools used to express *what* and, depending on its imperativeness and paradigm, *how* a device should perform a task. A device, in this context, means any device that is capable of performing sequential operations, such as a processor, a microcontroller or another device. However, programming languages are not limited to programming computers but are increasingly used for other tasks. So-called DSLs (*Domain Specific Language*) are languages designed for specific purposes that may intersect with traditional computing or describe traditional computing tasks but can also extend beyond traditional computing. DSLs can be used to program digital devices such as FPGAs but also to program and simulate analog systems such as *Verilog-AMS* or *SPICE* .

Additionally, programming languages can be used to build strong abstractions over traditional computing tasks. For example, SQL (*Structured Query Language*) is a language designed to describe database queries by describing the *what* and not the *how*, making it easier to reason about the queries being executed. Other examples include *Typst*, the language used to create this document.

Furthermore, some languages are designed to describe digital hardware, so-called RTL (*Register Transfer Level*) HDLs (*Hardware Description Language*) . These languages are used to describe the hardware in a way closer to the actual hardware; therefore, they are not used to describe the *what* but the *how*. These languages are not the focus of this thesis, but they are essential to understand the context of the research at hand, and they will be further examined in Section 3.4.1, where their applicability to the research at hand will be discussed.

As such, programming languages can be seen, in a more generic way, as tools that can be used to build abstractions over complex systems, whether software systems or hardware systems, and therefore, the ecosystem surrounding a language can be seen as a toolbox providing many amenities to the user of the language. Is it, therefore, important to understand these components and reason about their importance, relevance and how they can best be used for a photonic processor.

3.2 Typing in programming languages

Definition: A **type system** is a system made of rules that assign a property called a type to values in a program. It dictates how to create them, what kind of operations can be done on those values, and how they can be combined.

Adapted from [24]

Definition: **Static or dynamic typing** refers to whether the type of arguments, variables and fields is known at compile time or at runtime. In statically typed languages, the type of values must be known at compile time, while in dynamically typed languages, the type of values is computed at runtime.

Adapted from [24]

All languages have a type system; it provides the basis for the language to reason about values. It can be of two types: static or dynamic. Static typing allows the compiler to know ahead of executing the code what each value is and means. This allows the compiler to provide features such as type verification that a value has the correct type for an operation and to optimise the code to improve performance. On the contrary, dynamic typing does not determine the type of values ahead of time, instead forcing the burden of type verification on the user. This practice makes development easier at the cost of increased overhead during execution and the loss of some optimisations [25]. Additionally, dynamic typing is a common source of runtime errors for programs written in a dynamically typed language, something that is caught during the compilation process in statically typed languages.

Therefore, static typing is generally preferred for applications where speed is a concern, as is the case in *C* and *Rust*. However, dynamic typing is preferred for applications where iteration speed is more important, such as in *Python*. However, some languages exist at the intersection of these two paradigms, such as *Rust*, which can infer parts of the type system at compile time, allowing the user to write their code with fewer type annotations while still providing the benefits of static typing. This is achieved through a process called type inference, where the compiler generally uses the de facto standard algorithm called *Hindley-Milner* algorithm [26, 27], which will be discussed further in Section 5.

Definition: **Polymorphism** refers to the ability of a language to allow the same code to be used with different types.

Adapted from [24]

Polymorphism allows code to be re-used with different types; a typical example is a list. For a list to work, it does not matter what type of value is contained within the list. Therefore one can make the list polymorphic over the item type such that the list is defined at the type `List<V>` where `V` is a type argument defining the contents of the list. Additionally, polymorphic languages often offer a way to define types that meet specific criteria, such as a type that is comparable or a type that is copyable. This is called a *trait* in *Rust* and an *interface* in *Java* and *C#*. On the other hand, *C* does not have polymorphism nor interfaces or traits. Then, polymorphic types and functions can request that their type argument meet these requirements. This is called *bounded polymorphism* and is a common feature in modern programming languages [24].

3.3 Explicitness in programming languages

In language design, one of the most important aspects to consider is the explicitness of the language, that is, how many details the user must manually specify and how much can be inferred. This is a trade-off between the expressiveness of the language and the complexity of the language. A language that is too explicit is both difficult to write and to read, while a language that is too implicit is difficult to understand and reason about, while also generally being more complex to implement. Therefore, it is essential to find a balance between these two extremes. Another factor to take into account is that too much “magic”, that is, operations being done implicitly, can lead to difficult-to-understand code, unexpected results and bugs that are difficult to track down.

Therefore, it is in the interest of the language designer and users to find a balance where the language is sufficiently expressive while also being sufficiently explicit. This is, generally, a difficult balance to find and can take several iterations to achieve. This balance is not the same for every programming language either. The target audience of the language tends to govern, at least to some extent, which priorities are put in place. For example, performance-focused systems, such as HPC (*High-Performance Computing*) solutions, tend to be very explicit, with fine-grained control to eke out the most performance, while on the contrary, systems designed for beginners might want to be more implicit, sacrificing complexity and fine-grained control for ease of use.

3.4 Components of a programming ecosystem

An important part of programming any kind of programmable device is the ecosystem that surrounds that device. The most basic ecosystem components that are necessary for the use of the device are the following:

- a language reference or specification: the syntax and semantics of the language;
- a compiler or interpreter: to translate the code into a form that can be executed by the device;
- a hardware programmer or runtime: to physically program and execute the code on the device.

These components are the core elements of any programming ecosystem since they allow the user to translate their code into a form the device can execute. And then to use the device. Therefore, without these components, the device is useless. However, these components are not sufficient to create a user-friendly ecosystem. Indeed, the following component list can also be desirable:

- a debugger: to aid in the development and debugging of the code;
- a code editor: to write the code in, it can be an existing editor with support for the language;
- a formatter: to format the code consistently;
- a linter: a tool used to check the code for common mistakes and to enforce a coding style;
- a testing framework: to test and verify the code;
- a simulator: to simulate the execution of the code;
- a package manager: to manage dependencies between different parts of the code;

- a documentation generator: to generate documentation for the code;
- a build system: to easily build the code into a form that the device can execute.

With the number of components desired, one can conclude that any endeavour to create such an ecosystem is a large undertaking. Such a large undertaking needs to be carefully planned and executed. And to do so, it is important to look at existing ecosystems and analyse them. This section will analyse the ecosystems of the following languages, when relevant:

- *C*: a low-level language that is mainly used for embedded systems and operating systems;
- *Rust*: a modern systems language primarily used for embedded systems and high-performance applications;
- *Python*: a high-level language that is used chiefly for scripting and data science;
- VHDL (VHSIC Hardware Description Language) : an HDL (*Hardware Description Language*) that is used to describe digital hardware;
- Verilog-AMS : an analog simulation language that has been used to describe photonic circuits [28];

Each of these ecosystems comes with a particular set of tools in addition to the aforementioned core components. Some of these languages come with tooling directly built by the maintainers of the languages, while others leave the development of these tools to the community. However, it should be noted that, generally, tools maintained by the language maintainers tend to have a higher quality and broader usage than community-maintained tools.

Additionally, the analysis done in this section will give pointers towards the language choice used in developing the language that will be presented in Section 5, a custom DSL language for photonic processors. As this language will not be self-hosted – its compiler will not be written in itself – it will need to use an existing language to create its ecosystem.

3.4.a Language specification & reference

Definition: A **programming language specification** is a document that formally defines a programming language, such that there is an understanding of what programs in that language mean. This document can be used to ensure that all implementations of the language are compatible with one another.

Adapted from [29]

Definition: A **programming language reference** is a document that outlines the syntax, features and usage of a programming language. It serves as a simplified version of the specification and is usually written during the development of the language.

Adapted from [29]

A programming specification is useful for languages that are expected to have more than one implementation, as it outlines what a program in that language is expected to do. Indeed, code that is written following this specification should therefore be able to be executed by any language implementation and produce the same output. However, this is not always the case. Several languages with proprietary implementations, such as *VHDL* and *SystemC* – two languages used for hardware description of digital electronics – have issues with vendored versions of the language [30].

This previous point is particularly interesting for the application at hand: assuming that the goal is to reuse an existing specification for the creation of a new photonic HDL , then it is crucial to select a language that has a specification. However, if the design calls for an API (*Application Programming Interface*) implemented in a given language instead, then it does not matter. Indeed, in the latter case, the specification is the implementation itself.

Additionally, when reusing an existing specification for a different purpose than the intended one, it is essential to check that the specification is not too restrictive. Indeed, as previously shown in Section 2.1, the programming of photonic processors is different from that of electronic processors. Therefore, special care has to be taken that the specification allows for the expression of the necessary concepts. This is particularly important for languages that are not designed for hardware description, such as *C* and *Python*. Given that photonics has a different target application and different semantics, most notably the fact that photonic processors are continuous analog systems – rather than digital processes – these languages may lack the needed constructs to express the necessary concepts, they may not be suitable for the development of a photonic HDL . Given the effort required to modify the specification of an existing language, it may be better to create a new language dedicated for photonic programming.

Furthermore, the language specification is only an important part of the ecosystem being designed when reusing an existing language. However, if creating a new language or an API , then the specification is irrelevant. It is desirable to create a specification when creating a new language, as it can be used as a thread guiding development. With the special consideration that a specification is only useful when the language is mature, immature languages change often and may break their own specification. And maintaining a changing specification as the language evolves may lower the speed at which work is done. For example, *Rust* is widely used despite lacking a formal specification [31].

3.4.b Compiler

Definition: A **compiler** is a program that translates code written in a higher-level programming language into a lower-level programming language or format so that it can be executed by a computer or programmed onto a device.

Adapted from [32]

The compiler has an important task; they translate the user's code from a higher-level language, which can still remain quite low-level, as in the case of *C*, into a low-level representation that can be executed. The type of language used determines the complexity of the compiler. In general, the higher the level of abstraction, the more work the compiler must perform to create executable artefacts.

An alternative to compilers are interpreters who perform this translation on the fly; such is the case for *Python*. However, HDLs tend to produce programming artefacts for the device. However, a compiler is more appropriate for the task at hand. This, therefore, means that *Python* is not a suitable language for the development of a photonic HDL . Or, at least, it would require the development of a dedicated compiler for the language.

One of the key aspects of the compiler, excluding the translation itself, is the quality of errors it produces. The easier the errors are to understand and reason about, the easier the user can fix them. Therefore, when designing a compiler, extreme care must be taken to ensure that the errors are as clear as possible. Languages like *C++* are notorious for having frustrating errors [33], while languages like *Rust* are praised for the quality of their errors. This is important to consider when designing a language, as it can make or break the user experience. Following guidelines such as the ones in [33] can help in the design of a compiler and greatly improve user experience.

Components

Compilers vary widely in their implementation. However, they all perform the same basic actions that may be separated into three distinct components:

- the frontend: which parses the code and performs semantic analysis;
- the middle-end: which performs optimisations on the code;
- the backend: which generates the executable artefacts.

The frontend checks whether the program is correct in terms of its usage of syntax and semantics. It produces errors that should be helpful for the user [33]. Additionally, in statically typed languages, it performs type checking to ensure that types are correct and operations are valid. In general, the frontend produces a simplified, more descriptive version of the code to be used in further stages [27]. The middle-end performs multiple functions but generally performs optimisations on the code. These optimisations can be of various types, and are generally used to improve the performance of the final executable. As will se discussed in Section 5, while performance is important, it is not the main focus of the proposed language. Therefore, the middle-end can be simplified. Finally, the backend, has the task of producing the final executable. This is a complex topic in and off itself, as it requires the generation of code for the target architecture. In the case of *C* using *Clang* – a common compiler for *C*– this is done by the LLVM compiler framework [34]. However, as with the middle-end, the final solution suggested in this work will not require the generation of traditional executable artefacts. Instead, some of the tasks that one may group under the backend, such as place-and-route, will still be required and are complex enough to warrant their own research.

3.4.c Hardware-programmer & runtime

Definition: The **hardware-programmer** is a tool that allows the user to write their compilation artefacts to the device. It is generally a piece of software that communicates with the device through a dedicated interface, such as a USB port. Most often, it is provided by the manufacturer of the device.

Adapted from [35]

The hardware-programmer is an important part of the ecosystem, as it is required to program the physical hardware. Usually it is also involved in debugging the device, such as with interfaces like JTAG (*Joint Test Action Group – A standard for testing integrated circuits*) . However, as this may be considered part of the hardware itself, it will not be further discussed in this section. However, it must be considered as the software must be able to communicate with the device.

Definition: The **runtime** is a program that runs on the device to provide the base functions of the device, such as initialization, memory management, and other low-level functions [32]. It is generally provided by the manufacturer of the device.

Adapted from [35]

In the case of a photonic processor, it is as of yet unclear what tasks and functions it will perform for the rest of the ecosystem, and warrants its own research and work. The runtime is a device-specific component, and as such, it is not possible to design it as a generic, reusable, component. Therefore, it is mentioned as a necessary component, and will be discussed in further details in Section 5 but will not be further considered in this section.

In general, the hardware-programmer and the runtime work hand-in-hand to provide the full programmability of the device. As the hardware-programmer is the interface between the user and the device, and the runtime is the interface between the device and the user's code compiled artefacts. Therefore, these two components are what allows the user's code to not only be executed on the device, but also to have access to the device's resources.

3.4.d Debugger

> Definition: A **debugger** is a program that allows the user to inspect the state of the program as it is being executed. In the case of a hardware debugger, it generally works in conjunction with the hardware-programmer to allow the user to inspect the state of the device, pause execution and step through the code.
>
> Adapted from [32]

The typical features of debuggers include the ability to place break-points – point in the code where the execution is automatically paused upon reaching it – step through the code, inspect the state of the program, then resume the execution of the program. Another common feature is the ability to pause on exception, essentially, when an error occurs, the debugger will pause the execution of the program and let the user inspect what caused this error and observe the list of function calls that lead to the error.

Some of the functions of a debugging interface are hard to apply to analog circuitry such as in the case of photonic processors. And it is evident that traditional step-by-step debugging is not possible due to the real-time, continuous nature of analog circuitry. However, it may be possible to provide mechanisms for inspecting the state of the processor by sampling the analog signals present within the device.

Due to the aforementioned limitations of existing digital debuggers, no existing tool can work for photonic processors. Instead, traditional analog electronic debugging techniques, such as the use of an oscilloscope are preferable. However, traditional tools only allow the user to inspect the state at the edge of the device, therefore, inspecting issues inside of the device require routing signals to the outside of the chip, which may not always be possible. However, it is interesting to note that this is an active area of research [36, 37, 38], for analog electronics at least, and it would be interesting to see what future research yields and how much introspection will be possible with "analog debuggers".

3.4.e Code formatter

> Definition: A **code formatter** is a program that takes code as input and outputs the same code, but formatted according to a set of rules. It is generally used to enforce a consistent style across a codebase such as in the case of the *BSD project* [39] and *GNU style* [40].
>
> Adapted from [41]

Most languages have code formatters such as *rustfmt* for *Rust* and *ClangFormat* for the *C* family of languages. These tools are used to enforce rules on styling of code, they play an important role in keeping code bases readable and consistent. Although not being strictly necessary, they can enhance the programmer's experience. Additionally, some of these tools have the ability to fix certain issues they detect, such as *rustfmt*.

Most commonly, these tools rely on *Wadler-style* formatting [42]. Due to the prominence of this formatting architecture, it is likely that, when developing a language, a library for formatting code will be available. This makes the development of a formatting code much easier as it is only necessary to implement the rules of the language.

3.4.f Linting

Definition: A **linter** is a program that looks for common errors, good practices, and stylistic issues in code. It is used with a formatter to enforce a consistent style across a codebase. They also help mitigate the risk of common errors and bugs that might occur in code.

Adapted from [41]

As with formatting, most languages have linters made available through officially maintained tools or community maintained initiatives. As these tools provide means to mitigate common errors and bugs, they are an important part of the ecosystem. They can be built as part of the compiler, or as a separate tool that can be run on the codebase. Additionally, linters often lack support for finding common errors in the usage of external libraries. Therefore, when developing an API , linters are limited in checking for proper usage of the API itself. Care must be done to ensure that the API is used correctly, such as making the library less error-prone through strong typing.

Nonetheless, linters are limited in their ability to detect only common errors and stylistic issues, as they can only check errors and issues for which they have pre-made rules. They cannot check for more complex issues such as logic errors. However, the value of catching common errors and issues cannot be understated. Therefore, whether selecting a language to build an API or creating a custom language, it is important to consider the availability and quality of linters.

As for the implementation of linters, they generally rely on a similar architecture to formatters, using existing compiler components to read code. However, they differ by matching a set of rules on the code to find common errors. Creating a good linter is, therefore, more challenging than creating a good formatter as the number of rules required to catch common errors may be quite high. For example, *Clippy*, *Rust*'s linter, has 627 rules [43].

Interestingly, as in the case of *Clippy*, some rules can also be used to suggest better, more readable ways of writing code, colloquially called good practices. For example, *Clippy* has a rule that suggests lowering cognitive load using the rule `clippy::cognitive_complexity` [43]. This rule suggests that functions that are too complex, as defined in the literature [44], should be either reworked or split into smaller, more readable code units.

3.4.g Code editor

Definition: A **code editor** is a program that allows the editing of text files. It generally provides features aimed at software development, such as syntax highlighting, code completion, and code navigation.

Adapted from [45]

As previously mentioned, most code editors also provide features aimed at software development. Features such as syntax highlighting: which provides the user with visual cues about the structure of the code, code completion: which suggest possible completions for the code the user is currently writing. And code navigation: allows the user to jump to the definition or user of a function, variable, or type. These features help the user be more productive and navigate codebases more easily.

In general, it is not the responsibility of the programming language to make a code editor available. Fully featured programming editors are generally called IDEs (*Integrated Development Environment*) . Indeed, most users have a preferred choice of editor, with the most popular being *Visual Studio Code*, *Visual Studio* – both from *Microsoft* – and *IntelliJ* – a *Java*-centric IDE from *JetBrains* [46]. Additionally, most editors have support for more than one language, either officially or through community-maintained plugins – additional software that extends the editor's functionality.

When creating a new language, effort should not go towards creating a new editor as much as supporting existing ones. This is usually done by creating plugins for common editors. However, this approach leads to repetition, has editors use different languages for plugin development. Over the past few years, a new standard, LSP (*Language Server Protocol*) , has established itself as a de-facto standard for editor support [47]. Allowing language creators to provide an LSP implementation and small wrapper plugins for multiple editors greatly reducing the effort required to support multiple editors. LSP was originally introduced by *Microsoft* for *Visual Studio Code*, but has since been adopted by most editors [47].

3.4.h Testing & simulation

Definition: **Testing** is the process of checking that a program produces the correct output for a given input. It is generally done by writing a separate program that runs parts – or the entirety – of the tested program and checks that it produces an output and that the produced output is correct.

Adapted from [48, 49]

Testing can generally be seen as checking that a program works as intended. They check for logical errors rather than syntactic errors, as the compiler would. Tests can be written ahead of the writing of the program. This is then called TDD (*Test Driven Development*) [50]. Additionally, external software can provide metrics such as *code coverage* that inform the user of the proportion of their code being tested [51].

Testing also comes in several forms; one may write *unit tests* that test a single function, *integration tests* that test the interaction between functions or modules, *regression tests* that test that a bug was fixed and does not reappear in newer versions, *performance tests* – also called *benchmarks* – which test the performance of the programs or parts of the program, and *end-to-end tests* which test the program as a whole.

Additionally, there also exists an entirely different kind of test called *constrained random* which produces random but correct input to a program and checks that, under no conditions, the program crashes. This is generally utilised to find edge cases that are not correctly handled and test the program's robustness, especially in areas concerning security and memory management.

Most modern programming languages, such as *Rust* provide a testing framework as part of the language ecosystem. However, these testing frameworks may need to be expanded to provide library-specific features to test more advanced usage. As an example, one may look at libraries like *Mockito*, which provides features for HTTP (*Hypertext Transfer Protocol – the protocol used for web navigation*) testing in *Rust* [52].

Therefore, when developing an API , it is important to consider how the API itself will be tested and how the user is expected to test their usage of the API . Additionally, when creating a language, it is important to consider how the language will be tested and what facilities will be provided to the user to test their code.

Definition: **Simulation** is the process of running a program that simulates the behaviour of a physical device. It is used to test that HDLs produce the correct state for a given input and starting state while also checking that the program does so in the correct timing or power consumption limits.

Adapted from [48, 49]

Simulation is more specific to HDLs and embedded development than traditional computer development, where the user might want to programmatically test their code on the target platform without needing the physical device to be attached to a computer. For this reason, the hardware providers make simulators available to their users. These simulators run the user's code as if it was running on real hardware, providing the user with tools for introspection of the device and checking that the program behaves as expected. As an example, *Xilinx* provides a simulator for their FPGAs called *Vivado Simulator*. This simulator allows the user to run their code on a simulated FPGA and check that the output is correct. This is an essential tool for the users of HDLs as it allows them to test their code without needed access to physical devices. Furthermore, it allows programmers working on ASICs (*Application Specific Integrated Circuit*) to simulate their code and design before manufacturing a prototype.

There are many simulation tools, such as *Vivado Simulator*, which allows users to test their FPGA code, other tools, such as *QEMU* which allow users to test embedded platforms. Additionally, many analog simulation tools exist, most notably the *SPICE* family of tools, which allow the simulation of analog electronics. There is also work being done to simulate photonic circuits using *SPICE* [9].

Finally, there also exist tools for physical simulation, such as *Ansys Lumerical* which are physical simulation tools that simulate the physical interactions of light with matter. These tools are used during the creation of photonic components used when creating PICs . However, they are generally slow and require large amounts of computation power [7, 8]. Therefore, when creating an API or a language for photonic processor development, it is desirable to consider how simulations will be performed and the level of details that this simulator will provide. The higher the amount of details, the higher the computational needs.

Verification As previously mentioned, when writing HDL code, it is desirable to simulate the code to check that it behaves correctly. Therefore, it may even be desirable to automatically simulate code in a similar way that unit tests are performed. This action of automatically testing through simulation is called *verification*, as verification is an integral part of the HDL workflow and ecosystem. Any photonic programming solution must provide a way to perform verification. This would be done by providing both a simulator and a tester and then providing a way of interfacing both together to perform verification.

3.4.i Package manager

<u>**Definition**</u>: A **package manager** or **dependency manager** is a tool that allows users to install and manage dependencies of their projects. These dependencies are generally libraries but can also be tools such as testing frameworks, etc.

<u>Adapted from [53]</u>

Package management is an integral part of modern language ecosystems. It allows users to easily install dependencies from the community and share new ones with the community. This is done through the use of a global repository of packages. Additionally, some package managers provide a way to create private repositories to protect intellectual property.

This last point is of particular interest for hardware description. It is common in the hardware industry to license the use of components – generally called IPs (*Intellectual Property*) . Therefore, any package manager designed to be used with an HDL must provide a way of protecting the intellectual property of package providers and users alike.

Additionally, package managers often offer version management, allowing the user to specify which version of a package they wish to use. As well as allowing package providers to update their packages as they get refined and improved. The same can be applied to hardware description as additional features may be added to a component, or hardware bugs may be fixed.

Finally, package managers usually handle nested dependencies, that is, they are able to resolve the dependencies of the dependencies, making the experience of a user wishing to use a specific package easier. This lets creators of dependencies build on top of existing community solutions, providing a more cohesive ecosystem. It is also important to point out that nested dependencies can cause conflicts, so package managers must provide a way to resolve these conflicts. This is usually done using *semantic versioning* which is a way of specifying version number that allow, to some degree, automatic conflict resolution [54].

3.4.j Documentation generator

<u>**Definition**</u>: A **documentation generator** is a tool that allows users to generate documentation for their code using their code. This is usually done using special comments in the code that are extracted and interpreted as documentation.

<u>Adapted from [55]</u>

The most common document generators are *Doxygen* used by the *C* and *C++* communities and *Javadoc* used by the *Java* community. Generally, documentation generators produce documentation in the form of a website, where all the documentation and components are linked together automatically. This makes navigating the documentation easier for the user. Additionally, some documentation generators, such as *Rustdoc* for the *Rust* ecosystem, provide a way to include and test examples directly in the documentation. This makes it easier for users to understand and use new libraries they might be unfamiliar with. For this reason, when developing an API , having a documentation generator built into the language is highly desirable as the documentation can serve as a way for users to learn the API but also for maintainers to understand the implementation of the API itself. Additionally, when creating a new language, care might be given to documentation generators, as they can provide a way for users to document their code and maintainers to document the language and its standard library. Finally, as technical documentation is the primary source of information for developers [46], it is essential to consider this need from users.

3.4.k Build system

Definition: A **build system** is a tool that allows users to build their projects.

Adapted from [32]

Build systems play an essential role in building complex software. Modern software is generally composed of many files that are compiled together, along with dependencies, configuration and many other resources, so it is challenging to compile modern software projects by hand. For these reasons, build systems are available. They provide a way to specify how a project should be built, this can be done in an explicit way: where the user specifies the steps that should be taken, the dependencies and how to build them. This approach would be similar to the popular *CMake* build system for the *C* family of languages. Other build systems like *Cargo* for *Rust* provide a mostly implicit way of building projects, where the user only specifies the dependencies and, by using a standardised file structure, the build system is able to infer how to build the project. This approach is easier to use and leads to a more uniform project structure. This means that, in combination with other tools such as formatters and linters, projects built using tools like *Cargo* all *look* alike, making them easy to navigate for beginners and experienced users alike. Additionally, not having to create *CMake* files for every new project follows the DRY (*Don't Repeat Yourself*) principle, which is a common mantra in programming.

Additionally, build systems can provide advanced features that are of particular interest to hardware description languages. Features such as *feature flags* are particularly useful. A feature flag is a property that can be enabled during building that is additive. It adds additional features to the program. As a simple example, consider the program in Listing 1: it will print `"Hello, world!"` when it is called. A feature flag called `custom_hello` may be used to add the function in Listing 2, which allows the user to specify a name to greet. It is purely additive: adding functionality to the previous library and using the `custom_hello` feature flag to enable the additional feature conditionally. This example is trivial, but this idea can be expanded.

Another example might be a feature flag that enables an additional type of modulator in a library of reusable photonic components. Some libraries even take a step further, where almost all of their features are gated, which allows them to be very lean and fast to compile. However, this is not a common occurrence.

```rust
fn print_hello_world() {
    println!("Hello, world!");
}
```

Listing 1 | Simple function that prints `"Hello, world!"`, in *Rust*.

```rust
#[cfg(feature = "custom_hello")]
fn print_hello_world(name: String) {
    println!("Hello, {name}!");
}
```

Listing 2 | Function that prints `"Hello, {name}!"` with a custom name, in *Rust*.

Whether providing the user with an API or creating a new language, it is essential to consider how the user's program must be built, as this task can quickly become quite complex. Enforcing a fixed folder structure and providing a ready-made build system that handles all common building tasks can significantly improve the user experience. And especially the experience of newcomers as it might prevent them from having to do obscure tasks such as writing their own *CMake* files.

3.4.1 Summary

As has been shown, many components are necessary or desirable to build a complete, user-friendly ecosystem. Official support for these components might be preferred as they lead to lower fracturing of their respective ecosystems. In Table 1, an overview of required components, desirable or not needed, along with a short description and their applicability for different scenarios are mentioned. Some components are more critical than others and are required to build the ecosystem. Most notably, the compiler, hardware-programmer, and testing and simulation tools are critical to be able to utilise the hardware platform. Without these components, the ecosystem is not usable for hardware development. However, while the other components are not strictly needed, several of them are desirable: having proper debugging facilities makes the ecosystem easier to use. Similarly, having a build system can help the users get started with their projects faster.

In Table 1, there is a distinction made on the type of design that is pursued, as will be discussed in Section 5, this thesis will create a new hardware description language, but the possibility of creating an API was also discussed. And while an API is not the retained solution, one can use this information for the choice of the language in which this new language, called *PHÔS* , will be implemented. Indeed, the same components that make API designing easy also make language implementation easier. As will be discussed in Section 3.9, *PHÔS* will be implemented in *Rust*. The language meets all requirements by having first-party support for all of the required and desired components for an API design. Its high performance and safety features make it a good candidate for a reliable implementation of the *PHÔS* ecosystem.

Component	Description	Importance	
		API design	Language design
Language specification	Defines the syntax and semantics of the language.	~	~
Compiler	Converts code written in a high-level language to a low-level language.	✓	~ (interpreted[1])
Hardware-programmer & runtime	Allows the execution of code on the hardware.	✓	✓
Debugger	Allows the user to inspect the state of the program at runtime.	~	~
Code formatter	Allows the user to format their code in a consistent way.	~	~
Linter	Allows the user to check their code for common mistakes.	✗	~
Code editor	Allows the user to write code in a user-friendly way.	✗ (provided by the ecosystem[2])	✗

Testing & simulation	Allows the user to test their code.	✓	✓
Package management	Allows the user to install and manage dependencies.	~	~
Documentation generator	Allows the user to generate documentation for their code.	✓	~
Build system	Allows the user to more easily build their codebase.	~	~

Table 1 This table shows the different components that are needed (✓), desired (~) or not needed (×) for an ecosystem. It compares their importance for different scenarios, namely whether developing an API that is used to program photonic processors or whether creating a new language for photonic processor development.

1. Interpreted languages are languages that are not compiled to machine code, but rather interpreted at runtime. This means that they do not require a compiler per se, but rather an interpreter.
2. A code editor is provided as an external tool, however, support for the language must be provided by the ecosystem. That being said, it is not a requirement and is desired rather than required.

Finally, Table 2 compares the ecosystem of existing programming and hardware description languages and their components. It shows that some ecosystems, like *Python*'s, have many components but that not all of them are first-party, nor is there always an agreement within the community on the best tool. However *Rust* is a particularly interesting candidate in this regard, as it has first-party support for all of the required components except hardware-programming and debugging tools. However, as noted in Table 2, most other languages do not come with first-party support for these tools either. However, as will be discussed in Section 3.5, it is not easy to learn, has not seen use in hardware synthesis and is, therefore not a good fit for regular users. But its robust ecosystem makes it a good candidate for a language implementation, something for which it has a thriving ecosystem of many libraries, colloquially called *crates*, fit for this purpose.

One can also see from Table 2 that simulation and hardware description ecosystems tend to be highly proprietary and incomplete. This problem can be solved by providing a common baseline for all tasks relating to photonic hardware description, where only the lowest level of the technology stack: the platform-support is vendored. Forcing platforms, through an open source license such as GPL-3.0 (*GNU General Public License version 3.0*) , to provide a standard interface for their hardware will allow a standardised ecosystem to be built on top of it. This is the approach that *PHÔS* will hopefully take.

Components	Traditional languages			Hardware description & simulation languages	
	C	Rust	Python	*Verilog-AMS*	*VHDL*
Language specification	✓ [56]	✗ [31]	✗ [57]	✓ [58]	✓ [59]
Compiler	∼ [1] (*Clang* & *GCC*)	✓ (*rustc*)	∼ (*PyPy* & *Numba*)	✗ (simulated)	∼ (synthesised)
Hardware-programmer & runtime	∼ [2] (vendored)	∼ [2] (vendored)	∼ [2] (vendored)	∼ [3] (vendored)	∼ (vendored)
Debugger	∼ [4] (*GDB* & *LLDB*)	∼ [4] (*GDB* & *LLDB*)	✓ (*PDB*)	∼ (vendored)	∼ (vendored)
Code formatter	∼ (*clang-format* & *uncrustify*)	✓ (*rustfmt*)	∼ (*Black*)	✗ [5]	✗ [5]
Linter	∼ (*clang-tidy* & *uncrustify*)	✓ (*Clippy*)	∼ (*Black*)	✗ [5]	✗ [5]
Code editor support	∼ (*clangd* & *ccls*)	✓ (*rust-analyzer*)	∼ (*Pyright*)	✗ [5]	✗ [5]
Testing	∼ (*CUnit*)	✓ (*rustc*)	∼ (*Pytest*)	∼ (*SVUnit*)	∼ (*VUnit*)
Simulation	∼ [2] (vendored)	∼ [2] (vendored)	∼ [2] (vendored)	∼ (vendored)	∼ (vendored)
Package management	✗	✓ (*Cargo*)	✓ (*PyPI*)	✗ [6]	✗ [6]
Documentation generator	∼ (*Doxygen*)	∼ (*Rustdoc*)	∼ (*Sphinx*)	✗ [5]	✗ [5]
Build system	∼ (*CMake*)	✓ (*Cargo*)	∼ [7] (*Poetry*)	∼ (vendored)	∼ (vendored)

Table 2 This table compares the ecosystems of different programming and hardware description languages. It shows whether the components are first-party (✓), third-party but well-supported (∼) or third-party but not well-supported or non-existent (✗). Each component also lists the name of the tool that is most commonly used for that purpose.

1. *C* has multiple, very popular, compilers, such as *GCC* and *Clang*. However, these are third-party, and for embedded and HLS (*High Level Synthesis*) development, there is no de facto standard.

2. Traditional programming languages usually rely on programmers and runtime provided by the hardware vendor of the targetted embedded hardware.
3. Verilog-AMS is a language used for simulation, not hardware description.
4. *C* and *Rust* generally share debuggers due to being native languages.
5. There do seem to exist some formatters, linters, code editor support and documentation generators for Verilog-AMS and VHDL , but they are not widely used and are sparsely maintained.
6. Due to the difficulty in handling intellectual property in hardware, there is no ubiquitous package manager for hardware description languages.
7. Python being interpreted, it does not need a build system, but some dependency and environment automation tools such as *Poetry* are widely used.

With the previous sections, it can be seen that creating a user-friendly ecosystem revolves around creating tools to aid development. The compiler and language cannot be created in isolation, and the entire ecosystem has to be considered to achieve the broadest possible adoption.

Depending on the implementation choice, the ecosystem's components will change. However, whether the language already exists or is created to program photonic processors, special care needs to be taken to ensure high usability and productivity through the availability or creation of tools to aid in development.

As will be discussed in Section 5, the chosen solution will be the creation of a custom DSL for photonic processors. This will be done due to the unique needs of photonic processors and the lack of existing languages that can be used for development targetting such devices. Moreover, this ecosystem will need to be created from scratch. However, the analysis done in this section will be used to guide the development of this ecosystem.

3.5 Overview of syntaxes

Following the analysis of programming ecosystem components, this section will analyse the syntaxes employed by various common programming languages. This section aims at building intuition on what these syntaxes look like, what they mean and how they can be applied to photonics. Additionally, this section will also analyse the syntaxes of existing HDLs and other DSLs that are used to program digital electronics – most notably FPGAs – and analog electronics. This analysis will also provide insight into whether these languages are suitable for programmable photonics. As programmable photonics works using different paradigms than digital and analog electronics, it is crucial to understand these differences and why they make these existing solutions unsuitable.

The first analysis, which looks at traditional programming languages, will look at the syntaxes of the following languages: *C*, *Rust*, and *Python*. These languages have been chosen as they are some of the most popular languages in the world, but also because they each bring different strengths and weaknesses with regards to the following aspects:

- *C* is a low-level language that is used as the building block for other non-traditional computation types such as FPGAs by being used for HLS [60], but is also being used for novel use cases such as quantum programming [61].

- *Rust* is another low-level language, it has not seen wide use in HLS or other non-traditional computation types, but it has modern features that make it a good candidate for API development. However, *Rust* has a very steep learning curve, making it unsuitable for non-programmers [62].
- *Python* is a common language that is used by a vast proportion of researchers and engineers [46, 63], which makes it a great candidate as the starting point of any language development. It is also used for some HDL development [64] and is used for the development of the existing photonic processor APIs , as well as for other non-traditional computation types such as quantum computing. However, it is a high-level, generally slow language with a syntax generally unsuitable for hardware description, as will be further discussed later.

The second analysis will focus on different forms of HDLs (*Hardware Description Language*) and simulation languages. Most notably, the following languages will be analysed:

- *SytemC* is a language that has seen increased use in HLS for FPGAs .
- *MyHDL* is a library for *Python* that gives it hardware description capabilities.
- *VHDL*: a common HDL used for FPGA development and other digital electronics [65].
- Verilog-AMS : a superset of *Verilog* that allows for the description of analog electronics. It has seen use in the development of photonic simulation, most notably in *Ansys Lumerical* [28].
- SPICE : a language that is used for the simulation of analog electronics. SPICE as seen use in the development of photonic simulation [9].

The goal of the second analysis will be to see whether any of these languages can be reused or easily adapter for photonic simulation. In the end, none of these languages fit the needs of photonic development, most notably with regard to ease of use. Nonetheless, the analysis provides insight that can be useful when designing a new language. It is also important to note that two distinct families of languages are in the aforementioned list: digital HDLs and analog simulation-centric languages. Therefore this comparison will be made in two parts, one for each family of languages.

3.5.a Traditional programming languages

To compare traditional programming languages, a simple yet classical example will be used: *FizzBuzz*, which is a simple program that prints the number from one to one hundred, printing *Fizz* when the number is divisible by three, *Buzz* when the number is divisible by five and *FizzBuzz* when the number is divisible by both three and five. The *C* implementation of *FizzBuzz* is shown in Listing 3. The *Rust* implementation of *FizzBuzz* is shown in Listing 4. The *Python* implementation of *FizzBuzz* is shown in Listing 5. For each of those languages, many different implementations are possible. However, a simple and representative version was used. As performance is not the focus of this comparison, choosing the most optimised implementation is not necessary.

Programming languages often take inspiration from one another. As such, most modern languages are inspired by *C*, which is itself inspired by *B*, *ALGOL 68* and *FORTRAN* [66]. *C* has had a large influence on languages such as *Python* [67] and *Rust* [31] – through *C++* and *Cyclone* – but also on HDLs such as *Verilog* (and therefore *Verilog-AMS*). As such, this section will start with an outlook on the syntax of *C* and discuss some of its shortcomings regarding more modern languages. Additionally, the more difficult aspects of the language will be discussed, most notably manual memory management and pointer semantics, as these two aspects are error-prone and even considered to be the root cause for most security vulnerabilities [68].

A simple *C* implementation of *FizzBuzz* can be found in Listing 3, it shows several important aspects of *C*:

- blocks of code are surrounded by curly braces (`{` and `}`);

- statements are terminated by a semicolon (`;`); however, curly braces can be omitted for single-line statements;
- variables are declared with a type and a name and optionally initialised with a value;
- functions are declared with a return type, a name, a list of arguments and a body;
- ternary operators are available for shorter but less readable conditional statements;
- *C* lacks a lot of high-level constructs such as string, relying instead on arrays of characters;
- *C* has a lot of low-level constructs, such as pointers, which are used to pass arguments by reference;
- *C* is not whitespace or line-space sensitive, and statements can span multiple lines;
- *C* uses a preprocessor to perform text substitution, such as importing other files;
- *C* needs a `main` function to be defined, which is the program's entry point.

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

void main() {
    char buffer[88];
    for(int i = 0; i <= 100; i++) {
        int len = sprintf(
            buffer,
            "%s%s",
            i % 3 ? "" : "Fizz",
            i % 5 ? "" : "Buzz");
        if (len == 0)
            sprintf(buffer, "%d", i);
        printf("%s\n", buffer);
    }
}
```

Listing 3 | *FizzBuzz* implemented in *C*, based on the *Rosetta Code* project [1].

The *Rust* implementation of *FizzBuzz* can be found in Listing 4, it shows several important aspects of *Rust*:

- blocks of code are surrounded by curly braces (`{` and `}`);
- statements are terminated by a semicolon (`;`);
- loops use the range syntax (`..`) instead of manual iteration;
- printing is done using the `print` and `println` macros, which are similar to *C*'s `printf`;
- variables do not need to be declared with a type, as the compiler can infer it;
- *Rust* is not whitespace or line-space sensitive, and statement can span multiple lines;
- *Rust* needs a `main` function to be defined, which is the program's entry point.

```rust
use std::fmt::Write;

fn main() {
    for i in 1..=100 {
        let out = format!(
```

```
            "{}{}",
            if i % 3 == 0 { "Fizz" } else { "" },
            if i % 5 == 0 { "Buzz" } else { "" }
        );


        if out.len() == 0 {
            println!("{i}");
        } else {
            println!("{out}");
        }
    }
}
```

Listing 4 | *FizzBuzz* implemented in *Rust*, based on the *Rosetta Code* project [1]

The *Python* implementation of *FizzBuzz* can be found in Listing 5, it shows several important aspects of *Python*:

- blocks of code are delimited by indentation;
- a newline terminates statements;
- loops use the `range` function instead of manual iteration;
- printing is done using the `print` function;
- variables do not need to be declared with a type, as the language is dynamically typed;
- *Python* is whitespace and line-space sensitive;
- *Python* does not need a `main` function to be defined, as the file is the program's entry point.

Python

```python
for n in range(1,101):
    response = ''
    if not (n % 3):
        response += 'Fizz'
    if not (n % 5):
        response += 'Buzz'
    print(response or n)
```

Listing 5 | *FizzBuzz* implemented in *Python*, based on the *Rosetta Code* project [1].

This simple example shows some fundamental design decisions for *C*, *Rust*, and *Python*, most notably that *Python* is whitespace and line-space sensitive, while *C* and *Rust* are not. This is a design feature of *Python* that aids in making the code more readable and consistently formatted regardless of whether the user uses a formatter or not. Then, focusing on typing, *Python* is dynamically typed, making the work of any compiler more difficult. Dynamic typing is a feature that generally makes languages easier to use at the cost of runtime performance, as type-checking has to be done as the code is running. Per contra, *Rust* takes an intermediate approach between *Python*'s dynamic typing and *C*'s manual type annotation: *Rust* uses type inference to infer the type of variables, which means that users still need to annotate some types. However, overall most variables do not need type annotations. This makes *Rust* easier to use than *C*, but also more challenging to use than *Python* from a typing point of view.

Additional features that the languages offer:

- *Python* and *Rust* both offer iterators, which are a high-level abstraction over loops;
- *C* and *Rust* both offer more control over data movement through references and pointers;
- *Python* and *Rust* both have an official package manager, while *C* does not;
- *Python* and *Rust* are both memory safe, meaning that memory management is automatic and not prone to errors;
- *Rust* is a thread-safe language, meaning that multithreaded programs are easier to write and less prone to errors;
- *C* and *Rust* are both well suited for embedded development. While *Python* has seen use in embedded development, it is not as well suited as the other two languages due to performance constraints;
- *Rust* does not have truthiness: only `true` and `false` are considered boolean values, while *Python* and *C* have truthiness, meaning several types of values can be used as boolean values.

> It was shown that traditional programming languages generally lack the features required to be used as a photonic HDL . However, *Python* is a strong candidate for creating an API , and *Rust* is a strong candidate for implementing a compiler.

3.5.b Digital hardware description languages

Unlike traditional programming languages, digital HDLs try and represent digital circuitry using code. This means that the code is not executed but rather synthesised into hardware that can be built. This hardware generally has one of two forms: logic gates that can be built discretely or LUTs (*Look Up Table*) programmed on an FPGA. Both processes involve "running" the code through a synthesiser that produces a netlist and a list of operations that are needed to implement the circuit. As previously discussed, in Section 3.1, languages can serve as the foundation to build abstractions over complex systems. However, most HDLs tend to only have an abstraction over the RTL (*Register Transfer Level*) level, which is the level that describes the movement, and processing of data between registers. Registers are memory cells commonly used in digital logic that store the result of operations between clock cycles. This means that the abstraction level of most HDLs is shallow.

This low-level of abstraction can be better understood by understanding three factors regarding digital logic programming. The first is the economic aspect: custom ICs are very expensive to design and produce. As such, the larger the design, the larger the dies needed, which increases cost; and FPGAs are costly devices, the larger the design, the more space it physically occupies inside of the FPGA , increasing the size needed and therefore the cost. The second factor is the design complexity: the more complex the design, the more difficult it is to verify and the slower it is to simulate, which decreases productivity. The third factor is with regard to performance. Three criteria characterise the performance of a design: the speed of the algorithm being implemented, the power consumed for a given operation, and the area that the circuit occupies. These performance definitions are often referred to be the acronym PPA (*Power, Performance, Area*) . As such, the design is generally done at a lower-level of abstraction to try and meet performance targets.

High-level synthesis

> **Definition**: **High-level Synthesis (HLS)** is the process of translating high-level abstractions in a programming language into RTL (*Register Transfer Level*) level descriptions. This process is generally done by a compiler that takes as input the high-level language and translates the code into a lower-level form.
>
> Adapted from [60, 69]

In recent years, there has been a push towards higher-level abstraction for digital HDLs . It takes the form of so-called HLS (*High Level Synthesis*) languages. These languages allow the user to build their design at a higher-level of abstraction, which is generally more straightforward and more productive [70]. Allowing the user to focus on the feature they are trying to build and not the low-level implementation of those designs. As discussed in Section 3.1, this can be seen as a move towards declarative programming or a less imperative programming model. Coupled with the rise of hardware accelerators in the data center and cloud markets, which are generally either GPUs (*Graphics Processing Unit – also commonly used for highly parallel computing*) or FPGAs , there has been an increased need for software developers to be able to use these FPGA -based accelerators. Because these software developers are generally not electrical engineers, and due to the high complexity of FPGAs , developing for such devices is not an easy skill to acquire. This has provided an industry drive towards economically viable HLS languages and tools that software developers can use to program FPGA -based accelerators.

Another advantage of HLS is the ability to test the hardware using traditional testing frameworks, as discussed in Section 3.4. l, testing systems for HDLs tend to be vendored and therefore difficult to port. Additionally, they are based on simulation of the behaviour, which is generally slower than running the equivalent CPU (*Central Processing Unit*) instructions. Therefore, testing the hardware using traditional frameworks is a significant advantage of HLS languages. In the same way that it allows the use of regular testing frameworks, it also enables the reuse of well-tested algorithms that may already be implemented in a given ecosystem which can drastically lower the development time of a given design and reduce the risk of errors. In addition to being able to use existing testing frameworks, the code can be verified using provers and formal verification tools, which can prove the correctness of an implementation, something that does not exist for traditional RTL level development.

Given that HLS development is generally easier, more productive and allows for the reuse of existing well-tested resources, it is a sensible alternative to traditional RTL level development. However, it does come at the cost of generally higher resource usage and lower performance. This is due to the fact that the HLS abstractions are still not mature enough to meet the performance of hand-written HDL code. However, there has been a push towards a greater level of optimisation, such as using the breadth of optimisation available in the *LLVM* (*Low-Level Virtual Machine*) compiler. This has allowed HLS to reach a level of performance acceptable for large swath of applications, especially when designed by non-specialists [71]. Other techniques, such as machine learning based optimisation techniques have been used to increase performance even further [72].

Modern RTL languages

In parallel to HLS development, a lot of higher-level RTL languages and libraries have been created, such as *MyHDL*, *Chisel*, and *SpinalHDL*. These alternatives are positioned as replacements to traditional HDLs such as *SystemVerilog*. They are often libraries for existing languages such as *Python*, and therefore inherit their broad ecosystems. As discussed in Section 3.4.l, HDLs , tend to be lackluster – or highly vendor-locked – with regard to development tools. And just as in the case of HLS , this can be an argument in favour of using alternatives, such as these HDLs implemented inside of existing languages.

These HDLs are generally implemented as translators, where, instead of doing synthesis down to the netlist level, they translate the user's code into a traditional HDL . As such, they are not a replacement for traditional HDLs but offer a higher-level of abstraction and better tooling through the use of more generic languages. This places these tools in an interesting place, where users can use them for their nicer ecosystems and easier development but still have the low-level control that traditional HDLs offer. This is in contrast to HLS , where this control is often lost due to the higher-level of abstraction over the circuit's behaviour. Additionally, these tools often integrate well with existing package-managers which are available for the language of choice, allowing for easy reuse and sharing of existing libraries.

3.5.c Comparison

For the comparison, three HDLs of varying reach and abstraction levels will be used: VHDL , *MyHDL*, and *SystemC*. They each represent one of the aforementioned categories: traditional HDLs , modern RTL -level languages, and HLS languages. For this comparison, a simple example of an n-bit adder will be used, where n is a design parameter. This will allow the demonstration of procedural generation of hardware and the use of modules and submodules to structure code.

> Most HDL languages come with pre-built implementations of adders. Usually, the compiler or synthesis tool chooses the best adder implementation based on the user's constraints. These constraints can relate to the area, power consumption or timing requirements.

In the first example, in Listing 6, it can be seen that the *VHDL* implementation is verbose, giving details for all parameters and having to import all of the basic packages (line $2-3$). In *VHDL* , the ports and other properties are defined in the `entity`, and the logic is implemented in an `architecture` block. This leads to functionality being spread over multiple locations, generally reducing readability. Assignments are done using the `<=` operator. Unlike most modern counterparts, the language does not use indentation or braces to denote code blocks but rather the `begin` and `end` keywords, which is a dated practice. However, *VHDL* does support parameterisation of the design, as can be seen on line 6 with the declaration of the generic `n`. This allows for the generation of hardware based on parameters, which is a useful feature for hardware design.

</>VHDL

```vhdl
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  entity Adder is
6  Generic(n:integer:=8);
7      Port ( Cin : in  STD_LOGIC;
8             A : in  STD_LOGIC_VECTOR (n-1 downto 0);
9             B : in  STD_LOGIC_VECTOR (n-1 downto 0);
10            Result : out  STD_LOGIC_VECTOR (n downto 0));
11            Cout : out STD_LOGIC
12 end Adder;
13
14 architecture Behavioral of Adder is
15 begin
16     Result <= A + B + Cin;
17     Cout <= Result(n);
18 end Behavioral;
```

Listing 6 | Example of a n-bit adder in *VHDL* , based on [2].

The second example based on *MyHDL*, in Listing 7, shows a combinatorial implementation of an adder. It shows that *MyHDL* relies on decorators to perform code transformations, something that may be useful when designing custom languages based on *Python* [73]. Despite using decorators, the code for the *Python* example is very short, relying on the `@always_comb` annotation to denote the combinatorial logic. The `@block` annotation is used to denote a block of code that will be translated to a module. Overall, code in *MyHDL* is generally easy to read and has a low barrier to entry for *Python* developers.

```python
from myhdl import *

@block
def nbit_adder(A, B, Cin, S, Cout):
    """ n-bit adder with carry in and carry out """
    @always_comb
    def logic():
        S.next = A + B + Cin
        Cout.next = (A + B + Cin) >> 4
    return logic
```

Listing 7 | Example of a n-bit adder in *MyHDL*.

The final and third sample is in *SytemC*, in Listing 8. It is verbose, using lots of macros, it does not directly support generics due to its *C* heritage, and requires the use of defined macros to configure the number of bits. Overall, it does not provide a pleasant user experience even for a simple example. Despite being a HLS language, it is seemingly less readable and user-friendly than *MyHDL*.

```cpp
#include "systemc.h"
#define N 8

SC_MODULE(NBIT_ADDER) {
    sc_in<sc_lv<N> > a, b;
    sc_in<sc_logic> cin;
    sc_out<sc_lv<N> > sum;
    sc_out<sc_logic> cout;
    SC_CTOR(NBIT_ADDER) {
        SC_METHOD (process);
    }
    void process() {
        sc_lv<N> a_val = a.read();
        sc_lv<N> b_val = b.read();
        sc_lv<N> sum_val = a_val + b_val + cin.read();
        sum.write(sum_val);
        cout.write(sum_val[N-1]);
    }
```

```
19 };
```

Listing 8 | Example of a n-bit adder in *SystemC*.

Three languages were shown, starting with *VHDL* , which is widely used in the industry and has a long history of support and use in hardware synthesis toolchains. A newer, very modern RTL language based on *Python* with a compelling feature set, *MyHDL*, was also shown. Finally, a HLS language, *SystemC*, was shown. It was shown that *MyHDL* is a very user-friendly language, with a low barrier to entry and a very modern feature set. It was also shown that *SystemC* is a very verbose language and does not provide a good user experience. It was also shown that *SystemC* does not support generics and requires the use of macros to achieve the same functionality. This is in contrast to *MyHDL*, which supports generics and parameterisation of designs. It was also shown that *MyHDL* is a very modern language, with a very modern feature set and a very low barrier of entry. This is in contrast to *SystemC*, a very verbose language that does not provide a good user experience. It was also shown that *SystemC* does not support generics and requires the use of macros to achieve the same functionality. This is in contrast to *MyHDL*, which implicitly supports generics and parameterisation of designs. However, this implicitness can be error-prone, which in the case of ASIC design would be very expensive.

Finally, none of the aforementioned HDLs provide any facilities for analog hardware description. Some, like *VHDL* , can provide analog modelling, but not analog hardware description. This is a significant limitation of all digital electronic HDLs . Additionally, the signal semantics they all use of *driven-once*, *drive_many* could lead to issues with signal splitting, as will be discussed in Section 5.2.d.

> It was shown that traditional RTL HDLs are not suitable for photonic development. They are not easily approachable for non-expert and lack the correct semantic for analog processing. However, *MyHDL* shows a promising approach to HDL creation based on *Python*.

3.5.d Analog simulation languages

There are several analog simulation languages. However, there are very few analog hardware description languages, and they mostly seem to be research languages [74, 75]. Due to this overall unavailability of analog HDLs , this comparison will instead rely on analog simulation languages, namely *SPICE* and *Verilog-AMS* . These two languages are very different, designed for different purposes and at different times. However, they are both actively used. Their uses differ significantly as *SPICE* aims to provide a netlist description of analog electrical circuitry to be simulated, whereas *Verilog-AMS* aims to provide models of analog systems compatible with mixed-signal simulations of digital and analog electronics.

SPICE *SPICE* is not a programming language but a configuration language: the user declares a list of nets and the components that connect these nets. As such, *SPICE* is very explicit, and little in the way of programmatic features are offered. Additionally, *SPICE* depends on models and is not meant to describe hardware. This means it is a very low-level representation of a circuit, which goes against the goal of using a high-level language, as discussed in Section 2.4.

Verilog-AMS *Verilog-AMS* is a modern mixed-signal simulation. It suffers from the same issues as *SPICE* , namely that it cannot be used for hardware description but rather hardware modelling. While *Verilog-AMS* has been used for photonic modelling, it is not a suitable candidate for use as a photonic HDL .

Existing analog modelling languages are unsuitable for photonic hardware description, as they are not hardware description languages but hardware modelling languages.

3.6 Analysis of programming paradigms

Definition: A **programming paradigm** is a style of programming, a way of thinking, structuring, and solving problems in a language.

Adapted from [76].

After an overview of existing programming languages, one must now consider the available programming paradigms. When selecting or creating a language, particular care must be taken when selecting one or more paradigms. This is because the choice of paradigms will affect the language's expressiveness and ease of use. Generally, most languages, like *Python* are imperative languages with elements from functional programming.

There are two broad categories of programming paradigms, imperative and declarative programming. As mentioned in Section 3.1, imperative languages are concerned with the "how" of programming, whereas declarative languages are concerned with the "what". A complete overview of all programming paradigms is available in Annex A [77]. In this comparison, the number of paradigms will be reduced as many of them exist. Instead, focusing on the most relevant ones, namely object-oriented, functional, logic, and dataflow programming. It is important to note that object-oriented programming is a subset of imperative programming and that functional programming is a subset of declarative programming, with dataflow programming being a subset of functional programming. This means that the aforementioned paradigms are not mutually exclusive and can, for example, be combined to create an object-oriented language with functional elements [78].

3.6.a Object-oriented programming

Object-oriented programming is one of the most common paradigms, being part of *Java*, *Python*, *C#*, and many others. It follows the idea that data is the most important part of an application and that it should be contained together in an object along with the methods acting upon it. For each piece of data, an instance of an object is created. In theory, this allows for the creation of complex data structures easily in a tree-like structure. Object-oriented also allows for inheritance, where one class of object inherits from another. The most typical example is shown in Listing 9, it shows a super class `Student` being inherited by a subclass `Sebastien`. This allows the subclass to override methods on the super class and share its initialisation function and state.

Python

```python
class Student:
    def __init__(self, name):
        self.name = name
    def print_thesis_grade(self, grade):
        print("Thesis grade of " + self.name + " is " + grade)
class Sebastien(Student):
    def __init__(self):
```

```python
        super().__init__("Sébastien d'Herbais de Thun")
    def print_thesis_grade(self, grade):
        print("Thesis grade of Sébastien is A+")
```

Listing 9 | Example of object-oriented programming in *Python*, showing inheritance and method overriding.

Criticism Object-oriented programming has been criticised for its tendency to create overly complex and sometimes confusing data structures. This is because it is very easy to create complex trees of classes, all interconnected, and all inheriting from one-another in not-always obvious ways. Additionally, one of the stated goals of object-oriented programming is to make code more modularised, therefore reusable. However, in practice, this is not always the case, as it is easy to create overly specialised classes [79].

3.6.b Functional programming

Functional programming views programs in the opposite way of object-oriented programming, instead emphasising procedures being done on data. In purely functional programming, all data is immutable, meaning that a value cannot be changed, only ever created or destroyed. This has advantages regarding limiting the number of side-effects, making the code easier to reason about. However, it can also make implementing some programs that require side-effects very difficult, such as communicating with other programs.

3.6.c Logic programming

Logic programming is a subset of functional programming, instead focused on logical relations. The most common example of a logic programming language is *Prolog*. In logic programming, the programmer defines a set of rules, and the program will try to find a solution to the problem. This is done by defining a set of rules and a set of facts. The program will then try to find a solution to the problem by applying the rules to the facts. Logic programming does not find its use in common programming, but rather in proving mathematical theorems and solving mathematical problems. As such, it is not suitable for hardware description.

3.6.d Dataflow programming

Dataflow programming is another subset of functional programming, where the program is represented as a graph of nodes, where each node performs a function, and the graph's edges are the data flowing between the nodes. Its data model is particularly interesting for hardware description, as it can represent the operations being done on a signal, with the "flow" of light being the edges of the graph. Indeed, this is the approach taken by *DFiant*, a *Scala* based RTL HDL that uses dataflow programming as its paradigm [80]. And as will be seen in Section 5, it is part of the paradigm used by *PHÔS* , the language created in this thesis.

3.7 Existing framework

There currently exists a framework developed at the PRG for the programming of photonic processors. However, its level of abstraction is low. It consists of manually setting the parameters of each photonic gate and then manually connecting them together. This is a very low-level approach, and as such, it is not suitable for the programming of complex photonic processors. However, it is still useful for the programming of simpler photonic circuits, and as such has been used for demonstrations of routing, switching, and circuit designing.

3.8 Hardware-software codesign

Definition: **Hardware-software codesign** is the process of designing a system where both the hardware and software components are designed together, with the goal of interoperating hardware components and software systems more easily. And optimise the system as a whole rather than the hardware and software components separately.

Adapted from [81].

Hardware-software codesign is difficult, requiring good communication and planning between the different parties. As such, some tools have been created to make this process easier. In FPGA development, this is usually done by having the synthesiser produce lists of registers which the software can configure. However, this is still error prone and limited in usefulness.

3.9 Summary

From the aforementioned criteria, one may give a score for each of the discussed languages based on its suitability for a given application. This is done in Table 3. The score is given on a scale of one to five, with one being the lowest and five being the highest. The score is given based on the following criteria: the maturity of the ecosystem and the suitability for different scenarios that were previously explored, notably: API design, root language – i.e. as the basis for reusing the existing ecosystem and syntax – and the implementation of a new language – i.e. using the language to build the ecosystem components of a new language. RTL languages implemented on top of *Python* are not included in the table. Neither is *SPICE* due to its restrictive scope.

From Table 3, one can see that for creating a new language, the best languages to implement it are *Rust* and *C*. And the best languages to inspire the syntax and semantics are *Python* and Verilog-AMS . Additionally, *C* is also a good inspiration due to its widespread use and the familiarity of its syntax. Finally, for the implementation of an API , the best choice is *Python* due to its maturity, simplicity and popularity in academic and engineering circles.

<table>
<tr><th rowspan="2">Language</th><th colspan="4">Applications</th></tr>
<tr><th>Ecosystem</th><th>API design</th><th>Root language</th><th>New language</th></tr>
<tr><td>C</td><td>●●●○○</td><td>●●○○○</td><td>●●○○○</td><td>●●●●○</td></tr>
<tr><td></td><td colspan="4">C is a fully featured low-level language; it is performant and has a simple syntax. However, it lacks some more modern ecosystem components and is error-prone. Because of this, it is unsuitable for API design since it would require the user to be familiar with memory management. It lacks many of the semantics of hardware description, making it unsuitable as a root language. However, its extensive array of language-implementation libraries makes it a good candidate for implementing a new language.</td></tr>
<tr><td>Rust</td><td>●●●●●</td><td>●●○○○</td><td>●●○○○</td><td>●●●●●</td></tr>
<tr><td></td><td colspan="4">Rust is a modern low-level language; it is very performant, has excellent first-party tooling, is quickly growing in popularity, and is memory safe. However, it has complicated syntax and semantics that is unwelcoming for non-developers, which makes it unsuitable for either API design or as a root language. However, its extensive array of language-implementation libraries and its memory and thread safety make it an excellent candidate for implementing a new language.</td></tr>
<tr><td>Python</td><td>●●●●○</td><td>●●●●●</td><td>●●●●○</td><td>●●○○○</td></tr>
<tr><td></td><td colspan="4">Python is a mature high-level language that sees wide use within the academic community; it has great third-party tooling and is easy to learn. These factors make it an excellent candidate for API design and as a root language. However, its slowness and error-prone dynamic typing make it an unsuitable candidate for implementing a new language.</td></tr>
<tr><td>Verilog-AMS</td><td>●○○○○</td><td>○○○○○</td><td>●●●○○</td><td>○○○○○</td></tr>
<tr><td></td><td colspan="4">Verilog-AMS is a mixed signal simulation software; its ecosystem is lackluster, with many proprietary tools which incur expensive licenses. It is not a generic language and is therefore not designed for an API to be implemented in the language, nor is it suitable for implementing a new language. However, it is a mature language with a familiar syntax to electrical engineers, which may make it suitable as the root language.</td></tr>
<tr><td>VHDL</td><td>●○○○○</td><td>○○○○○</td><td>●○○○○</td><td>○○○○○</td></tr>
<tr><td></td><td colspan="4">VHDL is a mature language with a large ecosystem but suffers from the same issues as Verilog-AMS , most notably that most tools are proprietary and licensed. Similarly, its nature as a hardware description language makes it unsuitable for API design or the creation of a new language. Its verbose syntax and semantics are challenging to learn and make the language difficult to read, which makes it unsuitable as a root language.</td></tr>
</table>

Table 3 | Comparison of the different languages based on the criteria discussed in Section 3.9.

4.

TRANSLATION OF INTENT & REQUIREMENTS

In Section 3, the different programming ecosystem components, paradigms and tradeoffs were discussed. In this section, the translation of the user's intent – i.e. the design they wish to implement – will be discussed in further detail. The translation of intent is how the user will write down their design and how the program translates that design into an actionable, programmable design. This section will also outline some of the features needed for easier intent translation. This will be done by discussing important features such as *tunability*, *reconfigurability*, and *programmability*. These features revolve around the ability of the user to tune the operation of their programmed photonic processor as it is running. For this purpose, this section will introduce novel concepts, such as *constraints* and their *solver* and *reconfigurability through branching*. These two essential concepts will be discussed in detail and synergise to create an easy-to-use, powerful programming ecosystem for photonic processors.

Additionally to the aforementioned points, several key features were discussed in Section 2.4. The features relate to real-time control, which works in pair with *reconfigurability* and *tunability*, and simulation, which will use *constraints* and its solver. Platform independence, which will be achieved through the design of a unified vendor-agnostic ecosystem and the visualisation of the design, which has led to the design of the *marshalling layers* which will be discussed in Section 5.8.

> **DEFINITION**: **Synthesis** is the process of transforming the description of a desired circuit into a physical circuit.
>
> Adapted from [82].

Synthesis is the process of transforming the user's code into a physical circuit on the chip. It is done in a multitude of stages that will be discussed in Section 5. These stages are all required to go from the user's code, which represents their intent, and turn it into an actionable design that can be executed on the photonic processor. The synthesis process is complex, involving many components that all must cooperate. Additionally, some of the tasks that synthesis must do, such as place-and-route, are computationally intensive and often regarded as NP-hard.

4.1 FUNCTIONAL REQUIREMENTS

> **DEFINITION**: A **functional requirement** is a requirement that specifies a function that a system or component of a system must be able to perform.
>
> Adapted from [83].

Before a user can design their circuit, they must list their functional requirement. These requirements are the functionality that they wish for their circuit to achieve. As previously discussed, in Section 3.1, these requirements are the most declarative form of the user's intent. However, some elements will generally be common to all those functional requirements. And can be seen as the functional requirements for intent translation. These requirements can be seen in Table 4 and are discussed in the following sections.

	Requirement	Description	Discussion
	Ideal behaviour	As discussed in Section 2.3, devices vary from device to device and over time and temperature. The user should be able to program the device without having to worry about these variations.	Section 4.9
Realtime feedback	Reconfigurability	Reconfigurability allows the user to change the topology of their device at runtime. This is useful for several reasons. The primary reason is the ability to change behaviour based on configurations or inputs.	Section 4.5
Realtime feedback	Tunability	On the other hand, Tunability does not change the topology in itself, but rather changes the behaviour of the mesh through tuning elements already present in the mesh. This may be used to vary gains, phase shifters, switches, or couplers to affect the behaviour of the mesh. This can also allow the user to build feedback loops that they control.	Section 4.5
Realtime feedback	Programmability	In order to be able to make use of tunability and reconfigurability, the user must be able to communicate with their programmed device programmatically. This is done using a HAL (*Hardware Abstraction Layer*) that handles communication with the device.	Section 4.2
	Simulation	Simulation allows the user to test their code, verify whether it works and debug it before running it on the device. This is an important feature as it also allows the user to experiment without having access to the device.	Section 4.6
	Platform independence	Platform independence allows the user to focus on their design rather than the specific device it is expected to run on. While some degree of platform dependence is to be expected, most of the code should be platform-independent and allowed to run on any device.	Section 4.7
	Visualisation	Visualisation allows the user to see the result of a simulation, what a finalised design looks like, and block diagrams of functionality. All of these features can help the user in their design process, but also help the user when sharing information with others. Therefore, providing visualisation is desirable.	Section 4.8

Table 4 | Functional requirements for intent translation

4.2 Programmability

Definition: A **HAL (*Hardware Abstraction Layer*)** is a library whose purpose is to abstract the hardware with a higher level of abstraction, allowing for easier use and programming of the hardware.

Adapted from [84].

Programmability refers to the ability of the user to interact with their circuit while it is running programmatically. This is done using a HAL , which allows for interoperation between their software and their hardware, completing the hardware-software codesign loop. The HAL is made of two parts: the core HAL provided by the device manufacturer and the user HAL generated by the compiler based on the user's code.

Core HAL As previously mentioned, the core HAL is provided by the device manufacturer and consists mostly of communication routines. It handles the communication between the user's software and the device. This HAL is, therefore, platform-specific and is not generated by the compiler. However, the core HAL must be able to communicate with the user HAL , which is generated by the compiler. This is done by enforcing that all HALs implement a common API that allows the user to interact with both the core HAL and the user HAL in a consistent way, making the code as portable as possible.

User HAL The user HAL is a higher-level part of the HAL built from the user's design. It encapsulates the tunable values, reconfiguration states and detectors that are defined within the design and allows the user to change these values, reconfigure the device and readout detectors. All the while using the names the user-defined for these different values. This allows the user to interact with the device in a way consistent with their design, making it easier to understand and use. This should improve productivity and reduce the risk of error in the hardware-software codesign interface.

User HAL template The user HAL needs to be generated from the user's design; however, there may be elements of this HAL that are platform specific and, therefore must be generated instead by the device support package. This is expected to be done by allowing the device support package to generate part of the user HAL through a template or custom code. This allows the device support package to provide platform-specific features to the user HAL or to optimise common implementations for the platform, further improving the quality and usability of the generated interface.

4.3 Intrinsic operations

From the physical properties and features of a photonic processor, as discussed in Section 2, one can extract a set of intrinsic operations that the processor must support. These operations in themselves are not required to be on the chip, but the support packages of the chip must be able to understand them and produce errors when they are not implemented. A complete list with a description can be found in Table 5. In this section, the intrinsic operators will be discussed in more detail.

Filter One of the core operations that almost all photonic circuits perform is filtering. It is a block that alters the amplitude of an input signal in a predictable manner based on its spectral content. Due to the prevalence of filters in photonic circuits, coupled with their special constraint – see below – they benefit from being an intrinsic operation for a photonic processor. During compilation and before place-and-route, the filter will be synthesised based on its arguments in order to produce a filter of the desired frequency response. This synthesised filter will therefore be optimised for the hardware platform.

There are many different types of filters, the most common ones that can easily be implemented on a mesh – are MZIs , ring resonators, and lattice filters. Additionally, compound filters that combine multiple types of filters or more than one filter can be created; such filters can have improved response or behave like bandpass filters. Therefore, the compiler's task is to choose the base filter based on the specification and performance criteria that the user has set. For example, the user might prioritise optimising for mesh usage rather than finesse or might optimise for flatness of the phase response rather than the mesh usage, etc.

Gain and loss All waveguides within a device will cause power loss in the optical signals. However, this loss may not be sufficient if the user is working with a high-power signal. Therefore some devices might include special loss elements whose loss is tunable or at least known. Besides, following the same principle, some users may want to compensate for this loss by using gain sections or amplify incoming signals. Optical gain is difficult to obtain on silicon platforms, just like sources, but it is possible to obtain gain through the use of rare-earth doped waveguides or other techniques such as micro-transfer printing. Therefore, the compiler must be able to synthesise gain and loss sections based on the user's specifications. However, if the device does not support gain or loss, the compiler should produce an appropriate error for the user.

Modulator and detectors Two key applications of photonic processors are telecommunication and processing RF signals. Therefore, it stands to reason that modulators and detectors are key components that are expected to be present in most photonic processors. Additionally, based on the device, there may be an optimal type of modulator for either type – phase modulation or amplitude modulation – and the compiler may choose an appropriate implementation of the modulator. Additionally, the same is true for detectors, although they would generally only be used for amplitude demodulation, with phase coherent demodulation being the responsibility of the user.

Splitters Signals are often split, and they may be split in specific ratios. For this reason, a splitter intrinsic operation that splits a signal into n new signals with weight provided by the user is desirable. Internally, the compiler will likely have to implement these splitters as 1-to-2 splitters with specific splitting ratios, but the user should not have to worry about this. Additionally, the compiler can optimise the placement of these splitters to minimise the mesh usage, or to minimise non-linear effects in high-power signals, or maintain phase coherence between signals.

Combiners and interferometers A combiner is the inverse of a splitter. It combines n signals together; it can operate in one of two modes, it can either try and reach a target power level – which can be the maximum power –, or it can interfere with the signals with their differential phase to create interference. The user is responsible for choosing which implementation to use. However in cases where the phase is well known, the compiler may be able to optimise the design by using a phase coherent combiner, thus not requiring a feedback loop and a phase shifter.

Switches Some devices may have hardware optical switches, while others may need to rely on feedback loops. Generally, all platforms should be able to support switching. Whether they rely on purpose-built hardware or on feedback loops does not matter. Switching can be useful in many applications, including telecommunication, signal processing, etc. It may be used to route test signals, route signals conditionally, or implement simple reconfigurability without the added cost of having more than one mesh.

Sources A lot of applications will need the generation of laser light; while this is difficult to achieve on silicon, it may be available on some devices. As laser sources are such an important part of photonics, it is important to at least plan for sources to be available in the future. Additionally, in some cases, the compiler may be able to synthesise a source from a gain source, reflectors and splitters. However, this is not always possible, and the compiler should be able to produce an error if the user requests a source and none is available.

Phase shifter Phase shifters are a necessary building block for a lot of more complex structures such as tunable MZIs , tunable filters, coherent communication, power combiners, etc. Therefore, as an integral part of the functioning of photonic processing, they must be present as an intrinsic operation. Additionally, they may be used in two different modes: the first mode is as a phase shifter, shifting the phase of a single signal, and the second mode is as a differential phase shifter, imposing a phase shift with respect to another signal. This case is especially interesting as it can be used to implement complex quadrature modulation schemes. In Section 6, examples regarding coherent communication will be presented that makes use of this intrinsic operation to implement complex modulation schemes, as well as to implement a beamforming network.

Delay lines Each waveguide being used on the chip adds latency to the signal. While this latency may be low at the scale of a modulated signal, it can still be relatively significant overall. For this reason, the device must provide ways for the user to align signals in time by using a delay line or multiple wires of different lengths to match the total optical length. It works nicely with the ability to express differential constraints, which will be discussed in Section 4.4.

Coupler Couplers are part of each photonic gate present in the processor. They have the ability to couple two signals based on a coupling coefficient. This is a key intrinsic, as it allows the user to couple signals directly, something that would otherwise be difficult to implement. In terms of the underlying hardware, it should be a direct one-to-one relationship with a coupler on the processor itself. However, the compiler should be able to optimise the coupling coefficient based on the signal's frequency content and the device's calibration curves.

Sink & empty source In some cases, the user might want to produce an empty signal or consume a signal without doing anything with it. For this reason, the compiler should be able to produce a sink intrinsic operation that consumes a signal with little to no return losses and an empty source that produces a signal with little to no power. This is especially useful in cases where the user might want to have a reference "empty" signal or a signal consumed without any effect on the rest of the system. An example of such cases is a spectrometer, which wants to switch in an empty signal to calibrate the dark current of the detectors.

Intrinsic operation	Description	Arguments
Filter	Filters a given signal at a given wavelength or set of wavelengths. The architecture and parameters are derived automatically from its arguments and the constraints on its input signal.	• Input signal • Through signal • Drop signal • Wavelength response
Gain/loss	Gain/loss sections allow the user to increase or decrease the power of a signal. The platform may not support gain or loss, in which case the operation will fail.	• Input signal • Output signal • Gain/loss
Modulator	A phase or amplitude modulator that uses an external electrical signal as the modulation source. The implementation is chosen by the support package based on the type of modulator.	• Input signal • Output signal • Modulation source • Modulation type
Detector	A detector that converts an optical signal to an electrical signal.	• Input signal • Output signal
Splitter	A splitter that splits an optical signal into multiple lower-power optical signals.	• Input signal • Output signals • Splitting ratios

Combiner	A combiner that combines multiple optical signals into a single optical signal. At the same time, maximising the total output power.	• ✓ Input signals • ✓ Output signal
Interferometers	A combiner that interferes multiple optical signals into a single optical signal. Does not perform any power optimisation.	• ✓ Input signals • ✓ Output signal
Switch	A switch that switches between two optical signals.	• ✓ Input signal • ✓ Output signal • ✓ Switch state
Source	A laser source that generates an optical signal.	• ✓ Output signal • ✓ Wavelength
Phase shifter	A phase shifter that shifts the phase of an optical signal. Optionally, performs the phase shift in reference to another signal.	• ✓ Input signal • ~ Reference signal • ✓ Output signal • ✓ Phase shift
Delay	A delay that delays an optical signal.	• ✓ Input signal • ✓ Output signal • ✓ Delay
Coupler	A coupler that couples two optical signals.	• ✓ 1st input signal • ✓ 2nd input signal • ✓ 1st output signal • ✓ 2nd output signal • ✓ Coupling factor
Sink	A perfect sink that consumes an optical signal with little to no return loss.	• ✓ input signal
Empty	A perfect empty signal source that produces an optical signal with little to no power.	• ✓ output signal

Table 5 Intrinsic operations in photonic processors, with their name, description and arguments. For each argument, an icon indicates whether the argument is required (✓) or optional (~). Additionally, the type of the value is also indicated by an icon; it can be optical ( ), electrical ( ), or a value ( ).

4.4 Constraints

Constraints are a technique for expressing requirements on values and signals. They are associated with each signal or value to give additional information regarding its contents. In Section 7, the concept of using constraints with *refinement types* will also be discussed as a potential future expansion of constraints. The core idea of constraints is that the user can use them to specify additional information about their signals at a given point in the code. Additionally, they can be used to check the code's validity and infer additional constraints. This is done by the *constraint solver*. This section will discuss the multiple aspects of constraints and their use.

> Constraints in themselves are not a new concept. However, how they are applied to include more complex constraints, simulate circuits, and inferrence, does *appear* to be novel.

Constraints for validation The primary use of constraints is for the validation of the code. The *constraint solver* does this, as discussed later. The constraint solver will use the constraints to check whether they are compatible with one another. This is done by annotating some functions with constraints and then checking whether the input signals are compatible with those constraints. If the constraints are not compatible, a warning or an error can be presented to the user.

Constraints can be of many types; the likely most common ones will be constraints on power, gain, wavelength, delay, and phase. The reason why delay and phase are different constraints is because they most often will have different semantics, where phase refers to the phase of the light within the waveguide, and the delay will mostly impact the delay of the modulated information on the signal. Since light operates at frequencies much higher than the RF range, one can consider the phase of the light to be mostly decoupled from the signal's phase. These constraints can be used to verify the code's validity and inform the compiler how to optimise and generate the design. For example, the user might have a high-power signal coming onto the chip that gets split. The place-and-route system can either place the splitter close to the input, increasing mesh usage but reducing non-linearities, or closer to the components using this light, increasing mesh usage but decreasing non-linearities. One can use the input power constraint to make a decision since at high-power, there will be increased non-linearities and losses within the waveguide. Therefore, the place-and-route can use this information to decide where to place the splitter. This is just one example of how constraints inform the compilation system and can be used to optimise the design.

Constraints for simulation Additionally to the aforementioned constraints, one can also express constraints that are useful for simulation, such as noise sources, modulation inputs, etc. These constraints do not make sense for the compilation process as they are not actionable at compile time; however, they are actionable for creating more realistic, closer-to-physical simulations. These special constraints can be used for a variety of things and are in essence non-synthesisable, whereas the other constraints are synthesisable.

These non-synthesisable constraints, can be coupled with synthesisable constraints to create a more realistic yet very inexpensive simulation. As will be discussed in Section 4.6, simulating circuits using constraints is extremely fast. And due to the integration of constraints within the language, as discussed in Section 5.2.r, makes them an inherent part of the user's design. This means that accurate, yet fast simulations are always available to the user.

Constraints for optimisation As previously mentioned, constraints can be used as indicators for stages within the synthesis process. Therefore, it stands to reason that constraints can be used to optimise the design. The compiler can use constraints to remove unnecessary components or to optimise the placement of said components. For example, a signal going through a filter might have a constraint on the wavelength, and the filter might also have a constraint on the wavelength. If the compiler can prove that the filter is not needed, it can remove the filter altogether. Alternatively, if it detects that there would be no signal left after the filter, it can remove the filter and all dependent components, simplifying the design. Additionally, the user might specify optimisation targets that the place-and-route may try to reach. These targets may relate to the mesh usage, non-linearities, or other metrics. The place-and-route can use these targets and constraints to optimise the design to the user's specifications.

Constraints as real-time feedback As discussed in Section 2.1.a.c, power detectors on the chip can be used for monitoring purposes. These detectors are expected to be implicitly and automatically used most of the time through intrinsic components and their platform-specific implementation. However, it can be interesting to give access to these monitors to the user. This can be done by using constraints on the power and gain. These constraints can be used to check whether constraints on power and gain are respected while the device is running, notifying the user if they are not. This allows the user to add detection of erroneous events, such as the loss of an input or failure to meet gain requirements. This can be used to notify the user's control software of the error so that they may react appropriately to it. Indeed, through the use of detectors, and especially implicit detectors, the user may gain insight into the state of the device.

Categories of constraints Using the aforementioned sections, one can categorise constraints into three distinct categories: synthesisable constraints, which are used for real-time feedback, simulation constraints, which are used for simulation, and meta constraints, which are only used by the compiler. These three categories are not mutually exclusive: most constraints can be used by the compiler and for simulation, but some of them are only used for these purposes. Therefore, one can see all constraints as being a hierarchy that can be seen in Figure 6. It shows that all constraints are simulation constraints, some of which are meta constraints, and some of those are also synthesisable constraints. In Table 6, the different types of constraints are listed along with their category and a short explanation.

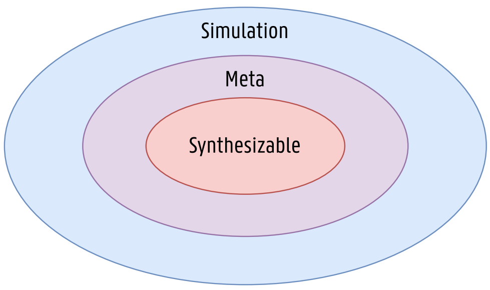


Figure 6 Hierarchy of constraints, showing that all constraints are simulation constraints, within that are meta constraints within which are synthesisable constraints.

	Constraint	Description
Synthesisable	Power	Power constraints are used to specify the power of a signal. At runtime, it can check whether signals are present and within certain power budgets by using detectors.
	Gain	Gain constraints are used to specify the gain created by a component. It can use detectors around a gain section to check whether a gain section is able to meet its parameters and to allow feedback control.
Meta	Wavelength	Wavelength constraints are used to specify the wavelength content of a signal. This is used for the optimisation of filters and other wavelength-dependent components.
	Delay	Delay constraints are used to specify the actual, minimum, or maximum delay of a signal. This can be used to meet delay requirements after place-and-route.
	Phase	Phase constraints are used to specify the differential phase of a signal. This can be used to ensure that phase-sensitive circuits are able to work as intended.
Simulation	Noise	Noise constraints are used to add noise to a signal. This can be used to simulate noise sources and to simulate the impact of noise on a device.
	Modulation	Modulation constraints are used to specify the modulation of a signal.

Table 6 | Different constraints on signals along with their category and a short explanation.

Constraints on values Expanding upon the concept of constraints further, adding constraints on values other than signals is possible. It can allow the user to set specific constraints, typically on numerical values, that can be used for two purposes. The first purpose is to allow the validation of values automatically without needing to write manual tests for values; this is often called a *precondition*. The second purpose, which is further explained in Section 4.5, is the ability to discover reconfigurable states that cannot be reached based on the constraints. This is an optimisation that can be done relatively easily by the compiler.

Constraint inference Constraints propagate through operations done in succession. Each intrinsic operation done on a signal adds its own constraints to the existing constraints. This allows the compiler to infer constraints on intermediary and output signals based on existing constraints and the constraints of the intrinsic operations. The compiler does this to allow the user to specify as few constraints as possible while still being able to infer the constraints on the signals. This feature is critical for the usability of the ecosystem, as it reduces the burden placed on the user of manually annotating their functions and signals with constraints. The constraints of entire functions can be computed and then summarised – i.e. simplified and grouped together – which simplifies the role of the simulator as it is simply using these simplified constraints and applying them to input spectrums and signals. This leads to a more efficient simulation which is much faster than traditional physical simulations. This is examined further in Section 5.7.

Constraint solver The solver is the tool that the compiler uses to summarise and check constraints. It will summarise the constraints on each signal such that it can easily be simulated, and it will verify that constraints are compatible. Additionally, in cases where the constraints depend on tunable values – i.e. values that can be changed at runtime – the solver can use a prover and the constraints on the tunable value to determine whether the constraints are compatible. This is done by using a prover such as *Z3* [85]. However, this is a very computationally expensive process and must therefore, only be performed when necessary. The compiler will only use the prover when the constraints depend on tunable values. Therefore, one can see the constraint solver as a tool composed of two subsystems, the first one, computing and verifying constraints based on known data, it is simpler and faster, and the second one, computing and verifying constraints based on tunable values, it is more complex, relying instead on a prover.

Limitations of constraints There are, however limitations of the constraint system, most notably that, using the aforementioned constraint solver, constraints are limited to exclusively feedforward system. This is due to constraints being calculated one after the other, not allowing cyclic dependencies. And as discussed in Section 2.2.a, one can represent any recirculating circuit as a feedforward system. However, there is one necessary condition for this to hold: it *must* be at a higher level of abstraction. When building the higher-level abstractions, this axiom cannot be assumed to be true, as the user is writing them at a lower level of abstraction. Therefore, the constraint system is limited in such cases, and the system must provide an "escape hatch" which allows the user to manually specify constraints at the edges of their abstractions, such that the compiler can use them outside of the abstraction while pausing constraint computation inside. When using this feature, the user can now express recirculating circuits easily by using the escape hatch to specify constraints on insufficient feedforward signals.

> Constraints can be used to validate signals, eliminate branches, and simulate the design. They are implemented using the constraint solver, combining constraints or using the *Z3* prover to verify constraints [85]. However, they are limited to feedforward systems, and therefore an escape-hatch is needed to specify constraints on recirculating circuits.

4.5 Tunability & Reconfigurability

> **Definition**: **Tunability** is the ability to change the value of a parameter at runtime to impact the behaviour of the programmed device.

Tunable values are values that can be interacted with, by the user, at runtime. They can be any non-signal value in the user's program, typically numerical values, that the user defines as being tunable values. These values can be seen as tuning knobs that the user can access at runtime to change their circuit's behaviour and implement their custom feedback loops. Tunable values can impact several parts of the design at once. For example, a single value may determine the centre frequency of operation of a bank of filters, all of them being tuned at once. This makes tunable values especially powerful as their impact can be propagated through the entire design.

The core idea behind tunable values is that the user can now represent parts of the parameters of their design as runtime values that can be interacted with while keeping all of the derived values within the circuit code itself. This design aims to make hardware-software codesign easier and more productive. Instead of having the complex relationships between parameters expressed within the "software" part of hardware-software codesign, they can be directly expressed on what they impact: the "hardware" part. This makes the design process more intuitive and removes the potential for discrepancies between the hardware and the software.

Additionally, the user should be able to name and access their tunable values by name within their code. This further improves the usability of the system, as it removes the need to maintain complex, error-prone tables of registers and their corresponding values. Instead, the user can address their tunable value naturally through its name, and the HAL can take care of the rest: translating these names and values into an appropriate set of registers and values.

This means that the physical parameters of each element can be represented as natural parameters – i.e. numerical values – while the underlying hardware uses lower-level likely binary values and flags. This also improves the development experience of the device provider, as they can now integrate the code required to do data conversion within their platform-support package, further simplifying the development of new support packages.

Furthermore, the use of constraints on values, such as explained in Section 4.4, can be used by the compiler to further detect the need for reconfigurability automatically without additional user input. It can also be used to validate that when a tunable value is changed in the user's software, it meets its requirements, ensuring that the user cannot change the value to an invalid one, where the device might then operate in an undefined state.

DEFINITION: **Reconfigurability** is the ability to change the topology of the device at runtime and, therefore its behaviour.

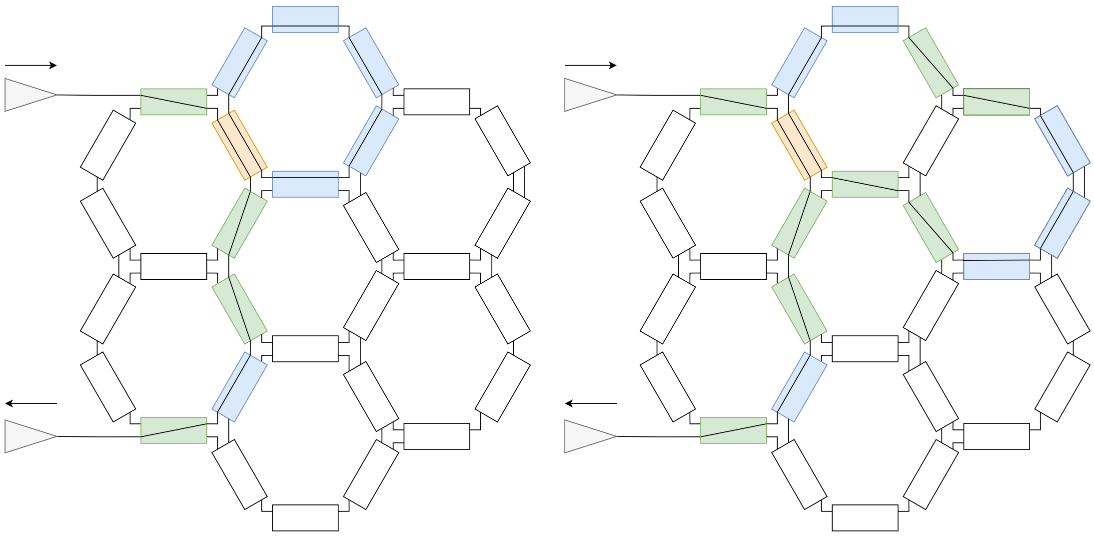
Additionally, one of the most important functional requirements is the reconfigurability. Its goal is to allow the user to reconfigure the mesh – or only parts of it – while the device is running. This can be achieved in several ways, but the most natural way is to use branches within the code to determine the boundaries between reconfigurability regions. Then, through the use of tunable values, these regions can be automatically selected based on their value.

However, this brings a set of difficult problems to solve, the first of which is the ability to determine whether a state is even reachable. However, this can be done using constraints; through the constraint solver for tunable constraints, one can verify which states are reachable and discard those unreachable. This is a powerful optimisation, as it greatly decreases the amount of states that need to be place-and-route, and, therefore, the amount of time needed to compile the mesh.

Indeed, considering the following example, the user instantiates a mesh containing 64 input signals and 64 output signals. Each input signal can go into one of two filters based on branching. This, therefore, means that there are 2^{64} possible states. It can easily be understood that this is an intractable problem, as synthesising the project would be almost impossible. However, if the system can determine that only two states are reachable for each signal, this comes down to 128 states, which is much more tractable. Therefore, one can see that there is an interest in reducing the number of states that need to be synthesised. One such way is by using the constraint solver to eliminate unreachable states. The second way is by finding subsets of the overall circuits that are independent of one another and, therefore, can be mostly synthesised in isolation.

Neither of those two tasks is trivial, they can be seen as similar to the halting problem, which is undecidable, and therefore, it is desirable to let the user specify some of the state reduction manually, letting the user take care of parts of the more complex cases. One can draw a parallel between this and the use of the escape-hatch for constraints, as it is a similar concept, where the user can specify constraints manually at the edge of abstraction, while here, the user can specify how to reduce the number of states manually. However, limiting the maximum number of iterations, the recursion depth, or both can make the halting problem decidable. However, despite being decidable, it still incurs a high computational cost. Therefore, one can expect that state reduction will also be computationally expensive.

In Figure 7, one may see what reconfigurability might look like on a fictitious device, where based on an input variable, a simple boolean in this case, the device will use either of two meshes. Each state (a) and (b) represents a different mesh that implements a different filter. In this example, the user would have created a tunable boolean that they can set at runtime, and based on its value, the appropriate mesh will be selected.



(a) (b)

FIGURE 7 Example of reconfigurability on a fictitious device. Each state (a) and (b) represent a different filter. The second (b) filter has a longer ring and a higher FSR (*Free Spectral Range*) than the first one (a). Squares of different colours represent photonic gates in different states: blue represents through gates, green represents cross gates, and yellow represents partial gates. The grey triangles represent optical ports.

Reconfigurability allows the user to create modular designs, where, at runtime, the user can select a different state to fit their needs. Reconfigurability is achieved through the branching of the code. The user can specify tunable values that are used to select the appropriate branch. The number of states is exponential but can be decreased using the constraint solver to remove unnecessary branches by finding independent mesh subsets and letting the user manually specify some of the state reduction.

4.6 Simulation

As previously discussed, the user must also be able to simulate their circuit. The traditional physical simulation approach is slow; therefore, finding solutions to make simulations faster may be desirable. As was discussed in Section 1.1, there is ongoing research in using *SPICE* to simulate photonic circuits. Additionally, some of this work is being conducted at the PRG . One of the main advantages of this solution, as opposed to the one presented below, is that it allows for recirculating meshes. However, it is not as fast as the solution presented in this document, and therefore, it is not as well suited for the use case of this project.

Additionally, the *SPICE* -based simulations may be able to incorporate more effects, such as the effects of the non-linearity of components, which may lead to more accurate simulations. Despite this, the user may not want a physically correct simulation. Instead, they may want a fast simulation representative of their circuit without all the physical hardware limitations. In essence, this is similar to simulations for electronic HDL development, where the simulations are not physical yet are still representative.

The simulation scheme that is suggested in this research is to use constraints to simulate the circuit. The idea is that the constraint solver can be used to summarise the constraints on each net. It can then be used to calculate analytically the value of each net and, therefore, the value of each signal. The main difference with other approaches is that this can be done very quickly with relatively simple code due to the relative simplicity of constraints. This simplicity improves the simulation's performance, as discussed in Section 5.7, but also decreases the work required to maintain and update this simulator as time goes on.

In practice, simulations would be separated into two categories: time domain simulation, which takes one or more signals modulated onto carrier optical signals and simulates their processing and a frequency domain simulation which looks at the frequency and phase response of the device. The reasoning behind this separation is as follows: due to the extremely high frequency of light, accurately representing light in the time domain is extremely difficult, as it requires very small time steps. Instead, if using the frequency domain, one can decouple the modulated signals by using the spectral envelope of the modulated signal as the input to the simulation. This, therefore, allows for easy analysis of the spectral performance without the computation cost of small timesteps. Then, in the time domain, the user specifies sets of wavelengths which are then modulated with the signal of interest, which can be passed through the device. This allows time domain simulation to use much bigger time steps on the modulating signal's period rather than timesteps on the light's period.

However, this introduces a limitation; due to this dichotomy, the user must simulate both effects separately and analyse the results. While this makes the simulation process more limited, it also makes it more flexible, as if the user only needs one of the simulation kinds, they can avoid needing to simulate the other, decreasing computation time further.

Simulation ecosystem There exist many tools for the simulation of photonic circuits. Additionally, there also exist a lot of tools for the kind of resolution that is being done. It is, therefore, of interest to reuse as many existing tools as possible. As long as these tools are free, they do not incur a cost on the user's end. Additionally, by reusing existing tools, the user can benefit the ecosystem surrounding these solutions and the community using them. Furthermore, it also simplifies the simulation ecosystem's development, as it no longer requires writing the entire simulation ecosystem from scratch. Instead, reusing existing tools and using the best-in-class tools for each task.

4.7 Platform independence

As the development of photonic processors continues, new devices must be expected to bring new features, different hardware programming interfaces, and characteristics. Ideally, all of the code would be backward and forward compatible, being able to be programmed on an older or a newer device with little to no adjustments. Therefore, one must plan for platform support right at the core of the design of a photonic processor ecosystem. In this document, several approaches will be suggested for tackling this issue. These approaches are meant to be used in conjunction with each other.

Standard library All platforms must share a common standard library containing base building blocks and more advanced synthesis tools – i.e., filter synthesis – that is common across all devices. This library must be able to be used by all devices. Therefore, it must be able to be compiled into the intrinsic operations mentioned in Section 4.3. Additionally, providing common building blocks and abstraction makes developing circuits targetting photonic processors easier. This is similar to the standard library for regular software development, where the language provides a set of functionalities out-of-the-box that the user can use.

Platform support packages Each platform must come with a platform-support package that implements several tools: a hardware programmer for programming the circuit onto the device; a compatibility layer for the standard library such that the standard library is compatible with the hardware; some device-specific libraries for additional features if needed; a place-and-route implementation, it may be shared across many devices, but the support package must at least list compatible place-and-route implementations. With these components, the user's circuit should be able to be compiled while using the standard library and then programmed onto the device for a working circuit.

Hardware abstraction layer Each platform must come with a HAL which allows the user to interact with the device programmatically at runtime. This HAL must provide features for communicating, setting tunable components and reading the state of the device. The HAL can be reused across devices as long as the devices have similar hardware interfaces. In Section 5.6.c, this will be further discussed, including how parts of the HAL can be generated based on the user's design for improved usability and easier hardware-software codesign.

Constraint packages It must also come with information regarding delays and phase response of its different components, as well as the capabilities of some of its components like amplifiers, modulators, etc. This information can be used by the constraint solver and the simulation ecosystem to more accurately represent the capabilities of the circuit and allow the user to make informed decisions. Additionally, a platform may come with additional simulation-specific constraints for more accurate simulations and the additional information provided by the constraint packages.

4.8 Visualisation

Several types of simulations may be useful for the user: the user might want to visualise the generated circuit mesh superimposed onto a schematic representation of the device to verify that no critical components were removed through constraints, to see the usage at a glance, or to visualise whether the place-and-route performed adequately. This visualisation is already presented in the existing library and exists in EDA (*Electronic Design Automation*) tools for photonics. Therefore, such visualisation facilities must be offered to the user. Mainly due to the fairly early research stage, the ability to communicate results visually is critical to the user's understanding of the results. Another kind of visualisation the user will want is the simulation results. Therefore, the ecosystem must provide easy visualisation of results.

Applying DRY As is the case of the simulation ecosystem, one can reuse existing tools and libraries for visualisation that are already on the market. This applies the DRY (*Don't Repeat Yourself*) principles, where one can reuse existing tools and libraries rather than rewriting them from scratch. This also allows the user to benefit from the large ecosystem of existing visualisation tools and to use the tools they are most familiar with or that give the best results for their application. Examples of such visualisations can be seen in Figure 7, which shows the mesh and the state of each gate, and a simulation result in Figure 8, which shows the results of a time-domain simulation using the aforementioned constraint solver.

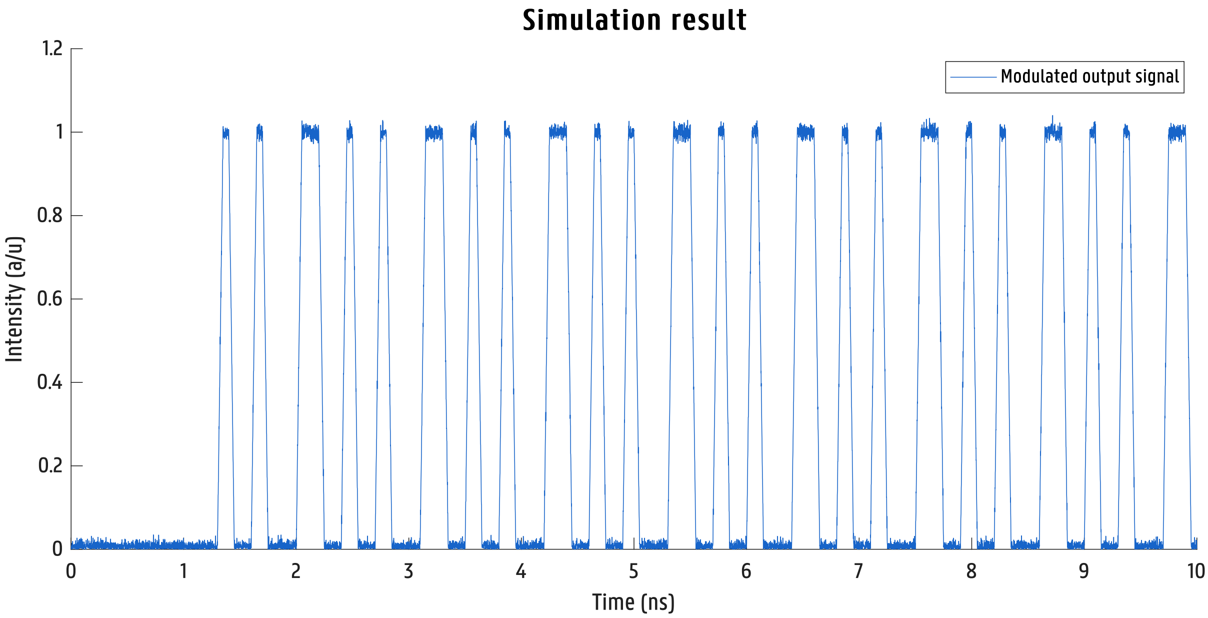


Figure 8 Example visualisation of a time-domain simulation result, showing a $10\ \mathrm{Gb/s}$ modulated PRBS (*Pseudo Random Binary Sequence*) 15 sequence on top of a $1550\ \mathrm{nm}$ carrier. The simulation was performed using the constraint solver. Shown is a $10\ \mathrm{ns}$ window of the simulation. The simulation was run for a total of $1\ \mathrm{\mu s}$ with an average execution time of $9\ \mathrm{ms}$. The simulation simulates a laser source with noise and the rise and fall time of the modulated signal, the rise and fall time being $50\ \mathrm{ps}$.

4.9 Calibration and variation mitigations

Photonic circuits can be susceptible to manufacturing variations and temperature variations. Therefore, each device must come with mitigation techniques that can aid in making the device behave as ideally as possible. This is expected to generally be done using calibration curves or calibration LUTs . And by using feedback loops to ensure that a component behaves as expected. For example, a power combiner might maximise power output using a feedback loop on a phase shifter to create constructive interference.

Feedback loops Feedback loops are essential to overcoming variations, especially those caused by temperature variations. A feedback loop can read a power monitor on the chip and adjust the tunable value of another element. Feedback loops can be built-in, as in added automatically by the compiler for specific tasks, based on the device support package and the intrinsics being used. Or they can be created manually by the user, in which case they must write code that, using the HAL , reads the sensors and then writes to whichever tunable value they need.

Wavelength dependence Additionally, to manufacturing variability and temperature dependence, the device's response is also wavelength dependent. This is due to the physical properties of the materials from which the device is made of. Hence there are no easy ways of mitigating these effects. However, by using constraints, the compiler can know which wavelengths are expected in which component, and similarly to using calibration curves for device-to-device variability, it can use similar response curves to adjust the circuit to the expected wavelength. This is expected to be done automatically by the compiler, but it requires that the user specifies wavelength constraints.

4.10 Resource management

Another aspect of designing circuits for programmable devices, whether they be traditional processors, FPGAs or photonic processors, is resource management. A limited number of elements are built into the hardware, and the user must be able to use these elements as efficiently as possible. This may be especially true for photonic processors where the number of gates is currently relatively small. Below is Table 7, which lists potential resources that may be present on the device. These resources are obtained from the description of intrinsic values in Section 4.3 and the components of a photonic processor detailed in Section 2.1.

Resource	Description
Photonic gate	The photonic gate is the core element of the photonic processor; it can be arranged in a grid, whether square, triangular or hexagonal. It generally contains a 2-by-2 tunable coupler and power detectors for monitoring. It is used to process the light and to route the light around the chip.
High-speed detector	High-speed detectors are used to demodulate the light; they can operate either in an amplitude demodulation scheme or be used with interference to perform phase demodulation.
High-speed modulator	High-speed modulators are used to modulate the light; they can operate either in a phase modulation scheme or be used with a MZI to perform amplitude modulation.
Laser source	Laser sources are used to generate light at a given wavelength directly inside the device. Due to the devices being made in silicon, there are none on existing prototypes. However, they may be added using epitaxial growth or micro-transfer printing.
Gain section	Gain sections are used to amplify the light, generally made of a semiconductor optical amplifier or an erbium-doped waveguide section. As with laser sources, there are currently no gain sections on prototypes.
Optical port	These ports at the edge of the device can be used to couple light in and out of the device.
Switch	Switches can be either implemented using the mesh and couplers, using a power splitter with its coupling coefficient controlled by a tunable value, or built into the device as dedicated hardware. There are no dedicated hardware switches, but they may be added in future devices.

Table 7 | List of device resources and their description.

4.11 Responsibilities and duties

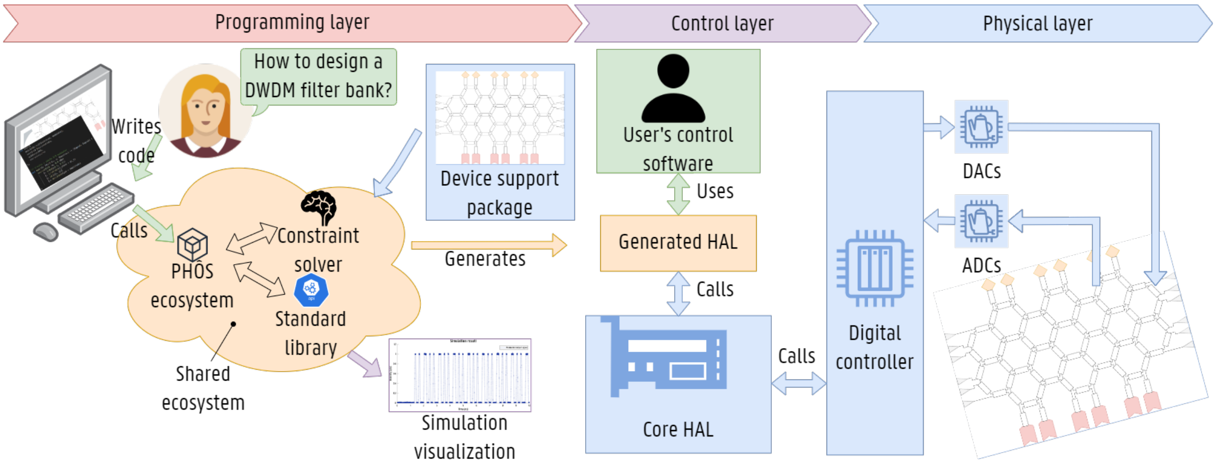


Figure 9 The responsibilities of each actor in the ecosystem, elements in orange, are the responsibility of the ecosystem developer. It includes the compiler, constraint solver and standard library. It also contains parts of the HAL generator. The elements in blue are the responsibility of the chip designer. It includes the device itself, the core HAL , and the device support package. Elements in green are the responsibility of the user. This includes the user's design and the user's control software. It also shows the different components of the ecosystem that have been discussed so far and their overall interaction with one another.

As with most ecosystems, the responsibilities for the development of different parts and the duties of maintaining these parts are split between different actors. In this case, one can see the ecosystem being designed in this thesis as having four actors: the user who is responsible for the design of their circuit, their own control software, the maintenance of their code, and the compatibility of this code with the ecosystem. The second actor is the developer of the ecosystem itself, their responsibility is spread among several tasks, from the programming ecosystem components discussed in Section 3, to the standard library, and the constraint solver. Due to the critical importance of these tools, the duties of maintaining some degree of backward and forward compatibility along with making sure that the tools are as bug-free as possible falls on the ecosystem developer. The third actor is the chip provider, they design the actual physical layer: the photonic processor. Because of this, they must also produce the device support package and the core HAL . Their responsibilities are to ensure that their device is compatible with the common parts of the ecosystem, that their devices can work in expected use scenarios, and to provide the HAL generator. The fourth and final actor are all of the external tool provider, those can be libraries developers, EDA tool developers, etc. Most of the time, their projects' licenses will remove any and all responsibilities from their user. Therefore, special care must be taken when integrating external tools and libraries so that trustworthy actors maintain them. A summary of these responsibilities and their interactions with one another can be seen in Figure 9.

5.

The PHÔS programming language

Based on all of the information that has been presented so far regarding the translation of intent and requirements (Section 4), programming paradigms (Section 3.6), and the inadequacies of existing languages (Section 3.9), it is now apparent that it may be interesting to create a new language which would benefit from dedicated semantics, syntax, and integrate elements from fitting programming paradigms for the programming of photonic processors. This language should be designed in such a way that it is able to easily and clearly express the intent of the circuit designer while also being able to translate this code into a programmable format for the hardware. Additionally, this language should be similar enough to languages common within the scientific community, making it easy for engineers to learn and leverage existing tools. Finally, this language should be created in such a way that it provides both the level of control needed for circuit design and the level of abstraction needed to clearly express complex ideas. Indeed, the language that is presented in this thesis, *PHÔS* , is designed to fulfil these goals.

The following sections will discuss the initial specification, syntax, constraint system, and other various elements of the language. Then, in Section 6, examples will be shown of how the language can be used to express various circuits. However, before discussing the language in itself, it is important to discuss the design of the language, the existing languages it draws inspiration from, and the lessons it incorporates from them.

5.1 Design

> The name of the language, *PHÔS* , is a reference to the ancient Greek word for light or daylight, φῶς (phôs).

Inspiration *PHÔS* primarily takes inspiration from *Python* and *Rust* for its syntax while incorporating elements from functional languages such as *Elixir*[2]. Its semantics, especially as they relate to signals, are inspired by traditional hardware description languages, most notably *SystemVerilog* and VHDL . Other semantics, as they relate to values, are inspired by *Rust*. Other elements, such as comments, are inspired by the *C* family of languages, while documentation comments are inspired by *Rust* as well.

Synthesisable *PHÔS* separates regular functions from synthesisable blocks. Whereas synthesisable blocks are able to interact with signals, regular functions are forbidden from operating on signals. This is done to ensure that branching reconfigurability computations are only done on synthesisable blocks. Ideally, synthesisable blocks would be kept as short as possible, while functions can be much longer.

Paradigm *PHÔS* is an imperative language with many functional elements, it purposefully keeps the easier aspects of imperative programming, while incorporating functional elements to make it both easier to reason about and easier to

[2] *Elixir* has not been discussed in this thesis, but it provides the piping operator who is incorporated into *PHÔS* .

synthesise into hardware. The elements from functional programming relate especially to dataflow programming, as it has the closest semantics to light flowing from one component to another. The language is purposefully kept simple, with only a few elements from each paradigm, to ensure that it is easy to learn and easy to use. Form follows function, and the language is designed to be used by engineers and researchers, not by computer scientists.

Constraints *PHÔS* can express constraints directly in its syntax; this is opposed to other languages, such as *VHDL* and *SystemVerilog*, which require timing constraints to be specified externally. This is done to ensure that the constraints are always in sync with the code and are always available at a glance. Additionally, *PHÔS* uses a constraint system that is compatible with both signals and regular values. This is done for several purposes, chief among them is to ensure that reconfigurability regions can be minimised, as discussed in Section 5.2.k.

5.2 PHÔS: an initial specification

This section serves as an initial specification or reference to the *PHÔS* programming language. It contains the elements and semantics that have already been well-defined and are unlikely to change. Therefore, this section is not a complete specification as that would require that the language be more mature. However, it serves as an in-depth introduction to the concepts, paradigms, and semantics of the language. Most parts of the specification are companied by a short example to illustrate the syntax and the semantics of the language. Additionally, some elements are further explored in subsequent section of this chapter, with only the basics being presented here.

5.2.a Execution model

PHÔS is a photonic hardware description language, due to its unique requirements, it is not designed in a traditional way and instead separates the compilation to hardware into three distinct steps: compilation, evaluation, and synthesis. The compilation step is responsible for taking the source code, written in human-readable text, and turning it into an executable form called the bytecode, see Section 5.4.g. Followed by the evaluation, the evaluation interprets the bytecode, performs several tasks discussed in Section 5.5, and produces a tree of intrinsic operations, constraints and collected stacks. This tree is then synthesised into the output artefacts of the language, namely, the user HAL and a programming file for programming the photonic processor. The execution model is shown graphically in Figure 10, showing all of the major components of the language and how they interact with each other. Further on, more details will be added as more components are discussed.

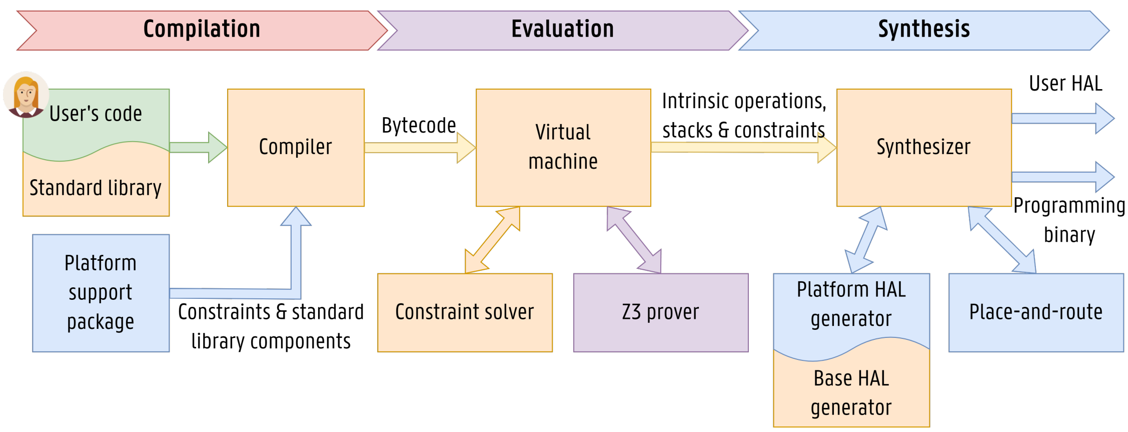


Figure 10 The execution model of the *PHÔS* programming language, showing the three distinct stages. Responsibilities use the same color code as Figure 9, showing the ecosystem components in orange, the user's code in green, the platform-specific code in blue, and the third-party code in purple.

Function execution model The execution model of functions in the *PHÔS* programming language is similar to that of *Java* and *C*, every statement is terminated by a semicolon (`;`), with the exception of automatic return statements. *PHÔS* is single threaded and exclusively execute statement in the order that they are written in. Branching causes the VM (*Virtual Machine*) to jump from one place in the code to another. Function calls are executed by jumping to the entry point of the callee function, executing it in sequence, then returning to the next statement in the caller. Statements are therefore indivisible units of work. Additionally, *PHÔS* has order of precedence for its operators, therefore the order of execution of operators is not necessarily the order in which they are written. However, the order of statements is always the order in which they are written. Statements being composed of operators and operands, the order of execution of operators is determined by their precedence, with the highest precedence being executed first. The precedence of operators is shown in Figure 11.

Precedence	Operator	Associativity
1	Conditional statements, loops, parenthesized expressions, unconstrained block, empty expressions	Left
2	Function calls	Left
3	Array indexing	Left
4	Field access	Left
5	Type casting	Left
6	Raising to a power	Right
7	Unary operators: negation, bitwise complement, logical complement	Right
8	Multiplication, division, remainder	Left
9	Addition, subtraction	Left
10	Bitwise shift left, bitwise shift right	Left
11	Bitwise and	Left
12	Bitwise xor	Left
13	Bitwise or	Left
14	Equality operators: equal, not equal	Left
15	Relational operators: less than, less than or equal, greater than, greater than or equal	Left
16	Logical and	Left
17	Logical or	Left
18	Pipe operator	Right
19	Range operators: inclusive range, exclusive range	Neither
20	Assignment operators: assign, add assign, subtract assign, multiply assign, divide assign, remainder assign, bitwise and assign, bitwise or assign, bitwise xor assign, bitwise shift left assign, bitwise shift right assign	Neither
21	Closures	Neither
22	Control flow operators: break, continue, return, and yield	Neither

Figure 11 Operator precedence in *PHÔS* , from highest to lowest. Operators with the same precedence are executed from left to right. The precedence of operators is used to determine the order of execution of operators in a statement. The associativity of the operator is also shown, it can be left-associative, right-associative or neither for special operators that only have one operand.

Synthesisable execution model Synthesisable blocks follow a different execution model due to reconfigurability: when a block of code cannot be evaluated, it is analysed to remove as much code as possible before being collected into a stack. The stack is then stored along with the intrinsic subtree of that reconfigurability region. Additionally, synthesisable block produce intrinsic operations and constraints that are stored in a global tree of operations in order to produce the expected output.

5.2.b Typing system

PHÔS uses a typing system similar to *C*, it does not support object oriented programming. It has a basic type system as the complexity of actual code is expected to be relatively low. This limitation is put in place such that the language is easier to use with constraints, and especially with provers for reconfigurability. This means that the typing system is generally simple to use and understand. Overall, *PHÔS* ' typing system is static and strong. Meaning that all values have a fixed type at compile time, and that implicit type conversions do not occur. This aims at reducing the potential for errors, as well as improving the clarity of APIs and IPs designed with *PHÔS* .

Static type checking and inference *PHÔS* is statically typed, but offers type inference as a means of reducing the annotation burden on the user. Essentially, type inference tries to figure out what the type of a value is based on the operations being performed and the type of the values it is produced from. In the case of *PHÔS* , this will be performed using the *Hindley-Milner* algorithm, which is a de-facto standard in modern programming languages [26, 27]. Additionally, due to the simple type system employed by *PHÔS* , it should be generally easier for the compiler to infer the types of values as conflicts are less likely to occur.

Constraints and typing In this initial design of *PHÔS* , constraints are seen as metadata on values and signals. This is a simpler approach that decreases the initial development complexity. However, it is limiting and makes design error only visible during the evaluation phase of the compiler. For this reason, it could be improved using the concept of *refinement types* [86], which would allow the compiler to check for errors in constraints earlier, this is further discussed in Section 7.

5.2.c Mutability, memory management and purity

PHÔS does only allow mutating of state, following the functional approach of limiting side effects [87]. This makes the work of the prover used for reconfigurability easier. It also ensures that all functions are pure functions, something that can be exploited by the compiler to optimise the code, as well as be used in future iterations of the design to provide features such as memoization for faster compile time. This also means that *PHÔS* does not have a garbage collector, as it does not need to manage memory. As the lifetime of all values is predictable, *PHÔS* can simply discard values when they fall out of scope. This is done by the compiler, and does not require any action from the user. This is a deliberate choice to reduce the complexity of the compiler.

Limitations of immutability and purity Immutability make global state management difficult, requiring the user of a state monad to manage state [88]. However, *PHÔS* does not need global mutable state, as it is not a general-purpose programming language. Indeed, due to the hardware-software codesign nature of *PHÔS* , it is expected that the user will manage global state within their own software layer, leaving their *PHÔS* codebase free of global state. Additionally, purity disallows the user of side effects, which removes the user's ability to perform I/O (*Input/Output*) operations, such as reading from a file, making the language inherently safer to use. However, this also means that *PHÔS* cannot be used for general-purpose programming, and is limited to the domain of photonic circuit design.

5.2.d Signal types and semantics

PHÔS distinguishes between the `electrical` and `optical` types. They mostly share the same semantics, with the exception that electrical signals cannot be operated upon. In the following paragraphs, the semantics of each will be discussed with examples.

Optical Optical signals follow a *drive-once*, *read-once* semantics, meaning that they must be produced by a source element and must be consumed by a sink element, signals cannot be empty or be discarded without being used. The goal of this semantic is to make signals less error prone by avoiding the possibility of signals being left unconnected. Additionally, the *drive-once* semantics ensure that signals are split explicitly by the user and not implicitly with difficult to predict results. Consider the example in Listing 10, depending on the compiler's implementation, it may lead to two splitting schemes, both shown in Figure 12, it shows that with an automatic scheme, based on the compiler architecture, it may lead to two different results. However, with an explicit scheme, the user is able to clearly define the splitting scheme, and the compiler no longer has to make any assumptions about the splitting scheme. Additionally, optical signals only support being passed into, used, or sourced inside of synthesisable blocks. This restriction aims at making the work of the compiler easier and creating a stronger distinction for users.

PHÔS
```
let a = source(1550 nm, -10 dBm)

let b = a |> gain(5 dB)
let c = a |> filter(center: 1550nm, bandwidth: 10 GHz)
let d = a |> modulate(external_signal, type_: Modulation::Amplitude)
```

Listing 10 Optical signal splitting example

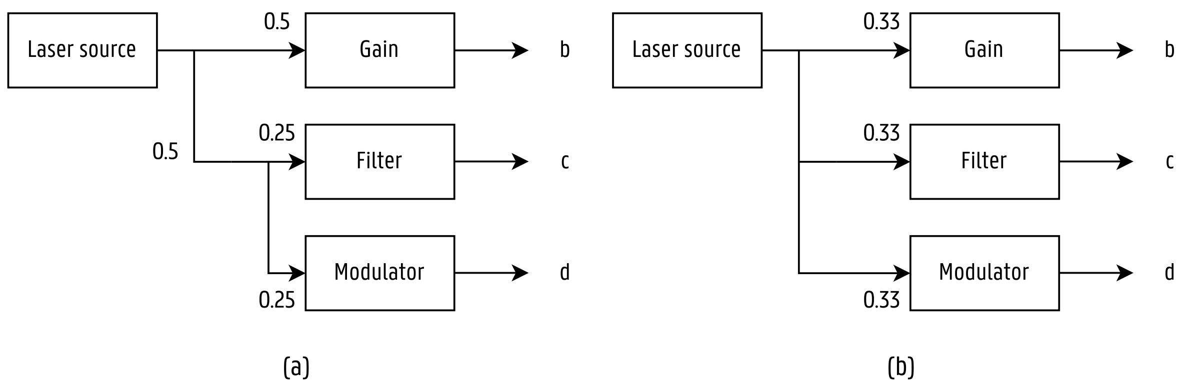


Figure 12 Automatic optical signal splitting schemes, showing the two possible automatic splitting scheme, using either a cascade architecture (a), or a parallel architecture (b). It leads to different power ratios for all of the downstream elements.

Electrical Electrical signals do not allow any operations on them apart from being used in `modulate` and `demodulate` intrinsic operators. The reasoning behind this limitation is that, at present, there are no plans for analog processing in the electrical domain. And this feature is not yet implemented. Therefore, electrical signals are only ever used to modulate optical signals, or are produced as the result of demodulating optical signals. It is possible that, in the future, some analog processing features may be added, such as gain, splitting, etc., but as it is currently not planned, electrical signals are not allowed to be used in any other way. Electrical signals follow the same semantics as optical signals: *drive-once*, *read-once*.

5.2.e Primitive types and primitive values

PHÔS aims at providing primitive types that are useful for the domain of optical signal processing. As such, it provides a limited set of primitive types, not all of which are synthesisable. To understand how primitive types are synthesisable, see Section 5.2.t. In Table 8, the primitive types are listed, along with a short description. Primitive types are all denoted by their lowercase identifiers. This is a convention to make a distinction between composite types and primitive types. These primitive types are very similar to those found in other high-level programming languages such as *Python* [57].

Primitive type	Description	Minimum value	Maximum value
`optical`	An optical signal, with *drive-once*, *read-once* semantics.	-	-
`electrical`	An electrical signal, with *drive-once*, *read-once* semantics.	-	-
`any`	Any value, used for generic functions.	-	-
`empty`	An empty value, equivalent to `null` is many other languages.	-	-
`string`	A character list, encoded at UTF-8.	-	-
`char`	A unicode scalar value [89].	-	-
`bool`	A boolean value, taking either `true` or `false` as a value.	-	-
`int`	A signed integer value, 64 bit wide.	-2^{63}	$2^{63}-1$
`uint`	A unsigned integer value, 64 bit wide.	0	$2^{64}-1$
`float`	A floating point number, 64 bit wide. Represents a number in the IEEE 754 format [90].	$-1.798 \cdot 10^{308}$	$1.798 \cdot 10^{308}$
`complex`	A complex floating point number, 128 bit wide. It consists of a real and imaginary part, both a floating point number.	-	-

Table 8 | Primitive types in *PHÔS* , showing the primitive types and their description. For numeric types, the minimum and maximum values are shown.

5.2.f Composite types, algebraic types, and aliases

Definition: **ADT** (*Algebraic Data Type*) is a type composed of other types, there exists two categories of ADT : **sum types** and **product types**. Product types are commonly tuples and structures. Sum types are usually enums, also referred to as **tagged unions**.

Adapted from [91]

PHÔS has the ability of expressing ADT in the forms of enums, enums are enumeration of n variants, each variant can be one of three types: a unit variant, that does not contain any other data, a tuple variant, that contains m values of different types, or a struct variant that also contains m values of different types, but supports named fields. Enums are defined using the `enum` keyword followed by an identifier and the list of variants. In Listing 11, one can see an example of an enum definition, showing the syntax for the creation of such an enum. Enums are a sum type, as they are a collection of variants, each variant being a product type. Enums are a very powerful tool for expressing ADT , and are used extensively in *PHÔS* and languages that support sum types.

```
enum EnumName {
    A,
    B(int, string),
    C {
        first_value: int,
        second_value: AnotherType
    }
}
```

Listing 11 Example in *PHÔS* of an ADT type, showing all three variant kinds: `A` a unit variant, `B` a tuple variant, and `C` a struct variant.

> **Definition**: **Composite types** are types that are composed of other types, whether they be primitive types or other composite types. They are also called **aggregate types** or **structured types**. They are a subset of ADT .
>
> Adapted from [92]

Additionally, *PHÔS* also supports product types and more generally composite types. Composite types are any type that is made of one or more other type. They can be one of five types: a unit structure, a tuple structure, a record structure with fields, a tuple, and an array of n items of the same type. The syntax of these five types can be seen in Listing 12. This variety in typing allows for precise control of values and their representation. It allows the user to choose the best type for their current situation, such as anonymous tuples for temporary values or named records for more complex structures.

```
/// A unit struct
struct A;

/// A tuple struct
struct A(uint, string, B);

/// A record struct
struct B {
    name: string,
    complex_signal: complex
}

/// an enum type is not named and can be declared inline.
(uint, string, A)

/// Arrays are defined with a type `A` and a size `10`.
[A; 10]
```

Listing 12 Example in *PHÔS* of composite types, showing all five kinds: `A` a unit structure, `B` a tuple structure, `C` a record structure, `D` a tuple, and arrays.

PHÔS also supports type aliases, which is the action of locally renaming a type. This can be used to indicate in code the semantic behind the value: instead of being a numeric value, it can now be expressed as a `Voltage`, etc. This is done by using the `type` keyword, followed by the new name and the type alias; this is shown in Listing 13.

```
type Voltage = int;
```

PHÔS

Listing 13 Example in *PHÔS* of type aliases, showing the creation of a `Voltage` type alias for `int`.

5.2.g Automatic return values

PHÔS support automatic return values, when the compiler sees that the final statement in a block is an expression that is not terminated by a semicolon, it will automatically return the value of that expression. This makes code more concise and readable, as shown in Listing 14, in (a) the example is shown with automatic return, and in (b) the example is shown with the explicit return. Additionally, it allows code blocks to be used as expressions, which is generally a useful feature.

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

PHÔS

(a)

```
fn add(a: int, b: int) -> int {
    return a + b;
}
```

PHÔS

(b)

Listing 14 Example in *PHÔS* of automatic return values, showing the difference between automatic return (a) and explicit return (b).

5.2.h Units, prefixes, and unit semantics

One of the unique features of *PHÔS* is the built-in SI (*Système international – the international system of units*) unit system. It is comprised of all of the SI units, with the exception of the candela, and some of the compound units, the list of units are: seconds, amperes, volts, meters, watts, hertz, joules, ohms, henries, farads, coulombs, and siemens. This set of units comprises more units than is typically used in photonics, and this is done such that more circuits may be designed using the language in the future, as discussed in Section 7.6. Additionally to SI units, SI prefixes are also supported from 10^{-18} to 10^{12}. Support for decibels is also included; the following decibel units are supported: relative (dB), relative to the carrier (dBc), relative to a milliwatt (dBm), and relative to a watt (dBW). Finally, due to the prevalence of angles in photonics for phase controls, radians and degrees are also natively supported. It is also important to note that this list can very easily be expanded to include more units, prefixes, as the language is designed to be extensible. In Listing 15, one can see the syntax for the usage of some of the units, decibels and angles.

```
/// 1 milliwatt
1 mW
0 dBm

// 1 degree
1 deg
0.01745 rad
1°

// 1 kilohertz
1 kHz
1e3 Hz
```

Listing 15 Example in *PHÔS* of SI units.

Unit semantic Instead of implementing a complete unit conversion system, at present, *PHÔS* is not intended to support conversion of units of different types, meaning that power is always power and that multiplying a power with time is invalid. To do so would require a complete unit conversion system, which is not planned for *PHÔS* . Units are the same regardless of prefixes, which the compiler converts into the base, unprefixed unit before execution begins.

5.2.i Tuples, iterable types, and iterator semantics

Tuples are a kind of product type that links one or more values together within a nameless container. They are often used as output values for functions, as they allow for multiple values to be returned. In *PHÔS* , tuples have two different semantics: on one the one hand, they can be used as storages for values, as in most modern languages, but on the other hand, they can be used as iterable values, which is a feature that is not present in many languages. Rather than having the concept of a list or collection, *PHÔS* supports unsized tuples. The general form of tuples as containers can be seen in Listing 16.

```
/// A tuple container containing b, c, and d
let a = (b, c, d)

/// A tuple as a type containing values of type B, C, and D
type A = (B, C, D)
```

Listing 16 Example in *PHÔS* of tuples as containers.

Unsized tuples Unsized tuples are a special kind of tuple which allows the last element to be repeating; it uses a special ellipsis (`...`) syntax to indicate that the last element is repeating. This is useful for representing lists, as *PHÔS* does not support lists otherwise. This is a purposeful decision, as it allows to extend the concept of pattern matching, discussed in Section 5.2.j, beyond simple fixed sized tuples and into the realm of dynamically sized lists. The general form of unsized tuples can be seen in Listing 17.

```
/// A simple unsized tuple: (B, B, B, B, ...)
type E = (B...)

/// A more complex unsized tuple: (B, C, D, D, D, ...)
/// On the type `D` is repeating
type A = (B, C, D...)
```

Listing 17 | Example in *PHÔS* of unsized tuples as containers.

Iterable tuples Unsized tuples lead well into the idea of tuples as iterable values. Iterable values are values that can be used for enumerations in a loop or for using iterators. In *PHÔS* , all tuples are iterable, and iterable collections can have heterogeneous types, meaning that they can iterate over values of different types. This allows the user to iterate over tuples of different signal types, values, etc. The general form of iterable tuples can be seen in Listing 18.

```
// Iterating over a list of elements
// It would print:
// 1
// 2
// 3
for a in (1, 2, 3) {
    print(a)
}
```

Listing 18 | Example in *PHÔS* of iterable tuples.

5.2.j Patterns

Patterns are used for pattern matching and destructuring, and are a core part of the language. They are used for matching values, tuples, and other types. They are also used to destructure complex values into their constituents. Patterns are used in many statements, such as the `match` statement, the `let` variable assignment statement, the `for` loop statement, and function argument declarations. The general form of the pattern can be seen in Listing 19.

```
// Destructuring of tuples: (a = 1, b = 2, c = 3)
let (a, b, c) = (1, 2, 3)
// Destructuring of tuples with trailing: (a = 1, b = (2, 3))
let (a, b...) = (1, 2, 3)
// Destructuring of a data structure: (a = 1, b = 2, c = 3)
let MyStruct { a, b, c } = MyStruct { a: 1, b: 2, c: 3 }
// Pattern matching for branching:
match (a, b, c) {
    // Matches exactly a tuple of three elements containing 1, 2, and 3
    (1, 2, 3) => print("Matched"),
    // Wildcard pattern
    _ => print("Not matched")
}
```

Listing 19 | Example in *PHÔS* of patterns.

Exhaustiveness In order for match statements to be correct, the compiler must be able to prove, based on the types, that a match statement is exhaustive. This is intended to be implemented using a similar algorithm to *Rust*'s compiler [27]. This is a very important feature, as it allows the compiler to prove that all possible cases are covered, and that the program will not crash due to a missing case. This is especially important with a point discussed in Section 4.5, which enables reconfigurability to work reliably.

Pattern matching and reconfigurability As previously mentioned, exhaustiveness helps ensure that reconfigurability is reliably achievable, as it allows the compiler to prove that all possible cases are covered. Additionally, the compiler can use constraints to remove branches that it can prove will never be taken. This is a very important and powerful feature as it decreases the configuration space of the user's design, and it allows the compiler to be faster and optimise the user's design further.

5.2.k Branching and reconfigurability

PHÔS supports branching as many other languages do. However, due to its use as a photonics HDL , *PHÔS* has the special ability to use branching as boundaries for reconfigurability regions in synthesisable contexts. This feature was previously discussed in Section 4.5. The general form of branching can be seen in Listing 20. Reconfigurability through branching is designed to be very simple to use, the user can simply branch in their code. Suppose the compiler detects that signals are being used across the branches. In that case, it will automatically create a new reconfigurability region, meaning that the work is implicit and the user does not need to do anything special to enable reconfigurability.

```
// Simple branching
if a == 1 {
    print("a is 1")
} else {
    print("a is not 1")
}

// Branching with multiple conditions
if a == 1 && b == 2 {
    print("a is 1 and b is 2")
} elif a == 1 && b == 3 {
    print("a is 1 and b is 3")
} else {
    print("a is not 1 or b is not 2 or 3")
}

// Branching using match statements
match (a, b) {
    (1, 2) => print("a is 1 and b is 2")
    (1, 3) => print("a is 1 and b is 3")
    _ => print("a is not 1 or b is not 2 or 3")
```

```
}
```

Listing 20 Example in *PHÔS* of branching.

5.2.1 Variables, mutability, and tunability

Variables in *PHÔS* are declared with the `let` keyword, followed by a pattern for the variable name, followed optionally by the type of the variable, and then followed by the assignment to the value. In *PHÔS* , variables cannot be uninitialised, and must be assigned a value when they are declared, with the notable exception of signals in unconstrained contexts, see Section 5.2.5, which have special semantics. Variables are immutable, this makes the work of the prover for reconfigurability easier, if the user whishes to update a value, they can simply recreate it. This also makes the language simpler, as the user does not need to worry about the value of a variable changing. The general form of variable declaration can be seen in Listing 21.

```
// Simple declaration
let a = 5

// Destructuring declaration
let (a, b...) = (1, 2, 3)

// Declaration with type
let a: uint = 5

// Declaration with destructuring and type
let (a, b...): (uint, uint...) = (1, 2, 3)

// Updating of a value
let a = 6
let a = a + 5
```

Listing 21 Example in *PHÔS* of variable declaration, assignment, and update.

Tunability Tunability is handled by passing a tunable value as an argument to the top-level component of the design. From then on, all subsequent use of this tunable value will also be turned automatically into tunable values. It is a simple implementation that allows the user to provide tunability and reconfigurability easily with minimal impact on the code. This, therefore, means that all code is generally tunable, and the user does not need to worry about the tunability of their code, as the compiler will handle it for them. However, if the user were to require a function not to be tunable due to its complexity, they can simply make it `static` , indicating that it cannot be tunable. An example of both use cases is provided in Listing 22.

```
// Tunable function
fn add(a: uint, b: uint) -> uint {
    return a + b
}

```

```
// Untunable function with static
fn add(a: static uint, b: static uint) -> uint {
    return a + b
}
```

Listing 22 Example in *PHÔS* of tunability.

5.2.m Piping operator and semantics

One of the key features of the *PHÔS* programming language that makes it easier to use for photonic circuit design is the ability to use the piping operator `|>` to chain functions together. This allows the user to write code in a more natural way, and allows the user to write code that is more readable. The piping operator is a binary operator with semantics that are more advanced than other binary operators: first, it can operate on any value, passing the output of one expression into the input of another, and second is that it can pattern match the values to create more complex calls. The general form of the piping operator can be seen in Listing 23.

```
// Function that performs the addition of two numbers using piping
fn add_with_pipe(a: uint, b: uint) -> uint {
    return (a, b) |> add
}

// Simple addition function
fn add(a: uint, b: uint) -> uint {
    return a + b
}

```

Listing 23 Example in *PHÔS* of the piping operator.

> **Definition**: **Monadic operations** are operations that take a value and a function and return a new value. They are common in functional programming languages, and are useful for manipulating data. They generally use the function provided as an argument to process the value, and return a new value.
>
> Adapted from [93].

Operation on iterators In addition to operating on values, the standard library will contain many common operations on iterators, such as mapping using the `map` and `flat_map` functions, two types of monadic bind operations, and filtering using the `filter` function. These operations are common in functional programming languages and are useful for manipulating data. The general form of these operations can be seen in Listing 24.

```
// Function that performs the addition of two numbers using piping
fn add_with_pipe(a: uint, b: uint) -> uint {
    return (a, b) |> fold(0, |acc, x| acc + x)
}
```

Listing 24 Example in *PHÔS* of the piping operator on iterable values.

5.2.n Function and synthesisable blocks

PHÔS separates functions into three categories: functions denoted by the keyword `fn`, and synthesisable blocks denoted by the keyword `syn`. They are designated in such a way to create clearer separation between the concerns of the user and to allow the compiler to better separate the different functions of the code. All functions in *PHÔS* , regardless of their type, are subject to constraints and the constraint solver, whether these constraints are expressed on values or signals.

Functions Function represents code that cannot consume nor produce signals, whether electrical or optical. They can only process primitive types, composite types, and ADTs . They are intended to separate any coefficient computation from the signal path, creating a strong separation between the two. An example could be a lattice filter, which implements a series of coefficients, which are likely to be computed by a function. Instead of computing them inline with the signal, they can be computed in a function and then joined with the signal in a synthesisable block. Function branches do not represent reconfiguration boundaries, which greatly simplifies the work of the compiler. An example of a function can be seen in Listing 25.

```
// Function that performs the sum of $n$ numbers
fn sum(a: (uint...)) -> uint {
    a |> fold(0, |acc, x| acc + x)
}
```

Listing 25 Example in *PHÔS* of a function.

Synthesisable blocks Synthesisable functions have the added semantic of being able to source and sink signals. They are intended to represent the signal path and are the only functions that can be used to create reconfiguration boundaries. They can be used to create a synthesisable block that can be tuned or reconfigured at runtime. An example of a synthesisable block can be seen in Listing 26.

```
// Synthesisable block that performs the filtering of an input signal using an MZI
// The MZI is built using the following actions:
//  1. The `input` signal is split into two signals using a 50/50 splitter
//  2. The two signals are constrained to have a phase difference of 30 degrees
//  3. The two signals are interfered using a 50/50 combiner
// This forms the basic structure of a Mach-Zehnder interferometer
syn filter(input: optical) -> optical {
    input |> split((0.5, 0.5)) |> constrain(d_phase: 30 deg) |> merge()
}
```

Listing 26 Example in *PHÔS* of a synthesisable block, showing an MZI built discretely using the `split`, `constrain` and `merge` functions.

5.2.o Modules and imports

Most programming languages have module systems, which allow the user to organise their code into different files or folders with nested modules. It generally avoids files being overly long and makes the code tidier and easier to understand. *PHÔS* is no different in that regard. It adopts the module system of *Python*, where each file represents a different module, with files in the folder representing submodules of the folder. The module system of *PHÔS* is very simple, but it does allow for cyclic dependencies. While cyclic dependencies tend to increase the complexity of the compiler, it is relatively easy to overcome and makes the language easier to use.

Modules are then imported using the `import` keyword, importing allows the user to import code from a module, and they can then choose what they want to import, whether it is everything, a specific submodule, or a specific function. The import system also allows the user to locally rename imported elements, such that they can avoid conflicting names. An example of an import can be seen in Listing 27.

```
// Importing the module `std::intrinsic` and renaming it to `intrinsic`
import std::intrinsic as intrinsic;

// Importing the syn `filter` from the module `std::intrinsic`
import std::intrinsic::syn::filter;

// Importing everything from the module `std::intrinsic`
import std::intrinsic::syn::*;

// Importing the syn `filter` and `gain` from the module `std::intrinsic`
import std::intrinsic::syn::{filter, gain};
```

Listing 27 | Example in *PHÔS* of an import.

Visibility In *PHÔS* , all elements that are declared are always public. Due to the expected low number of users of the language, it makes sense that all elements declared into a module be made public, such that code reuse can be maximised. This also means there is no need for the concept of visibility as it exists in many other languages, with special keywords like `pub`, `public`, or `private` to define the visibility of an item.

5.2.p Closures and partial functions

Closures are anonymous functions that are defined inline with the rest of the code. As with most modern languages, *PHÔS* supports closures. Closures are a source of complication for the compiler, it is very difficult for the compiler to keep track of value movement in closures and it is even more difficult to keep track of signal movement and usage for closures. Therefore, while signals are allowed to be used within closures, this cannot be checked at compile time and therefore must be checked by the VM at a much later stage. An example of a closure can be seen in Listing 28.

```
// Closure that performs the sum of $n$ numbers
let sum = |a: (uint...)| -> uint {
    a |> fold(0, |acc, x| acc + x)
};
```

Listing 28 | Example in *PHÔS* of a closure.

Signals in closures Due to their *drive-once*, *read-once* semantics, signals are an especially difficult case for closures. Normally, closures are allowed to capture variables from their environment, and are equivalent to functions. However, signals are not normal variables, and the capturing mechanism has to be different. For this reason, closures are separated into three types: `Fn`, which are closures that operate like regular functions, `Syn`, which are closures that operate like regular synthesisable blocks. While they can process signals, they cannot capture signals, `SynOnce` which are closure that is synthesisable, but must be called once, essentially following the same *drive-once*, *read-once* semantic as other signals. The difference between the three kinds can be seen in Listing 29.

```
// (a) `Fn` closure
let c = 1;
let fn_closure = |a: uint| -> uint {
    a + c
};

// (b) `Syn` closure
let syn_closure = |a: optical| -> (optical...) {
    a |> split((0.5, 0.5))
};

// (c) `SynOnce` closure
let a = source(1550 nm)
let syn_once_closure = || -> (optical...) {
    a |> split((0.5, 0.5))
};
```

Listing 29 | Example in *PHÔS* of an `Fn` closure (a), a `Syn` closure (b), and a `SynOnce` closure (c).

Definition: **Partial functions** are functions which are **partially applied**, meaning that parts of their argument are already applied and that the new function can be called with only the missing arguments. Partial functions are, therefore functions with lower arity.

Adapted from [94]

Partial functions In *PHÔS* , partial functions are created with the keyword `set`. It produces a closure with the same semantics as previously discussed but with the added ability to be called with fewer arguments than the closure expects. Additionally, the caller of the closure may override already `set` arguments by referring to them by name. An example of a partial function can be seen in Listing 30.

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

let add_1 = set add(1);

print(add_1(2)); // prints 3
```

Listing 30 | Example in *PHÔS* of partial function in *PHÔS* .

5.2.q Loops, recursion, and Turing completeness

Definition: **Turing completeness** is used to express the power of a data manipulation system. It is a measure of the ability of a system to perform all calculable computations.

Adapted from [95].

PHÔS does not aim to be a Turing complete, as it does not need to be used for generic purposes. Due to the exponential complexity of reconfigurability states as the program is allowed to loop and recurse, *PHÔS* places a hard limit on the following elements: the number of iterations, the depth of recursion, and the number of optical intrinsic operations that can be performed. The goal of these measures is to avoid the compiler taking too long to try and compile a program for which no device is big enough to fit it. The compiler will therefore reject programs that exceed these limits. The limits are set to be high enough that they should not be reached in most cases but low enough that the compiler can reject programs that are too complex to be compiled. Additionally, some of these limits can be changed by the user using the marshalling layers (see Section 5.8).

For the aforementioned reason, *PHÔS* does not have infinite loops but only iterative loops that iterate over an input value. This limits the risk of the user falling into the iteration limit, as the number of iterations is known. An example of a loop can be seen in Listing 31.

PHÔS

```
for i in 0..5 {
    print(i);
}
```

Listing 31 | Example in *PHÔS* of a loop.

5.2.r Constraints

It has been discussed that the syntax of constraints should be changed to declutter function/synthesisable block signatures. This would allow constraints to be cleaner and would ideally be expressed as its own part of the signature, rather than being defined with the arguments. However, this has not yet been designed and is therefore not discussed further in this document.

PHÔS models constraints as additional data carried by values and signals. It applies the semantics discussed in Section 4.4. In the current iteration of the design, constraints are, therefore, a evaluation-time concept that cannot be checked by the compiler. This is a limitation of the current design and will be addressed in future iterations. An example of a constraint can be seen in Listing 32.

PHÔS

```
// Performs an optical gain on an input signal,
// the maximum input power is `10dBm - gain`,
// the gain is constrained to be between 0 and 10dB.
syn gain(
```

```
    @max_power(10dBm - gain)
    input: optical,
    @range(0dB, 10dB)
    gain: Gain,
) -> @gain(gain) optical {
    ...
}
```

Listing 32 | Example in *PHÔS* of a constrained synthesisable block.

5.2.s Unconstrained

As mentioned in Section 4.4, constraints only work for non-cyclic cases[3], however, this limitation removes the advantage of having a recirculating mesh inside the photonic processor. Therefore, as was previously mentioned, *PHÔS* must provide a way to express blocks where the constraints are not automatically inferred but must be manually specified. This is done by using the `unconstrained` keyword, which allows the user to specify the constraints manually at the boundary of a synthesisable block. An example of an unconstrained block can be seen in Listing 33.

Additionally, unconstrained blocks allow the user to create their signals without needing to use a source intrinsic. This semantic is useful for creating recirculating elements in the photonic processor, as it allows the user to create temporary variables containing signals.

```
// A ring resonator implemented using an unconstrained block
unconstrained syn ring_resonator(
    input: optical,

    @range(0.0, 1.0)
    coupling: float,

    @min(6)
    length: Length,
) -> @frequency_response(response(coupling, length)) optical {
    // Create a new internal signal
    let ring: optical;

    // Create the output signal
    let output: optical;

    // Use an intrinsic coupler
    (input, ring)
        |> std::intrinsic::coupler(coupling)
        |> constrain(dlen = length)
        |> (output, ring);
```

[3] That is, constraints that do not depend on themselves.

```

    output
}

// Returns the frequency response of a ring resonator given its arguments.
fn response(coupling: float, length: Length) -> FrequencyResponse {
    ...
}
```

Listing 33 Example in *PHÔS* of an unconstrained synthesisable block.

5.2.1 Stack collection and synthesisable non-signal types

One of the issues that arises from tunability, and especially with reconfigurability, is that the tunable values can be of any type and therefore need to be converted into the values that the physical hardware can interpret. When designing the circuit, or the hardware platform package, it is natural to convert these meaningful high-level values into lower-level, less explicit values using *PHÔS* . Ideally, as much of the conversion as possible should be done in *PHÔS* , such that the circuit code can act as a source of truth and decrease the complexity of the hardware-software codesign. But this creates an issue: when using tunable value, the *PHÔS* VM cannot compute these low-level values directly, as it does not know the value of the tunable value. Additionally, when tunable values lead to reconfigurability, the VM cannot evaluate the conditions that lead to reconfigurability statically.

Therefore, the parts of the code that perform the conversion between high-level and low-level values must be synthesisable. Of course, the photonic processor being an analog processor, it is not possible to perform this synthesis on the actual mesh. However, as mentioned in Section 4.2, one of the compilation artefacts is the user HAL , which is generated for the user based on their design. It would therefore be possible to collect these operations and package them into the HAL , such that when the user programmatically tunes their design, the conversion is done inside of the user HAL and sent to the processor's controller as a low-level value. The name is based on the fact that *PHÔS* uses a stack-based virtual machine and that the operations are collected into the user HAL and converted into *Rust* (the implementation language of the HAL) automatically by the compiler.

> **Definition**: **Stack collection** is the process of collecting the operations that convert high-level values into low-level values and packaging them into the user HAL .

Stack collection is, therefore, an automatic feature performed by the compiler, its goal is to evaluate as much as possible at compile time as a means to decrease the amount of work being done inside of the user HAL and collect the stack operations that are relevant to the conversion. From these collected stacks, it can easily evaluate the conditions that lead to reconfigurability and package them into the user HAL . Nonetheless, one problem remains: which reconfigurability states must be kept past this point. As explained in Section 4.5, the compiler can discard as many reconfigurability states as possible based on constraints and using a prover like *Z3* [85].

5.2.u Language items and statements

Language items and statements are the hierarchy of language elements that can be used to create a program. They are the most basic elements of the language and are used to create more complex elements. They are the building blocks of the language and are the elements that are used to create the AST (*Abstract Syntax Tree*) . They are the most important elements of the language and are the ones that are the most likely to be modified in the future. The language items and statements of *PHÔS* are listed in Table 9, along with a short description and a short example.

<table>
<thead>
<tr><th>Element</th><th>Item</th><th>Statement</th><th>Description</th><th>Example</th></tr>
</thead>
<tbody>
<tr><td>Import</td><td>✓</td><td>✓</td><td>Imports a module into the current module.</td><td><code>import std::intrinsic as intrinsic;</code></td></tr>
<tr><td>Function</td><td>✓</td><td>✓</td><td>Declares a function.</td><td><code>fn sum(a: (uint...)) -> uint {</code>
<code>   a |> fold(0, |acc, x| acc + x)</code>
<code>}</code></td></tr>
<tr><td>Synthesisable</td><td>✓</td><td>✗</td><td>Declares a synthesisable function.</td><td><code>syn gain(input: optical) -> optical {</code>
<code>   input |> std::intrinsic::gain(10 dB)</code>
<code>}</code></td></tr>
<tr><td>Type alias</td><td>✓</td><td>✗</td><td>Declares a type alias.</td><td><code>type Voltage = float;</code></td></tr>
<tr><td>Constant</td><td>✓</td><td>✓</td><td>Declares a constant.</td><td><code>const PI = 3.141592;</code></td></tr>
<tr><td>Structure</td><td>✓</td><td>✗</td><td>Declares a new data structure.</td><td><code>struct Point {</code>
<code>   x: float,</code>
<code>   y: float</code>
<code>}</code></td></tr>
<tr><td>Enumeration</td><td>✓</td><td>✗</td><td>Declares a new ADT .</td><td><code>enum Color {</code>
<code>   Red,</code>
<code>   Green,</code>
<code>   Blue,</code>
<code>}</code></td></tr>
<tr><td>Local</td><td>✗</td><td>✓</td><td>Declares a new local variable</td><td><code>let (a, b...) = (1, 2, 3);</code></td></tr>
<tr><td>Expression</td><td>✗</td><td>✓</td><td>Declares a new expression.</td><td><code>1 + 2</code></td></tr>
</tbody>
</table>

Table 9 The list of language items and statements supported in *PHÔS* , along with a short description and a short example. Additionally lists whether the element is an item or a statement, or both, where a statement is a language element that can be used as an expression, and an item is a language element that cannot be used as an expression, but can be used as a top level declaration. Legend: ✓ means yes, ✗ means no.

5.2.v Expressions

Expressions are a subset of statements that operate on one or more values and may produce an output value. A complete list of *PHÔS* expressions is available in Table 10.

	Expression	Description	Example

Control flow	**Block**	Declares a new block of code.	`{`
			`    let a = 10;`
			`    a + 20`
			`}` (PHÔS)

	If/Else/Elif	A conditional statement for branching.	`if a > b { a } else { b }` (PHÔS)

	Match	A conditional statement for branching.	`match a {`
			`    1 => "one",`
			`    _ => "other"`
			`}` (PHÔS)

	Loop	A loop statement.	`for i in 0..5 {`
			`    print(i)`
			`}` (PHÔS)

	Return	Returns a value from a function.	`return 1` (PHÔS)

	Break	Breaks out of a loop.	`for i in 0..5 {`
			`    break;`
			`}` (PHÔS)

	Continue	Continues a loop, terminating the current iteration and moving on the the next one.	`for i in 0..5 {`
			`    continue;`
			`}` (PHÔS)

	Yield	Yields a value from an iterator.	`for i in 0..5 {`
			`    yield i;`
			`}` (PHÔS)

Constant	**Path**	A path to a value, constant, or other item.	`std::intrinsic::gain` `float::PI` (PHÔS)
	Identifier	A name that refers to a value, constant, or other item.	`gain` `PI` (PHÔS)

	Literal	A literal value.	`1 dBc`
			`true`
			`0.5 MHz` (PHÔS)

	None	The none value.	`none` (PHÔS)

UNARY	**NEGATION**	Negates a value.	`-1` PHÔS
	NOT	Negates a boolean value.	`!true` PHÔS
	BINARY NOT	Negate a binary value.	`!0xFF` PHÔS
BINARY	**ADDITION**	Adds two values.	`1 + 2` PHÔS
	SUBTRACTION	Subtracts two values.	`1 - 2` PHÔS
	MULTIPLICATION	Multiplies two values.	`1 * 2` PHÔS
	DIVISION	Divides two values.	`1 / 2` PHÔS
	MODULO	Calculates the remainder of a division.	`1 % 2` PHÔS
	EXPONENTIATION	Raises a value to a power.	`1 ** 2` PHÔS
	BITWISE AND	Performs a bitwise and operation.	`1 & 2` PHÔS
	BITWISE OR	Performs a bitwise or operation.	`1 | 2` PHÔS
	BITWISE XOR	Performs a bitwise xor operation.	`1 ^ 2` PHÔS
	BITWISE SHIFT LEFT	Performs a bitwise shift left operation.	`1 << 2` PHÔS
	BITWISE SHIFT RIGHT	Performs a bitwise shift right operation.	`1 >> 2` PHÔS
	LESS THAN	Checks if a value is less than another.	`1 < 2` PHÔS
	LESS THAN OR EQUAL	Checks if a value is less than or equal to another.	`1 <= 2` PHÔS
	GREATER THAN	Checks if a value is greater than another.	`1 > 2` PHÔS
	GREATER THAN OR EQUAL	Checks if a value is greater than or equal to another.	`1 >= 2` PHÔS
	EQUAL	Checks if a value is equal to another.	`1 == 2` PHÔS
	NOT EQUAL	Checks if a value is not equal to another.	`1 != 2` PHÔS
	LOGICAL AND	Checks if two boolean values are both true.	`true && false` PHÔS
	LOGICAL OR	Checks if either of two boolean values are true.	`true || false` PHÔS

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr><td rowspan="13">Special</td><td>Pipe</td><td>Pipes a value into a function.</td><td><code>1 |> f</code> PHÔS</td></tr>
<tr><td>Parenthesized</td><td>Groups an expression.</td><td><code>(1 + 2) * 3</code> PHÔS</td></tr>
<tr><td>Tuple</td><td>Creates a tuple.</td><td><code>(1, 2)</code> PHÔS</td></tr>
<tr><td>Cast</td><td>Casts a value to a different type.</td><td><code>1 as uint</code> PHÔS</td></tr>
<tr><td>Index</td><td>Indexes into a value.</td><td><code>a[1]</code> PHÔS</td></tr>
<tr><td>Member access</td><td>Accesses a member of a value.</td><td><code>a.b</code> PHÔS</td></tr>
<tr><td>Function call</td><td>Calls a function.</td><td><code>f(1, 2)</code> PHÔS</td></tr>
<tr><td>Method call</td><td>Calls a method.</td><td><code>a.f(1, 2)</code> PHÔS</td></tr>
<tr><td>Partial</td><td>Partially applies a function.</td><td><code>set f(1)</code> PHÔS</td></tr>
<tr><td>Closure</td><td>Creates a closure.</td><td><code>|x| x + 1</code> PHÔS</td></tr>
<tr><td>Range</td><td>Creates a range.</td><td><code>1..2 1..=2 ..3 4..</code> PHÔS</td></tr>
<tr><td>Array</td><td>Creates an array.</td><td><code>[1, 2]</code> PHÔS</td></tr>
<tr><td>Object instance</td><td>Creates an object instance.</td><td><code>A {a: 1, b: 2}</code> PHÔS
<code>B(1, 2)</code>
<code>MyEnum::A</code>
<code>C</code></td></tr>
</table>

Table 10 The list of language expressions supported in *PHÔS* , along with a short description and a short example. Additionally lists whether the expression is a control flow expression, constant expression, unary expression, binary expression, or special expression. Where a control flow expression is an expression that can be used to control the flow of the program, a constant expression is a value that can be determined at compile time, a unary expression is an expression that takes only one argument, a binary expression is an expression that takes two arguments, and a special expression is an expression that is not easily categorised and that performs more complex actions, with well-defined semantics.

5.3 Standard library

In addition to the language itself, *PHÔS* will come with a standard library: a library of functions and synthesisable blocks that come with the language. The standard library will be written in a mixture of *PHÔS* for synthesisable blocks and functions and some native *Rust* code for either performance-critical sections or areas where external libraries are required. The standard library will be organised as logically as possible, providing the necessary building blocks for new users to be productive with the language. However, the standard library will be limited in scope such that it does not become a burden to maintain. Relying instead on third-party libraries and IPs for more complex functionality. A notable goal of the standard library is to provide synthesisable blocks for all common functions, like modulators, filters, and so on. But not for more complex functionality like larger components or even entire systems.

Most intrinsic operations discussed in Section 4.3 are more complex than they first appear. As previously mentioned in Section 2, most photonic components are actually reciprocal, meaning that they are the same whether light is travelling forwards or backwards. Additionally, waveguides support two modes, one in each direction, meaning that each device may be used for two purposes. Removing the user's ability to exploit these properties would be greatly limited, and as such, the standard library must provide ways of accessing these fully-featured intrinsic operations. However, the user cannot be expected to only program using these low-level primitives. Furthermore, they are mostly unconstrained and would require constrained blocks to be used to their full extent due to the limitation on constraints regarding cyclic dependencies. Therefore, one of the main goals of the standard library is to provide higher-level primitives that wrap these unconstrained intrinsic operations into constrained blocks following the feedforward approximation (Section 2.2.a).

The standard library should also decouple synthesisable blocks from computational methods. For example, a filter block may need other functions to compute the coupling coefficient or the length of a ring resonator. These functions will need to be part of the standard library to offer filter synthesis, but they should be accessible separately, such that if a user wishes to implement some function themselves, they can rely on the existing code present in the standard library to make their work easier. This also means that the standard library should be as modular as possible, perhaps even in the future, allowing users to replace default behaviour in the standard library with their own implementations. This modularity also helps in the development of the platform-support packages, as these need to be able to support the standard library, something that may be done by replacing parts of the standard library with platform-specific implementations while keeping the exposed API the same.

Finally, the standard library can serve as a series of examples for new users. A photonic engineer that is knowledgeable in photonic circuit design would benefit from the standard library as a source of high-quality examples onto which they may base themselves. Similarly, a software engineer knowledgeable with software development but not photonic circuit design, would benefit from the standard library as a source of high-quality examples of basic building blocks of photonic circuits. The standard library should be written in a way that is easy to understand and that is well-documented, such that it can serve as a learning resource for new users.

Constrain A unique feature of the standard library is the `constrain` method. It is used to impose differential constraints on signals. This means that constraints over delay or phase, which are always relative, can be expressed between signals. This tells the synthesiser to ensure that these constraints are respected between signals. Indeed, if it were not for this method, the user could not easily represent phase-matched or delay-matched signals. During the examples, in Section 6, the use of this method will become clear, and its implication and why it is so important will be discussed.

5.4 Compiler architecture

The design of the *PHÔS* compiler is inspired in parts by *Clang*'s, *Rust*'s, and *Java*'s compilers [34, 27, 96]. As previously mentioned, the compilation of a *PHÔS* program into a circuit design that can be programmed onto a photonic processor is a three-step process: compilation, evaluation, and synthesis. The compiler, as it is referred to in this section, performs the compilation step. Therefore, as previously mentioned in Section 5.2.a, it takes the user's code as an input and produces bytecode for the VM (*Virtual Machine*) . The compiler is written in *Rust* and is split into several components, each with a specific purpose. As will be discussed in subsequent sections, the *PHÔS* compiler is composed of a *lexer*, a *parser*, an *AST (*Abstract Syntax Tree*)* , a *desugaring* step, a *high-level intermediary representation*, a *medium-level intermediary representation*, and a *bytecode* generator. The multiple stages of the compiler are illustrated in Figure 13.

> The overall architecture and components of the *PHÔS* compiler are similar in design to *Rust*'s compiler [27]. This is done purposefully for two reasons. First, *Rust* as a language is advanced and has a well designed compiler, and as such, is a good source of reference materials. Second, the *PHÔS* compiler is written in *Rust*, and as such, it can reuse existing code and algorithms from both the extended ecosystem of language development libraries in *Rust* and from the *Rust* compiler itself. As the *Rust* compiler is released under an MIT (*The* MIT *license*) license, it can legally be used as a source of inspiration and code for the *PHÔS* compiler.

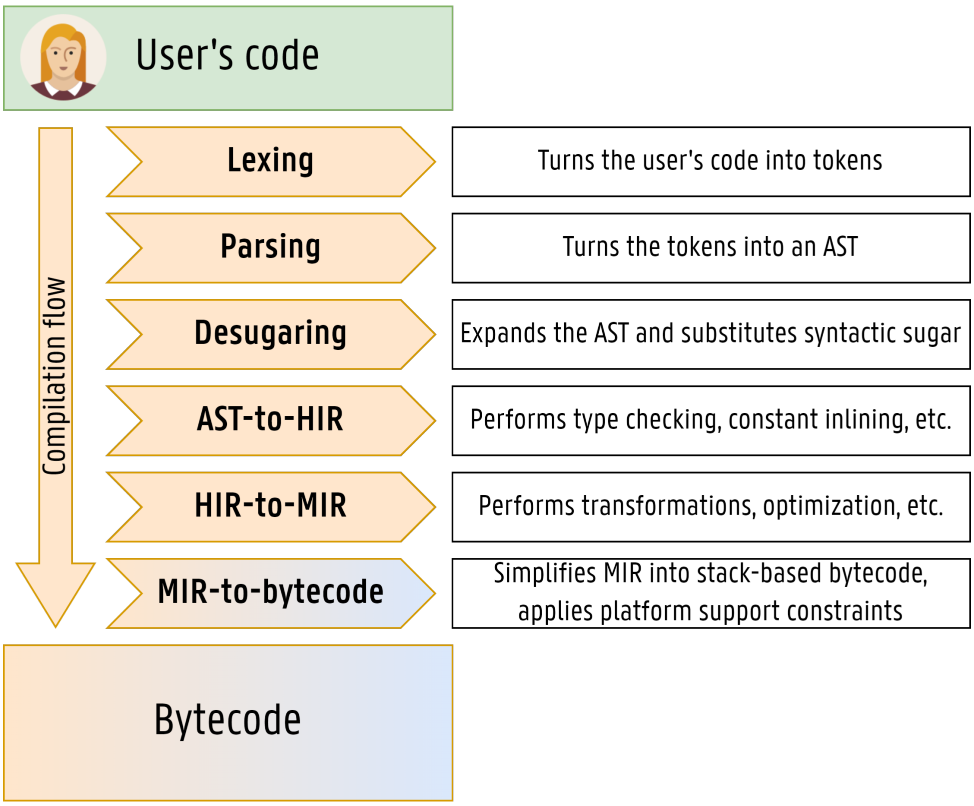


Figure 13 Compiler architecture of the *PHÔS* programming language, showing that the user code flows into the different stages of the compiler. All of these stages produce the bytecode that can be evaluated.

This figure uses the same colour scheme as Figure 9, showing the ecosystem components in orange, the user's code in green, and the platform-specific code in blue.

5.4.a Lexing

Definition: **Lexing** is the process of taking a stream of characters and converting it into a stream of tokens. A token is a sequence of characters that represent a unit of meaning in the language.

Adapted from [97].

As this definition implies, this lexer turns a series of characters from the human-readable *PHÔS* code into a series of tokens, which can be seen as words of the language. Some of those words have special significance, such as `+`, which represents an addition, and others such as an open parenthesis (`(`) simply represents an open parenthesis. At this stage of the compilation process, there is no meaning associated with each token, meaning that they are separate entities that the compiler is yet to associate into bigger, more meaningful units. The *PHÔS* lexer is implemented using the *Logos* library in *Rust* [98]. *Logos* is a lexer generator, meaning that it allows the creation of extremely fast lexers very quickly, and is used by other open-source projects [99].

In Listing 34, one can see the process that turns a simple piece of code into a series of tokens that can be used by the parser. The code is first split into a series of characters, which are then fed into the lexer. The lexer then produces a series of tokens, which are then printed to the console. One can also see that the comment in line one has been removed. Indeed, the lexer discards all unnecessary tokens, such as linebreak, whitespace, and comments. This is done to simplify the parsing process, as the parser does not need to deal with those tokens and can focus on the tokens that are meaningful to the language.

Additionally, and not shown in Listing 34, the *PHÔS* lexer preserves the span where the token is found in the source code. This is used to generate more meaningful errors by indicating the location of the error in the source code. This is done by associating the said span with each token produced by the lexer.

PHÔS
```
1 // Adds two numbers together.
2 fn add(a: int, b: int) -> int {
3     a + b
4 }
```

(a)

Tokens
```
Keyword("fn") Identifier("add")
    OpenParen
        Identifier("a") Colon Identifier("int")
Comma
        Identifier("b") Colon Identifier("int")
    CloseParen Arrow Identifier("int") OpenBrace
        Identifier("a") Plus Identifier("b")
    CloseBrace
```

(b)

Listing 34 Example of lexing in *PHÔS* , showing a code sample (a) and the output of the lexer (b). Note that the indentation in (b) is solely for readability purposes and is not part of the output of the lexer.

5.4.b Parsing

Definition: **Parsing** is the action of taking a stream of tokens and turning it into a tree of nested elements that represent the grammatical structure of the program.

Adapted from [97].

Before parsing the language, one must first describe the grammar of the language. There exist many families of grammar, as seen in Figure 14. The more complex the grammar is, the more complex of a parser it will require. It is important to note that Figure 14 describes grammar, not languages. Languages can be expressed using multiple grammars, and some grammars for a given language can be simpler [100]. Every grammar can be expressed with a higher-level grammar, but the reverse is not valid.

PHÔS has an LL(K) grammar, meaning that it can be read from **L**eft-to-right with **L**eftmost derivation, with k token lookahead, meaning that the parser can simply move left-to-right, only ever applying rules to elements on the left, and needs to look up to k tokens ahead of the current position to know which rule to apply. This is a fairly complex grammar to express and parse. The parser for *PHÔS* is implemented using the *Chumsky* library (named after *Noam Chomsky*) in *Rust* [101]. *Chumsky* is a parser combinator generator, meaning that it allows the creation of complex parsers with relatively little code. Because of the properties of *Chunmsky*, the parser for the *PHÔS* language is fairly simple, at only 1600 lines of code, and is relatively easy to understand. It is the task of the parser to use the *PHÔS* grammar to produce an AST (*Abstract Syntax Tree*) this tree represents the syntax and groups tokens into meaningful elements.

Additionally, the grammar of *PHÔS* contains the priority of operations, meaning that the resulting AST is already correct regarding the order of operations, something that otherwise would need to be done using AST transformations in the next step of the compilation process. This further increases the complexity of the grammar and the complexity of the parser but simplifies the next steps of the compilation process.

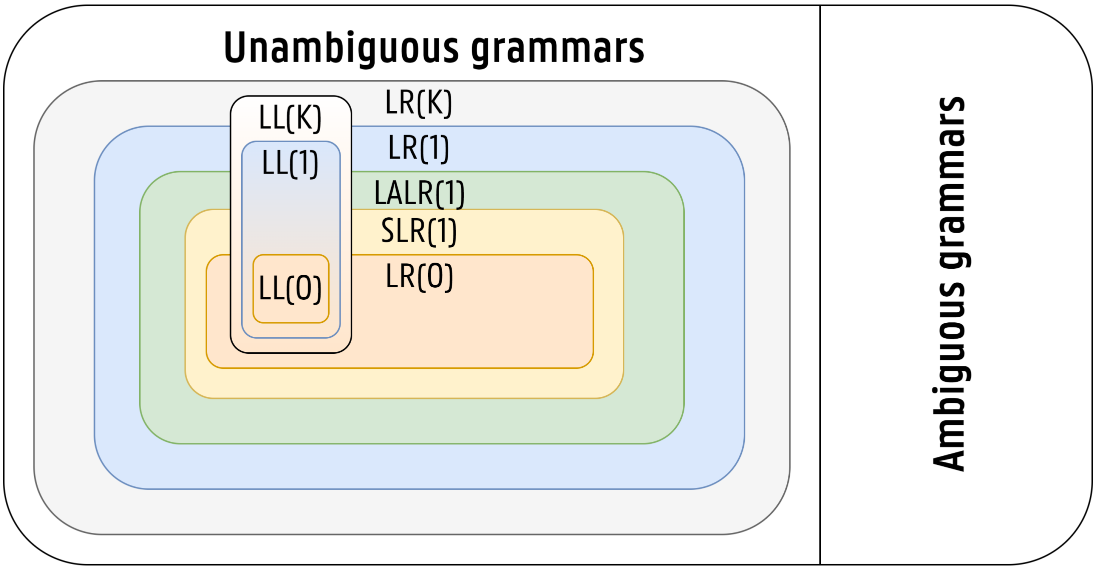


FIGURE 14 Hierarchy of grammar that can be used to describe a language. The grammars are ordered from the most powerful to the least powerful. The most powerful grammars are able to describe any unambiguous language, whereas the least powerful grammars are only able to describe a subset of the languages [100]. As *PHÔS* does not use ambiguous grammar and they are very difficult to describe and parse, they are not discussed further.

5.4.c The abstract syntax tree

The AST (*Abstract Syntax Tree*) is the result of the previous compilation step – parsing –, and it is a tree-like data structure that represents the syntax of the user's code in a meaningful way. It shows the elements as bigger groups than tokens, such as expressions, synthesisable blocks, etc. The AST is the base data structure on which all subsequent compilation steps are based. The AST would also be used by the IDE to provide code completion, syntax highlighting, and code formatting.

Just as is the case for parsing, syntax trees have a hierarchy; it generally consists of two categories: the CST (*Concrete Syntax Tree*) and the AST (*Abstract Syntax Tree*) . The CST aims at being a concrete representation of the grammar, being as faithful as possible, and keeping as much information as possible. On the other hand, an AST only keeps the information necessary to perform the compilation. Therefore, it is generally simpler and smaller than an equivalent CST . However, while this can be seen as a hierarchy, it is more of a spectrum, as the AST can be made more concrete and closer to a CST , depending on the needs. In fact, the AST of *PHÔS* keeps track of all tokens, and their position in the source code, making it possible to reconstruct the original source code from the AST . The only thing it discards is whitespaces, linebreaks, and comments. The AST of *PHÔS* also keeps track of spans where the code comes from, just like in the lexer. It is used to provide better error messages.

Building on top of the example shown in Listing 34, the AST for the function `add` would look like Figure 15. Additionally, an overview of the data structure required to understand this part of the AST is shown in Annex B. Some details have been omitted for brevity and to focus on the relevant parts of the AST . However, one can still see the tree-like structure of the AST and the many different kinds of data structures that it requires. In fact, the current AST for *PHÔS* is composed of 250 different data structures. This shows how complex the AST can be and how much work is required to build it. However, having a good AST as the basis of the compilation process is crucial as it can be easily modified, expanded, and transformed to perform the compilation. Additionally, the breadth of data it contains can be used to implement other elements of the ecosystem, such as code formatters, code highlighters, and linters, all tools that have been discussed at length and are essential to provide a good developer experience.

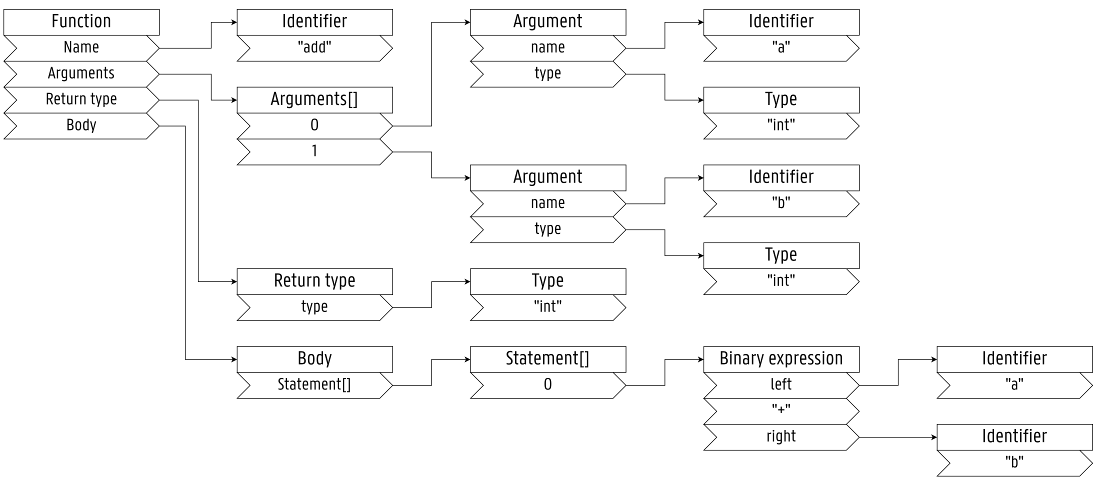


Figure 15 Partial result of parsing Listing 34, showing the tree-like structure of nested data structures. The AST is a tree-like data structure that represents the syntax of the user's code. In this case, it shows a function whose name is an identifier `add`, and that has two arguments: `a` and `b`, both of type `it`, it has a return type of type `int`, and a body that is a block containing a single expression, which is a call to the `+` operator, with the arguments `a` and `b`.

5.4.d Abstract syntax tree: desugaring, expansion, and validation

From this point in the document, the language is not yet implemented and therefore does not exist. Therefore, the following steps are not implemented and only describe what will be done when the language is fully implemented.

Definition: **Syntactic sugar** is the part of the language's syntax that is intended to make things easier to read, express, or understand.

Adapted from [97].

The first step in the compilation process is to remove syntactic sugar, as it is not useful for the compiler and is only intended to make the code easier to read and write. Therefore, this syntactic sugar can be simplified into simpler syntactic blocks. Additionally, in the same stage, the AST is expanded to include more information. In the case of the *PHÔS* programming language, this will involve the following:

- Feature gate checking;
- Transforming all automatic return statements into explicit return statements;
- Name resolution;
- Macro expansion;
- and AST validation.

For this section, the example shown in Listing 35 will be used. It shows a simple circuit involving a filter, a gain section that is gated behind a feature flag, implicit returns and imports. Since macros don't yet have a fixed syntax for a language and are left as future work, see Section 7.7, there will not be an example involving macros.

PHÔS

```
import std::{filter, gain};
/// Processes a signal using one of two filters based on feature flags
syn process_signal(signal: optical) -> optical {
    signal |> internal_process()
}
/// If the library supports gain, add some gain after the filter
#[feature(gain)]
syn internal_process(signal: optical) -> optical {
    signal |> filter(1550 nm, 10 GHz) |> gain(10 dB)
}
/// If the library does not support gain, just filter the signal
#[feature(not(gain))]
syn internal_process(signal: optical) -> optical {
    signal |> filter(1550 nm, 10 GHz)
}
```

Listing 35 | An example used to show the desugaring, AST expansion, and AST validation in *PHÔS* .

> **Note:** The processes being described in this section are done at the AST level. However, for clarity, the source code is shown instead of the AST for each step in this compilation stage.

Feature gate checking In this step, the syntax is checked for feature gates, and only the parts of the AST not gated by feature gates are kept. Meaning that feature flags are evaluated, and the part of the AST that cannot be compiled with the current set of feature flags is removed. For code involving feature flags, this reduces the complexity of the AST and the amount of work the compiler must do. This step is done as early as possible to further reduce the amount of work the compiler must do. Continuing on from Listing 35, in Listing 36, it can be observed that the `gain` section has been removed, assuming that the `gain` feature flag is not enabled. The comments are also removed as the parser does not include comments in the AST .

```
import std::{filter, gain};
syn process_signal(signal: optical) -> optical {
    signal |> internal_process()
}
syn internal_process(signal: optical) -> optical {
    signal |> filter(1550 nm, 10 GHz)
}
```

Listing 36 Demonstration of feature gate checking in *PHÔS* , using the example from Listing 35.

Automatic return statement In this process, all automatic return statements that are present at the end of a block are transformed into explicit return statements, such that the rest of the compiler need only look for explicit return statements. This is done to simplify subsequent compilation steps. Building on from Listing 35, one can see in Listing 37 that the automatic return statements have been transformed into explicit return statements.

```
import std::{filter, gain};
syn process_signal(signal: optical) -> optical {
    return signal |> internal_process();
}
syn internal_process(signal: optical) -> optical {
    return signal |> filter(1550 nm, 10 GHz);
}
```

Listing 37 An example used to show the desugaring of return statements, AST expansion, and AST validation in *PHÔS* .

Name resolution At this stage, the compiler has not yet resolved the path to the different items being used. Therefore, all import statements and path expressions are inlined and resolved in these steps. This means that import statements will no longer be needed or used after this process, and their absolute path will replace all relative paths to items. Additionally, it is at this stage that the compiler checks for the existence of the items being used. If they do not exist, the compiler will return an error. Continuing on from Listing 35, in Listing 38, one can see that the import statements have been removed, and the paths to the items have been resolved, further simplifying the AST (shown here as code for clarity). As the type `optical` is a built-in type in *PHÔS* , it does not get resolved. It is always valid.

```
syn process_signal(signal: optical) -> optical {
    return signal |> self::internal_process();
}

syn internal_process(signal: optical) -> optical {
    return signal |> std::filter(1550 nm, 10 GHz);
}
```

Listing 38 | Example of name resolution in *PHÔS* , using the example from Listing 35.

Macro expansion Depending on the type of macros being implemented in *PHÔS* , they may be operating at the AST level. If that were to be the case, macros would be expanded in this stage. Macro expansion refers to the compiler replacing the macro calls within the code with the output produced by the macro. This is done at this stage because the AST has not yet been checked. Meaning that the code produced by the macro, if it were to be erroneous, would still be verified and not assumed correct. As the example in Listing 35 does not contain any macros, the AST remains unchanged.

It is also important to note that *PHÔS* may benefit more from reflection-level macros rather than AST -level macros. This is because *PHÔS* is, at its core, a high-level language, and therefore, macro creators may be interested in also operating at a higher level of symbolic representation than the AST . However, this is neither a requirement nor has either solution been implemented yet. Additionally, both solutions are suitable and can be implemented at the same time.

AST validation AST validation involves the process of verifying that the AST is correct. This means that the more complex syntactic rules that were not expressed in the grammar are checked. They are checked on the AST because it makes the grammar of the language simpler, simplifying the parser too. Additionally, it can perform basic checks based on rules. While these rules are not yet clear for *PHÔS* , they will need to be defined in the future. It is important to note that AST validation should not involve any complex analysis, such as type checking, as these are easier to implement on the HIR (*High-level Intermediate Representation*) .

5.4.e AST lowering, type inference, and exhaustiveness checking

At this point in the compilation pipeline, much of the initial complexity of the user's code has been removed. However, many critical aspects of compiling the language have not been performed yet, most notably with regard to type inference, type checking and exhaustiveness checking. These functions are all performed on the HIR , but at this point, the compiler still only has the AST . Therefore, the first process in this step is to lower the AST . Essentially, the compiler must transform the AST into a lower-level, simpler form of code called the HIR .

AST lowering Lowering from the AST to the HIR requires removing all of the elements of the language that are not needed for type analysis which is the focus of this compiler step. Due to the previous step having decreased language complexity using desugaring, the AST is already less complex. However, it still contains elements that are not needed for type analysis, such as names. Indeed, variable names, type names, etc. are only useful for humans. It is easier for a computer to understand these as numerical identifiers (IDs). Therefore, in this process, the name of all values are replaced with generated IDs. Additionally, the tree-like data structure can be flattened by using node IDs instead of pointers to nodes. This decreases the complexity of the data structure and makes traversal and, most importantly, queries easier to perform. Indeed, queries need to be performed on the HIR to find elements and apply rules for HIR to MIR (*Mid-level Intermediate Representation*) lowering.

Type inference After lowering the AST into the HIR , the compiler will try to infer all values' types based on existing annotation and some rules. When the compiler has inferred the type of a value, it will explicitly annotate that value with its type, that way, each value has its type known at each point in the code, such that further checks can be performed. Continuing with Listing 35, one can see what the resulting, fully annotated code would look like in Listing 39. It shows what the code would look like if this were valid syntax after AST lowering and explicit type annotation.

PHÔS

```
syn $0($1: optical) -> optical {
    return (($1: optical) |> ($2(): optical)): optical;
}
syn $2($1: optical) -> optical {
    return (($1: optical) |> ($3((1550 nm: Wavelength), (10 GHz: Frequency)); optical)): optical;
}
```

Listing 39 Example of name stripping and type inference in *PHÔS* , showing the process of name stripping and type inference, using the example from Listing 35. All nodes are annotated with their type, and all variables, arguments, function names, etc. have been renamed with an ID shown here as `$n`, where `n` is a number.

Note that this is not valid *PHÔS* syntax and is only used to illustrate the process of name stripping and type inference. In practice, the HIR would be a flattened tree-like data structure, similar to the AST , not a textual representation.

As with most modern programming languages with type inference, *PHÔS* will use the *Hindley-Milner* algorithm [26, 102]. It is an algorithm that is capable of inferring the type of a value with little to no type annotations. This makes development easier, as less manual work of annotating types is required. Additionally, it is a convenient algorithm to use as many resources are available, and many libraries are implementing it already, meaning that the development burden caused by type inference is significantly lower.

Additionally, the *Hindley-Milner* algorithm supports polymorphism. While this feature is not yet integrated into the syntax and feature set of the *PHÔS* language, it may be of interest as it can allow more advanced types to be expressed. However, this is not a priority for the language; therefore, it is not yet designed into the language nor into the compiler architecture.

Exhaustiveness checking Exhaustiveness checking is the process of verifying that the user has covered all possible cases in a pattern-matching statement. This can be done with the algorithm presented by *Karachalias et al.* [103, 104]. It is an algorithm that is capable of checking for pattern exhaustiveness even in very complex cases, such as when using GADTs (*Generalized Algebraic Data Type*) , a generalised form of the previously mentioned ADTs . However, in the case of *PHÔS* , this task is made more complex by constraints. Indeed, the compiler tries to verify that all cases are covered, but constraints may reduce the number of cases that are valid. Take the example shown in Listing 40: gain as a numerical value can take any value in the range $[0, \infty[$, however in this example, it can take a value only in the range $[0\ \mathrm{dB}, 10\ \mathrm{dB}]$. When looking at this code, the compiler, if it does not exploit constraints, will declare that the `match` statement on line 9 is not exhaustive, despite it being exhaustive in the context created by the constraints.

To alleviate this issue, the compiler can either force the user to always be exhaustive even when it is not necessary or can exploit constraints and utilise them to improve exhaustiveness checking. This can be done in multiple ways, including using the prover to verify that all cases are covered. However, this approach is likely to be slow and difficult to implement. Others revolved around the use of refinement types and guarded recursive data type constructors [105]. However, these techniques are not yet fully explored and are not yet integrated into the compiler architecture, and further research and experimentation are needed to determine which technique is the most appropriate for the *PHÔS* language. This topic is further discussed in Section 7.2.

```
PHÔS
1  // Performs gain on an optical signal, depending on the gain, it will
2  // either use a short gain section or a long gain section.
3  syn example(
4      signal: optical,
5
6      @range(1dB, 10dB)
7      gain: Gain,
8  ) -> optical {
9      match gain {
10         1dB..5dB => signal |> short_gain(gain),
11         5dB..=10dB => signal |> long_gain(gain),
12     }
13 }
```

LISTING 40 Example of exhaustiveness checking in *PHÔS* when using constraints. In this example, the compiler should be able to detect that the `match` statement on line 9 is exhaustive given the constraints on line 6, but it is a difficult problem to solve and requires further research.

5.4.f Constant evaluation, control flow, liveness, and pipe desugaring

After processing the HIR , it is reduced into an even simpler form based on a CFG (*Control Flow Graph*) . During this stage, almost all of the elements of the language are removed. Even conditional branching is now implemented using `goto` operations. All of this is with the aim of making the code as easy to analyse as possible. From this stage, as everything has been reduced to the most basic elements, the compiler can now do some optimisation. It can compute the values of all constants and inline them where they are used. It can also perform control flow optimisations, such as performing liveness analysis, which is the process of determining which code is used and which is not and removing the parts that are not used. Finally, it can remove pipe operators, replacing them instead of function calls, performing the pattern matching at compile time.

Using the aforementioned analysis, the compiler can now also detect whether all signals are being used, and if not, it can create an error for the user, indicating which part of the code is problematic. This is a useful feature of the compiler, as it enforces that all signals are at least *read-once*, which is part of the signal semantics discussed in Section 5.2.d. The way in which liveness and dead code elimination can be done is through the use of the CFG , as it allows the compiler to easily determine which code does not contribute to the final result and, therefore, can be removed. It detects unused signals simply by checking whether any signals are used within dead code.

Control Flow Graph A CFG , as the name implies, is a graph data structure that represents each operation being done as a node of the graph, with the different branches of the code being represented as edges. This means that all orphan sections are not accessed through the main entry point and can therefore be easily discarded. In Listing 41, one can see a simple example checking whether a number is prime (a) and its expanded version (b). The expanded version corresponds to an approximation of what code **equivalent** to the MIR would look like, with all types specified, the `for` loop replaced with a `goto` statement and labels. As was the case in previous examples of lower-level constructs, this code is not valid *PHÔS* and is purely for demonstration purposes.

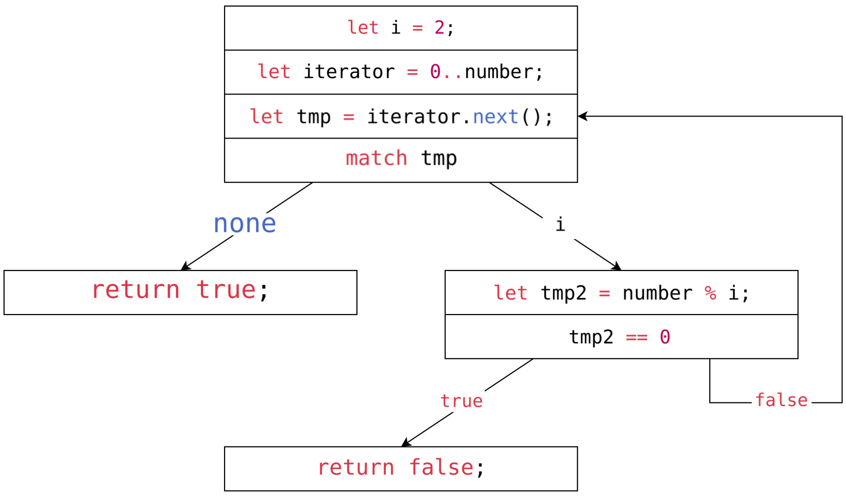


Figure 16 CFG created from the code in Listing 41. It shows the different branches of the code, with the first branch relating to the iteration from `2` to `number` and the second branch relating to the `if` statement checking whether the number is divisible by `i`. It shows that each branch is made of individual statements and that the `if` and `match` statements are represented as branches with two possible outcomes.

```
// Checks if the number is prime
fn is_prime(number: int) -> bool {
    let i = 2;
    for i in 0..number {
        if number % i == 0 {
            return false;
        }
    }

    true
}
```

```
// Checks if the number is prime
fn is_prime(number: int) -> bool {
    let i: int = 2;
    let iterator: Iterator = 0..number;

    top:
        let tmp = iterator.next();
        match tmp {
            i => {
                let tmp2: int = number % i;
                if tmp2 == 0 {
                    return false;
                }
```

```
                goto top;
            }
            none => goto ret,
        }

    ret:
        return true;
}
```

(a) (b)

Listing 41 The code example in *PHÔS* and code-equivalent representation of its MIR expanded version. Shows a function computing whether a number is prime before (a) and after expansion (b). The code in (b) is not valid *PHÔS* and is purely for demonstration purposes.

Constant inlining It is at this stage of the compilation pipeline that all constants are evaluated and replaced. The reason why *PHÔS* does this step is not for performance, as in most other languages, but for simplicity. As the VM will use a prover to verify aspects of the code in scenarios where reconfigurability is used, the code produced by the compiler must be as simple as possible to simplify this process as much as possible. For this reason, as constant evaluation is relatively easy to perform, it is done during compilation. In Listing 42, one can see a piece of code before constant inlining (a), after constant inlining (b), and after constant evaluation (c). It also shows that operations that produce constant values within the user's code are also computed, further reducing the complexity of the code.

```
const MY_CONST: int
    = 32;

fn add_my_const(
    a: int
) -> int {
    a + 2 * MY_CONST + 5
}
```

(a)

```
fn add_my_const(
    a: int
) -> int {
    a + 2 * 32 + 5
}
```

(b)

```
fn add_my_const(
    a: int
) -> int {
    a + 69
}
```

(c)

Listing 42 The code example in *PHÔS* shows the original code (a), the code after constant inlining (b), and the code after constant evaluation (c).

These operations would normally be done in the MIR stage and, therefore, would not be visible in code, but for demonstration purposes, they are shown here as valid *PHÔS* code.

Pipe desugaring The final syntactic sugar that has yet to be simplified is the pipe operator (`|>`). This operator is used on iterable tuples to pass data from one function or synthesisable block to another easily while keeping the code readable. The pipe operator performs pattern matching on its input values and into the function arguments it is piping into. Doing so is quite difficult, which is why it is performed close to the end of all transformations. At this stage, the types are all known, and operations have been simplified to the maximum. And therefore, it is the easiest point in the compilation process for the compiler to perform this transformation. In Listing 43, one can see a piece of code before pipe desugaring (a) and after pipe desugaring (b). While both of these expressions are equivalent, the former is easier to read and understand. In more complex cases, where the pipe operator is used to pattern match over multiple values, this simplification is more complex.

PHÔS

```
// Returns the sum of all inputs,
// also returns true if the sum is even
fn sum(inputs: (int...)) -> (int, bool) {
    inputs
        |> fold(0, |acc, x| acc + x)
        |> map(|x| (x, x % 2 == 0))
}
```

(a)

PHÔS

```
// Returns the sum of all inputs,
// also returns true if the sum is even
fn sum(inputs: (int...)) -> (int, bool)  {
     map(fold(inputs, 0, |acc, x| acc + x), |x| (x, x % 2 == 0))
}
```

(b)

Listing 43 The code example in *PHÔS* shows the original code (a) and the code after pipe desugaring (b).

These operations would normally be done in the MIR stage and, therefore, would not be visible in code, but for demonstration purposes, they are shown here as valid *PHÔS* code.

5.4.g PHÔS bytecode

Definition: The **bytecode** is a binary representation of the original source code that has been processed by the compiler to be verified for correctness, simplified, and optimised. It is an executable representation, made of instructions, that can be executed by the VM (*Virtual Machine*) .

Adapted from [106].

From the CFG built in the previous step, it is now relatively easy to move to a bytecode representation. This is done by replacing all simplified expressions with bytecode instructions. The bytecode of the *PHÔS* language is greatly inspired by *Java*'s bytecode [106], with the addition of a few key features, most notably special instructions representing the intrinsic signal operations, discussed in Section 4.3 and constraints which are added as special instructions on values. The *PHÔS* bytecode also removes some of the instructions that are not needed since *PHÔS* does not distinguish between 32-bit and 64-bit values. *PHÔS* is not object-oriented and does not need object-related information and instructions. Finally, *PHÔS* does not have a concept of exceptions and therefore does not need instructions related to exception handling.

Because of these properties, the instruction set of the *PHÔS* language is fairly simple. Additionally, some instructions are more generic than in *Java*. For example, *PHÔS* does not distinguish between integer and floating-point operations and therefore has a single instruction for arithmetic operations, which can be used for both integer and floating-point values. The *PHÔS* bytecode is also stack-based, meaning that all operations are performed on a stack, and all values are pushed and popped from the stack. This will be discussed in more detail in Section 5.5.b. The full instruction set of the *PHÔS* VM (*Virtual Machine*) can be found in Table 11.

Finally, along with the bytecode, the previously built CFG is also included, with its node now replaced with the position of the relevant instructions. The reasoning behind this inclusion is as follows: as the VM will need to prove that some branches can be taken while others cannot, it will need to build branching information either way. Instead of having to rebuild the

CFG in the VM , it is instead packed along with the bytecode as a means of reducing computation time. This gives a dual purpose to the CFG , as it is used both in the compiler and in the VM .

> **Note:** It is likely that as the development of the *PHÔS* language continues, the instruction set will be expanded and refined, and therefore the instruction set shown in Table 11 may not be the final instruction set of the *PHÔS* VM (*Virtual Machine*) .

Instruction	Stack operations [before] → [after]	Description

`call_fn[`
`    <function_id>`
`]`
BC (*Bytecode*)
	`[`
	`    arg0,`
	`    arg1,`
	`    ...`
	`] → result`
		Calls the function with the given ID, the arguments are obtained from the function definition and then popped from the stack, and the result of the function call is pushed onto the stack.

`call_method[`
`    <type>,`
`    <function_id>`
`]`
BC
	`[`
	`    arg0,`
	`    arg1,`
	`    ...`
	`] → result`
		Calls the method with the given ID on the given type, the arguments are obtained from the function definition and then popped from the stack, and the result of the function call is pushed onto the stack.

`goto[`
`    <label>`
`]`
BC
	`[] → []`	Jumps to the given label. Used when branching.

`pop[`
`    <n>`
`]`
BC
	`[ a0, a1, ... ] → []`	Pops the given number of values `n` from the stack and discards them.

`repeat[`
`    <n1>,`
`    <n2>`
`]`
BC
	`[ a0, a1, ... ] → [`
	`    [a0, a1, ... ],`
	`    [a0, a1, ...] ,`
	`    ...`
	`]`
		Repeats the `n2` top values of the stack `n1` times and pushes the result onto the stack.

`const[ <value> ]` BC	`[] → [ <value> ]`	Pushes the given constant value onto the stack, can be an `int/uint`, a `float`, a `bool`, a `string`, a `char`, a `complex`, or a function, which are used for passing closures to other functions.

`return[]` BC	`[ a0, a1, ... ] → []`	Returns from the current function, popping all of the remaining values from the stack and returning them as a tuple. If the stack is empty, it returns an empty tuple, which is equivalent to the `none` value.
`none[]` BC	`[] → [ none ]`	Pushes the `none` value onto the stack.

`load[ <id> ]` BC	`[] → [`
	`a0,`
	`a1,`
	`...,`
	`len`
	`]`
		Loads the value with the given local variable ID onto the stack. If it is a tuple, it expands the tuple into `len` values on the stack and pushes the tuple length onto the stack. The first `n` local variables are reserved for the arguments of the function, where `n` is the number of arguments of the function.

`store[`
`<id>,`
`]` BC
	`[`
	`a0,`
	`a1,`
	`...,`
	`len`
	`] → []`
		Stores the top `len` values of the stack into the local variable with the given ID. If `len` is more than one, then store them as a tuple. The first `n` local variables are reserved for the arguments of the function, where `n` is the number of arguments of the function.

`get[`
`<type_id>,`
`<field_id>`
`]` BC
	`[ instance ] → [`
	`a0,`
	`a1,`
	`...,`
	`len`
	`]`
		Gets the field with the given ID `field_id` from the given type `type_id`, and pushes it onto the stack. If it is a tuple, expands the tuple into `len` values on the stack, also pushes the `len` of the tuple onto the stack. Pops the instance of the type from the stack.

`push[`
`<type_id>,`
`<field_id>,`
`],` BC
	`[`
	`instance,`
	`a0,`
	`a1,`
	`...,`
	`len`
	`] → [ ]`
		Stores the top `len` elements from the stack into the field with the given ID `field_id` from the given type `type_id`. If it is a tuple, expands the tuple into `len` values on the stack. Pops the instance of the type from the stack.

`new[`
`<type_id>,`
`<variant_id>,`
`],` BC
	`[`
	`a0,`
	`a1,`
	`...`
	`] → [ instance ]`
		Creates a new instance of the variant `variant_id` of given type `type_id`, and pushes it onto the stack. The arguments are obtained from the type definition, and then popped from the stack. For structs, the `variant_id` are ignored.

```
branch[
    <false_offset>
]
```

BC

```
[
    a0,
] → []
```

Takes the top value from the stack, if it is `false`, then jumps to the given offset. Otherwise, continues execution at the next instruction.

```
flag[ <flag> ],
```

BC

```
[] → []
```

Gets the given flag, and pushes it onto the stack as a boolean value. The flags are produced by the previous operation. The valid flags are `overflow`, `underflow`, `div_by_zero`, `invalid`, `inexact`, `unimplemented`, `unreachable`. Used for branching.

```
unary[
    <op>
]
```

BC

```
[ a0, ] → [ a1 ]
```

Takes the top value from the stack, applies the given unary operator `op` to it, and pushes the result onto the stack. The valid unary operations are numerical negation (`-`), logical negation (`!`), and bitwise negation (`~`).

```
binary[
    <op>
]
```

BC

```
[ a0, a1 ] →
[ a2 ]
```

Takes the top two values from the stack, applies the given binary operator `op` to them, and pushes the result onto the stack. The valid binary operations are addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), modulo (`%`), exponentiation (`**`), bitwise and (`&`), bitwise or (`|`), bitwise xor (`^`), bitwise left shift (`<<`), bitwise right shift (`>>`), equality (`==`), inequality (`!=`), less than (`<`), less than or equal (`<=`), greater than (`>`), greater than or equal (`>=`), logical and (`&&`), and logical or (`||`).

```
cast[
    <type_id>
]
```

BC

```
[ a0, ] → [ a1 ]
```

Takes the top value from the stack, cast it to the given type `type_id`, and pushes the result onto the stack. The valid conversions are `int`, `uint`, `float`, `bool`, `string`, `char`, and `complex`. Only primitive values may be converted in this way.

`insert[]` BC	`[`
	`value,`
	`offset,`
	`len`
	`] -> [ ... ]`
		Inserts the given `value` into the stack at the given `offset`, `len` times. This allows the insertion of values into the middle of the stack, as well as interspersing values into the stack. In the cast of the stack `[a0, a1, a2, "hello", 1, 3]`, calling `insert` will result in the stack `[a0, "hello", a1, "hello", a2, "hello"]`.

`intrinsic[`
`<intr>`
`]` BC
	`[ a0, a1, ... ] →`
	`[ a2, a3, ... ]`
		Execute the given photonic intrinsic operation `intr`, see Section 4.3, based on the intrinsic value, pops the arguments from the stack, and pushes the result onto the stack.

`constraint[`
`<constr>`
`]` BC
	`[`
	`signal,`
	`a0,`
	`a1,`
	`...`
	`] → [ ]`
		Applies the given photonic constraint `constr`, see Section 4.4, based on the constraint value, pops the arguments from the stack, and pushes the result onto the `signal`.

Table 11 Instruction set of the *PHÔS* VM (*Virtual Machine*) . Showing the instruction and its static arguments (i.e. arguments that are produced during compilation) and the operations that each instruction does on the stack.

5.4.h Compiler complexity

As one can see from the previous sections, the *PHÔS* compiler is more complex than one might expect. However, there are good reasons why the compiler is so complex. A lot of the features that have been discussed in Section 4 and in this chapter are rather difficult to implement. They require a lot of modern features and tight coupling with provers for constraints and intrinsic. These tasks are not easy in isolation, but when they are combined, they become even more difficult to implement. When taking this into account, the complexity of the compiler is not that surprising. Essentially, the compiler simplifies the code as much as reasonably possible, such that the resulting bytecode is easy to interpret and execute, easy to collect stacks for tunable values, and such that it is easy to process using a prover. Additionally, this complexity makes the compiler modular, which allows for easy extension and rework of the language, something that will need to be done as the language evolves.

> The *PHÔS* compiler translates source code into a bytecode format used by the VM for evaluation. It first turns the code into a computer-understandable representation called the AST . Then the AST goes through three transformation stages. Called the HIR , the MIR , and finally, the bytecode.

5.5 The virtual machine

After investigating the components and stages of the compiler, the analysis of the VM (*Virtual Machine*) can proceed. Recalling the execution model of *PHÔS* , discussed in Section 5.2.a, the role that the VM fills is the evaluation of the bytecode. Therefore, the behaviour of the virtual machine is discussed, including how the VM works, how it uses a stack for computation, and how it executes the bytecode. Additionally, the section discusses the result of the evaluation, namely, the tree of intrinsic operations, collected stacks, and constraints. However, the suitability of existing virtual machines must also be discussed, as it is important to understand why an existing virtual machine was not used.

5.5.a Why not use an existing VM?

One may wonder why *PHÔS* does not use an existing virtual machine and requires a custom-built one. The reason for this is that existing VMs are not suitable for the semantics and execution model of *PHÔS* . This can be explained by looking at the artefacts produced by the execution process, previously shown in Figure 10. It must produce three components, all of which would be hard, if not impossible, to properly extract and process within an existing implementation.

Stack collection One of the key functions of the virtual machine is to detect which parts of the code cannot be evaluated based on tunable values, as was discussed in Section 4.5 and collect them to be included in the user HAL . This requires tight integration in the VM , as it must support doing partial computation and collect all of the instructions it cannot execute. This is not a feature that is supported by any existing VM that was investigated, and it would be difficult to implement in an existing VM .

Constraints and intrinsic Another feature of the *PHÔS* VM is the ability to collect constraints and intrinsic operations and to produce a tree of these values, representing the signal flow of the circuit. While this can be implemented in a traditional language, it would require an extensive library to be included along with the bytecode, which would make the bytecode harder to generate. Additionally, this means that these operations would no longer be expressed as dedicated bytecode instructions but rather as regular function calls, tightly coupling the aforementioned library and the bytecode, significantly increasing the burden of maintenance and development of the language.

Simplicity Existing VMs often support many features that are simply not needed for *PHÔS* , such as object-oriented programming or memory recollection schemes like garbage collection. *PHÔS* is not a general-purpose language and as such, can work with a limited set of features. This means that the complexity of the VM can be significantly reduced to only contain the elements relevant to *PHÔS* . This can help improve performance and reduce the size of the VM , making it easier to distribute to users.

Licensing Finally, existing VMs may be subject to licenses. While intellectual property has not been discussed in this document, it is essential to be aware of issues that can arise when using other people's code or, more generally, intellectual property. Therefore, if the VM is written from scratch, it can be licensed in a way that is compatible with the *PHÔS* license, whichever that may be.

5.5.b Stack-based architecture

> **Definition**: A **stack** is a data structure that follows the last-in-first-out (LIFO) principle. This means that the last element that was added to the stack is the first one to be removed. Stacks usually only implement two operations: `pop` to remove the last element and `push` to add an element to the top of the stack.
>
> Adapted from [107].

So far, the mention of the *stack* has been made several times. However, no clear definition has been provided. This definition, along with the reasoning behind the choice of a stack-based architecture, is discussed in this section. The stack is a data structure that holds the values required during the evaluation of a given block of code. The stack is used to store all intermediary values needed for computation. When a new value is needed, it is pushed onto the stack, and when a bytecode instruction is executed, full list in Section 5.4.g, the instruction pops the elements that it needs from the stack, processes them, and once it is done, it pushes all of the output values to the stack again. Later on in this section, an example is provided showing the execution of a simple function.

The stack is convenient because it can be very effectively implemented; it is a common data structure that is easy to implement and very fast. Additionally, it means that all short-lived values are stored inline and do not require any additional memory allocation. This is important, as it means that the VM does not need to implement a garbage collector, which would be a significant burden on the development of the VM . Furthermore, the stack also plays very nicely with the automatic memory management scheme used by *Rust* – the language in which *PHÔS* will be implemented – as it allows values that are removed from the stack and no longer used to be automatically discarded, further simplifying the implementation of the VM .

One of the special aspects of the *PHÔS* VM that one may notice from the list of instructions in Section 5.4.g is that arrays of values in *PHÔS* are all pushed to the stack and that *PHÔS* allows for quite complex operations on the stack. The reasoning behind this decision is to make the VM very powerful and to be able to express complex operations in relatively few instructions. This allows the work of the user HAL generator, discussed in Section 5.6.c, to be even easier. It also means that fewer instructions must be provided to the prover for branch elimination and constraint checking. Therefore the interface between the prover and the VM will be easier to create.

5.5.c Signals, constraints, and intrinsics

As previously mentioned, one of the tasks of the VM is to collect the different intrinsic operations and their constraints to build a tree representing the signal processing. This is done through the `intrinsic` and `constraint` instructions. These special instructions take signals and arguments, and instead of only pushing results to the stack, they also can internally add information to a global tree of signals. This is done transparently for the user and is how the VM can store these signals. In Listing 44, one can see a simple program splitting, filtering and then adding gain to a signal. Then in Figure 17, one can see the signal processing tree with the added constraints. In cases where reconfigurability would also be present, collected stacks would also be appended to a special reconfigurability node, which would be used to differentiate the different configurations of the circuit.

```
// Processes a signal by splitting it and
// filtering one of the split branches.
syn process(
    input: optional
) -> (optional, optical) {
    input |> split((0.5, 0.5))
        |> map(filter(1550 nm, 10 GHz))
        |> (_, gain(10 dB))
}
```

LISTING 44 Code example in *PHÔS* , showing a simple photonic circuit splitting a signal, then filtering both signals and finally adding gain to one of the signals.

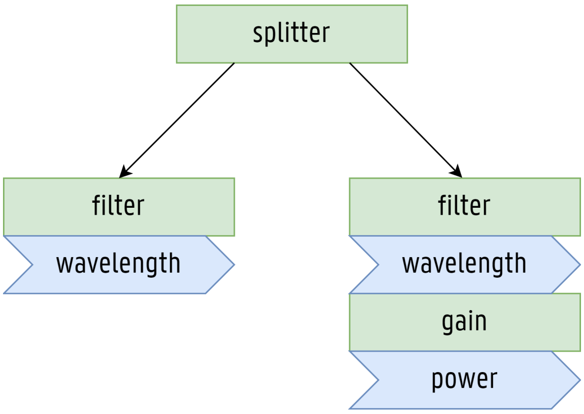


FIGURE 17 The signal flow diagram of Listing 44 shows the splitting of the input signal, followed by the filtering of both signals, and finally, the gain of one of the signals. The green boxes represent the intrinsic operations, the blue boxes represent the constraints, and the arrows represent the flow of the signal.

5.5.d EXAMPLE OF BYTECODE EXECUTION

In this section, a simple example of code will be shown, along with its resulting bytecode. The code is shown in Listing 45 (a), and the resulting bytecode is shown in Listing 45 (b). The function being compiled is a simple accumulating sum that also computes whether the sum is even or not. In (b), one can see the bytecode contains a total of 22 instructions. The instructions would be purely binary values. However, they are shown as text for convenience and readability. One can see that the two closures at line 5 and 6 were respectively turned into two anonymous functions called `__anonym_0` and `__anonym_1`, normally these would have numeric IDs instead of names, but names are provided for clarity. This code uses the `load`, `pop`, `binary`, `return`, `repeat`, `const` and `call_fn` instructions, which can all be found in the previously shown Table 11.

An example of execution and the state of the stack will now be shown. The function `sum` will be called with the iterable tuple `(1, 2, 3, 4, 5)`. The execution diagram of the `sum` function can be seen in Annex C, in Figure C.1. It shows the stack after each step of the execution of the function. It also gives symbolic meaning to the values, showing integers as `int(x)`, length as `len(x)`, and functions as `fn(x)`. From this figure, one can see that the `load` instruction pushes all of the argument `inputs` values onto the stack, followed by the length of the argument 5 in this case. Then, when constants are pushed, the whole stack moves up, and a new value is added. When calling functions, the whole stack is consumed, and the result is pushed onto the stack. The `pop` instruction is used to remove the top value from the stack, and the `return` instruction is used to return the top values from the stack. Additionally, one can see that the `const[ 1 ]` being performed is used to add the length of the argument before calling `map`. This is because map expects an iterable tuple as an input, and produces an iterable tuple. The result of this execution is then returned with the tuple `(15, false)`.

PHÔS

```
// Returns the sum of all inputs,
// also returns true if the sum is even
fn sum(inputs: (int...)) -> (int, bool) {
    inputs
        |> fold(0, |acc, x| acc + x)
        |> map(|x| (x, x % 2 == 0))
}
```

(a)

BC

```
fn @__anonym_0(@0: int, @1: int) -> int
  load[ @0 ]    pop[ 1 ]
  load[ @1 ]    pop[ 1 ]
  binary[ + ] return[]

fn @__anonym_1(@0: int) -> (int, bool):
  load[ @0 ]     pop[ 1 ]
  repeat[1, 1]   const[ 2 ]
  binary[ % ]    const[ 0 ]
  binary[ == ]   return[]

fn @sum(@0: (int...)) -> (int, bool):
  load[ @0 ]               const[ 0 ]
  const[ @__anonym_0 ] call_fn[ @fold ]
  const[ 1 ]               const[ @__anonym_1 ]
  call_fn[ @map ]          pop[ 1 ]
  return[]
```

(b)

LISTING 45 Code example in *PHÔS* , showing the original code (a), and the bytecode after compilation (b).

The bytecode would normally be, as the name implies, binary. However, here it is shown in a textual format for clarity.

5.5.e Partial evaluation

Definition: **Partial evaluation** is a technique for specialising a program with respect to some of its arguments. The result is a new program that only requires the remaining arguments to run. The new program is generally smaller.

Adapted from [108]

It has been shown that the VM executes bytecode. However, one may wonder how the VM handles tunable values. The answer is that the VM will use *partial evaluation*. Meaning that the VM will collect the code impacted by the tunable values and will try and evaluate as much of the code as possible while leaving the code. It cannot evaluate as is. This means that the VM will produce a new program that performs the same functionality as the user's original program but is specialised based on the static inputs, needing only the tunable values as inputs for it to be complete.

Additionally, it will still analyse the intrinsic operations – impacted by tunability – present within the user's design and collect them separately so that they can still be synthesised. These operations also depend on tunable values, but using the constraints on those tunable values, if any, the compiler will still be capable, in most cases, of synthesising them into a photonic mesh. This will allow the VM to produce a special subtree of the signal flow tree, which represents tunable sections and their intrinsic operations and constraints but requires tunable values for finalisation.

Tunability failures In some cases, if tunable values are not sufficiently constrained, the synthesis of these components may fail. In such cases, the user will be invited to provide more constraints, such that the synthesiser can produce the photonic mesh for the given intrinsic operation. If, however, the user were not to be able to provide these additional constraints, they would need to rework their design either to avoid the use of broad range tunable values or to be able to provide the additional constraints. Therefore, one may understand tunable values as a tool to help the user tune their photonic circuit but not as a tool for broad-range reconfiguration.

Broad-range reconfiguration As previously mentioned, if tunability has failed, the user must be able to constrain their tunability values more, in some cases, this may prove difficult. However, *PHÔS* also provides the ability of creating reconfigurability regions, which are regions of the photonic circuit that can be reconfigured. The way in which the user may be able to constrain their tunable values, is by using reconfigurability regions. If they can partially constrain their tunable values, such that it can be used for reconfigurability, in addition to tunability, then the synthesiser will be able to produce a photonic mesh for their design.

Looking at Listing 46, in (a), one can see a circuit that relies on broad-range reconfigurability if parameter `gain` is not constrained, and the platform only supports a short gain section in the range $[0\mathrm{dB}..5\mathrm{dB}]$, or a long gain section in the range $]5\mathrm{dB}, 10\mathrm{dB}]$, the synthesiser will fail to make the circuit. However, if the user were to constrain the `gain` in the range supported by the platform then match on the gain to create either a `short_gain` section or a `long_gain` section, the code would be synthesisable, this second case is shown in (b).

However, when looking at this code (b), one might think that it is much longer than the original code (a), however, most of this complexity would actually be contained within the standard library, and the user would only need to add the `range` constraint on their gain to match their platform's capabilities.

PHÔS

```
syn my_module(
    input: signal,
    the_gain: Gain
) -> signal {
    input |> gain(the_gain)
}
```

(a)

PHÔS

```
syn my_module(
    input: signal,
    @range(0 dB ..= 10 dB)
    the_gain: Gain
) -> signal {
    match the_gain {
        0 dB..=5 dB => input
            |> short_gain(the_gain),
        5 dB..=10 dB => input
            |> long_gain(the_gain),
    }
}
```

(b)

Listing 46 | Code example in *PHÔS* , showing the original unsynthesisable code (a), and the fixed code (b)

5.6 Synthesis

Now that the first two steps in the overall synthesis of a *PHÔS* design, namely compilation and evaluation, have been explained, the last step is to synthesise the design. This step is likely to be the most complex, and a lot of work is still needed both at the design stage, and at the algorithm stage. However, the goal of this section is to explain the general idea behind the synthesis of a *PHÔS* design. Therefore, this section can be assumed to be less precise and formal than previous ones, focusing more on overall ideas and concepts, rather than on specific details.

The core goal of the synthesis stage is to take the signal flow tree produced by the VM and turn it into two things: the user HAL that the user can use to interact with their design and the binary programming files used by the actual hardware. These two goals are very different, and the simplest is the generation of the user HAL . Generating the binary programming files requires processing all of the constraints and intrinsics into gate descriptions, then placing them on the chip and routing between them. This is a problem that is already incredibly difficult for traditional FPGAs to solve, but it is exacerbated by both the two modes that can be supported in waveguides also by the recirculating hexagonal nature of the mesh. Therefore, the synthesis of a *PHÔS* design is incredibly difficult and computationally expensive and is still an active area of research.

Among the ongoing work that has been happening, including at the PRG , the modelling of the circuit meshes in a graph structure has been done [109]. In Figure 18, one can see the graph representation of a single photonic gate and a junction in a hexagonal photonic mesh, and in Figure D.1, one can see a set of gates in a mesh. In their work, *Xiangfeng Chen et al.* show that by incorporating relevant metrics in the mesh edges, they can achieve efficient routing in a photonic mesh. This is a very promising result and can be used as the basis for future research. Indeed, research is already ongoing at Ghent University to further this routing. However, they are not yet at the point of incorporating more complex photonic components inside of the mesh [110]. Other work has implemented the automatic realisation of circuits on photonic meshes, which is closer to what is needed for synthesis, but it is still incomplete [111].

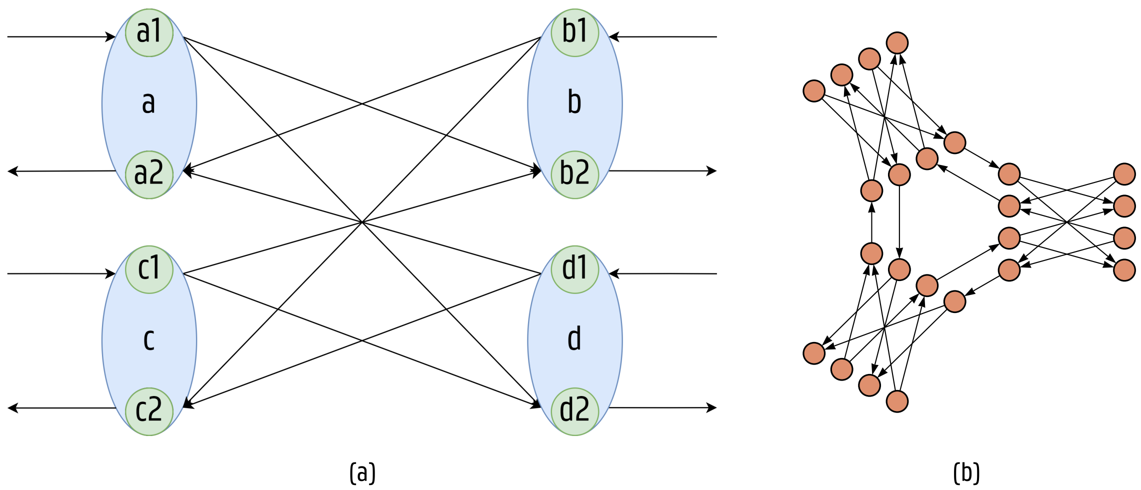


Figure 18 Graph representation of a single photonic gate (a) and of a complete junction in a photonic mesh (b). Based on the work of *Xiangfeng Chen et al.* [109]. (b) is composed of three unit cells shown in (a), showing the direction that light is travelling in and all of the possible connections.

5.6.a From intrinsic operations to gates

The first step in synthesising the circuit, is determining for each intrinsic operation being done in the circuit, what gates are required to implement it. Some intrinsics have one-to-one mapping with photonic gates, such as phase shifters, splitters, and couplers, for these this tasks should be relatively easy. However, other intrinsic operations, such as modulators, detectors, and sources are not part of the mesh and are, in fact, components placed on the edges of the mesh. Finally, other intrinsic operations are actually compounded operations, that can result in more than one photonic gate. This means that each type of intrinsic operation, will need to be decomposed into their component photonic gates. Some of them, such as edge devices, do not need to become a specific gate, rather they need to be assigned a location, such that they can be routed to during place-and-route.

Filter synthesis Filters are purposefully made into their own intrinsic operation despite being compound components. As was explained in Section 4.3, filters may be optimised based on the platform, for example, some platforms may even have built-in tunable filters placed on the edge. Therefore, the platform is responsible for the synthesis of the filters. As the synthesiser is provided with the input wavelength constraint, and the expected wavelength response, wether it be a bandpass filter, or any other filter, it should be able to synthesise the filter into its component gates. In some cases, it is possible that filter synthesis might fail, in such cases, the synthesiser should produce an error for the user.

Tunability At this point, some components may depend on tunable values, nonetheless, the synthesis tool must be capable of handling tunable components. Meaning that it must understand that components are tunable within a certain range, indicated with constraints, and still produce the appropriate gates. This task should generally be relatively similar to regular intrinsic-to-gate translation, assuming that the parts of the standard library implemented for the platform were done correctly. Indeed, the task of separating widely tunable values into smaller tunable ranges, is in parts, the task of the designer, in other parts, the task of the standard library as it is implemented by the chip designer. This means that, at this point in the synthesis pipeline, all tunable intrinsic operations should be synthesisable. If they are not, then the platform-support package would be to blame, since it would mean that its implementation of the standard library is invalid.

5.6.b Place-and-route

As was previously discussed, there are currently no algorithms that can place and route all of the components that can be present in a mesh. Since *PHÔS* provides the constraints on each signals to the place-and-route engine, it is expected that it will utilize those constraints to improve its placement and routing. Furthermore, the place-and-route algorithm will be provided by optimisation targets by the user, these targets should be an indication of what matters most for the user: the area that a given circuit occupies, the power consumed by the circuit, or the optical losses in the circuits. Additionally, in some cases, the user may want to create their own metric for further customization. Finally, despite there being no place-and-route algorithm, there are a number of routing algorithms that are being developed, including at Ghent University, which are showing promising results [112]. Once routing has been improved, these algorithms may be able to be included in place-and-route implementations.

5.6.c Hardware abstraction library

Another task of the synthesiser is to generate the user HAL . To do this, it will use pre-made routine, that have yet to be designed, coupled with the platform-support package, which will provide information with regards to interconnecting the generated, high-level user HAL and the low-level core HAL provided my the chip designer. This task is expected to be relatively simple, as it is mostly a matter of connecting the dots between the two HAL . Additionally, the user HAL would be generated in *Rust*, in a way that is compatible with FFI (*Foreign Function Interface*[4]) for interoperability with *C* and *C++*.

At this point, little is known about the exact way that the synthesiser will work, however, this section has hopefully provided pointers that may be used in future research for the implementation of this stage.

5.7 Constraint solver and provers

In the previous sections, the mention of the constraint solver and of the use of a prover has been discussed extensively. However, the exact way in which these tools will be used has not been discussed. This section will attempt to provide a brief overview of the way that these tools will be used. First, the constraint solver will be discussed, followed by the prover.

5.7.a Constraint solver

The constraint solver is a software that can, given a set of intrinsic operations and constraints, solve the state of a signal at any point in the signal chain. It does this in one of two modes: in frequency domain analysis, it will only look at a subset of relevant intrinsic, and compute the spectrum at each step of the signal chain. In time domain analysis, it will process complex amplitude signals modulated onto carrier wavelength, and at each time t, it will process the effect that each constraint has on the signal.

Limitations of frequency domain analysis In frequency domain mode, the constraint solver must have the values of all tunable values set before it starts executing. This is because, for proper frequency domain analysis, the system must be time invariant, meaning that it must be in steady state. While it is possible, assuming slow varying tunable values, to perform frequency domain analysis, it is currently not planned, and the constraint solver is not yet designed with this in mind.

Co-simulation Through the marshalling layers, which will be discussed in Section 5.8, the constraint solver will be able to communicate with user code, in order to co-simulate both the user's software, and the user's photonic design. This will allow the user to test their design programmatically rather than manually. This will also allow the user to simulate tunability and reconfigurability.

Simulating tunability The constraint solver is capable of simulating tunability in the time domain, by updating the signal flow graph based on the tunable values, it can easily reflect any changes in the tunable values. This can be leverage, in combination with co-simulation, to test whether the user's feedback loops work as intended. Allowing simulation and verification of the overall design, rather than just parts of it. All of the values that might still need to be computed, can be done using the VM , since the VM produces a partially evaluated bytecode, the constraint solver only needs to provide the VM with the values of the tunable values to obtain the signal flow graph.

5.7.b Prover

Definition: **SMT problems** are decision problems of logical formulas with respect to combinations of background theories. This means that it verifies whether mathematical formulas are satisfiable.

Adapted from [113].

[4]Foreign Function Interface: a way to call functions from other languages

Theorem provers like *Z3* are called SMT provers, they can be provided with set of theories and rules and verify whether they are satisfied [113]. Provers like *Z3* are especially well suited for program verification, which is the area of interest in this thesis. In the case of *PHÔS* , the prover is expected to be used for multiple areas: verifying constraint compatibility, verifying that tunable code respects constraints, verifying exhaustiveness of pattern matches, and determining which reconfigurability branches are reachable. In the following sections, each of these use cases will be discussed.

> **Note:** The features discussed in this section are all complex, and while they may appear simple on the surface, translating them into SMTs is a complex task, and will require further research.

Constraint compatibility A prover can express the mathematical relations between constraints; this means that a prover like *Z3* can be used to check whether two constraints are compatible. This would be used as part of the compilation and evaluation processes.

Tunable code Tunable code is turned into a partially evaluated program, as discussed in Section 5.5.e, these programs can be fed into a prover, which can verify whether the program, irrespective of its inputs, respects the constraints that were provided to it. This would be used as part of the evaluation process.

Exhaustiveness As was discussed, when presented with constraints, it is very difficult for the compiler to verify exhaustiveness, however, a prover can be used to verify whether a pattern matching expression is exhaustive given a set of constraints. This would be used as part of the compilation process.

Reachability The evaluation stage, when encountering tunability, may have more reconfigurability states than needed given the current constraints. Just like with exhaustiveness checking, a prover can be used to verify whether branches are even reachable. If they are not, then they can safely be discarded, and the synthesiser will have less work to perform.

5.8 Marshalling library

Now that all of the synthesis steps of the *PHÔS* programming language have been discussed, one must now focus on interoperating all of these components. As well as how *PHÔS* can leverage existing software. This section will discuss the marshalling library, which is the component that will be used to interconnect all of the elements of the *PHÔS* ecosystem.

What's in a name? Marshalling is a term used in Computer Science to describe the transforming of representation of objects into formats suitable for transmission [114]. Indeed, in the case of *PHÔS* , the marshalling library will be used to move data around between the different step, but also be used to allow the user to configure each step in the synthesis chain to their requirements. In that way, it performs both the traditional marshalling role – that of assembling and arranging [115] – but also the Computer Science term of transforming and moving data. Therefore, the goal of the marshalling library is to facilitate moving the data around the different pieces of the *PHÔS* ecosystem in a programmatic way. Thinking back to the ecosystem analysis performed in Section 3, this is a replacement to the build tools and the compiler. Offering ways for the user to programmatically and dynamically configure the different steps of the synthesis chain.

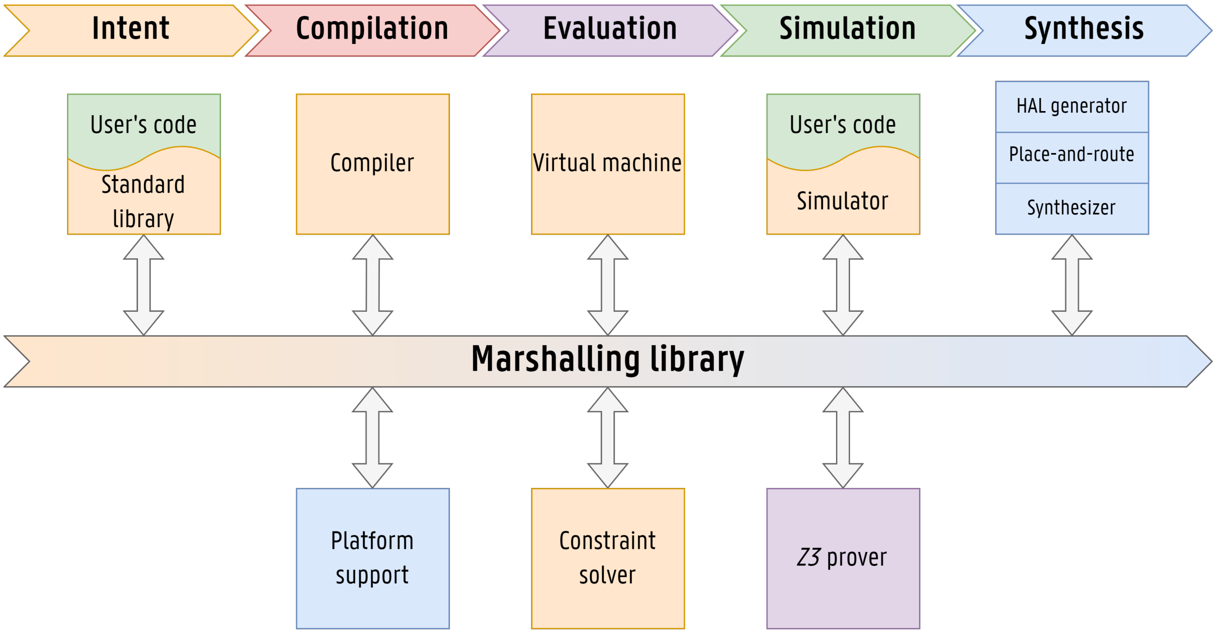


Figure 19 Overview of the marshalling library, it shows all of the different components of the synthesis toolchain of *PHÔS* , and how they are interconnected using the marshalling library. Additionally, it also shows the simulation stage, which would couple user simulation code with the simulator. Below the marshalling library are all of the common components that do no belong to one particular stage of the synthesis toolchain.

The color scheme used in the same as the one used in Figure 9: blue represents the responsibility of the chip designer, orange the responsibility of the ecosystem developer, green the responsibility of the user, and purple are external tools.

Overview In Figure 19, one can see the overview of the entire toolchain proposed in this thesis, it shows all of the different components that have been discussed so far, all interconnected using the marshalling library. It shows that all components are interconnected through this library, and that the marshalling library is the only component that is aware of all of the other components. Additionally, the marshalling library provides all of the data required by a given component to perform its task, this means that the user should easily be able to intercept the data, and modify it to their needs. This is the primary advantage of the marshalling library: providing an easy way of communicating, configuring, and tuning the synthesis process.

Choice of language During the discussion on ecosystems, in Section 3, *Python* was shown to be a good candidate as a language to create libraries in, and as such, *Python* is a good candidate for writing the marshalling library in. As the marshalling library is not a performance critical section, nor expected to be particularly complex, it can be written in *Python* such that the user can easily script the synthesis toolchain, using a common academic language.

5.8.a Example

> The marshalling library does not exist yet, therefore this example is a mockup of what the final library may look like, and how it may be used.

Due to its length, the code of this example is shown in Annex E, where the *PHÔS* code being simulated is shown in Listing E.1, the code to build the modulate into a programmable form is in Listing E.2, and the code to simulate the modulator is in Listing E.3. In this example, a simple *PHÔS* circuit is being built, it consists of a splitter of which one of its outputs is modulated by a PRBS 12-bit sequence. In Figure 20, one can see the result of the simulation code, showing the modulate output in blue, and the unmodulated output in orange. One can see that the noise source is applied to both signals, but that the modulated signal is modulated by the PRBS sequence. Additionally, the circuit can be seen in Figure E.1, showing the generated mesh on a rather large chip, showing the ports, the modulator, and the splitter. On that figure, one can also see that the place-and-route engine may utilize the two modes of the waveguides to perform more efficient routing. In practice, when looking Figure E.1, one may notice that the losses experienced by the modulated signals, which should be significantly higher, are not modeled in the simulation shown.

Synthesis One can see in Listing E.2, that the user starts by importing the marshalling library `phos` (line 2), and their device support package `prg_device` (line 5) – a fictitious device from the PRG . A device instant is then created from the `phos` library and the device support package (line 8). From this, the module can be loaded from its `phos` file. Moving on to the creation of the inputs and outputs (I/O) of the device (lines $14 - 17$), the electrical input is created, with its device-specific identifier being 0. Then, each of the three optical ports are created, depending on whether they are used as inputs, outputs, their remaining port is discarded. Here as well, the device-specific ID is being used. The reason why device-specific IDs are being used is to assign the ports of the device to the logical ports of the module. Then, the module is instantiated, given a name, and all of its inputs and outputs are assigned. It can then finally be synthesised. In a real design, one would likely specify more parameters and more than one module, indeed, the marshalling library can be used to compose modules together, and to synthesise more than one module. In this example, the synthesis stage has only one parameter set, the optimisation of the design set to area optimisation. Finally, from the synthesised design, the user HAL and programming files can be generated.

Simulation This example shows that a PRBS sequence is generated in a *Numpy* array, it shows one of the core goals of the marshalling library: broad compatibility with the existing *Python* ecosystem. A simulator is then created from the device, a noise source and a laser source are created, from which the design can be simulated using the previously instantiated module. This runs the simulation, and the result can be plotted using libraries such as *Matplotlib*, giving the result seen in Figure 20.

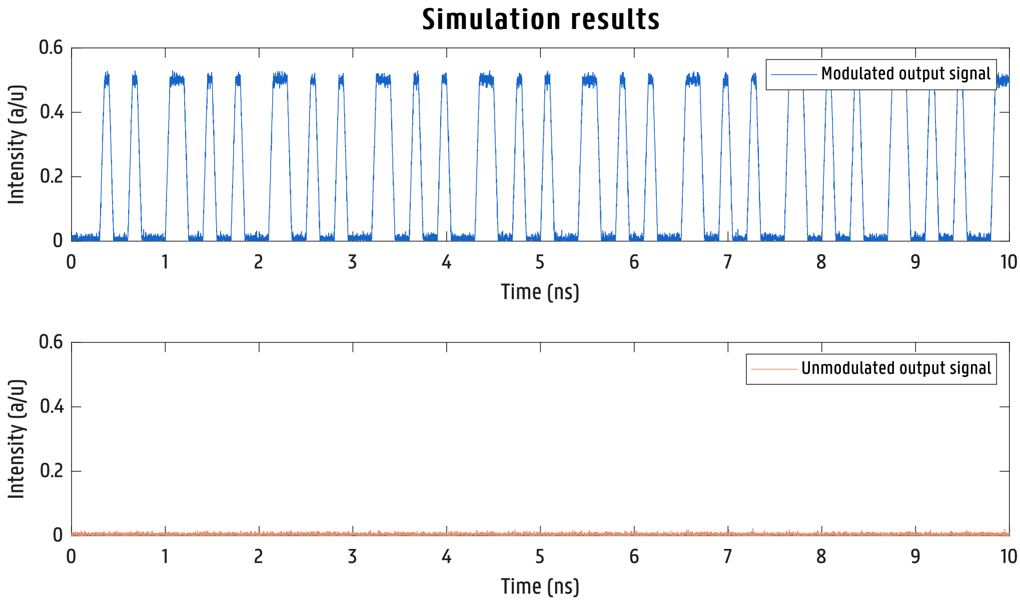


FIGURE 20 Simulation results of the marshalling layer example, showing the output of the modulator, and the output directly from the splitter. The output of the modulator is the same as the output of the splitter, but with the PRBS sequence modulated onto it.

> The marshalling library aims at providing an easy-to-use, productive interface for configuring, synthesising, and simulating *PHÔS* circuits. Through the use of a *Python* API , it makes it easy for people with relatively little programming knowledge to get started. Finally, its ability to reuse existing libraries from the *Python* ecosystem makes it easy to integrate into **existing workflows**.

6.

Examples of photonic circuit programming

Several different application areas were mentioned in Section 2.1.c, and in this section, some of these areas will be demonstrated using the *PHÔS* programming language. The examples are meant to be mockups of real applications and not complete implementations, focusing solely on the *PHÔS* part of their design. These examples will be explored in different levels of detail depending on the complexity of the application and how much of the capabilities of *PHÔS* they demonstrate. The full code, comments, and type annotations are available starting at Annex F.

6.1 Beamforming

Optical beamforming is being used to build new solid-state LiDAR systems [116]. These LiDARs need precise phase matching between all of the produced optical signals, as any imprecision over the phase and delay will negatively impact the precision of the overall system. Conveniently, *PHÔS* offers an easy way to ensure that signals are phase and delay matched: the `constrain` synthesisable block. It imposes a differential constraint over a number of signals, in this case, as will be visible in Listing 47, it is used to enforce equal phase into the modulators, and equal delay when going back towards the outputs.

6.1.a Theoretical background

Beam forming allows a system to control the directionality of a signal emitted by its antennae. It requires multiple antennae at the transmitter. The transmitter then controls the phases of the emitted signals to create constructive interference in the desired direction of interest and destructive interference in the others. This allows the transmitter to focus its signal in a specific direction. This has several advantages, it can allow a transmitter to reach longer distances at the same transmitted power, it can be used to decrease interference with other transmitters, and it can be used to increase the directional precision of a system, such as in the case of a LiDAR [117, 118].

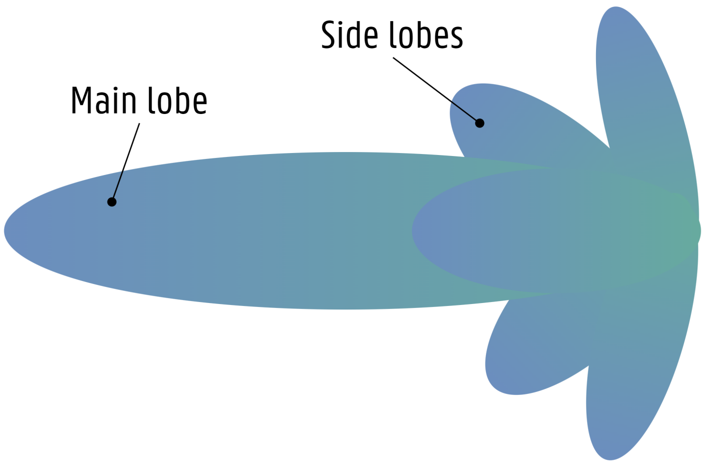


Figure 21 | Demonstration emission pattern of a beamforming system, showing the main lobe and side lobes.

6.1.b *PHÔS* implementation

The *PHÔS* implementation relies on several key features of the *PHÔS* language. It utilises the `split` function which is used to split a signal based on a list of weights. These weights are provided by the `splat` function which creates a list of n elements all of the same value. Those signals are then constrained to have the same phase before being phase modulated using the `modulate` function. The resulting signals are then constrained to have the same delay before being sent to the outputs. The code for this example is available at Listing 47, with the fully commented code being available in Annex F.

Constrain `constrain` is a synthesisable block that allows the user to create constraints between signals. It can be used to impose one of two constraints, either a phase constraint, matching the phases of the different signals, or a delay constraint, matching the delays of the different signals. In this case, the phase constraint is used to ensure that all of the signals have the same phase when reaching a certain component, and the delay constraint is used to ensure that all of the signals have the same delay. Recalling Section 4.4, these constraints are different due to the large order of magnitude difference between the frequency of light and the frequency of modulated content; a phase shift on the light will have a negligible impact on the modulated content, but a delay shift on the light will have a large impact on the modulated content.

Modulate `modulate` is a synthesisable block used to modulate an optical signal, it can perform either phase modulation or amplitude modulation. In the case of amplitude modulation, the synthesis stage may create a MZI to perform a phase-to-amplitude conversion. In this example, it is used to modulate the external phase shifts onto the optical signals.

Partial function `set` allows the creation of a partial function where parts of the arguments have already been filled. In this case, it is used to create a partial function of `modulate` where the type of modulation is already set to phase modulation.

Zipping & mapping `zip` allows two lists to be zipped together, creating a list of tuples, where each tuple contains the elements of the two lists at the same index. `map` allows a function to be applied to each element of a list. In this case, `zip` is used to zip the list of phase shifts with the list of optical signals, creating a list of tuples where each tuple contains an optical signal and a phase shift. This list is then mapped to a function that sets the phase shift of the optical signal to the phase shift of the tuple.

```
syn beam_forming(
    input: optical,
    phase_shifts: (electrical...),
) -> (optical...) {
    input
        |> split(splat(1.0, phase_shifts.len()))
        |> constrain(d_phase = 0)
        |> zip(phase_shifts)
        |> map(set modulate(type_: Modulation::Phase))
        |> constrain(d_delay = 0)
}
```

Listing 47 *PHÔS* implementation of a configurable optical beamforming system. A fully commented version is available in Annex F.

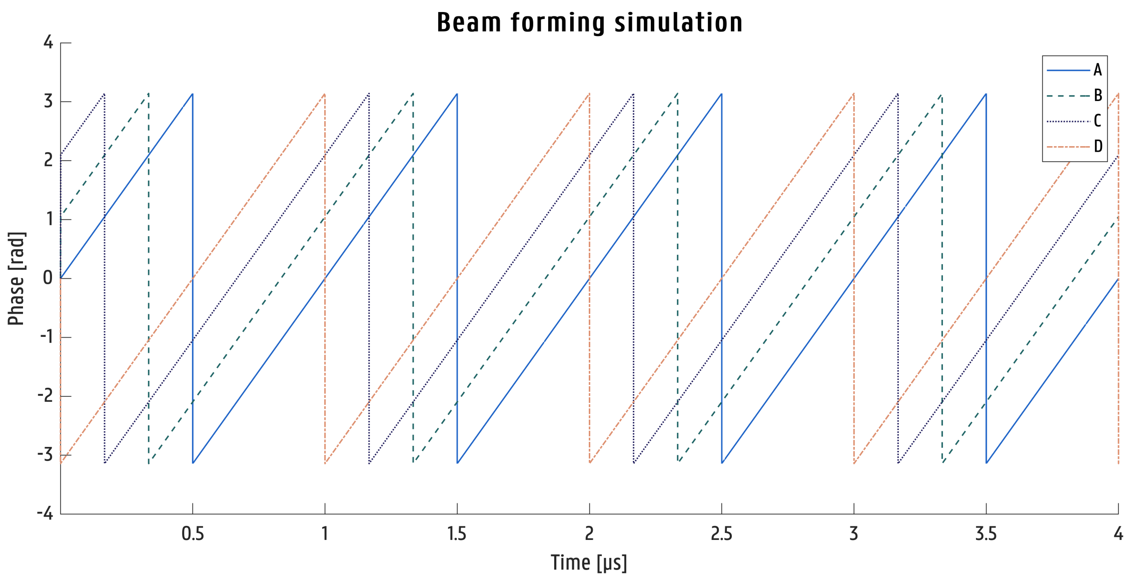
6.1.c Results

The time-domain simulation can easily be performed using the constraint solver, yielding the results shown in Figure 22. In this simulation, only four channels were simulated, each with a time-dependent phase shift at a frequency of 1 MHz. In the first simulation (a), the phases are following Equation 1, where k refers to the channel number starting at zero. And in the second simulation (b), they are following Equation 2. The simulation shows that the phase shifts are correctly applied to the optical signals, and that the optical signals are correctly constrained to have the same phase and delay.

$$\varphi_k(t) = \frac{k \cdot \pi}{3} + 2\pi \cdot 1\ \mathrm{MHz} \cdot t \tag{1}$$

$$\varphi_k(t) = \sin\left(\frac{k \cdot \pi}{3} + 2\pi \cdot 1\ \mathrm{MHz} \cdot t\right) \tag{2}$$

(a)



(b)

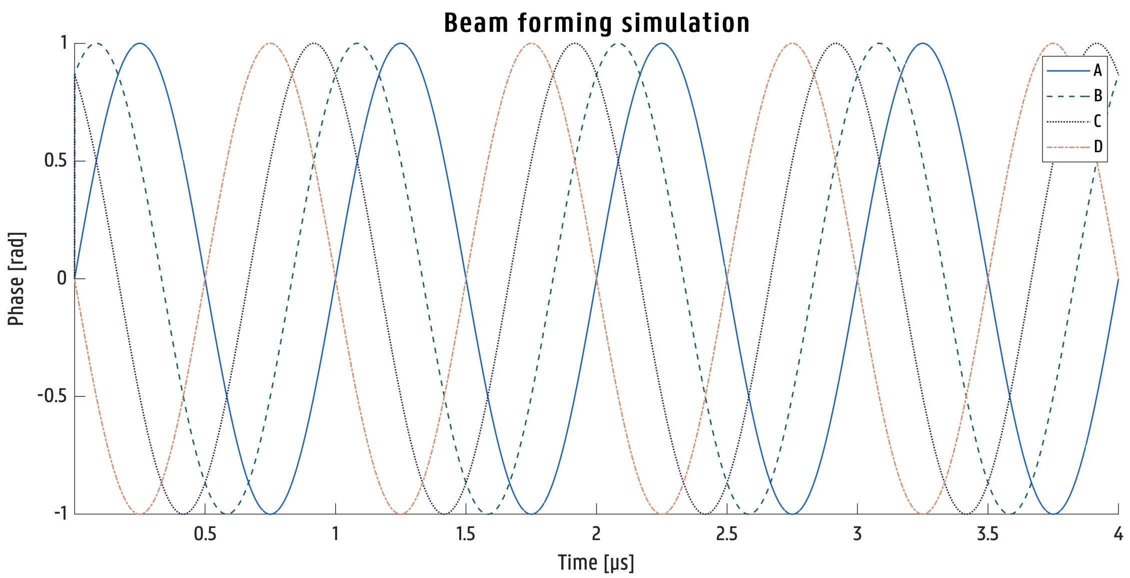


Figure 22 Simulation results of the beamforming system, showing the time-dependent phase shifts applied to the optical signals.

6.2 Coherent 16-QAM transmitter

In this next example, a simple 16- QAM transmitter will be demonstrated along with simulation results. The code for this example is available at Annex G. This example will first cover the theoretical background needed to understand the motivation for the example and the measurements taken, followed by the transmitter's modulation aspects.

6.2.a Theoretical background

Definition: **QAM** refers to **quadrature amplitude modulation**, a modulation scheme where the information is encoded in both the amplitude and the phase of the signal. The **16** refers to the number of symbols in the constellation, which is the number of different values the signal can take. In this case, the data encoded is 4 bits per symbol for 16 possible values.

Adapted from [119].

In telecommunications, and especially in high-speed communication, engineers need to be able to transmit as much information as possible in a given bandwidth while still maintaining good immunity to noise and other impairments. One way to achieve these higher throughputs is by using more advanced modulation schemes, state-of-the-art in photonic communication being 64- QAM [120]. In this example, however, state-of-the-art will not be reproduced and will instead be focused on a simpler 16- QAM modulation scheme, based on the work by *Talkhooncheh, Arian Hashemi, et al.* [121].

Modulations are often visualised using two types of diagrams: so-called *eye diagrams* which show the transitions between symbols, and *constellation diagrams* which show the actual symbols after sampling. These two visualisations are used to measure the quality of the received signal and to visualise any impairment it might have suffered during transmission. Eye diagrams are built by overlaying many transitions between symbols over one another, slowly building a statistical representation of the signal. Constellation diagrams are built by sampling the signal at a given rate and plotting its magnitude and phase in a complex plane. The resulting plot is a point cloud that can be used to visualise the symbols that were transmitted.

Finally, the measure that will be used to quantify the quality of the transmitter is not the BER (*Bit Error Rate*) , as the measurement will be taken at the transmitter's output. Therefore the bit error rate will be zero, but rather the EVM (*Error Vector Magnitude*) . The EVM is a measure of the difference between the ideal constellation and the actual constellation and is defined as in Equation 3, with N the number of samples, I_{err} and Q_{err} the error in the in-phase and quadrature components of the constellation, $\text{EVM}_{\%}$ the EVM in percentage, and $\text{EVM Normalization Factor}$ is a normalisation factor that depends on the modulation scheme used, for 16- QAM , it is the maximum magnitude of the constellation [122]. A visualisation of EVM can be found in Figure 23. With this definition, one can see that the EVM measures the average distance between the ideal constellation and the actual constellation and should, therefore, be minimised.

$$\text{EVM}_{\%} = \frac{\sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left(I_{\text{err}}[i]^2 + Q_{\text{err}}[i]^2\right)}}{\text{EVM Normalization Factor}} \cdot 100\% \tag{3}$$

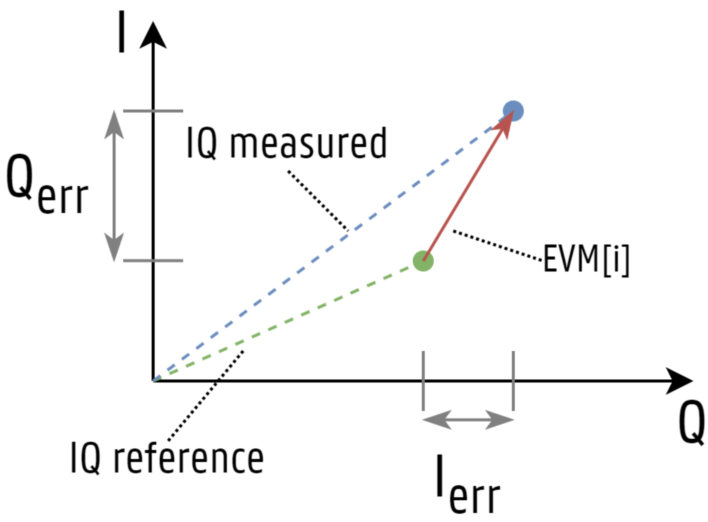


FIGURE 23 Error vector magnitude – reference plot, showing the reference and measured IQ points, and the EVM vector of the sample.

6.2.b PHÔS IMPLEMENTATION

The circuit being built is shown in Figure 24, with its code in Annex G. It consists of a laser source, which, in the *PHÔS* code shown in Listing G.1, is considered to be external to the device. The light is then split into four parts, two of which are split into one-quarter of the total light, while the remaining two each receive half of the total light. Each signal is then modulated, on a real chip. This could be done using an electro-absorption modulator (EAM) or a MZI -based modulator. The signals are then phase-shifted to form the *I* and the *Q* modulation. The first two signals are form the in-phase modulation, while the remaining two form the quadrature modulation. The four modulated signals are then combined and sent to the output.

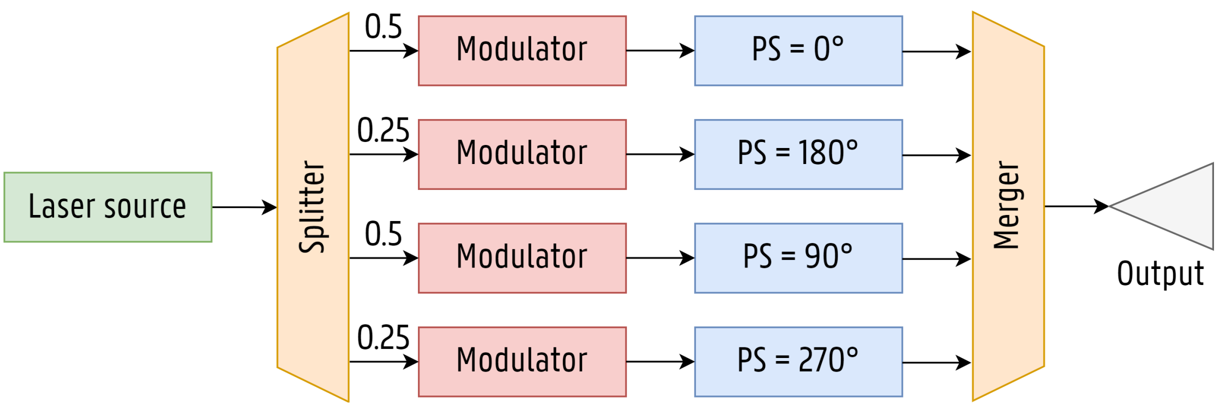


FIGURE 24 16- QAM modulator circuit, showing the splitter, modulators, phase shifters, and interferometer.

The input signal is first split into four parts (line 15) into four parts with weights 1.0, 1.0, 0.5, and 0.5. These four signals are zipped (line 16), meaning that they are combined into a single value containing an optical and an electrical signal each. All of those values are then amplitude modulated (line 17). The second, third, and fourth signals are then phases shifted such that they are 0, 90, and 180 degrees out of phase with the first signal (line 18). The four signals are then interfered together (line 19) and sent to the output. The resulting signal is a 16- QAM modulated signal composed of four binary values per symbol.

One can build a signal flow diagram from this code containing all of the intrinsic operations and constraints. Note that the input was replaced with a source of intensity in an arbitrary unit of 1.0, with an AWGN (*Additive White Gaussian Noise*) with mean 0.0 and standard deviation 0.025 added to it. The resulting signal flow diagram can be found in Figure 25.

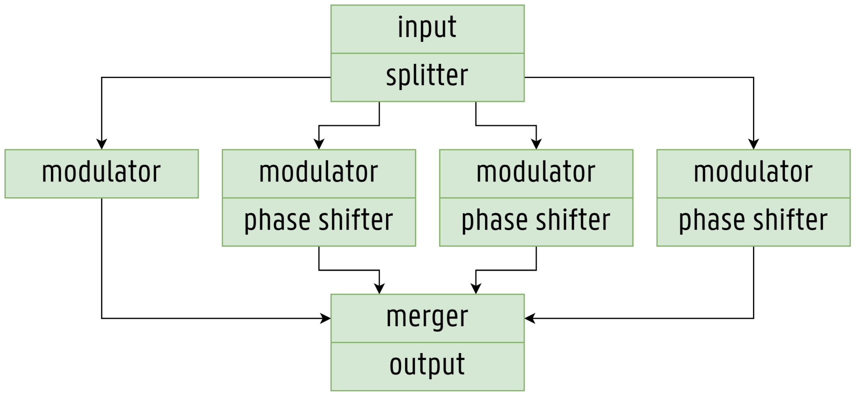


FIGURE 25 | Signal flow diagram of a 16- QAM modulator, showing the different components.

6.2.c Results

This example is trivial to simulate for the constraint solver, with four $100\ \mathrm{Gb}\ /\ s$ binary sources, it finishes simulating a 1ns window in $45\ \mathrm{ms}$ on a recent *AMD* CPU . In Figure 26, one can see the simulation results, showing the input signal with its simulated noise in (a), the output signal in (b), and the intermediary signals in (c) and (d). Finally in (e), one can see the constellation, from which the EVM can be calculated as 4.41%, which is $-17.11\ \mathrm{dB}$ when expressed logarithmically. At these speeds, with a fairly high noise, this can be considered a good result as it leads to a BER of zero.

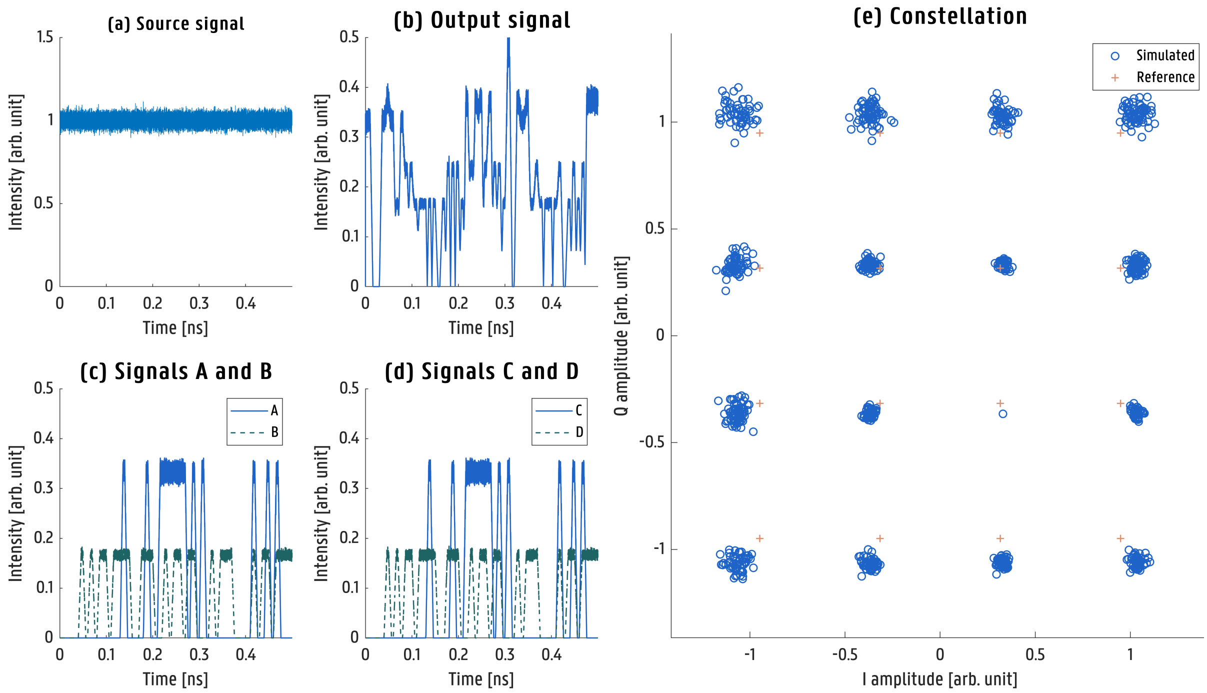


FIGURE 26 | Simulation results of a 16- QAM modulator, showing the input signal, the output signal, and the intermediary signals, along with the constellation points, and reference points. The constellation points have been normalized before being shown.

6.3 Lattice filter

Lattice filters are a type of filter that can be easily built from MZIs and couplers, they allow the user to easily build a filter with the frequency response that matches their needs [123]. Specifically, lattice filters are ideal components to use as they are completely passive, allowing for very low power signal processing, of particular interest for microwave signal processing in the optical domain [124]. This example will shortly discuss the theoretical background of lattice filters, then show how to build such a filter in *PHÔS* , showing how easy and expressively the language can be used to build such a filter. As the constraint solver is not yet able to solve frequency domain problems, the filter will not be simulated, instead relying on theoretical results. The general form of an MZI based lattice filter can be seen in Figure 27. This example is based off of *Ruocco et al.*'s and *Guan et al.*'s works [123, 124].

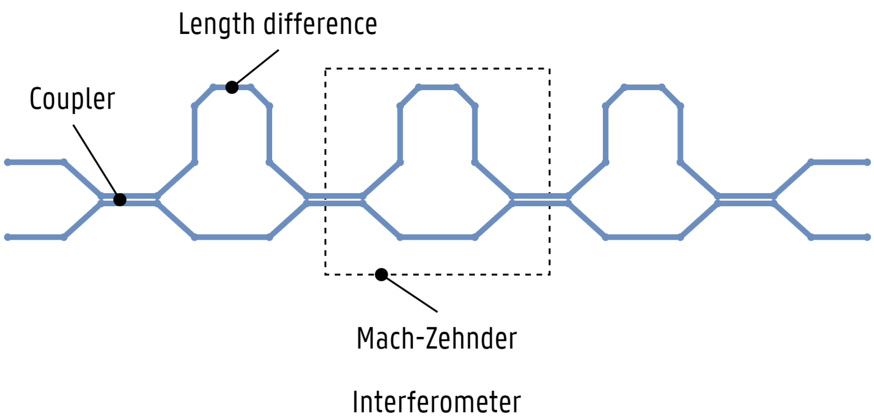


FIGURE 27 | MZI based lattice filter built of three MZIs with the same path length difference.

6.3.a Theoretical background

Lattice filters are built from two elements, couplers and sections with a length difference. Assuming that there are no reflections, one can model these elements as 2×2 matrices, the first one being the coupler, and the second one being a phase shifter. The *S* matrix of a coupler can be seen in Equation 4, where τ_i corresponds to the coupling coefficient of the i -th coupler. The *S* matrix of a phase shifter can be seen in Equation 5, where β corresponds to the propagation constant of the waveguide, and ΔL_i corresponds to the length difference of the i-th section. The *S* matrix of a complete lattice filter can then be calculated by multiplying the *S* matrices of the couplers and sections together, as seen in Equation 6.

$$S_{\text{coupler},i} = \begin{bmatrix} \tau_i & -j \cdot \sqrt{1-\tau_i^2} \\ -j \cdot \sqrt{1-\tau_i^2} & \tau_i \end{bmatrix} \tag{4}$$

$$S_{\text{delay},i} = \begin{bmatrix} e^{-j \cdot \beta \Delta L_i} & 0 \\ 0 & 1 \end{bmatrix} \tag{5}$$

$$S = S_{\text{coupler},n+1} \cdot \prod_{i=1}^{n} S_{\text{delay},i} \cdot S_{\text{coupler},i} \tag{6}$$

6.3.b Building the filter

MZI based lattice filters are very simple to build in *PHÔS* , assuming that one has a function to compute the coefficients required, which would be part of a filter synthesis toolbox, here named `filter_kind_coefficients` , then one can build a filter with the code in Listing 48. The code first computes the coefficients, then folds them, meaning that it iterates over them while accumulating a result, in this case the result are the final output signals. For each coefficient, the accumulator signals are coupled with the computed coefficient, and then constrained to the differential phase computed. Finally, the last two signals are coupled together with the final computed coefficient. The result is a filter with the frequency response of the coefficients, which can be seen in Figure 28, showing the theoretical results of a 4th and 8th order filter.

```
syn lattice_filter(a: optical, b: optical, filter_kind: FilterKind) -> (optical, optical) {
    let (coeffs, final_coupler) = filter_kind_coefficients(filter_kind)

    coeffs |> fold((a, b), |acc, (coeff, phase)| {
            acc |> coupler(coeff) |> constrain(d_phase = phase)
    }) |> coupler(final_coupler)
}
```

Listing 48 MZI based lattice filter in *PHÔS* , parametrically generated for the user, fully commented example in Annex H.

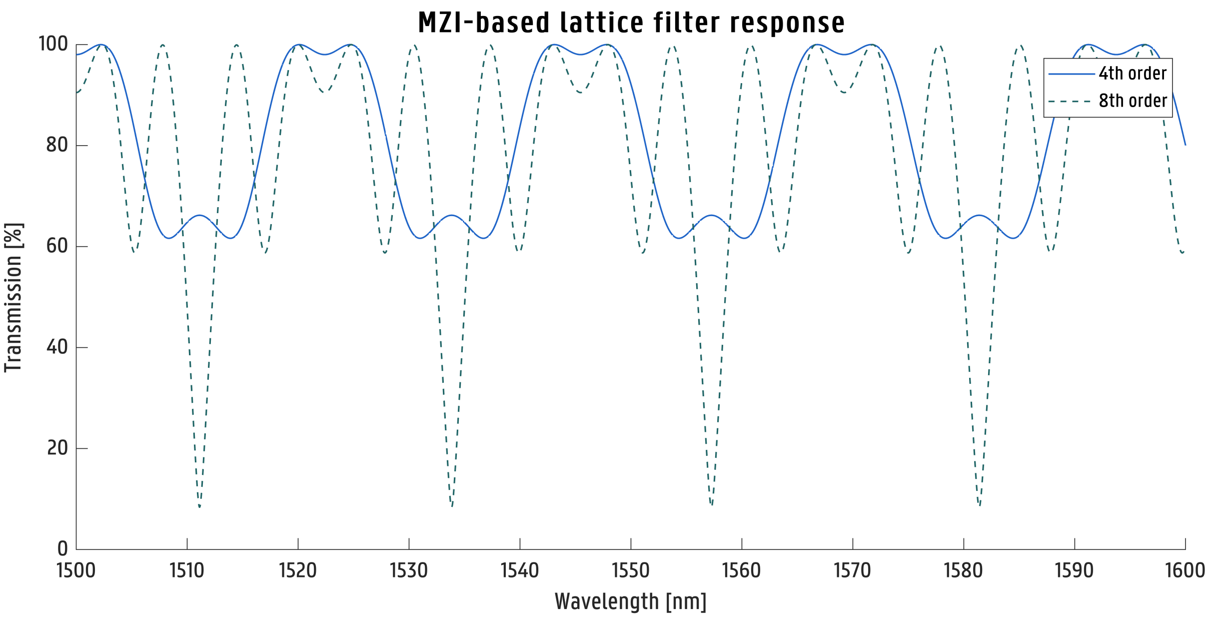


Figure 28 Theoretical frequency response of a MZI based lattice filter in *PHÔS* . Fourth order example with coefficients:

$$(\tau_i, \Delta L_i) = (0.5, 30\ \mu\text{m}), (0.8, 30\ \mu\text{m}), (0.5, 30\ \mu\text{m}), (0.8, 30\ \mu\text{m}) \quad (7)$$

and a final coupler with a coefficient of $\tau_5 = 0.04$, in waveguide with an effective refractive index of $n_{\text{eff}} = 2.4$. An addition 8th order is also shown, with the coefficients from the 4th order filter repeated twice.

6.4 Analog matrix-vector multiplication

As previously mentioned in Section 2.1, there are two major kinds of programmable PICs , and while this work has mostly focused itself on recirculating mesh-based photonic processors, they are capable of building the same circuits as feedforward PICs . A typical use case of feedforward meshes is MVM (*Matrix-Vector Multiplication*) . This is useful for very quickly and efficiently performing MVM , a common machine learning operation. This example will demonstrate how such a MVM photonic circuit is built in *PHÔS* and how to use it to perform MVM . The example shown in this example is based on *Shokraneh et al.*'s work [125].

This circuit is built from individual MZIs , with an added phase shifter, these groupings, which can be seen in Figure 29, are equivalent to the photonic gates from which a photonic processor is built, see Section 2.1.a. For this circuit, they are configured in a triangular shape, as can be seen in Figure 30. The circuit is built from 6 gates and can multiply a vector of size 4 with a 4×4 matrix.

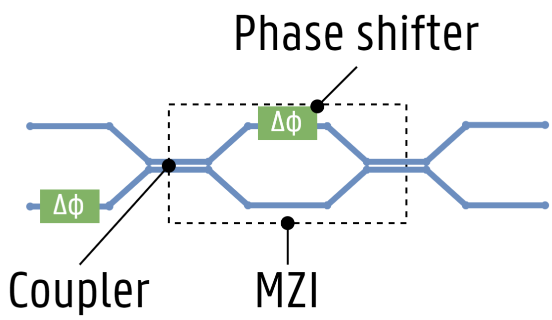


Figure 29 Diagram of a single MZI gate used when building an MVM circuit. The MZI gate is built from two couplers and two phase shifters. The first coupler is used to split the input signal into two, the second coupler is used to recombine the two signals, the first phase shifter is used to add a phase shift to the top signal, and the second phase shifter is used to add a phase shift to the bottom signal.

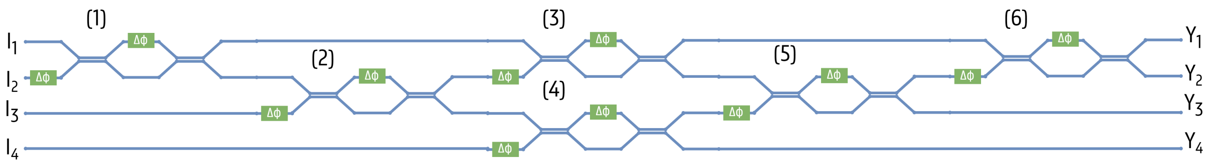


Figure 30 Diagram of the full MZI MVM circuit. With the inputs annotated I_1 through to I_4, and the output annotated Y_0 through to Y_3. The circuit is built from 6 MZI gates, with four inputs and four outputs.

From these diagrams, it becomes clear that the matrix-vector multiplication is not trivial, assuming that the final operation being performed is $Y = \boldsymbol{M} \cdot X$, where Y and X are vectors, and $\boldsymbol{M}$ is a matrix, these cannot be mapped one-to-one with the values of the phase shifters on the circuit. The transformation from the matrix $\boldsymbol{M}$ into the corresponding phase shifts is not the focus of this thesis. Therefore, the ones from *Shokraneh et al.*'s work will be used instead [125]. It is interesting to note that, by performing the matrix multiplication in the analog domain, while the circuit can be made to be extremely fast, it also introduces noise and imprecision. Therefore, while machine-learning models that rely on low-precision arithmetic may not be a problem, they would have limited use in applications that depend on higher-precision arithmetic.

The code to create this circuit in *PHÔS* is rather long and is therefore available in Annex I. It can successfully be simulated using the constraint solver. The tests were done with the following input vectors: $X = (0,0,0,0), (1,0,0,0), (0,1,0,0), (0,0,1,0), \text{and } (0,0,0,1)$. From these values, one can verify that the circuit is indeed performing the correct operation by comparing that the first vector produces an empty vector and that the other vectors return the corresponding column of the matrix.

7. Future work

This section will give short introductions to some of the interesting research topics and areas that are related to PHÔS.

7.1 Implementation

As of writing this thesis, *PHÔS* is still very much in its infancy. While the first two elements of its compilation pipeline are implemented, many of its components need implementation. However, this document outline in great detail what each component should do and how they should generally do it. This means that the implementation of *PHÔS* is mostly a matter of time and effort.

7.2 Dependent types & refinement types

Definition: **Dependent types** are types that depend on values. For example, in the case of *PHÔS* , they could depend on an argument such as the centre wavelength of a filter.

Definition: **Refinement types** are types that are refined by predicates. In the case of *PHÔS* , they could be used to refine the types using constraints.

One potential improvement to the *PHÔS* language is the use of dependent and refinement types. Together, this would allow the typing of different synthesisable blocks to be stronger and have stronger guarantees over the validity of the design. While *PHÔS* already implements those partly in its current architecture, they are not integrated as part of the types, instead being additional values carried by the types. In its current design iteration, this means that the compiler is limited on the complexity of the constraint it can properly verify. This limitation can be lifted by integrating dependent and refinement types as part of the type system, where constraints would be part of the type definition, and the compiler could fully verify them at compile time, further verifying the user's design. However, refinement types, in particular, are not trivial to implement and would require a lot of work and research to be appropriately integrated into the language.

7.3 Advanced constraint solving & constraint inference

In its current iteration, the constraint system cannot infer new constraints, only existing ones. This means that a circuit that behaves as a filter must be manually annotated with a filter constraint to be recognised as one. This is a limitation that may be able to be lifted by being able to infer new constraints rather than just existing ones. This would allow the compiler to recognise more complex circuits without the user annotating them as much manually. This could be implemented by automatically running frequency domain analysis over portions of the code, using a set of predefined rules to infer constraints from the circuit's structure, using machine learning to infer constraints from the circuit's structure, or using a combination of those methods. This area also has the potential for new and interesting research with applications outside of photonics.

7.4 Co-simulation with digital electronic

While it is not implemented yet, it would be relatively easy and interesting to interface the constraint solver with digital electronic simulation. This could be done with a tool such as *Verilator* and by using an event-driven simulation model. The constraint solver could run until a digital event is triggered, such as the rising edge of a clock, and then the digital simulation would be run, at which point the constraint solver would be running again, taking into effect the changes done by the digital simulation. This would continue as long as the simulation is running. This would allow the co-simulation of *PHÔS* code with digital electronic and would allow the simulation of larger parts of a system.

7.5 Place-and-route

While algorithms for routing photonic components on photonic processors have been created, there are no complete place-and-route solutions for photonic processors.

7.6 Programming of generic photonic circuits

This thesis is mostly focused on photonic processors. However, *PHÔS* can be used for the creation of generic photonic circuits. Replacing the platform-support package with a photonic development kit would allow the user to create any circuit. This development kit would then interface with a manufacturer's PDK, allowing the user to create custom chips from their design. Some parts of the design would need to be done externally, such as placement, but there are already tools for that, such as *Luceda's IPKISS*. This is made especially easy through the marshalling library, discussed in Section 5.8, which allows the user to easily interface with a *Python*-based PDK.

7.7 Language improvements

Several language improvements and additions should be made to make development easier, such as adding error handling, generics, bitflags, macros, reflection, meta programming, algebraic effects, incremental compilation, and more. This section will briefly introduce each of those topics and how they could be used in *PHÔS* .

Bitflags Bitflags are similar in principle to enumerations, but they allow for more than one value to be set at a time. This is useful for the creation of flags, replacing a list of booleans with a single value containing all of the boolean values, each encoded on one bit.

Generics Generics were discussed in Section 3.2 under the more formal name of polymorphism. Currently, the *PHÔS* design does not involve generics for simplicity. However, due to the complexity of the type system, generics would be very useful and would allow for less "magic" functions, such as `map` which can take any type. This would allow the user to implement complex polymorphic functions themselves rather than being forced to defer to implementation within the compiler.

Traits and type classes If generic types were to be implemented, traits would need to also be implemented; traits define a set of functions that a type must implement to be considered to implement the trait. This is similar to interfaces in object-oriented programming, but with the difference that traits can be implemented on types that are not defined in the same module as the trait. This is useful for the creation of generic functions that can be used on any type that implements the trait. This can be done rather easily be relying on *Rust*'s *Chalk* library, which is a trait solver [126].

Error handling Currently, *PHÔS* has no facilities for evaluation errors, this is an oversight, and an error-handling model must be added. This could be done with a `Result` type or with exceptions. The advantage of the `Result` type is that it is explicit and that it forces the user to handle the error, while the advantage of exceptions is that they are implicit. However, having a `Result` type would require generics to be implemented first, as it would need to be generic over the value and error types.

Macros and reflection It was discussed Section 5.4.d, that *PHÔS* could benefit from macros. Macros are functions called by the compiler during compilation that transform the input AST into a new AST . They can be used to generate complex code from simpler code. However, *PHÔS* could also benefit from compile-time reflection, allowing the user to write compiler plugins that can introspect and modify the program while it is compiling. It could be used to have the same effect as macros but would be much more capable, as it would allow the user to introspect the program and modify it rather than just transform the AST .

Incremental compilation Currently, *PHÔS* does not support incremental compilation. This is a limitation that should be lifted. Incremental compilation is the ability to only recompile the parts of the program that have changed rather than recompiling the whole program. However, in the case of *PHÔS* , compilation is not the slowest part of the process, and it would be useful to be able to perform increment place-and-route, rather than just incremental compilation. This could make the development of large circuits much faster by reusing the placement and routing of unchanged circuit parts.

This page is intentionally left blank.

8. Conclusion

In this thesis, a new approach for the description and subsequent programming of photonic processors combined an easy-to-use programming language with powerful new features, such as constraints, reconfigurability through branching, and tunability. Together, these features create a modern, flexible, and novel hardware description language that is able to describe a wide range of photonic circuits and can be used for both the development of circuits for photonic processors, as well as the development of standalone photonic circuits.

This design was created after a thorough analysis of existing programming languages and paradigms was performed, shedding light on each language's strengths, weaknesses, and applicability. The analysis showed that modern languages with fully-featured first-party ecosystems are good candidates for developing a new language and implementing libraries but that a new hardware description language was needed overall. This new language, called *PHÔS* , results from months of research and development and is the first hardware description language to combine aspects of modern programming languages with unique features designed specifically for photonic circuits.

Through the analysis of relevant, real-world examples, this thesis has shown that *PHÔS* is a powerful language that is appropriate and useful for the development of photonic circuits. The examples have shown that *PHÔS* is able to concisely describe complex circuits, their constraints, and functionality while still being easy to read and understand. Additionally, it was demonstrated that the constraint solver used for simulations is, despite its early stage, already usable to create meaningful results and that the simulation of photonic circuits can be done quickly and efficiently.

PHÔS has also been designed so that it can be created by a relatively small team, by reusing existing algorithms and libraries available in both the *Rust* and *Python* ecosystems. This makes *PHÔS* an ideal candidate for future development of photonic circuit programming, as it is easy to expand, modify, and maintain.

Finally, the design of the *PHÔS* language has shed light on new areas of research into which photonic circuit programming can be expanded, including the development of a new, more powerful constraint solver, the use of state-of-the-art type systems for improved correctness, and the ability to use the language for the development of non-traditional computing such as analog computing.

Bibliography

[1] e. a. Rosetta Code, "Sieve of eratosthenes," Rosetta Code. https://rosettacode.org/wiki/Sieve_of_Eratosthenes (accessed: May 21, 2023).

[2] A. Mohamed, "N-bit adder in VHDL." https://ahmedmohamed45am.wixsite.com/fpgagate/single-post/2018/02/02/n-bit-adder-in-vhdl (accessed: Jun. 3, 2023).

[3] W. Bogaerts, and A. Rahim, "Programmable Photonics: An Opportunity for an Accessible Large-Volume PIC Ecosystem," *IEEE J. Sel. Topics Quantum Electronics*, vol. 26, no. 5, pp. 1–17, Sep. 2020, doi: 10.1109/JSTQE.2020.2982980. Accessed: Mar. 10, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9049105/

[4] W. Bogaerts, D. Pérez, et al., "Programmable photonic circuits," *Nature*, vol. 586, no. 7828, pp. 207–216, Oct. 2020, doi: 10.1038/s41586-020-2764-0. Accessed: Mar. 10, 2023. [Online]. Available: https://www.nature.com/articles/s41586-020-2764-0

[5] R. L. Geiger, P. E. Allen, and N. R. Stader, "VLSI design techniques for analog and digital circuits," 1990. [Online]. Available: https://www.researchgate.net/profile/Arturo-Salz-2/publication/4218954_IRSIM_an_incremental_MOS_switch-level_simulator/links/54909e260cf214269f27c7eb/IRSIM-an-incremental-MOS-switch-level-simulator.pdf

[6] "Imperative programming: Overview of the oldest programming paradigm," 2020. Accessed: Mar. 27, 2023. [Online]. Available: https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/

[7] W. Bogaerts, and L. Chrostowski, "Silicon Photonics Circuit Design: Methods, Tools and Challenges," *Laser & Photon. Reviews*, vol. 12, no. 4, p. 1700237, Apr. 2018, doi: 10.1002/lpor.201700237. Accessed: Mar. 27, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/lpor.201700237

[8] E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration," *J. Biomed. Opt.*, vol. 13, no. 6, p. 60504, 2008, doi: 10.1117/1.3041496. Accessed: Mar. 27, 2023. [Online]. Available: http://biomedicaloptics.spiedigitallibrary.org/article.aspx?doi=10.1117/1.3041496

[9] Y. Ye, T. Ullrick, W. Bogaerts, T. Dhaene, and D. Spina, "SPICE-Compatible Equivalent Circuit Models for Accurate Time-Domain Simulations of Passive Photonic Integrated Circuits," *J. Lightw. Technol.*, vol. 40, no. 24, pp. 7856–7868, Dec. 2022, doi: 10.1109/JLT.2022.3206818. Accessed: Mar. 10, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9893343/

[10] D. Marpaung, J. Yao, and J. Capmany, "Integrated microwave photonics," *Nature Photon.*, vol. 13, no. 2, pp. 80–90, Feb. 2019, doi: 10.1038/s41566-018-0310-5. Accessed: Mar. 10, 2023. [Online]. Available: https://www.nature.com/articles/s41566-018-0310-5

[11] C. Sorace-Agaskar, J. Leu, M. R. Watts, and V. Stojanovic, "Electro-optical co-simulation for integrated CMOS photonic circuits with VerilogA," *Opt. Electron.*, vol. 23, no. 21, p. 27180, Oct. 2015, doi: 10.1364/OE.23.027180. Accessed: Mar. 27, 2023. [Online]. Available: https://opg.optica.org/abstract.cfm?URI=oe-23-21-27180

[12] A. M. Smith, J. Mayo, et al., "Digital/analog cosimulation using cocotb and xyce.", 2018, doi: 10.2172/1488489. [Online]. Available: https://www.osti.gov/biblio/1488489

[13] D. Pérez-López, A. López, P. DasMahapatra, and J. Capmany, “Multipurpose self-configuration of programmable photonic circuits,” *Nature Commun.*, vol. 11, no. 1, p. 6359, Dec. 2020, doi: 10.1038/s41467-020-19608-w. Accessed: Mar. 10, 2023. [Online]. Available: https://www.nature.com/articles/s41467-020-19608-w

[14] J. Capmany, I. Gasulla, and D. Pérez, “The programmable processor,” *Nature Photon.*, vol. 10, no. 1, pp. 6–8, Jan. 2016, doi: 10.1038/nphoton.2015.254. Accessed: Mar. 10, 2023. [Online]. Available: http://www.nature.com/articles/nphoton.2015.254

[15] D. Perez, I. Gasulla, and J. Capmany, “Programmable Multifuctional Photonics ICs,” arXiv, 2019. Accessed: Mar. 28, 2023. [Online]. Available: http://arxiv.org/abs/1903.04602

[16] A. Ghatak, and K. Thyagarajan, *Introduction to Fiber Optics*, Cambridge: Cambridge University Press, 1998. Accessed: Mar. 28, 2023. [Online]. Available: http://ebooks.cambridge.org/ref/id/CBO9781139174770

[17] C. Demirkiran, F. Eris, et al., “An Electro-Photonic System for Accelerating Deep Neural Networks,” 2021, doi: 10.48550/ARXIV.2109.01126. Accessed: Jun. 15, 2023. [Online]. Available: https://arxiv.org/abs/2109.01126

[18] D. Pérez-López, A. Gutiérrez, and J. Capmany, “Silicon nitride programmable photonic processor with folded heaters,” *Opt. Electron.*, vol. 29, no. 6, pp. 9043–9059, Mar. 2021, doi: 10.1364/OE.416053. Accessed: Jun. 15, 2023. [Online]. Available: https://opg.optica.org/oe/abstract.cfm?uri=oe-29-6-9043

[19] M. G. Daher, S. A. Taya, et al., “Design of a nano-sensor for cancer cell detection based on a ternary photonic crystal with high sensitivity and low detection limit,” *Chin. J. Phys.*, vol. 77, pp. 1168–1181, Jun. 2022, doi: 10.1016/j.cjph.2022.03.032. Accessed: Jun. 15, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0577907322000880

[20] M. N. Trutzel, K. Wauer, et al., “Smart sensing of aviation structures with fiber optic Bragg grating sensors,” in *Smart Structures Materials 2000: Sensory Phenomena Meas. Instrum. Smart Structures Materials*, vol. 3986, Jun. 2000, pp. 134–143, doi: 10.1117/12.388099. Accessed: Jun. 15, 2023. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/3986/0000/Smart-sensing-of-aviation-structures-with-fiber-optic-Bragg-grating/10.1117/12.388099.full

[21] I. Ashry, Y. Mao, et al., “A Review of Distributed Fiber–Optic Sensing in the Oil and Gas Industry,” *J. Lightw. Technol.*, vol. 40, no. 5, pp. 1407–1431, Mar. 2022, doi: 10.1109/JLT.2021.3135653. Accessed: Jun. 15, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9652039/

[22] W. contributors, “Netlist,” Wikimedia Foundation. https://en.wiktionary.org/wiki/netlist (accessed: Jun. 15, 2023).

[23] Y. Xing, M. U. Khan, A. Ribeiro, and W. Bogaerts, “Behavior Model for Directional Coupler,” in *Proc. Symp. IEEE Photon. Soc. Benelux*, Delft, The Netherlands, Jan. 2017.

[24] L. Cardelli, and P. Wegner, “On understanding types, data abstraction, and polymorphism,” *ACM Comput. Surveys*, vol. 17, no. 4, pp. 471–523, Dec. 1985, doi: 10.1145/6041.6042. Accessed: May 22, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/6041.6042

[25] G. Dot, A. Martinez, and A. Gonzalez, “Analysis and Optimization of Engines for Dynamically Typed Languages,” in *2015 27th Int. Symp. Comput. Architecture High Perform. Comput. (SBAC-Pad)*, Florianopolis, Brazil, Oct. 2015, pp. 41–48, doi: 10.1109/SBAC-PAD.2015.20. Accessed: May 22, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/7379832/

[26] R. Milner, “A theory of type polymorphism in programming,” *J. Comput. System Sciences*, vol. 17, no. 3, pp. 348–375, Dec. 1978, doi: 10.1016/0022-0000(78)90014-4. Accessed: May 22, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0022000078900144

[27] Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/ (accessed: May 16, 2023).

[28] ANSYS, “Lumerical photonic verilog-a platform.” Accessed: May 20, 2023. [Online]. Available: https://www.ansys.com/products/photonics/verilog-a

[29] D. M. Jones, “Forms of language specification,” 2007. Accessed: May 16, 2023. [Online]. Available: http://www.knosof.co.uk/vulnerabilities/langconform.pdf

[30] K. S. Chacko, “Case study on universal verification methodology(uvm) systemc testbench for rtl verification,” 2019.

[31] The Rust Reference. https://doc.rust-lang.org/nightly/reference/ (accessed: May 16, 2023).

[32] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, “Compilers: principles techniques and tools. 2007,” *Google Scholar Google Scholar Digit. Library Digit. Library*, 2006.

[33] B. A. Becker, P. Denny, et al., “Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research,” in *Proc. Work. Group Reports Innov. Technol. Comput. Sci. Educ.*, Aberdeen Scotland Uk, Dec. 2019, pp. 177–210, doi: 10.1145/3344429.3372508. Accessed: May 16, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3344429.3372508

[34] e. a. Clang, “Clang internals manual,” Clang. https://clang.llvm.org/docs/InternalsManual.html (accessed: May 21, 2023).

[35] R. Czerwinski, and D. Kania, *Finite State Machine Logic Synthesis for Complex Programmable Logic Devices*, vol. 231, Springer Science & Business Media, 2013.

[36] S. Szczesny, “HDL-Based Synthesis System with Debugger for Current-Mode FPAA,” *IEEE Trans. Computer-Aided Des. Integr. Circuits Syst.*, p. 1, 2017, doi: 10.1109/TCAD.2017.2740295. Accessed: May 17, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/8010823/

[37] M. C. Felgueiras, G. R. Alves, and J. M. M. Ferreira, “A built-in debugger for 1149.4 circuits,” 2007.

[38] V. Motel, “Simulation and debug of mixed signal virtual platforms for hardware-software co-development,” 2014.

[39] F. documentation project mailing list, “Style(9),” in *Freebsd Kernel Developer's Manual*, FreeBSD Kernel Developer's Manual. https://man.freebsd.org/cgi/man.cgi?query=style&sektion=9 (accessed: May 17, 2023).

[40] R. S. et Al., “5.1 formatting your source code,” in *GNU Coding Standards*, GNU Coding Standards. https://www.gnu.org/prep/standards/standards.html#Formatting (accessed: May 17, 2023).

[41] M. A. Nono, “What's the difference between code linters and formatters?,” 2022. Accessed: May 20, 2023. [Online]. Available: https://nono.ma/linter-vs-formatter

[42] P. Wadler, and J. Kilmer, “A prettier printer,” 2002. Accessed: May 17, 2023. [Online]. Available: https://homepages.inf.ed.ac.uk/wadler/papers/prettier/prettier.pdf

[43] R. Clippy, “Clippy lints,” GNU Project. https://rust-lang.github.io/rust-clippy/master/index.html (accessed: May 17, 2023).

[44] e. a. G. Ann Campbell, “Cognitive Complexity - a new way of measuring understandability.” [Online]. Available: https://www.sonarsource.com/docs/CognitiveComplexity.pdf

[45] e. a. PCMag, “Source code editor,” PCMag. https://www.pcmag.com/encyclopedia/term/source-code-editor (accessed: May 21, 2023).

[46] S. Overflow, “Stack overflow developer survey 2022.” https://survey.stackoverflow.co/2022 (accessed: May 18, 2023).

[47] J. Kjær Rask, F. Palludan Madsen, N. Battle, H. Daniel Macedo, and P. Gorm Larsen, “The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions,” *Electronic Proc. Theor. Comput. Sci.*, vol. 338, pp. 3–18, Aug. 2021, doi: 10.4204/EPTCS.338.3. Accessed: May 18, 2023. [Online]. Available: http://arxiv.org/abs/2108.02961v1

[48] e. a. PCMag, “Unit test,” PCMag. https://www.pcmag.com/encyclopedia/term/unit-test (accessed: May 21, 2023).

[49] e. a. PCMag, “Simulation,” PCMag. https://www.pcmag.com/encyclopedia/term/simulation (accessed: May 21, 2023).

[50] J. E. McDonough, “Test-driven development,” *Automated Unit Testing Abap*, 2021.

[51] M. Ivanković, G. Petrović, R. Just, and G. Fraser, “Code coverage at Google,” in *Proc. 2019 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, Tallinn Estonia, Aug. 2019, pp. 955–963, doi: 10.1145/3338906.3340459. Accessed: May 18, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3340459

[52] e. a. Florin Lipan, “Mockito.” https://github.com/lipanski/mockito (accessed: May 18, 2023).

[53] e. a. Aptitude, “What is a package manager?,” Aptitude. http://aptitude.alioth.debian.org/doc/en/pr01s02.html (accessed: May 21, 2023).

[54] P. Lam, J. Dietrich, and D. J. Pearce, “Putting the semantics into semantic versioning,” in *Proc. 2020 ACM SIGPLAN Int. Symp. New Ideas, New Paradigms, Reflections Program. Softw.*, Virtual USA, Nov. 2020, pp. 157–179, doi: 10.1145/3426428.3426922. Accessed: May 18, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3426428.3426922

[55] Sai Zhang, Cheng Zhang, and M. D. Ernst, “Automated documentation inference to explain failed tests,” in *2011 26th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE 2011)*, Lawrence, KS, USA, Nov. 2011, pp. 63–72, doi: 10.1109/ASE.2011.6100145. Accessed: May 30, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/6100145/

[56] t. e. ISO/IEC JTC 1/SC 22 Programming languages , and system software interfaces, “Information technology — programming languages — c,” International Organization for Standardization, Geneva, CH, Jun. 2018, vol. 520.

[57] P. S. Foundation, “The Python Language Reference.” https://docs.python.org/3/reference/ (accessed: May 21, 2023).

[58] “Verilog-ams language reference manual,” Accellera Systems Initiative, Elk Grove, CA 95758, USA, Jun. 2014.

[59] “IEEE Standard for VHDL Language Reference Manual,” IEEE. Accessed: Mar. 10, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/8938196/

[60] B. C. Schafer, and Z. Wang, “High-Level Synthesis Design Space Exploration: Past, Present, and Future,” *IEEE Trans. Computer-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, Oct. 2020, doi: 10.1109/TCAD.2019.2943570. Accessed: May 19, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/8847448/

[61] A. Mccaskey, T. Nguyen, et al., “Extending C++ for Heterogeneous Quantum-Classical Computing,” *ACM Trans. Quantum Comput.*, vol. 2, no. 2, pp. 1–36, Jun. 2021, doi: 10.1145/3462670. Accessed: May 19, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3462670

[62] N. Tietz, “Why rust's learning curve seems harsh, and ideas to reduce it.” https://ntietz.com/blog/rust-resources-learning-curve/ (accessed: May 20, 2023).

[63] T. Rohner, "Why is python good for research? benefits of the programming language," netguru. https://www.netguru.com/blog/python-research (accessed: May 20, 2023).

[64] J. Villar, J. Juan, et al., "Python as a hardware description language: A case study," in *2011 VII Southern Conf. Programmable Log. (Spl)*, Cordoba, Argentina, Apr. 2011, pp. 117–122, doi: 10.1109/SPL.2011.5782635. Accessed: May 19, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/5782635/

[65] e. a. Jan Decaluwe, "Myhdl." Accessed: May 20, 2023. [Online]. Available: http://www.myhdl.org/

[66] D. M. Ritchie, "The development of the C language," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 201–208, Mar. 1993, doi: 10.1145/155360.155580. Accessed: May 21, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/155360.155580

[67] G. Rossum, "Python for unix/c programmers copyright 1993 guido van rossum 1," in *Proc. NLUUG Najaarsconferentie*, 1993.

[68] R. Levick, and S. Fernandez, "We need a safer systems programming language," *Microsoft Secur. Response Center*, Jul. 2019. Accessed: May 21, 2023. [Online]. Available: https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/

[69] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Des. Automat. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, Sep. 2012, doi: 10.1007/s10617-012-9096-8. Accessed: May 24, 2023. [Online]. Available: http://link.springer.com/10.1007/s10617-012-9096-8

[70] H. Ye, C. Hao, et al., "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," in *2022 IEEE Int. Symp. High-Performance Comput. Architecture (Hpca)*, Seoul, Korea, Republic of, Apr. 2022, pp. 741–755, doi: 10.1109/HPCA53966.2022.00060. Accessed: May 24, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9773203/

[71] S. Lahti, P. Sjovall, J. Vanne, and T. D. Hamalainen, "Are We There Yet? A Study on the State of High-Level Synthesis," *IEEE Trans. Computer-Aided Des. Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019, doi: 10.1109/TCAD.2018.2834439. Accessed: May 24, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/8356004/

[72] H. Shahzad, A. Sanaullah, et al., "Reinforcement Learning Strategies for Compiler Optimization in High level Synthesis," in *2022 IEEE/ACM Eighth Workshop LLVM Compiler Infrastructure HPC (LLVM-Hpc)*, Dallas, TX, USA, Nov. 2022, pp. 13–22, doi: 10.1109/LLVM-HPC56686.2022.00007. Accessed: May 24, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10027131/

[73] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation* in Pldi 2022, New York, NY, USA, 2022, pp. 703–718, doi: 10.1145/3519939.3523446. Accessed: Jun. 15, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3519939.3523446

[74] T. Murayama, and Y. Gendai, "A top-down mixed-signal design methodology using a mixed-signal simulator and analog HDL," *Proc. EURO-DAC '96. Eur. Des. Automat. Conf. EURO-VHDL '96 Exhib.*, pp. 59–64, 1996, doi: 10.1109/EURDAC.1996.558078. Accessed: Jun. 17, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/558078/

[75] S. Mitra, A. Ain, P. Ghosh, and P. Dasgupta, "A study of modeling techniques in use in digital and mixed-signal domains for semi-formal verification," in *2010 IEEE Students Technol. Symp. (Techsym)*, Apr. 2010, pp. 103–108, doi: 10.1109/TECHSYM.2010.5469221.

[76] "Programming paradigms: What are the principles of programming?," 2020. Accessed: Jun. 18, 2023. [Online]. Available: https://www.ionos.com/digitalguide/websites/web-development/programming-paradigms/

[77] P. Van Roy, "Classification of the principal programming paradigms," 2009. Accessed: Jun. 18, 2023. [Online]. Available: https://www.info.ucl.ac.be/~pvr/paradigms.html

[78] P. Van Roy, "Programming Paradigms for Dummies: What Every Programmer Should Know," Apr. 2012.

[79] L. Cardelli, "Bad engineering properties of object-orient languages," *ACM Comput. Surveys*, vol. 28, no. 4es, 1996, doi: 10.1145/242224.242415. Accessed: Jun. 18, 2023. [Online]. Available: https://doi.org/10.1145/242224.242415

[80] O. Port, and Y. Etsion, "DFiant: A dataflow hardware description language," in *2017 27th Int. Conf. Field Programmable Log. Appl. (Fpl)*, Sep. 2017, pp. 1–4, doi: 10.23919/FPL.2017.8056858.

[81] T. Darwish, and M. Bayoumi, "Trends in Low-Power VLSI Design," in *Elect. Eng. Handbook*, Elsevier, pp. 263–280. Accessed: Jun. 8, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780121709600500220

[82] "Logic synthesis," 2023. Accessed: Jun. 17, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Logic_synthesis&oldid=1157529091#cite_note-Verilog_2005-1

[83] M. Martin, "What is a Functional Requirement in Software Engineering?," 2023. Accessed: Jun. 17, 2023. [Online]. Available: https://www.guru99.com/functional-requirement-specification-example.html

[84] D. Huang, and H. Wu, "Virtualization," in *Mobile Cloud Comput.*, Elsevier, pp. 31–64. Accessed: Jun. 4, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B978012809641300003X

[85] L. De Moura, and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools Algorithms Construction Anal. Syst.*, D. Hutchison, T. Kanade, et al., Eds., vol. 4963, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 337–340. Accessed: Jun. 9, 2023. [Online]. Available: http://link.springer.com/10.1007/978-3-540-78800-3_24

[86] T. Freeman, and F. Pfenning, "Refinement types for ML," *ACM SIGPLAN Notices*, vol. 26, no. 6, pp. 268–277, Jun. 1991, doi: 10.1145/113446.113468. Accessed: Jun. 9, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/113446.113468

[87] T. Mailund, *Functional Programming in R*, Berkeley, CA: Apress, 2017. Accessed: Jun. 10, 2023. [Online]. Available: http://link.springer.com/10.1007/978-1-4842-2746-6

[88] HaskellWiki, "State Monad." https://wiki.haskell.org/State_Monad (accessed: Jun. 10, 2023).

[89] The Unicode Consortium, "The Unicode Standard, Version 10.0," The Unicode Consortium, 2017. Accessed: Jun. 7, 2023. [Online]. Available: https://www.unicode.org/versions/Unicode10.0.0/

[90] "IEEE Standard for Binary Floating-Point Arithmetic," IEEE. Accessed: Jun. 7, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/30711/

[91] HaskellWiki, "Algebraic data type." https://wiki.haskell.org/Algebraic_data_type (accessed: Jun. 7, 2023).

[92] F. O.-L. D. of Computing, "Aggregate type." https://foldoc.org/aggregate (accessed: Jun. 7, 2023).

[93] B. O'Sullivan, J. Goerzen, and D. B. Stewart, *Real World Haskell: Code You Can Believe In*, 1. ed., [nachdr.], Beijing: O'Reilly, 2010.

[94] S. Peyton Jones, "How to make a fast curry: push/enter vs eval/apply," in *Int. Conf. Functional Program.*, Sep. 2004, pp. 4–15. [Online]. Available: https://www.microsoft.com/en-us/research/publication/make-fast-curry-pushenter-vs-evalapply/

[95] O. D. University, "CS390: Turing Completeness," 2023. Accessed: Jun. 7, 2023. [Online]. Available: https://www.cs.odu.edu/~zeil/cs390/latest/Public/turing-complete/index.html

[96] J. Rose, "OpenJDK Wiki HotSpot." https://wiki.openjdk.org/display/HotSpot (accessed: Jun. 11, 2023).

[97] R. Nystrom, *Crafting Interpreters*, Pacific Grove: Smashwords Edition, 2021.

[98] M. Hirsz, "Logos: create ridiculously fast lexers."

[99] crates.io, "Logos dependents." https://crates.io/crates/logos/reverse_dependencies (accessed: Jun. 10, 2023).

[100] M. Johnson, and J. Zelenski, "CS143: Compilers - miscellaneous parsing," 2008. Accessed: Jun. 7, 2023. [Online]. Available: https://suif.stanford.edu/dragonbook/lecture-notes/Stanford-CS143/12-Miscellaneous-Parsing.pdf

[101] e. a. Barretto Joshua, "Chumsky: A Rust Parser Combinator Library." https://github.com/zesterer/chumsky (accessed: Jun. 10, 2023).

[102] R. Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," *Trans. Amer. Math. Soc.*, vol. 146, p. 29, Dec. 1969, doi: 10.2307/1995158. Accessed: Jun. 11, 2023. [Online]. Available: https://www.jstor.org/stable/1995158?origin=crossref

[103] G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. P. Jones, "GADTs meet their match: pattern-matching warnings that account for GADTs, guards, and laziness," in *Proc. 20th ACM SIGPLAN Int. Conf. Functional Program.*, Vancouver BC Canada, Aug. 2015, pp. 424–436, doi: 10.1145/2784731.2784748. Accessed: Jun. 11, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/2784731.2784748

[104] P. Kalvoda, and T. S. Kerckhove, "Structural and semantic pattern matching analysis in Haskell," 2019, doi: 10.48550/ARXIV.1909.04160. Accessed: Jun. 11, 2023. [Online]. Available: https://arxiv.org/abs/1909.04160

[105] H. Xi, C. Chen, and G. Chen, "Guarded recursive datatype constructors," in *Proc. 30th ACM SIGPLAN-SIGACT Symp. Princ. Program. Languages*, New Orleans Louisiana USA, Jan. 2003, pp. 224–235, doi: 10.1145/604131.604150. Accessed: Jun. 11, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/604131.604150

[106] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, "The Java® Virtual Machine Specification," Oracle Corporation, Feb. 8, 2019. Accessed: Jun. 11, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se12/html/index.html

[107] T. H. Cormen, Ed., *Introduction to Algorithms*, 3rd ed, Cambridge, Mass: MIT Press, 2009.

[108] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, New York: Prentice Hall, 1993.

[109] X. Chen, P. Stroobant, M. Pickavet, and W. Bogaerts, "Graph Representations for Programmable Photonic Circuits," *J. Lightw. Technol.*, vol. 38, no. 15, pp. 4009–4018, Aug. 2020, doi: 10.1109/JLT.2020.2984990. Accessed: Mar. 10, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9056549/

[110] F. V. Kerchove, X. Chen, D. Colle, W. Bogaerts, and M. Pickavet, "Adapting Routing Algorithms to Programmable Photonic Circuits," in *2022 Eur. Conf. Opt. Communication (Ecoc)*, Sep. 2022, pp. 1–4.

[111] Z. Gao, X. Chen, et al., "Automatic Realization of Light Processing Functions for Programmable Photonics," in *2022 IEEE Photon. Conf. (Ipc)*, Nov. 2022, pp. 1–2, doi: 10.1109/IPC53466.2022.9975757.

[112] F. V. Kerchove, X. Chen, et al., "An Automated Router with Optical Resource Adaptation," *J. Lightw. Technol.*, pp. 1–13, 2023, doi: 10.1109/JLT.2023.3275385.

[113] L. de Moura, N. Bjørner, L. Nechmanson, and W. Christoph, "Programming Z3," Microsoft Research. https://z3prover.github.io/papers/programmingz3.html#sec-solver-implementations (accessed: Jun. 14, 2023).

[114] Wikipedia, "Marshalling (computer science)," Wikipedia. https://en.wikipedia.org/wiki/Marshalling_(computer_science) (accessed: Jun. 14, 2023).

[115] C. Dictionary, "marshalling," Collins. https://www.collinsdictionary.com/dictionary/english/marshalling (accessed: Jun. 14, 2023).

[116] W. Xu, Y. Guo, et al., "Fully Integrated Solid-State LiDAR Transmitter on a Multi-Layer Silicon-Nitride-on-Silicon Photonic Platform," in *2022 Opt. Fiber Commun. Conf. Exhib. (Ofc)*, Mar. 2022, pp. 1–3.

[117] B. Van Veen, and K. Buckley, "Beamforming: a versatile approach to spatial filtering," *IEEE ASSP Mag.*, vol. 5, no. 2, pp. 4–24, Apr. 1988, doi: 10.1109/53.665.

[118] Y. Zou, W. Rave, and G. Fettweis, "Analog Beamsteering for Flexible Hybrid Beamforming Design in Mmwave Communications," arXiv, 2017. Accessed: Jun. 17, 2023. [Online]. Available: http://arxiv.org/abs/1705.04943

[119] "What is QAM or Quadrature Amplitude Modulation? - everything RF." Accessed: Jun. 16, 2023. [Online]. Available: https://www.everythingrf.com/community/what-is-qam-or-quadrature-amplitude-modulation

[120] S. Ishimura, T. Fukui, et al., "64-QAM Self-Coherent Transmission Using Symmetric Silicon Photonic Stokes-Vector Receiver," *Opt. Fiber Communication Conf. (OFC) 2022*, 2022, doi: 10.1364/OFC.2022.M4J.6. Accessed: Jun. 16, 2023. [Online]. Available: https://opg.optica.org/abstract.cfm?URI=OFC-2022-M4J.6

[121] A. H. Talkhooncheh, A. Zilkie, G. Yu, R. Shafiiha, and A. Emami, "A 200Gb/s QAM-16 Silicon Photonic Transmitter with 4 Binary-Driven EAMs in An MZI Structure," in *2023 Opt. Fiber Commun. Conf. Exhib. (Ofc)*, Mar. 2023, pp. 1–3, doi: 10.1364/OFC.2023.M1E.6.

[122] I. Keysight Technologies, "Error Vector Magnitude (Digital Demodulation)." Accessed: Jun. 16, 2023. [Online]. Available: https://rfmw.em.keysight.com/wireless/helpfiles/89600b/webhelp/subsystems/digdemod/content/digdemod_symtblerrdata_evm.htm

[123] A. Ruocco, D. Thourhout, and W. Bogaerts, "SOI Lattice Filters Design Framework: from Functional Parameters to Layout," 2013. Accessed: Jun. 17, 2023. [Online]. Available: https://www.semanticscholar.org/paper/SOI-Lattice-Filters-Design-Framework%3A-from-to-Ruocco-Thourhout/5d818417e4aea00d6041aec55964207053b9af43

[124] B. Guan, S. S. Djordjevic, et al., "CMOS Compatible Reconfigurable Silicon Photonic Lattice Filters Using Cascaded Unit Cells for RF-Photonic Processing," *IEEE J. Sel. Topics Quantum Electronics*, vol. 20, no. 4, pp. 359–368, Jul. 2014, doi: 10.1109/JSTQE.2013.2296233.

[125] F. Shokraneh, S. Geoffroy-Gagnon, M. S. Nezami, and O. Liboiron-Ladouceur, "A Single Layer Neural Network Implemented by a 4\textbackslash times 4 MZI-Based Optical Processor," *IEEE Photon. J.*, vol. 11, no. 6, pp. 1–12, Dec. 2019, doi: 10.1109/JPHOT.2019.2952562.

[126] R. project, "What is chalk?." https://rust-lang.github.io/chalk/book/what_is_chalk.html (accessed: Jun. 17, 2020).

ANNEX A: PROGRAMMING PARADIGM POSTER

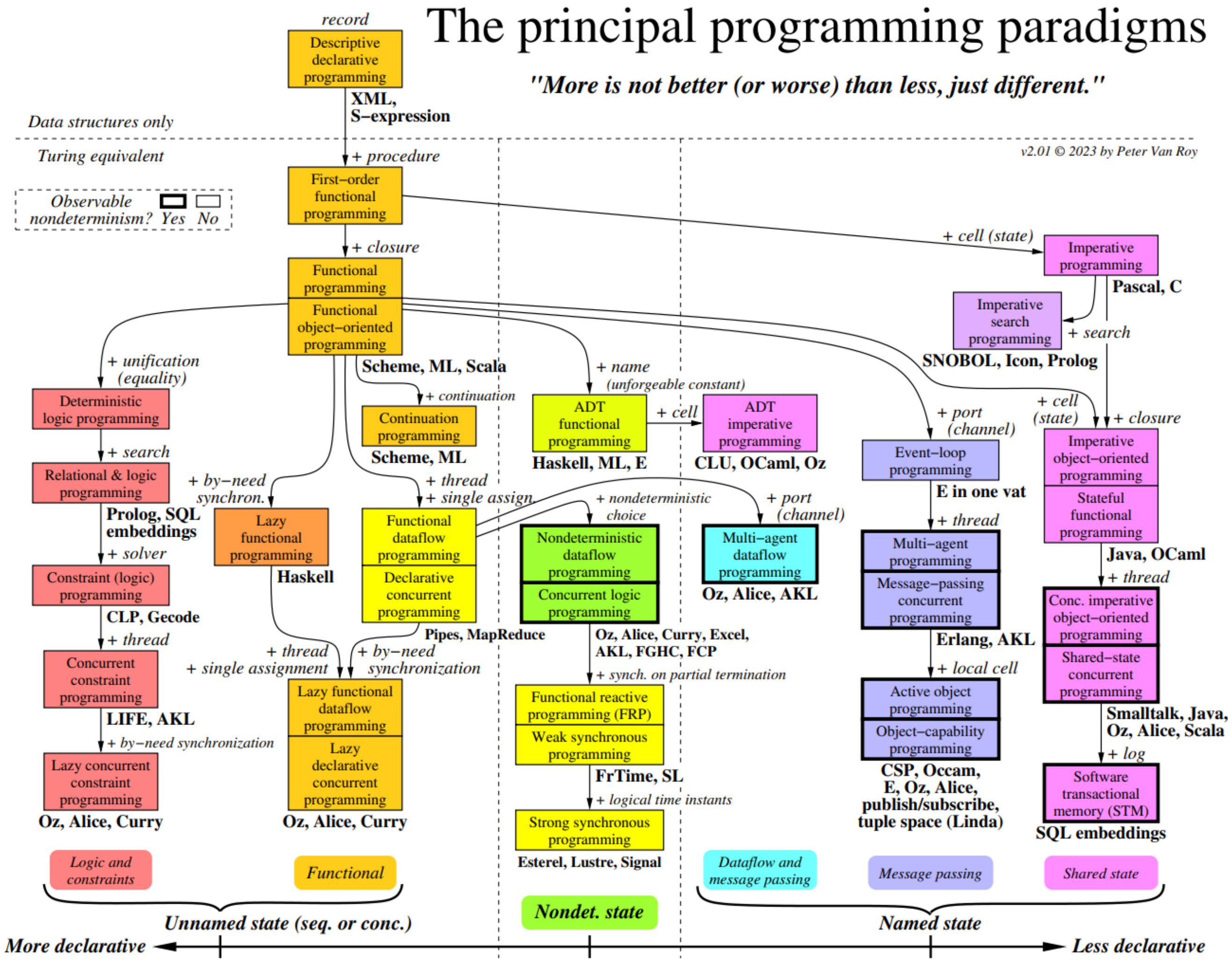


FIGURE A.1 | Programming paradigms poster, showing the different programming paradigms and their relationships. Created by *Peter Van Roy* [77].

Annex B: AST data structure: overview

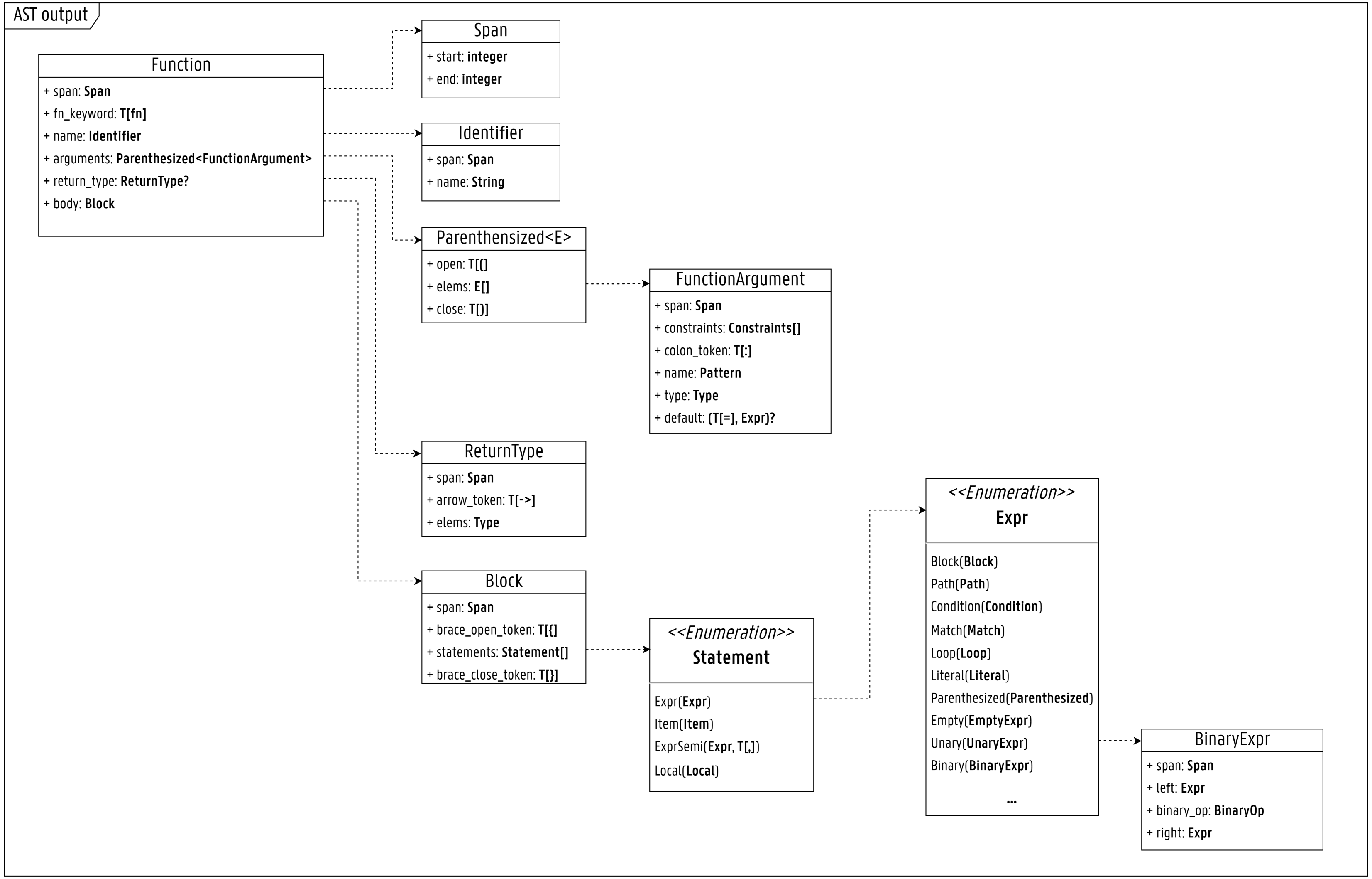


Figure B.1 | UML diagram of parts of the AST relevant for Section 5.4.c. It is incomplete since phos contains 120 data structures to fully represent the AST .

Annex C: Bytecode execution

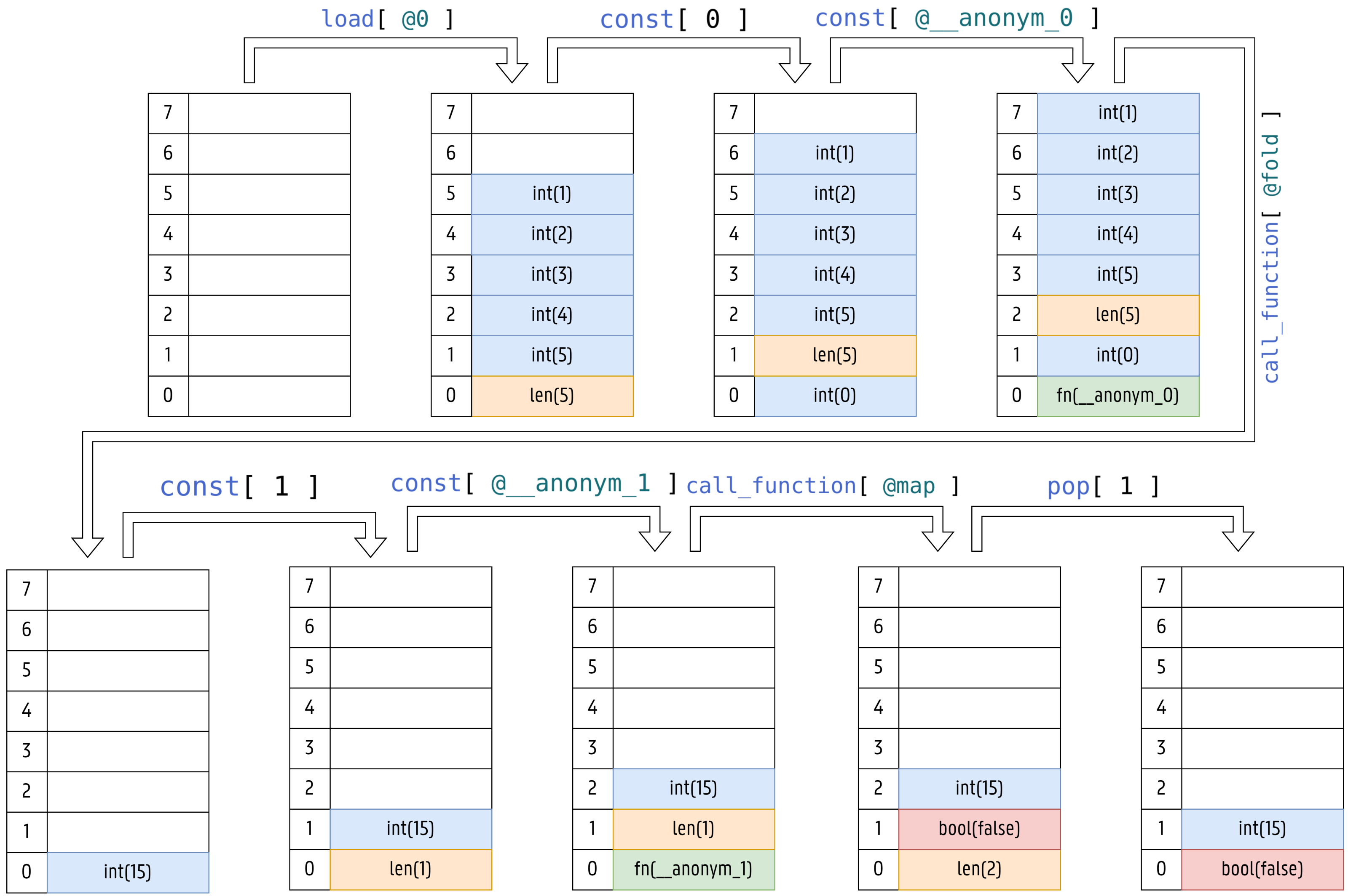


Figure C.1 | Execution diagram of the stack of Section 5.5.d, showing the stack before and after the execution of each of the bytecode instructions.

Annex D: Graph representation of a mesh

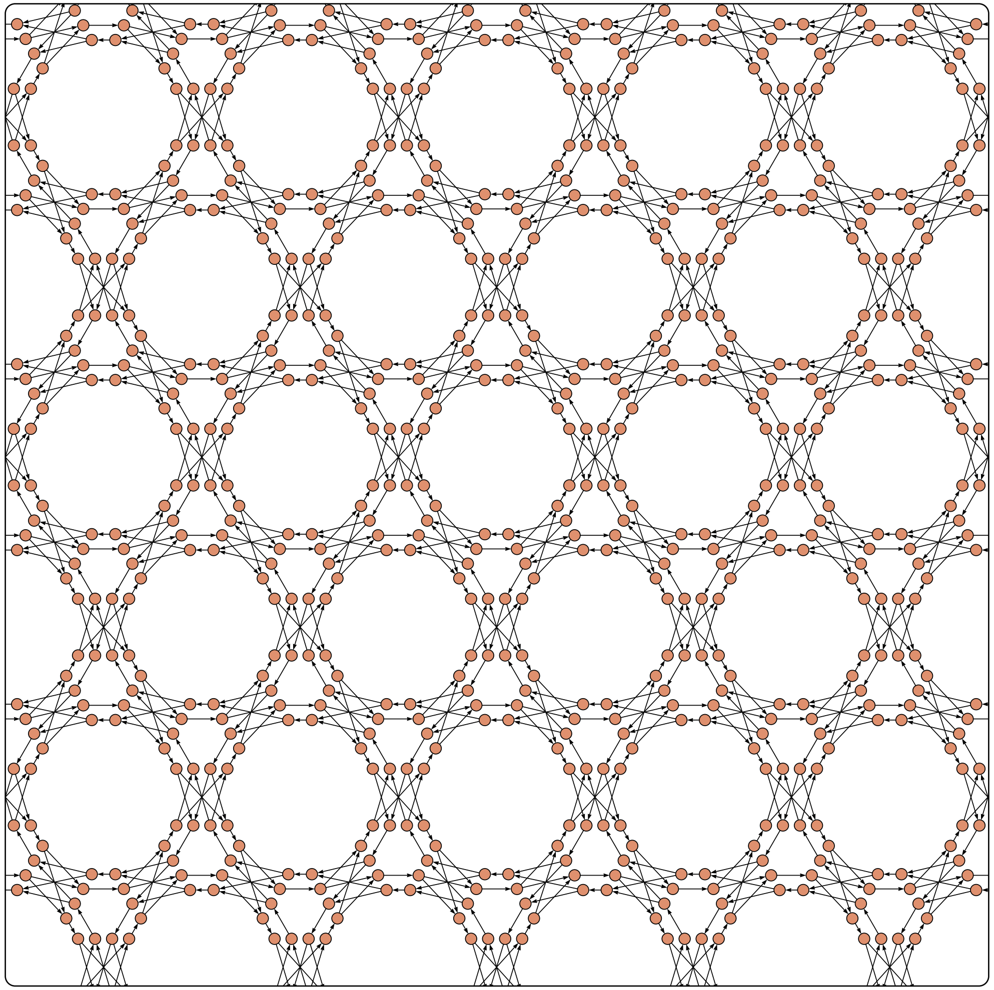


Figure D.1 Graph representation of a mesh, showing the direction that light is travelling in, and all of the possible connections. Based on the work of Xiangfeng Chen, et al. [109]. This visualisation was created with the collaboration of Léo Masson, as mentioned in the acknowledgements.

Annex E: Marshalling library example

```
syn main(input: optical, modulated: electrical) -> (optical, optical) {
    input |> split((0.5, 0.5))
          |> (modulator(modulated, type_: ModulationType::Amplitude), _)
}
```

PHÔS

Listing E.1 *PHÔS* code performing splitting and modulation, used in Listing E.2.

```python
# Import the marshalling library
import phos as ph

# Import the platform-support package
import prg_device as prg

# Create the device with the specific support package
device = ph.Device(prg.DeviceDescription())

# Try and load a module, this will compile the module
module = device.load_module("module.phos")

# Create the I/O, each `io` calls returns an input and an output
electrical    = device.electrical_input(0)
(input, _)    = device.io(0)
(_, output0) = device.io(1)
(_, output1) = device.io(2)

# Instantiate the module, first passing in the inputs and parameters
# and then the outputs. This run evaluation of the module.
instance = module(input, electrical, name="Module Instance")
                .output(output0, output1)

# Build the design, this will run synthesis, with area optimisation
built = device.synthesise(instance, optimisation="area")

# Create the user HAL in the `./iq_modulator` directory
built.generate_hal("./iq_modulator")
# Create the firmware in the `./iq_modulator.bin` file
build.generate_firmware("./iq_modulator.bin")
```

Python

Listing E.2 Example of the marshalling library, showing the configuration of the different components of the synthesis toolchain.

```python
# Import numpy
import numpy as np

# Import plotting library
import matplotlib.pyplot as plt

# We set the simulation parameters
dt = 1e-12
tstop = 1e-6
bitrate = 10e9
t = np.arange(0, tstop, dt)
bit_timing = 1 / bitrate

# generate a test PRBS sequence
prbs = gen_prbs(12, tstop / bit_timing, 0x17D)
prbs = np.array([1.0 if x else 0.0 for x in prbs])

# Create the simulator
simulator = device.simulator()

# Create the source with some noise
noise = simulator.noise_source(0, 0.01)
source = simulator.source(nm(1550), noise=noise)

# Simulate the module
(output0, output1) = simulator.simulate(module, t).with_input(source, prbs)

# Plot `output0` and `output1` with respect to `t`
plt.plot(t, output0)
plt.plot(t, output1)
plt.show()
```

Listing E.3 | Example of the marshalling library for simulation, showing the simulation of a module.

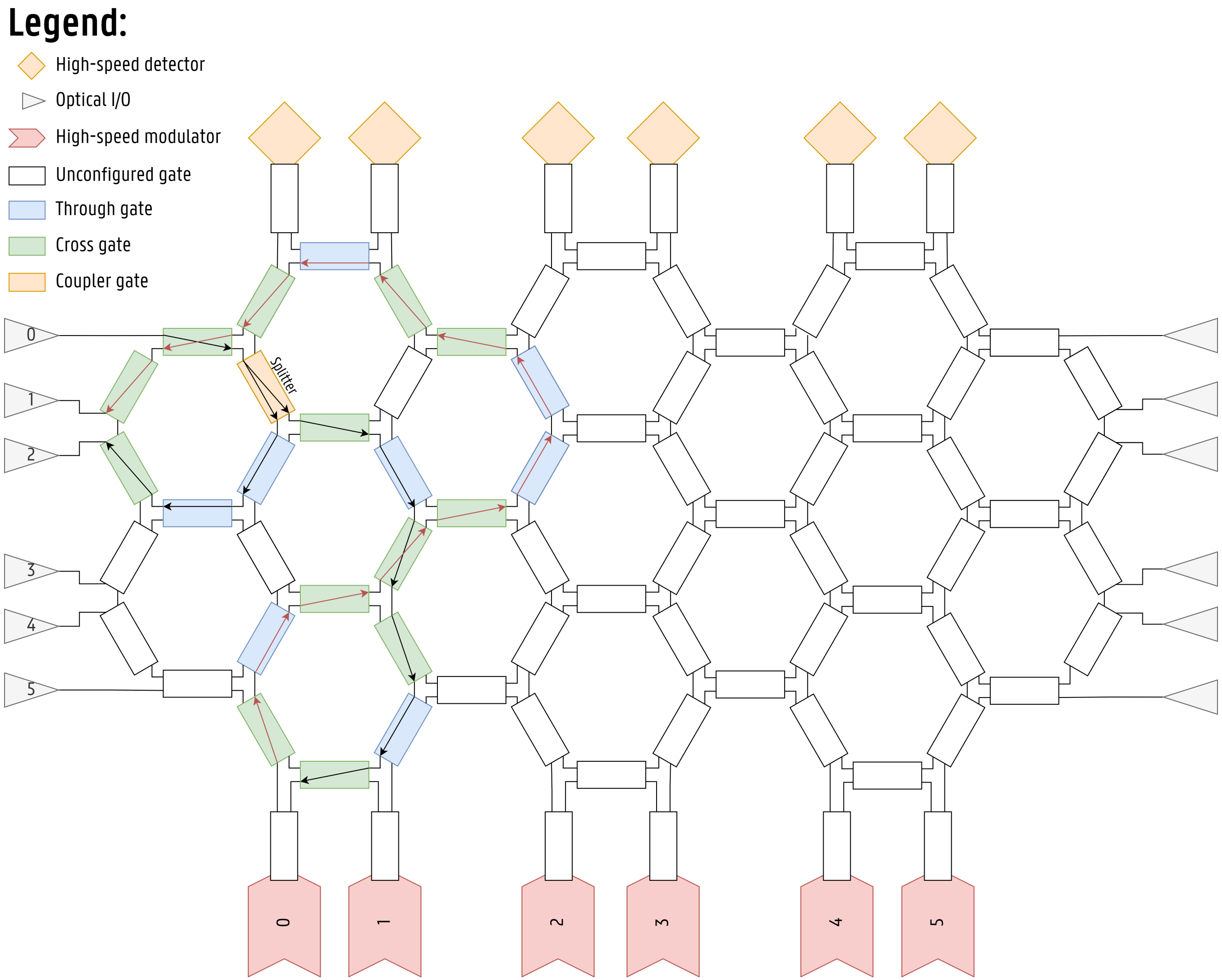


Figure E.1 Layout of the circuit in the marshalling library example, showing the path that the light takes inside of the photonic processor, as well as the state of each photonic gate. It also shows the path of the modulated light in red, and highlight the splitter.

Annex F: Example: Beam forming system

```
// Create a simple beamforming system
// This system takes an input optical signal and a set of electrical signals
//  1. It splits the input optical signal into N optical signals
//  2. It ensures that the phase of each of the optical signals is the same
//  3. It modulates each of the optical signals with the electrical signals
//  4. It ensures that the delay of each of the optical signals is the same
syn beam_forming(
    input: optical,
    phase_shifts: (electrical...),
) -> (optical...) {
    input                                       // optical
        |> split(splat(1.0, phase_shifts.len()))    // (optical...)
        |> constrain(d_phase = 0)                   // (optical...)
        |> zip(phase_shifts)                        // ((optical, electrical)...)
        |> map(set modulate(type: Modulation::Phase))   // (optical...)
        |> constrain(d_delay = 0)                   // (optical...)
}
```

PHÔS

Listing F.1 Example in *PHÔS* of beamforming system, parametric over the number of channels.

Annex G: Example: coherent 16-QAM transmitter

```
// Coherent transmitter, modulates four binary signals into a 16-QAM signal.
// 1. the signal is split into four, each signal is a fraction of the input signal
// 2. each signal is zipped with its corresponding electrical signal
// 3. each signal is modulated using amplitude modulation
// 4. the phase difference between the four signals is constrained to 90° between each
//    other
// 5. the four signals are merged back into one
// Note: the splitting ratios and order of modulation are chosen to match the modulation
//       order for the coherent transmission
syn coherent_transmitter(
    input: optical,
    (a, b, c, d): (electrical, electrical, electrical, electrical),
) -> optical {
    input                                       // optical
        |> split((1.0, 1.0, 0.5, 0.5))          // (optical, optical, optical, optical)
        |> zip((a, c, b, d))                    // ((optical, electrical), ...)
        |> modulate(type = Modulation::Amplitude)  // (optical, optical, optical, optical)
        |> constrain(d_phase = 90°)             // (optical, optical, optical, optical)
        |> merge()                              // optical
}
```

Listing G.1 Example in *PHÔS* of a 16- QAM modulator. The four binary sources are modulated on a common laser source, and then interfered together.

Annex H: Example: lattice filter

```phos
// The kinds of filter that can be used.
// For this example, we only support Chebyshev and Butterworth.
enum FilterKind {
    Chebyshev(uint),
    Butterworth(uint),
}

// Computes the coefficient of a given kind of filter
fn filter_kind_coefficients(
    filter_kind: FilterKind,
) -> ((Fraction, Phase)...) {
    ...
}

// Implements the latice filter
syn lattice_filter(
    a: optical,
    b: optical,
    filter_kind: FilterKind,
) -> (optical, optical) {
    // Steps:
    // 1. we compute the coefficients of the filter in the form ((Fraction, Phase)...)
    //    that is: a list of coefficients and phases
    // 2. we fold over the list of coefficients and phases, that is, we iterate over
    //    the list, and for each element, we apply a function to the current value
    //    and the element, and return the result as the new current value, turning a
    //    list of coefficients and phases into a single value, our starting value is
    //    the a tuple of both input signals. In the fold we:
    //   1. couple the signals together with the computed coefficient
    //   2. we constrain the phase difference between the two signals, imposing the computed
    //      phase difference
    filter_kind_coefficients(filter_kind)       // ((Fraction, Phase)...)
        |> fold((a, b), |acc, (coeff, phase)| { // Fold over the list of coefficients:
            acc                                   // (optical, optical)
                |> coupler(coeff)                 // (optical, optical)
                |> constrain(d_phase = phase)     // (optical, optical)
        })                                      // (optical, optical)
}
```

Listing H.1 | Example in *PHÔS* of a parametric lattice filter.

Annex I: Example: MVM

PHÔS

```
// A single Mach-Zehnder interferometer based gate
syn mzi_gate(
    a: optical,
    b: optical,
    (beta, theta): (Phase, Phase),
) -> (optical, optical) {
    (a, b)
        |> coupler(0.5)
        |> constrain(d_phase = beta)
        |> coupler(0.5)
        |> constrain(d_phase = theta)
}

// Produces a 4x4 matrix-vector multiplier
syn matrix_vector_multiply(
    source: optical,
    (a, b, c, d): (electrical, electrical, electrical, electrical),
    coefficients: (
        (Phase, Phase), (Phase, Phase), (Phase, Phase),
        (Phase, Phase), (Phase, Phase), (Phase, Phase)
    )
) -> (electrical, electrical, electrical, electrical) {
    let (ref_a, ref_b, ref_c, ref_d, rest...) = source |> split(splat(1.0, 8));
    let (a, b, c, d) = (a, b, c, d)
        |> zip((ref_a, ref_b, ref_c, ref_d))
        |> modulate(type_: Modulation::Amplitude)

    let (c1, d1) = mzi_gate(c, d, coefficients.0);
    let (b1, c2) = mzi_gate(b, c1, coefficients.1);
    let (y1, b2) = mzi_gate(a, b1, coefficients.3);
    let (c3, d2) = mzi_gate(c2, d1, coefficients.2);
    let (y2, c4) = mzi_gate(b2, c3, coefficients.4);
    let (y3, y4) = mzi_gate(c4, d2, coefficients.5);

    (y1, y2, y3, y4)
        |> zip(rest)
        |> demodulate(type_: Modulation::Coherent)
}
```

Listing I.1 | Example in *PHÔS* of an analog matrix-vector multiplier.