Acknowledgements

I would like to express my deepest gratitude to Prof. Wim Bogaerts and Prof. 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 will carry with me for the rest of my career. It is with great pleasure that I write this document to share my findings with them and others within the community.

I would also like to give my most heartfelt thanks to the best friend one could ever ask for: 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.

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.

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.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

Table of contents

Glossary

2x2 tunable coupler	*A tunable coupler with two inputs and two outputs* 5
ADT	*Algebraic Data Type* ..... XII, 56, 64, 70
API	*Application Programming Interface* 11, 14, 15, 16, 17, 18, 19, 20, 23, 33, 54
ASIC	*Application Specific Integrated Circuit* 16
AST	*Abstract Syntax Tree* 70
CPLD	*Complex Programmable Logic Device* 3
CPU	*Central Processing Unit* 27
DRY	*Don't Repeat Yourself* 18, 48
DSL	*Domain Specific Language* 8, 11, 22, 23
DSP	*Digital Signal Processor* .....
ECS	*Entity-Component-System* .....
EDA	*Electronic Design Automation* 47, 50
FIR	*Finite Impulse Response* 5, 6
FPGA	*Field Programmable Gate Array* 1, 3, 8, 16, 23, 26, 27, 46, 49
FPPGA	*Field Programmable Photonic Gate Array* 3
FSR	*Free Spectral Range* 45
GPL-3.0	*GNU General Public License version 3.0* 21
GPU	*Graphics Processing Unit – also commonly used for highly parallel computing and machine learning* 27
HAL	*Hardware Abstraction Layer* 35, 36, 43, 47, 48, 50, 52, 69
HDL	*Hardware Description Language* 8, 10, 11, 12, 16, 17, 23, 24, 26, 27, 28
HLS	*High Level Synthesis* 22, 23, 27, 28
HPC	*High Performance Computing* 10
HTTP	*Hypertext Transfer Protocol – the protocol used for web navigation* 16
IC	*Integrated Circuit* 26
IDE	*Integrated Development Environment* 15
IIR	*Infinite Impulse Response* 5, 6
I/O	*Input/Output* .....
IP	*Intellectual Property* 17, 54, 73
JTAG	*Joint Test Action Group – A standard for testing integrated circuits* 13
LLVM	*Low Level Virtual Machine* 27, 74
LSP	*Language Server Protocol* 15
LUT	*Look Up Table* 26, 48
MEMS	*Microelectromechanical Systems* 5
MZI	*Mach-Zehnder Interferometer* 36, 37, 49

PHÔS	*Photonic Hardware Description Language* X, XI, XII, 19, 21, 51, 52, 53, 54, 55, 56, 57, 58, 59, 61, 62, 63, 64, 65, 66, 67, 68
PIC	*Photonic Integrated Circuit* ..... X, 1, 2, 3, 4, 17
PPA	*Power, Performance, Area* 27
PRG	*Photonics Research Group* 46
RF	*Radio Frequency* 1, 37, 40
RTL	*Register Transfer Level* 8, 26, 27, 28, 32
SI	*Système international – the international system of units* ..... XII, 57, 58
SPICE	*Simulation Program with Integrated Circuit Emphasis* 1, 2, 8, 16, 23, 46
SQL	*Structured Query Language* 8
TDD	*Test Driven Development* 16
VERILOG-A	*A continuous-time subset of Verilog-AMS (*Verilog for Analog and Mixed Signal*)* 2
VERILOG-AMS	*Verilog for Analog and Mixed Signal* ..... IX, 2, 8, 10, 21, 22, 23, 33
VHDL	*VHSIC (*Very High Speed Integrated Circuit*) Hardware Description Language* ..... XII, 10, 21, 22, 28, 29, 51
VHSIC	*Very High Speed Integrated Circuit* ..... IX, XII, 10
VM	*Virtual Machine* 53, 65, 69

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 color blind and visually impaired people in mind. The colors used are generally chosen to have good contrast for color blind individuals, as can be seen here. Most colored elements are accompanied by an icon, generally 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. They are generally adapter from the literature and are used for important elements that are not common knowledge for photonic engineers.

Definition usually have a source in the footer

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

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

TODO

1.1 Motivation

This section serves as the motivation for the research and development of 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*) [3] or Photonic Processors. As with all other types of circuit designs, such as digital electronic circuits, analog electronic circuits and RF circuits, appropriate abstractions can allow the designer and/or engineer to focus on the functionality of their design rather than the implementation [4]. 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 [5].

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) [5]. In turn, this allows the designer to focus on what truly matters to them: the functionality of their design.

Currently, a lot of the work being done, on photonic circuits, is done at a very low level of abstraction, close to the component-level [6]. This leads to several issues for broader access to the fields of photonic circuit design. Firstly, it requires expertise and understanding of the photonic component, their physics and the, sometimes complex, relationship between all of their design parameters. Secondly, it requires a large amount of time and effort to design and simulate a photonic circuit. Physical simulation of photonic circuit is generally slow [6, 7], which has led to efforts to simulate photonic circuit using SPICE (*Simulation Program with Integrated Circuit Emphasis*) [8]. Finally, the design and implementation of photonic circuit is generally expensive, requiring taping-out of the design and working with a froundry for fabrication. This increases the cost, but also increases the time to market for the product [9].

This therefore means, that there is a large interest in constructing new abstractions, method of simulation and design tools for photonic circuit design. Especially for rapid prototyping and iteration of photonic circuits. This is the reason that research in the field of programmable photonics, and especially recirculating programmable photonics has had growing interest in the past few years. This master's thesis has for purpose to find new ways in which the user can easily design their photonic circuit and program them onto those programmable PICs (*Photonic Integrated Circuit*) [9].

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 electronic, to provide control and feedback loops and RF (*Radio Frequency*) to benefit from the high bandwidth, high speed capabilities of photonics [10]. Therefore, it is of interest to the user to co-simulate [6, 11] their photonic circuits with the other components of their systems. This is a problem that is partly addressed using SPICE simulation [8]. However, especially with regards to digital co-

simulation, SPICE tools are often lacking, making the process difficult [12], relying instead on alternatives such as Verilog-A (*A continuous-time subset of Verilog-AMS (*Verilog for Analog and Mixed Signal*)*).

In this work, the beginning of a solution to these problems will be offered by introducing a new way of designing photonic circuit using code, a novel way of simulating these circuits and a complete workflow for the design and programming of circuit will be presented. 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 as well.

1.1.a Research questions

The main goal of this work is to design a system to program photonic circuits, this entails the following:

1. How to express the user's intent?
    - What programming languages and paradigms are best suited?
    - What workflows are best suited?
    - How does the user test and verify their design?
2. How to translate that intent into a programmable 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?

The remainder of this work will be structured following these questions until the formulation of thorough answers to them; followed by a series of mockups to demonstrate the potential use of the workflow. Finally, demonstrations based on the prototype of the aforementioned workflow will be presented showing the potential and the capabilities of the simulation system.

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 basic of photonic processors. This chapter will therefore start by discussing what photonic processors are, what niche they fill and how they work. From this, the chapter will then move on to discuss the different types of photonic processors and how they differ from each other. Finally, this chapter will conclude with the first and most important assumption made in all subsequent design decisions.

In this document, the names Photonic FPGA and Photonic Processor are used interchangeably. They are both used to refer to the same thing, that being 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

In essence, a photonic FPGA or photonic processor is the optical analogue to the traditional digital FPGA. It is composed of a certain number of 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 contained within waveguide to perform analog processing tasks.

However, it is interesting to note that, just like traditional FPGAs, there are devices that are more general forms of programmable PIC (*Photonic Integrated Circuit*)[3] than others, just like CPLDs (*Complex Programmable Logic Device*) are less general forms of FPGAs. As any PIC that has 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 see that four large categories of PIC can be built based on their programmability. The first ones (a) are 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 function, the tunable element could be a tunable coupler, modulator, phase shifter, etc. and allows the designer to tweak the properties of their circuit during use, for purposes such as calibration, temperature compensation, signal modulation or more generally, altering the usage 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. Additionally, 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.g.

(a) (b) (c) (d)

FIGURE 1 A hierachy of programmable PICs (*Photonic Integrated Circuit*), starting at the non-programmable single function PIC (a), moving then to the tunable PIC (b), the feedfoward 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 will be discussed in Section 2.1.g, the recirculating mesh is the most general kind of programmable PIC, but also the most difficult to represent with a logic flow of operation due to the fact that the light can travel in either direction. Therefore, the following question may be asked:

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

This question, which will be the driving factor behind this first section, will be answered in Section 2.2.c. However, before answering this question, it is necessary to first discuss the different types of photonic processors and how they differ from each other. Additionally, the answer to that question will show that the solution suggested in this thesis is also applicable for feedfoward systems.

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 certain 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 low dispersion allowing for 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 important for high speed communication.

TUNABLE 2X2 COUPLERS

A 2x2 tunable coupler is a structure that allows two waveguides to interact in a pre-determined way. It is composed of two waveguides whose coupling, that being 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 coupler is given. 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 mostly 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, by allowing signals to cross with little to no interference. The final state allows the user to interfere two optical signals together based on predefined proportions. This is useful for applications such as filtering for ring resonators or splitting.

(a) (b) (c) (d)

Figure 2 Different states of a 2x2 optical coupler, (a) a simplified coupler, (b) in "bar" mode, (c) in "cross" mode, (d) in "partial" mode.

There are many construction techniques for building 2x2 couplers, each with their own advantages and disadvantages. The most common ones are the Mach-Zehnder interferometers with two phase shifters. However, other techniques involve the user of MEMS (*Microelectromechanical Systems*) or liquid crystals [3, 14, 15].

Detectors

todo

2.1.b Meshes

todo

2.1.c Fast modulators

2.1.d Fast detectors

2.1.e Amplifiers

2.1.f Filters

2.1.g Feedforward and recirculating mesh

Both architecture rely on the same components [3], 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 tuned to achieve the user's intent. Additionally, to provide more functionality, the meshed gates can also be connected to other devices, such as high speed modulators, amplifiers, etc. [3, 14, 15]

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

As it has been shown[15], recirculating meshes offer the ability to create more advanced structures such as IIR (*Infinite Impulse Response*) elements, whereas feedforward architectures are limited to FIR (*Finite Impulse Response*) systems. This is due to inherent infinite impulse response of the ring resonator cell, while in a feedforward architecture, the Mach-Zehnder

interferometers have a finite impulse response. But, not only does the recirculating mesh allow the creation of IIR cells, it still allows the designer to create FIR cells when needed.

2.1.h Potential use cases of photonic processors

2.1.i Embedding of photonic processor in a larger system

2.2 Circuit representation

2.2.a Netlist and nets

2.2.b Bi-directional systems

2.2.c Feedforward approximation

As mentioned in Section 2.1, a type of photonic processor is the feedfoward processor, which, assumes that light "flows" from an input port to an output port in a single direction. However, we are interested in the more complex, more capable processor that uses recirculating meshes. Therefore, one may wonder if the assumption that one can design a generic circuit using a feedforward approximation is valid. In this section, we will show that, given a sufficient level of abstraction, any bi-directional photonic circuit can be represented by an equivalent, higher-level, feedforward circuit.

Theory has shown that one may view a waveguide as a four port devices, with each end of the waveguide being composed of two ports: an incoming and an outgoing port. This is due to the fact that, in a waveguide, light can travel in both directions with little to no interactions. This means that, in a waveguide, one can have two guiding modes, one in each direction [17]. Therefore, one can already see that each physical port, as well as each waveguide in the device can be split into two ports, one for each direction. This is a common approximation done in many simulation tools that assume that each signal is an *analytical signal* at a fixed wavelength in a single mode [6].

This therefore gives the first approximation: each physical port is split into two ports, one for each direction. This means that, from the perspective of a user, they can easily split the light incoming from a port and process it in the desired way. And they can easily output light that has been processed into a port with little to no interactions with the rest of the circuit. This is the first step in the feedforward approximation.

The second step in this approximation is to show that, given a sufficient level of abstraction, any circuit can be represented as an element that has zero or more input ports and zero or more output ports. But, as previously mentioned, some circuits, such as ring resonators, have an IIR (*Infinite Impulse Response*) that requires a recirculating mesh to be built. This is where the abstraction comes into play. One can view a ring resonator as a black box that has input and output ports and that has a certain scattering matrix that links each pair of input and output port as can be seen in Figure 3.

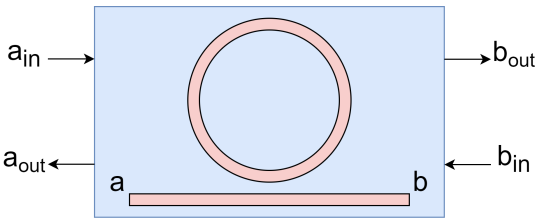


FIGURE 3 A black box representation of a ring resonator. The input and output ports are labeled as $a_{\text{in}}, b_{\text{in}}$ and $a_{\text{out}}, b_{\text{out}}$ respectively.",

It has been shown that, given a sufficient level of abstraction, any bi-directional photonic circuit can be represented by an equivalent, higher-level, feedforward circuit. This result is crucial for the formulation of the requirements for the programming interface of such a photonic processor. And is the basis on which the rest of this document is built.

2.3 Difficulties

2.3.a Wavelength as a continuum

2.3.b Amplitude as a continuum

2.3.c Temperature dependence

2.3.d Manufacturing tolerances

2.3.e Non-linearities

2.4 Initial design requirements

2.4.a Interfacing

2.4.b Programming

2.4.c Reconfigurability

2.4.d Tunability

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 an analysis of the existing software solutions and their limitations will be done. Finally, an analysis of relevant programming paradigms will be done. The secondary objective of this chapter is to make the reader familiar with concepts and terminology that will be used in the rest of the thesis. This chapter will also introduce the reader to different programming paradigms that are relevant for the research at hand, as well as programming language concept 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 that are relevant for 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* and *how*, or what the program should do and how it should do it.

Adapted from [5]

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 as a tool 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 that is closer to the actual hardware, and therefore, they are not used to describe the *what* but the *how*. These languages are not the focus of this thesis, but they are important to understand the context of the research at hand, and they will be further examined in Section 3.4.l where they 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 assigns 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 [18]

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 language the type of values is computed at runtime.

Adapted from [18]

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 – but also to optimize 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 to the user. This practice makes development easier at the cost of increase overhead during execution and the loss of some optimizations [19]. 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 [20, 21], which will be discussed further in Section 5.

3.3 Explicitness in programming languages

In language design, one of the most important aspect to consider is the explicitness of the language, that is, how much details must the user 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 important 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 languages 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 find 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 in a consistent way;
- 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 can be executed by the device.

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 mostly used for embedded systems and operating systems;
- *Rust*: a modern systems language mostly used for embedded systems and high performance applications;
- *Python*: a high-level language that is mostly used 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 [22];

Each of these ecosystems comes with a certain 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, in general, tools maintained by the language maintainers directly 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 the development of the language that will be presented in Section 5, which will be 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 means. This document can be used to ensure that all implementations of the language are compatible with one another.

Adapted from [23]

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 development of the language.

Adapted from [23]

A programming specification is useful for languages that are expected to have more than one implementation, as it outlines what a program in that languages is expected to do. Indeed, code that is written following this specification should therefore be able to be executed by any implementation of the language 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 [24].

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 important 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 important to check that the specification is not too restrictive. Indeed, as has been previously shown in Section 2.1, the programming of photonic processors is different from the programming 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 from 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 however desirable to create a specification when creating a new language, as it can be used 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 evolve may lower the speed at which work is done. For example, *Rust* is widely used despite lacking a formal specification [25].

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.

Adapter from [26]

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 which perform this translation on the fly, such is the case for *Python*. However, HDLs tend to produce programming artefacts for the device, 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. Language like *C++* are notorious for having frustrating errors [27], while languages like *Rust* are praised for the quality of their errors. This is an important aspect to consider when designing a language, as it can make or break the user experience. Following guidelines such as the ones in [27] 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 the syntax and semantics. It produces errors that should be helpful for the user [27]. Additionally, in statically typed languages, it performs type checking to ensure that types are correct and operations are valid. In general, it is the frontend that produces a simplified, more descriptive, version of the code to be used in further stages [21]. The middle-end performs multiple functions, but generally, it 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 [28]. 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 [29]

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 [26]. It is generally provided by the manufacturer of the device.

Adapted from [29]

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 [26]

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 realtime, 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 [30, 31, 32], 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* [33] and *GNU style* [34].

Adapted from [35]

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 [36]. 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 in conjunction 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 [35]

As with formatting, most languages have linters made available either through officially maintained tools or with 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 directly, 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 their ability check for proper usage of the API itself and 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 implementation of linters, they generally rely on a similar architecture than formatters, using existing components of the compiler 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. As en example *Clippy*, *Rust*'s linter, has 627 rules [37].

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` [37]. This rules suggests that functions that are too complex as defined in the literature [38] 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 that are aimed at software development such as syntax highlighting, code completion, and code navigation.

Adapted from [39]

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* [40]. Additionally, most editors have support for more than one language, either officially or through community maintained plugins – additional software that extends the functionality of the editor.

When creating a new language, effort should therefore not go towards creating a new editor as much as supporting existing editors. This is usually done by creating plugins for common editors, however this approach leads to repetition has editors use different language 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 [41]. 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 [41].

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 programs 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 [42, 43]

Testing can generally be seen as a way of checking that a program works as intended. Checking 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*) [44]. Additionally, external software can provide metrics such as *code coverage* that inform the user of how much of their code is being tested [45].

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 module, *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 tests called *constrained random* which produces random, but correct, input to a program and checks that, in no conditions, does the program crash. This is generally utilized to find edge cases that are not properly handled as well as testing the robustness of the program – especially areas concerning security and memory management.

Most modern programming language such as *Rust* provide a testing framework as part of the language ecosystem. However, these testing framework 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* [46].

Therefore, when developing an API, it is important to consider how the API itself will be tested, but also 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 for testing of their code.

Definition: **Simulation** is the process of running a program that simulates the behavior 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, power consumption limits, etc.

Adapted from [42, 43]

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 are used to 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 important tools for the users of HDLs as it allows them to test their code without needed access to the physical devices. Furthermore, it allows programmers working on ASICs (*Application Specific Integrated Circuit*) to simulate their code, and therefore their design before manufacturing of a prototype.

There exists a plethora of simulation tools, as previously mentioned, *Vivado Simulator* allows users to test their FPGA code, other tools such as *QEMU* allow users to test embedded platforms. Additionally, a lot of analog simulations 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 [8].

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 [6, 7]. Therefore, when creating an API or a language for photonic processor development, it is desirable to consider how simulation 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 important part of the HDL workflow and ecosystem. It is critical that any photonic programming solution provides a way to perform verification. This would be done by providing both a simulator and a tester and then providing a way of interface both together to perform verification.

3.4.i Package manager

Definition: 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 they can also be tools such as testing frameworks, etc.

Adapted from [47]

Package management is an integral part of modern language ecosystems. It allows users to easily install dependencies from the community as well as share dependencies 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 for protection of 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 manager 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 for 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 themselves build on top of existing community solutions, providing for a more cohesive ecosystem. It is also important to point out that nested dependencies can cause conflicts, and therefore, 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 [48].

3.4.j Documentation generator

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

Adapted from [49]

The most common document generators are *Doxygen* used by the *C* and *C++* community 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 for maintainers to document the language and it standard library. Finally, as technical documentation is the primary source of information for developers [40], it is essential to take this need from users into account.

3.4.k Build system

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

Adapted from [26]

Build systems play an essential role in building complex software. As modern software is generally composed of many files that are compiled together, along with having dependencies, configuration and many other resources, it is difficult 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 standardized file structure, the build system is able to infer how to build the project. This approach is easier to use and leads to 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 of 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 uses the `custom_hello` feature flag to conditionally enable the additional feature. 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
/// Prints `Hello, world!` in the console.
fn print_hello_world() {
    println!("Hello, world!");
}
```

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

```rust
/// Prints `Hello, {name}!` in the console.
#[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 important 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 greatly improve the user experience. And especially the experience of newcomers as it might avoid them having to do obscure tasks such as writing their own *CMake* files.

3.4.1 Summary

As has been shown, in order to build a complete, user friendly ecosystem, a lot of components are necessary or desirable. Official support for these components might be preferred as they lead to lower fracturing of their respective ecosystems. In Table 1, an overview of components that are required, desirable or not needed, along with a short description and their applicability for different scenarios are mentioned. Some components are more important than others and are required to build the ecosystem. Most notably the compiler, hardware-programmer, and testing and simulation tools. Without these components, the ecosystem is not usable for hardware development. However, while the other components are not strictly needed, a lot 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 (*Photonic Hardware Description Language*), will be implemented. Indeed, the same components that make API designing easy, also make language implementation easier. As will be discussed in Section 3.11, PHÔS will be implemented in *Rust*. The language meets all of the requirement by having first party support for all of the required and desired components for an API design. Additionally, 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 a lot of 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 strong candidate in this regard, as it has first party support for all of the required components with the exception of hardware-programming and debugging tools. But as also noted in Table 2, most other languages do not come with first party support fot these tools either. However, as will be discussed in Section 3.5, it is difficult to learn, has not seen use in hardware synthesis and is

therefore not a good fit for regular users. But its strong 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 is a problem that can be solved by providing a common baseline for all task 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 common interface for their hardware, will allow for a common ecosystem to be built on top of it. This is the approach that will hopefully be taken by PHÔS, which will be discussed in Section 5.11.

Components	Traditional languages			Hardware description & simulation languages	
	C	Rust	Python	Verilog-AMS	VHDL
Language specification	✓ [50]	✗ [25]	✗ [51]	✓ [52]	✓ [53]
Compiler	~ 1 (*Clang* & *GCC*)	✓ (*rustc*)	~ (*PyPy* & *Numba*)	✗ (simulated)	~ (synthesized)
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 (×). For each component, is also lists the name of the tool that is most commonly used for that purpose.

Notes:

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 (*Verilog for Analog and Mixed Signal*)* is a language used for simulation, not hardware description.
4. *C* and *Rust* generally share debuggers due to being native languages.
5. There does 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* exist and are widely used.

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

Depending on the choice of implementation, the components of the ecosystem will change. However, whether the language already exists or whether it is created for the purpose of programming 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. And 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 different syntaxes employed by various, common, programming languages. The goal of this section is to build an intuition on what these syntaxes look

like, what they mean and how they can be applied in Section 6.1. As that section will compare simple examples and how one might implement simple photonic circuits in each of these languages. 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 important to understand these differences and why they makes these existing solutions unsuitable.

The first analysis, which looks at traditional programming languages, will be done by looking 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 strength 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 [54], but is also being used for novel use cases such as quantum programming [55].
- *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 which makes it unsuitable for non-programmers [56].
- *Python* is a common language that is used by a wide proportion of researchers and engineers [40, 57], which makes it a great candidate as the starting point of any language development. It is also used for some HDL development [58] and has been 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 that is generally not suitable 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 analyzed:

- *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 [59].
- *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* [22].
- *SPICE*: a language that is used for the simulation of analog electronics. *SPICE* as seen use in the development of photonic simulation [8].

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 really fit the needs of photonic development, most notably with regards 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 there are two distinct families of languages in the aforementioned list: there are digital HDLs and analog simulation-centric languages. Therefore this comparison will be done in two parts, one for each family of languages.

3.5.a Traditional programming languages

To compare traditional programming language, a simple, yet classical example will be used: *FizzBuzz*, which is a simple program that prints the number from 1 to 100, printing *Fizz* when the number is divisible by 3, *Buzz* when the number is divisible by 5 and *FizzBuzz* when the number is divisible by both 3 and 5. 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 are not the focus of this comparison, choosing the most optimized 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* [60]. *C* has had a large influence on languages such as *Python* [61] and *Rust* [25] – through *C++* and *Cyclone* – but also on HDLs such as *Verilog* (and therefore *Verilog-AMS*). As such, this section will start with an outlook of the syntax of *C* and discuss some of its shortcomings with regards to 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 [62].

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 statement;
- variables are declared with a type and a name, and optionally initialized 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 statement 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 entry point of the program.

```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 entry point of the program.

Rust

```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;
- statements are terminated by a newline;
- 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 itself is the entry point of the program.

Python

```python
for n in range(1,101):
    response = ''
    if not (n % 3):
```

```
4         response += 'Fizz'
5     if not (n % 5):
6         response += 'Buzz'
7     print(response or n)
```

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

These simple example show us 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 which makes 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, that means that users still need to annotate some types, but overall most variables do not need type annotations. This makes *Rust* easier to use than *C*, but also more difficult 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 pointer;
- *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 that several types of values can be used as a boolean value.

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 synthesized into hardware that can be built. This hardware is generally in 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 synthesizer that produces a netlist and a list of operations that are needed to implement the circuit. As was 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 very low.

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 (*Integrated Circuit*) are very expensive to design and produce, as such, the larger the design, the larger the dies needed, which increases cost; and FPGAs are very expensive 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, performance of a design is characterized by three criteria, one criterion is the speed of the algorithm being implemented, another one is 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 an input the high-level language and translates the code into a lower-level form.

Adapter from [54, 63]

There has been a push in recent years towards higher level of 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 simpler and more productive [64]. Allowing the user to focus on the feature they are trying to build and not the low level implementation of those designs. As was discussed in Section 3.1, this can be seen as a move towards declarative programming, or at least, a less imperative programming model. Coupled with the rise of hardware accelerator in the data center and cloud markets, which are generally either GPUs (*Graphics Processing Unit – also commonly used for highly parallel computing and machine learning*) or FPGAs, there has been an increased need for software developer 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 can be used by software developers to program FPGA based accelerators.

Another advantage of HLS is the ability to test the hardware using traditional testing frameworks, as discussed in Section 3.4.1, 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, the ability to test the hardware using traditional testing frameworks is a major 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, as well as reduce the risk of errors. In addition to being able to use existing testing framework, 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 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 greater level of optimization, such as using the breadth of optimization available in the LLVM (*Low Level Virtual Machine*) compiler. This has allowed HLS to reach a

level of performance that is acceptable for large swath of applications, especially when designed by non-specialists [65]. Other techniques such as machine learning based optimization techniques have been used to increase performance even further [66].

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 alternative are positioned as an alternative 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.I, HDLs, tend to be lackluster – or highly vendor-locked – with regards to development tools. And just as in the case for HLS, this can be an argument in favour of using alternatives, such as these HDL 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, being 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 parameter of the design. This will allow the demonstration of procedural generation of hardware, as well as the use of modules and submodules to structure code.

Most HDL languages come with pre-built implementations of adders. Usually, the best implementation of the adder is chosen by the compiler or synthesis tool based on constraints defined by the user. These constraints can relate to the area, power consumption or timing requirements. In this case, the adders implemented are simple combinatory ripple-carry adders, which are generally not the best implementation.

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). In VHDL, the ports, as well as other properties are defined in the `entity` and the implementation of the logic is done in an `architecture` block. This leads to functionality being spread over multiple locations, generally reducing readability.

```vhdl
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Adder is
5  Generic(n:integer:=8);
6      Port ( Cin : in  STD_LOGIC;
7             A : in  STD_LOGIC_VECTOR (n-1 downto 0);
```

```vhdl
            B : in  STD_LOGIC_VECTOR (n-1 downto 0);
            Result : out  STD_LOGIC_VECTOR (n downto 0));
end Adder;

architecture Behavioral of Adder is
    -- Full Adder component
     COMPONENT FullAdder
     PORT(
         A : IN std_logic;
         B : IN std_logic;
         Cin : IN std_logic;
         S : OUT std_logic;
         Cout : OUT std_logic
         );
     END COMPONENT;
     signal carry : std_logic_vector(n downto 0);
begin
-- external carry input
carry(0) <= Cin;
-- Array of full adders
FA_array: For i in 0 to n-1 generate
    Inst_FullAdder: FullAdder PORT MAP(
         A => A(i),
         B => B(i),
         Cin => carry(i),
         S => Result(i),
         Cout => carry(i+1) -- connect the output carry to the input carry of the next FA
    );
end generate FA_array;
-- output carry
Result(n) <= carry(n);
end Behavioral;
```

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

```python
from myhdl import *

@block
def bit_adder(A, B, Cin, S, Cout):
    """ 1-bit adder with carry in and carry out """
```

```
    @always_comb
    def logic():
        S.next = A ^ B ^ Cin
        Cout.next = (A & B) | (A & Cin) | (B & Cin)

    return logic

@block
def nbit_adder(A, B, Cin, S, Cout):
    """ n-bit adder with carry in and carry out """
    @always_comb
    def logic():
        Carries = [Signal(bool(0)) for i in range(len(A))]
        Carries[0].next = Cin
        for i in range(len(A)):
            if i == len(A) - 1:
                bit_adder(A[i], B[i], Carries[i], S[i], Cout)
            else:
                bit_adder(A[i], B[i], Carries[i], S[i], Carries[i + 1])
    return logic
```

Listing 7 | Example of a n-bit adder in *MyHDL*, based on [2].

```c
#include "systemc.h"

#ifndef N
    #define N 8
#endif

SC_MODULE (BIT_ADDER) {
    sc_in<sc_logic> a, b, cin;
    sc_out<sc_logic> sum,cout;

    SC_CTOR (BIT_ADDER)
{
        SC_METHOD (process);
        sensitive << a << b << cin;
    }

    void process() {
        sc_logic aANDb, aXORb, cinANDaXORb;
```

```
19
20          aANDb = a.read() & b.read();
21          aXORb = a.read() ^ b.read();
22          cinANDaXORb = cin.read() & aXORb;
23
24          sum = aXORb ^ cin.read();
25          cout = aANDb | cinANDaXORb;
26      }
27  };
28
29  SC_MODULE (NBIT_ADDER) {
30      sc_in<sc_lv<N> > a, b;
31      sc_in<sc_logic> cin;
32      sc_out<sc_lv<N> > sum;
33      sc_out<sc_logic> cout;
34
35      sc_signal<sc_logic> ss[N], cc[N];
36
37      BIT_ADDER* add[N];
38
39      SC_CTOR (NBIT_ADDER)
40  {
41          SC_METHOD (process);
42      }
43
44      void process() {
45          int i = 0;
46          for(i = 0; i < N; i++) {
47              char name[25];
48              sprintf(name, "add_%d", i);
49              add[i] = new BIT_ADDER(name);
50
51              if (i == 0) {
52                  add[i] << a[i] << b[i] << cin << sum[i] << cc[i + 1];
53              } else if (i == N - 1) {
54                  add[i] << a[i] << b[i] << cc[i] << sum[i] << cout;
55              } else {
56                  add[i] << a[i] << b[i] << cc[i] << sum[i] << cc[i + 1];
57              }
58          }
```

```
59         }
60     };
61
```

Listing 8 | Example of a n-bit adder in *SystemC*, based on [2].

3.5.d Analog simulation languages

3.6 Analysis of programming paradigms

3.6.a Imperative programming

3.6.b Object-oriented programming

3.6.c Functional programming

3.6.d Logic programming

3.6.e Dataflow programming

3.7 Existing framework

3.8 Long term compatibility and cross-vendor support

3.9 A primer on calculability and complexity

3.10 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 optimizing the system as a whole, rather than optimizing the hardware and software components separately.

Adapted from [67].

3.11 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 1 to 5, with 1 being the lowest and 5 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 the creation of a new language, the best languages to implement it in 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 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.

Language	Applications			
	Ecosystem	API design	Root language	New language
C				
	C is a fully featured low-level language, it is performant and has a simple syntax. However, it lacks some of the 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 a lot of the semantics of hardware description which makes it unsuitable as a root language. However, its large array of language-implementation libraries makes it a good candidate for the implementation of a new language.			
Rust				
	Rust is a modern low-level language, it is very performant, has excellent first-party tooling, quickly growing in popularity, and is memory safe. However, it has difficult syntax and semantics that is unwelcoming for non-developers, which makes it unsuitable for either API design or as a root language. However, its large array of language-implementation libraries coupled with its memory and thread safety makes it an excellent candidate for the implementation of a new language.			
Python				
	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 the implementation of a new language.			
Verilog-AMS				
	Verilog-AMS is a mixed signal simulation software, its ecosystem is lackluster with many proprietary tools which incurs expensive licenses. It not a generic language and is therefore not design for an API to be implemented in the language, nor is it suitable for the implementation of 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.			
VHDL				
	VHDL is a mature language with a large ecosystem, but suffers from the same issues than 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 difficult to learn and make the language difficult to read, which makes it unsuitable as a root language.			

Table 3 | Comparison of the different languages based on the criteria discussed in Section 3.11.

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 the way in which the user will write down their design, and how the program translate that design into an actionable, programmable, design. This section will also outline some of the features that are needed for easier translation of intent. This will be done by discussing important features such as *tunability*, *reconfigurability*, and *programmability*. These features revolve around the ability for the user to tune the operation of their programmed photonic processor as it is running. For this purpose, this section will introduce some novel concepts, such as *constraints* and its *solver* and *reconfigurability through branching*. These two important concepts will be discussed in details and synergize to create an easy to use, yet powerful, programming ecosystem for photonic processors.

Additionally to the aforementioned points, several key features were discussed in Section 2.4, the features relate to realtime control, which works in pair with *reconfigurability* and *tunability*, 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 visualization of the design, which has lead to the design of the *marshalling layers* which will be discussed in Section 5.7.

Definition: **Synthesis** is the process of transforming the description of a desired circuit into a physical circuit.

Adapted from insert reference

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 different components that all need to cooperate. Additionally, some of the tasks that synthesis must do, such as place-and-route, are incredibly computationally intensive and are often regarded as being 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 insert reference

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, one can see these requirement as the most declarative form of the user's intent. Therefore, one can see this step as the user's intent.

However, there are elements that are generally going to be common to all of 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
	Tunability	Tunability on the other hand does not change the topology in itself, but rather changes the behaviour of the mesh through the tuning of 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
	Programmability	In order to be able to make use of tunability and reconfigurability, the user must be able to programmatically communicate with their programmed device. This is done through the use of 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
	Visualization	Visualization allows the user to see the result of a simulation, what a finalized design looks like, 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 visualization is desireable.	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 [68].

Programmability refers to the ability for the user to programmatically interact with their circuit while it is running. 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 which is provided by the device manufacturer and the user HAL which is 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 that is consistent with their design, and therefore 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 template or custom code. This allows the device support package to provide platform-specific features to the user HAL or to optimize 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 must be supported by the processor. These operation 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 full 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 circuit 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 synthesized based on its arguments in order to produce a filter of the desired frequency response. There are many different types of filters, the most common ones, that can easily be implemented on a mesh – are MZIs (*Mach-Zehnder Interferometer*), ring resonators, and lattice filters. Additionally, compound filters that combine multiple types of filters, or more than one filter can be created, such filter can have improved response or behave like band pass filters. Therefore, it is the task of the compiler to chose the base filter based on the specification and performance criteria that the user has set. For example, the user might prioritize optimizing for mesh usage rather than finesse, or might optimize 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 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 even 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 synthesize gain and loss sections based on the user's specification. However, if the device does not support gain or loss, the compiler should produce an appropriate error to the user.

Modulator and detectors Two of the key applications of photonic processor is in telecommunication and processing of 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 chose 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 optimize the placement of these splitters to minimize the mesh usage, or to minimize non-linear effects in high power signals, or to 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 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 optimize 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 other 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 signal conditionally, or to implement simple reconfigurability without the added cost of having more than one mesh.

Sources A lot of applications will need 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 synthesize 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, 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 make use of this intrinsic operation to implement complex modulation schemes, as well as to implement a beam forming 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, either by using a delay line, or by using multiple wires of different lengths in order of matching the total optical length. It works nicely with the ability to express differential constraint which will be discussed in Section 4.4.

Coupler Couplers are part of each photonic gate present on the processor, they have the ability to couple two signals based on a coupling coefficient. This is a key intrinsic, at 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 relation with a coupler on the processor itself. However, the compiler should be able to optimize the coupling coefficient based on the frequency content of the signal and the calibration curves of the device.

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 optical signals.	• ✓ Input signal • ✓ Output signals • ✓ Splitting ratios
Combiner	A combiner that combines multiple optical signals into a single optical signal. While maximizing the total output power.	• ✓ Input signals • ✓ Output signal
Interferometers	A combiner that combines multiple optical signals into a single optical signal. Does not perform any power optimization.	• ✓ 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

Table 5 Intrinsic operations in photonic processors, with their name, description and arguments. For each arguments, 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 constraints 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 validity of the code, and to 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, the way in which they are applied to include more complex constraints, to simulate circuits, and inferring them, does *appear* to be novel.

Constraints for validation The primary use of constraints is for the validation of the code. This is done by the *constraint solver* 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 are going to be constraints on power, gain, wavelength, delay, and phase. The reasoning behind 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 phase of the signal. These constraints can be used to verify the validity of the code, and to inform the compiler how to optimize and generate the design. Indeed, in some cases, 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 or closer to the components using this light. 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 make a decision on where to place the splitter. This is just one example of how constraints inform the compilation system and can be used to optimize 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-synthesizable, whereas the other constraints are synthesizable.

These non-synthesizable constraints, can be couple with synthesizable 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 will be discussed in Section 5.2.q, it makes them an inherent part of the user's design. Meaning that accurate, yet fast, simulations are available to the user at all times.

Constraints for optimization 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 optimize the design. The compiler can use

constraints to remove unnecessary components, or to optimize the placement of said components. A simple example, a signal going through a filter might have a constraint on the wavelength, and the filter might 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 after the filter there would be so signal left, it can remove the filter and all dependent components, simplifying the design. This is just one example of how constraints can be used to optimize the design.

Constraints as realtime feedback As discussed in Section 2.1.a.c, there are power detectors on the chip that can be used for monitoring purposes. These detectors are expected to be implicitly and automatically used most of the time through the use of 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. Where these constraints can be used to check, while the device is running whether constraints on power and gain are respected, notifying the user if they are not. This gives the user the ability to add detection of erroneous events, such as the loss of an input or failing 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 the error. 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 categorize constraints into three distinct categories: synthesizable constraints, which are used for realtime 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 4. It shows that all constraints are simulation constraints, some of which are meta constraints and some of those are also synthesizable constraints. In Table 6, the different types of constraints are listed along with their category and a short explanation.

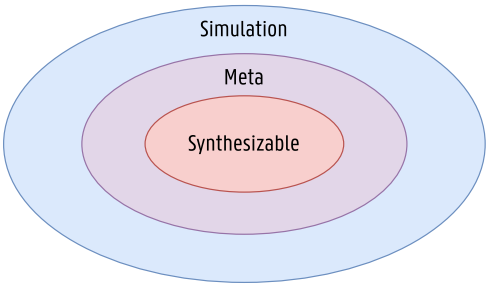


Figure 4 Hierarchy of constraints, showing that all constraints are simulation constraints, within that are meta constraints within which are synthesizable constraints.

	Constraint	Description
Synthesizable	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 optimization 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 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 onto 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 on constraints further, it is possible to add constraints on values other than signals. 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 validation of value 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 discord reconfigurable states that cannot be reached based on the constraints. This is an optimization 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. This is done by the compiler 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 summarized – 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 5.10.

Constraint solver The solver is the tool that the compiler uses to summarize and check constraints. It is used by the compiler for these two operations, where it will summarize the constraints on each signal such that it can be easily 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* [69]. However, this is a very computationally expensive process and must therefore only be performed when necessary. This is why 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, which is solver.

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

Constraints can be used to validate signals, eliminate branches, and simulate the design. They are implemented using the constraint-solver which can either combine constraints or using the *Z3* prover to verify constraints [69]. 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 value at runtime to impact the behavior 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 the behaviour of their circuit, and to implement their own custom feedback loops. Tunable values can impact several parts of the design at once, for example, a single value may determine the center frequency of operation of a bank of filters, all of them being changed when one tunable value has been changed. 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. The purpose of this design is to make hardware-software codesign easier and more productive. Where instead of having the complex relationships between parameters expressed within the "software" part of hardware-software codesign, they can instead be directly expressed on what they impact: the "hardware" part. This makes the design process more intuitive, and also removes the potential for discrepancies between the hardware and the software.

Additionally, the user should be able to name their tunable values and to be able to access them by name within their own code. This further improves the usability of the system, as it removes the need to maintain complex, error-prone table of registers and their corresponding values. Instead, the user can address their tunable value in a natural way through its name, and 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 improves the development experience of the device provider as well, as they can now integrate within their platform support package, the code required to do data conversion, 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, that 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 structure of the device at runtime to change the topology of the device 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 a number of way, 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 its value.

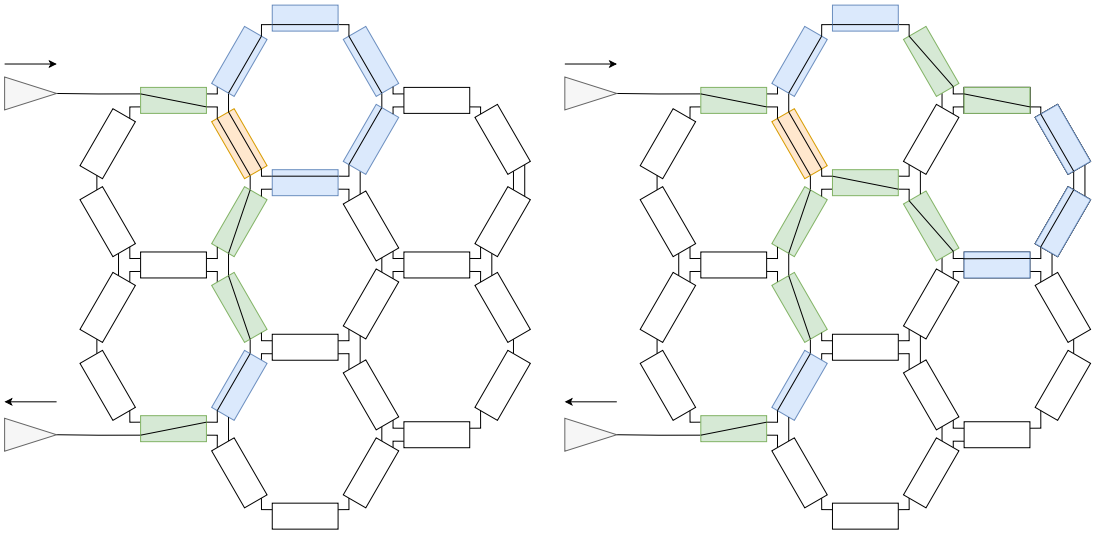
However, this brings a set of difficult problems to solve, the first of which is the ability to determine whether a state is even reachable. However, this can be done using constraints, through the constraint solver for tunable constraints, one can verify which states are reachable or not, and discard those that are unreachable. This is a powerful optimization, 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, consider the following example, the user instantiates a mesh containing 64 input signals and 64 output signals, based on branching, each input signal can go into one of two filters. This therefore means that there are 2^{64} possible states. It can easily be understood that this is an intractable problem, as it would be almost impossible to synthesize the project. However, if the system is able to determine that for each signal, only two states are reachable, this comes down to 128 states, which is much more tractable. Therefore, one can see that there is an interest in finding ways of reducing the amount of states that need to be synthesized. 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 from one another, and therefore can be mostly synthesized in isolation.

Neither of those two tasks are trivial, 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 amount of states manually. Additionally, this idea of figuring out which parts of the mesh are independent from one another, is similar to the halting problem, which is undecidable. However, by limiting the maximum number of iterations, the recursion depth, or both, one can make it decidable. But despite being decidable, it still incurs a heavy computational cost.

In Figure 5, 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 5 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 therefore a higher FSR (*Free Spectral Range*) than the first one (a). Squares of different color represent photonic gates in different states: blue represents through gates, green represent cross gates, and yellow represents partial gates. The gray 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 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 subsets of the mesh, and by letting the user specify some of the state reduction manually.

4.6 Simulation

As previously discussed, the user must also be able to simulate their circuit. The traditional approach of physical simulation is slow, and therefore, it may be desirable to find solutions to make simulations faster. 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 (*Photonics Research Group*). One of the main advantages of this solutions, as opposed to the one that will be 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 simulation that is fast, and representative of their circuit without all of the limitations of the physical hardware. In essence, this is similar to simulations for FPGA 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 summarize 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 due to the relative simplicity of constraints, this can be done very quickly, with relatively simple code. This simplicity both improves the performance of the simulation, as will be discussed in Section 5.10, 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 take one or more signals modulated onto carrier optical signals and simulated 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 order of the modulate signal's period, rather than timesteps on the order of the light's period.

However, this does introduce a limitation, due to this dichotomy, the user needs to simulate both effect separately and analyze the results themselves. While this makes the process of simulation 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 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 much as the existing tools out there 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 that surrounds these solutions and the community that uses them. Furthermore, it also makes the development of the simulation ecosystem simpler, as it no longer required writing the entire simulation ecosystem from scratch. Instead reusing existing tools and making use of the best-in-class tools for each tasks.

4.7 Platform independence

As the development of photonic processor continues, it must be expected that new devices will 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 that contains base building blocks and some 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, by providing common building blocks and abstraction, it makes the development of circuits targetting photonic processors easier. This is similar to the standard library that exists for regular software development, where the language provides a set of functionalities out-of-the-box that can be used by the user.

Platform support packages Each platform must come with a platform support packages that implements several tools: a hardware programmer for programming the circuit onto the device; 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, 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 5.9, 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 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, in addition to the additional information provided by the constraint packages.

4.8 Visualization

There are several types of simulations that may be useful for the user: the user might want to visualize 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 visualize whether the place-and-route performed adequately. This visualization is already presented in the existing library and exists in EDA (*Electronic Design Automation*) tools for photonics. Therefore, such visualization facilities must be offered to the user. Especially due to the fairly early stage of research, the ability to communicate results visually is critical to the user's understanding of the results. Another kind of visualization the user will want is the results of the simulation results. Therefore, the ecosystem must provide easy visualization of results.

Applying DRY As is the case of the simulation ecosystem, one can reuse existing tools and libraries for visualization that are already on the market. This is an application of 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 visualization tools that already exist, and to use the tools they are most familiar with, or that gives the best results for their application. Examples of such visualizations can be seen in Figure 5 that shows the mesh and the state of each gate, and a simulation result in Figure 6 which shows the results of a time-domain simulation using the aforementioned constraint-solver.

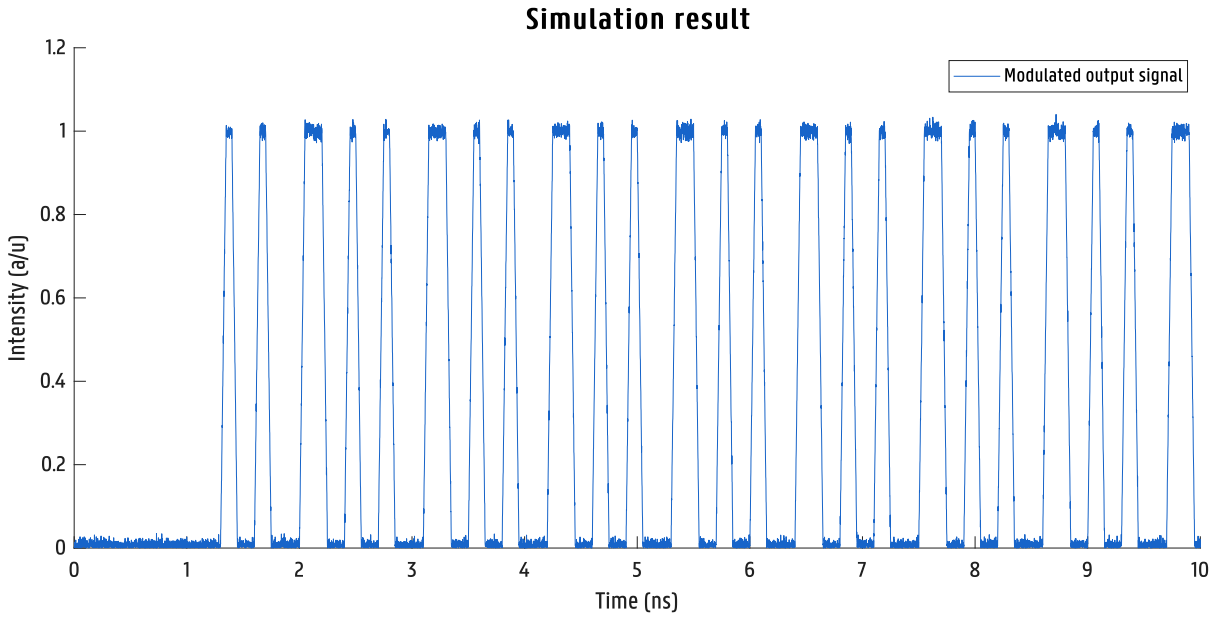


Figure 6 Example visualization of a time-domain simulation result, showing a $10\ \mathrm{Gb/s}$ modulated PRBS 15 sequence on top of a $1550\ \mathrm{nm}$ carrier. The simulation was performed using the constraint-solver. Shown is a $10\ \mathrm{ns}$ window of the simulation. The simulation was ran for a total of $1\ \mu\mathrm{s}$ 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 very sensitive 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 through the use of calibration curves or calibration LUTs. And by using feedback loops to ensure that a component behaves as expected. For example, a power combiner might maximize power output using a feedback loop on a phase shifter to create constructive interference.

Feedback loops Feedback loops are an essential part of being able to overcome variations, especially those caused by temperature variations. A feedback loop can be used to read a power monitor present 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 are 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 design circuits for programmable devices, whether they be traditional processors, FPGAs or photonic processor is resource management. Built into the hardware a limited number of elements, and the user must be able to use these elements as efficiently as possible. This may be especially true for photonic processors where, currently, the number of gates are relatively small. Below is Table 7 that 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 an 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 of the device. Currently, due to the devices being made in silicon, there are none on prototypes, however, in the future they may be added using epitaxial growth or micro-transfer printing.
Gain section	Gain sections are used to amplify the light, they are 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 are the ports at the edge of the device that 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 being controlled by a tunable value, or built into the device itself as dedicated hardware. Currently, 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

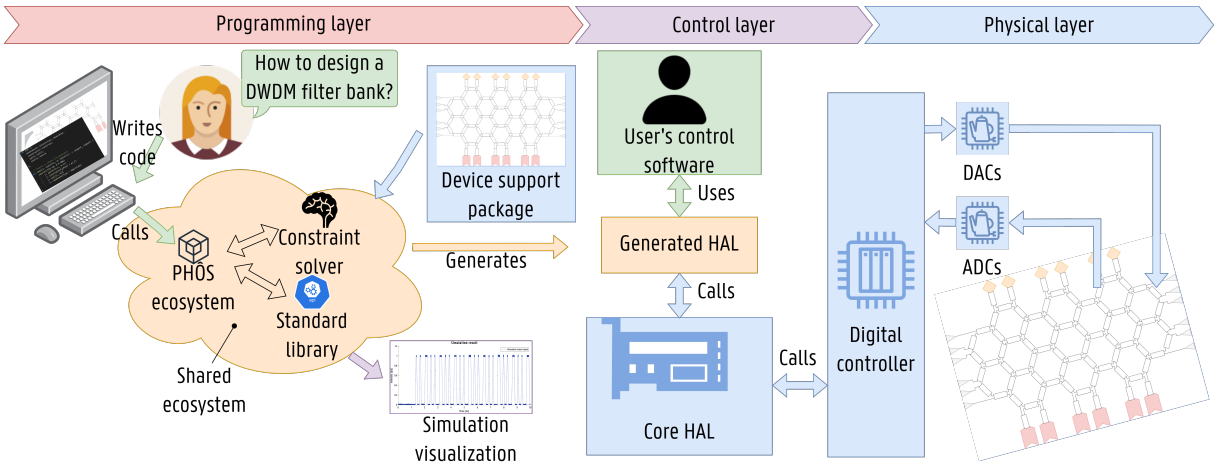


Figure 7 Responsibilities of each actor in the ecosystem, elements in orange are the responsibility of the ecosystem developer, it includes the compiler, constraint solver and standard library, it also contains parts of the HAL generator. Elements in blue are the responsibility of the chip designer, it includes the device itself, the core HAL, and the device support package. 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 has having four actors: the user who is responsible for the design of their circuit and their own control software and they are also responsible for the maintenance of their own 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, 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 that they are maintained by trustworthy actors. A summary of these responsibilities and their interactions with one another can be seen in Figure 7.

5.
The PHÔS programming language

Based on all of the information that has been presented so far regarding translation of intent and requirements (Section 4), programming paradigms (Section 3.6), and with the inadequacies of existing languages (Section 3.11), it is now apparent that it may be interesting to create a new language which would benefit from dedicated semantics, syntax, and integrates 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 that are common within the scientific community such that it is easy to learn for engineers, 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 fulfill these goals.

In the following sections, the initial specification, the syntax, constraint system, and other various elements of the language will be discussed. Then, in Section 6, examples will be shown 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 languages, 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*[1]. 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. **Synthesizable** PHÔS separates regular functions from synthesizable blocks, whereas synthesizable 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 synthesizable blocks. Ideally, synthesizable blocks would be kept as short as possible while functions can be much longer.

Paradigm PHÔS is an imperative language with functional elements.

Constraints

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

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 into 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 synthesized 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 8, 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.

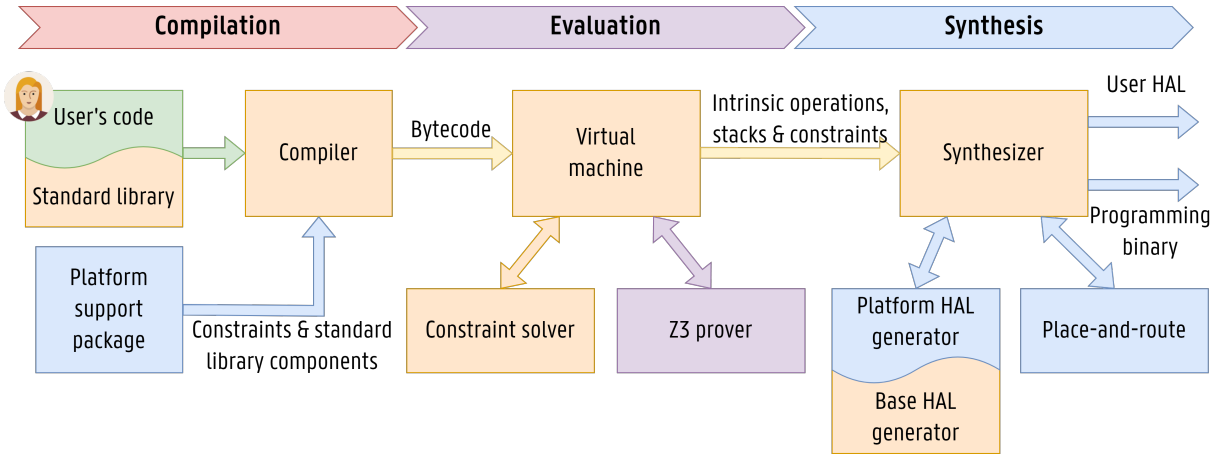


FIGURE 8 Execution model of the PHÔS programming language, showing the three distinct stages: compilation, evaluation and synthesis. The compilation stage takes the user's source code along with the source code of the standard library, the platform support package – that contains device-specific constraints and component implementations – and produces bytecode. The evaluation stage takes the bytecode and produces a tree of intrinsic operations, constraints on those operations, and collected stacks. The evaluation uses the constraints solver and the *Z3* prover to check for constraint satisfiability [69]. Finally, the synthesis stage takes the output of the evaluation stage, along with the HAL generator for the platform and the place-and-route implementation to produce the user HAL and the programming file for the photonic processor.

This figure uses the same color scheme as Figure 7, 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 9.

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 9 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.

Synthesizable execution model Synthesizable blocks follow a different execution model due to reconfigurability: when a block of code cannot be evaluated, it is analyzed 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, synthesizable 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 [20, 21]. 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* [70], which would allow the compiler to check for errors in constraints earlier, this is further discussed in Section 7.

5.2.c 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 9, depending on the compiler's implementation, it may lead to two splitting schemes, both shown in Figure 10, 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 synthesizable blocks. This restriction aims at making the work of the compiler easier and creating a stronger distinction for users.

```phos
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: Modulate::Amplitude)
```

Listing 9 | Optical signal splitting example

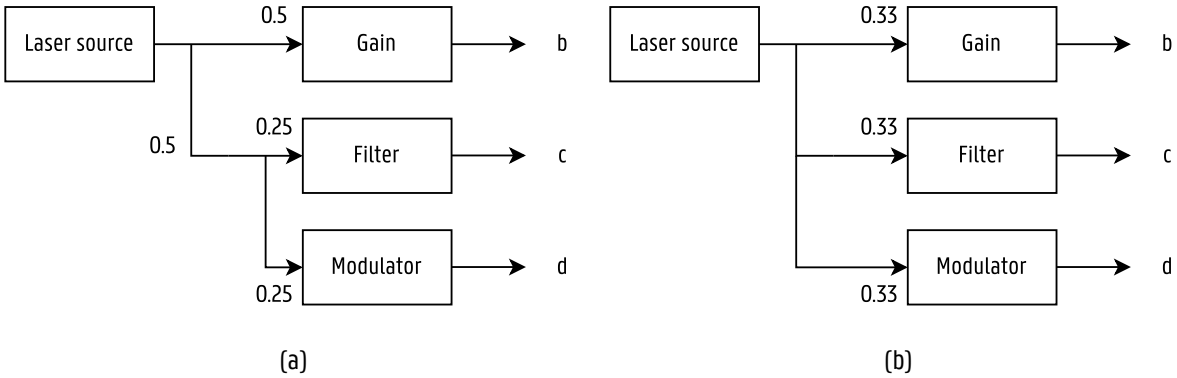


Figure 10 | Automatic optical signal splitting schemes, showing the two possible automatic splitting scheme, using either a cascade architecture (a), or a parallel architecture (b). It leads in different power ratios to all of the downstream elements.

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, as present, there are no plans for analog processing in the electrical domain. 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, 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-one*.

5.2.d 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 synthesizable. To understand how primitive types are synthesizable, see Section 5.2.s. 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* [51].

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 [71].	-	-

`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 [72].	$-1.798 \cdot 10^{308}$	$1.798 \cdot 10^{308}$
`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.e 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**.

Adapter from [73]

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 10, 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.

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

LISTING 10 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.

Adapter from [74]

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 see in Listing 11. This variety in typing allows for precise control of values and their representation. It allows the user to chose the best type for their current situation, such as anonymous tuples for temporary values, or named records for more complex structures.

PHÔS

```
/// 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 11 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 12.

PHÔS

```
type Voltage = int;
```

Listing 12 Example in PHÔS of type aliases, showing the creation of a `Voltage` type alias for `int`.

5.2.f Automatic return values

5.2.g 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 unit 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 7.10. 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 13, one can see the syntax for the usage of some of the units, decibels and angles.

```phos
/// 1 milliwatt
a = 1 mW;
b = 0 dBm;

// 1 degree
c = 1 deg;
d = 0.01745 rad;
e = 1°;

// 1 kilohertz
f = 1 kHz
g = 1e3 Hz
```

PHÔS

Listing 13 | 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 are converted by the compiler into the base, unprefixed, unit before execution begins.

5.2.h 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: 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, typst supports unsized tuples. The general form of tuples as container can be seen in Listing 14.

```phos
/// A tuple container
a = (a, b, c)

/// A tuple as a type
type A = (B, C, D)
```

PHÔS

Listing 14 | 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.i, beyond simple fixed sized tuples and into the realm of dynamically sized lists. The general form of unsized tuples can be seen in Listing 15.

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

/// A more complex unsized tuple
type A = (B, C, D...)
```

Listing 15 | 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 16.

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

Listing 16 | Example in PHÔS of iterable tuples.

5.2.i Patterns

Patters 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 its 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 pattern can be see in Listing 17 .

```PHÔS
// 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 17 | 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 [21]. 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 optimize the user's design further.

5.2.j 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 synthesizable contexts. This feature was previously discussed in Section 4.5. The general form of branching can be seen in Listing 18. Reconfigurability through branching is designed to be very simple to use, the user can simply branch in their code, if the compiler detects that signals are being used across the branches, 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.

```phos
// 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")
}
```

PHÔS

LISTING 18 | Example in PHÔS of branching.

5.2.k 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 uninitialized, and must be assigned a value when they are declared, with the notable exception of signals in unconstrained contexts, see Section 5.2.r, which have a 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 19.

```
// 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
```
PHÔS

Listing 19 | 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 to 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 as `static`, indicating that it cannot be tunable. An example of both use cases is provided in Listing 20.

```
// 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
}
```
PHÔS

Listing 20 | Example in PHÔS of tunability.

5.2.1 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 the 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 21.

```phos
// 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
}

```

PHÔS

Listing 21 | 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 [75].

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 22.

```phos
// Function that performs the addition of two numbers using piping
// Folding is a common operation on iterators, and is used to reduce
// the values of an iterator into a single value, with a starting value
// and a closure that takes the accumulator and the current value and
// returns the new accumulator
fn add_with_pipe(a: uint, b: uint) -> uint {
    return (a, b) |> fold(0, |acc, x| acc + x)
}
```

PHÔS

Listing 22 | Example in PHÔS of the piping operator on iterable values.

5.2.m Function and synthesizable blocks

PHÔS separates functions into three categories: functions denoted by the keyword `fn`, and synthesizable 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 be expressed on values, or on signals.

Functions Function represent 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, it 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 synthesizable 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 23.

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

Listing 23 | Example in PHÔS of a function.

Synthesizable blocks Synthesizable 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 synthesizable block that can be tuned or reconfigured at runtime. An example of a synthesizable block can be seen in Listing 24.

```PHÔS
// Synthesizable block that performs the filtering of an input signal using an FIR
syn filter(in: optical) -> optical {
    a |> split((0.5, 0.5))
      |> constrain(d_phase: 30 deg)
      |> interfere()
}
```

Listing 24 | Example in PHÔS of a synthesizable block.

5.2.n Modules and imports

Most programming languages have module systems, which allow the user to organize their code into different files or folders, with nested modules. It generally avoids file 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 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 chose 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 25.

```phos
// 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};
```

PHÔS

Listing 25 | 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 maximized. This also means that there is no need for the concept of visibility as it exists in many other languages, that have special keywords like `pub`, `public`, or `private` to define the visibility of an item.

5.2.0 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 26.

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

PHÔS

Listing 26 | 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 synthesizable blocks, while they can process signals, they cannot capture signals, `SynOnce` which are closure that are synthesizable, but must called once, essentially following the same *drive-once*, *read-once* semantic as other signals. The difference between the three kind can be seen in Listing 27.

```phos
// (a) `Fn` closure
let c = 1;
```

PHÔS

```
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 27 Example in PHÔS of a `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 [76]

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 less 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 28.

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

let add_1 = set add(1);

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

Listing 28 Example in PHÔS of partial function in PHÔS.

5.2.p 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 [77].

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 5.7).

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 29.

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

Listing 29 Example in PHÔS of a loop.

5.2.q Constraints

It has been discussed that the syntax of constraints should be changed to be declutter function/synthesizable 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 30.

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 30 Example in PHÔS of a constrained synthesizable block.

5.2.r Unconstrained

As mentioned in Section 4.4, constraints only work for non-cyclic constraints, however this limitations removes the advantage of having a recirculating mesh inside of 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 synthesizable block. An example of an unconstrained block can be seen in Listing 31.

Additionally, unconstrained block allow the user to create their own signal, 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                PHÔS
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);

    output
}

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

Listing 31 | Example in PHÔS of a constrained synthesizable block.

5.2.5 Stack collection and synthesizable 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 as 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, it is required the parts of the code that perform conversion between high-level values and low-level values be synthesizable. 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 artefact 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 off of 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* [69].

5.2.t 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.

Element	Item	Statement	Description	Example
Import	✓	✓	Imports a module into the current module.	`import std::intrinsic as intrinsic;`

Function	✓	✓	Declares a function.	`fn sum(a: (uint...)) -> uint {`
				`  a |> fold(0, |acc, x| acc + x)`
				`}`

Synthesizable	✓	✗	Declares a synthesizable function.	`syn gain(in: optical) -> optical {`
				`  a |> std::intrinsic::gain(10 dB)`
				`}`

Type alias	✓	✗	Declares a type alias.	`type Voltage = float;`
Constant	✓	✓	Declares a constant.	`const PI = 3.141592;`

Structure	✓	✗	Declares a new data structure.	`struct Point {`
				`  x: float,`
				`  y: float`
				`}`

Enumeration	✓	✗	Declares a new ADT.	`enum Color {`
				`  Red,`
				`  Green,`
				`  Blue,`
				`}`

Local	✗	✓	Declares a new local variable	`let (a, b...) = (1, 2, 3);`
Expression	✗	✓	Declares a new expression.	`1 + 2`

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.u 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 expression is available in Table 10.

	Expression	Description	Example
Control flow	**Block**	Declares a new block of code.	`{ let a = 10; let b = 20; a + b }`
	If/Else/Elif	A conditional statement for branching.	`if a > b { a } else { b }`
	Match	A conditional statement for branching.	`match a { 1 => "one", _ => "other" }`
	Loop	A loop statement.	`for i in 0..5 { print(i) }`
	Return	Returns a value from a function.	`return 1`
	Break	Breaks out of a loop.	`for i in 0..5 { break; }`
	Continue	Continues a loop, terminating the current iteration and moving on the the next one.	`for i in 0..5 { continue; }`
	Yield	Yields a value from an iterator.	`for i in 0..5 { yield i; }`
Constant	**Path**	A path to a value, constant, or other item.	`std::intrinsic::gain` `float::PI`
	Identifier	A name that refers to a value, constant, or other item.	`gain` `PI`

	Literal	A literal value.	`1 dBc`
			`true`
			`0.5 MHz`

	None	The none value.	`none`
Unary	**Negation**	Negates a value.	`-1`
	Not	Negates a boolean value.	`!true`
	Binary not	Negate a binary value.	`!0xFF`

Binary	Addition	Adds two values.	`1 + 2`
	Subtraction	Subtracts two values.	`1 - 2`
	Multiplication	Multiplies two values.	`1 * 2`
	Division	Divides two values.	`1 / 2`
	Modulo	Calculates the remainder of a division.	`1 % 2`
	Exponentiation	Raises a value to a power.	`1 ** 2`
	Bitwise and	Performs a bitwise and operation.	`1 & 2`
	Bitwise or	Performs a bitwise or operation.	`1 | 2`
	Bitwise xor	Performs a bitwise xor operation.	`1 ^ 2`
	Bitwise shift left	Performs a bitwise shift left operation.	`1 << 2`
	Bitwise shift right	Performs a bitwise shift right operation.	`1 >> 2`
	Less than	Checks if a value is less than another.	`1 < 2`
	Less than or equal	Checks if a value is less than or equal to another.	`1 <= 2`
	Greater than	Checks if a value is greater than another.	`1 > 2`
	Greater than or equal	Checks if a value is greater than or equal to another.	`1 >= 2`
	Equal	Checks if a value is equal to another.	`1 == 2`
	Not equal	Checks if a value is not equal to another.	`1 != 2`
	Logical and	Checks if two boolean values are both true.	`true && false`
	Logical or	Checks if either of two boolean values are true.	`true || false`
Special	Pipe	Pipes a value into a function.	`1 |> f`
	Parenthesized	Groups an expression.	`(1 + 2) * 3`
	Tuple	Creates a tuple.	`(1, 2)`
	Cast	Casts a value to a different type.	`1 as uint`
	Index	Indexes into a value.	`a[1]`
	Member access	Accesses a member of a value.	`a.b`
	Function call	Calls a function.	`f(1, 2)`
	Method call	Calls a method.	`a.f(1, 2)`
	Partial	Partially applies a function.	`set f(1)`
	Closure	Creates a closure.	`|x| x + 1`
	Range	Creates a range.	`1..2 1..=2 ..3 4..`
	Array	Creates an array.	`[1, 2]`
	Object instance	Creates an object instance.	`A {a: 1, b: 2} B(1, 2)` `MyEnum::A C`

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 categorized 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 synthesizable blocks that come with the language. The standard library will be written in a mixture of PHÔS for synthesizable 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 organized 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 synthesizable 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 limiting, 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, as 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 block following the feedforward approximation (Section 2.2.c).

The standard library should also decouple synthesizable 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 with 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, that is 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.

5.4 Compiler architecture

The design of the PHÔS compiler is inspired in parts by LLVM, *Rust*'s compiler, and *Java*. 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 as in this section performs the compilation step

5.4.a Lexing

5.4.b Parsing

5.4.c The abstract syntax tree

5.4.d Desugaring

5.4.e AST to high-level intermediary representation

5.4.f HIR to medium-level intermediary representation

5.4.g MIR to bytecode

5.5 Virtual machine

5.6 Execution artefacts

5.7 Marshalling library

5.7.a Moving data around

5.7.b Modularity

5.8 Place-and-route

5.9 Hardware abstraction library

5.10 Constraint solver

5.11 Adopting PHÔS

5.12 State of the project

Due to the complexity of implementing a software ecosystem, PHÔS is still in its infancy. While some components were created and tested, such as the *parser*, the *abstract syntax tree*, and a *syntax highlighter*, the language is not currently

usable. Therefore, the language is a work in progress and the syntax is subject to changes. Additionally, examples serve as a way to illustrate the language and are not necessarily valid.

5.13 Putting it all together

6.

EXAMPLES OF PHOTONIC CIRCUIT PROGRAMMING

6.1 USING TRADITIONAL PROGRAMMING LANGUAGES

7.

Future work

7.1 Implementation

7.2 Dependent types & refinement types

7.3 Advanced constraint solving

7.4 Improve simulation using an ECS

7.5 Co-simulation with digital electronic

7.6 Towards co-simulation with analog electronic

7.7 Place-and-route

7.8 Programming of generic photonic circuits

7.9 Language improvements

7.9.a Bitflags

7.9.b Error handling

7.9.c Generics

7.9.d Triggers and clocks

7.9.e Macros, reflection and meta programming

7.9.f Traits and type classes

7.10 PHÔS for generic circuit design

8.
Conclusion

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, 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

[4] 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

[5] "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/

[6] 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

[7] 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

[8] 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/

[9] 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/

[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] 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.

[18] 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

[19] 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/

[20] 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

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

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

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

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

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

[26] 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.

[27] 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

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

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

[30] 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/

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

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

[33] 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).

[34] 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).

[35] 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

[36] 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

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

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

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

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

[41] 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

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

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

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

[45] 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

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

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

[48] 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

[49] 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/

[50] 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.

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

[52] "Verilog-ams language reference manual," Accellera Systems Initiative, Elk Grove, CA 95758, USA, Jun. 2014.

[53] "IEEE Standard for VHDL Language Reference Manual," IEEE. Accessed: Mar. 10, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/8938196/

[54] 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/

[55] 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

[56] 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).

[57] 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).

[58] 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/

[59] e. a. Jan Decaluwe, "Myhdl." Accessed: May 20, 2023. [Online]. Available: http://www.myhdl.org/

[60] 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

[61] G. Rossum, "Python for unix/c programmers copyright 1993 guido van rossum 1," in *Proc. NLUUG Najaarsconferentie*, 1993.

[62] 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/

[63] 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

[64] 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/

[65] 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/

[66] 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/

[67] 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

[68] 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

[69] 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

[70] 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

[71] 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/

[72] "IEEE Standard for Binary Floating-Point Arithmetic," IEEE. Accessed: Jun. 7, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/30711/

[73] HaskellWiki, "Algebraic data type." https://wiki.haskell.org/Algebraic_data_type (accessed: Jun. 7, 2023).

[74] F. O.-L. D. of Computing, "Aggregate type." https://foldoc.org/aggregate (accessed: Jun. 7, 2023).

[75] B. O'Sullivan, J. Goerzen, and D. B. Stewart, *Real World Haskell: Code You Can Believe In*, 1. ed., [nachdr.], Beijing: O'Reilly, 2010.

[76] 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/

[77] 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

1.

Annexes