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

1 Introduction 2
1.1 Motivation 2
1.1.a Research questions 3
2 Programmable photonics 4
2.1 Photonic processors 4
2.1.a Components 5
2.1.b Meshes 6
2.1.c Fast modulators 6
2.1.d Fast detectors 6
2.1.e Amplifiers 6
2.1.f Filters 6
2.1.g Feedforward and recirculating mesh 6
2.1.h Potential use cases of photonic processors 7
2.1.i Embedding of photonic processor in a larger system 7
2.2 Circuit representation 7
2.2.a Bi-directional systems 7
2.2.b Feedforward approximation 7
2.3 Difficulties 8
2.3.a Wavelength as a continuum 8
2.3.b Amplitude as a continuum 8
2.3.c Temperature dependence 8
2.3.d Manufacturing tolerances 8
2.3.e Non-linearities 8
2.4 Initial design requirements 8
2.4.a Interfacing 8
2.4.b Programming 8
2.4.c Reconfigurability 8
2.4.d Tunability 8
3 Programming of photonic processors 9
3.1 Programming languages as a tool 9
3.2 Components of a programming ecosystem 10
3.2.a Language specification & reference 11
3.2.b Compiler 12
3.2.c Hardware-programmer & runtime 13
3.2.d Debugger 13
3.2.e Code formatter 14
3.2.f Linting 14

3.2.g Code editor ... 15
3.2.h Testing & simulation ... 15
3.2.i Package manager ... 17
3.2.j Documentation generator ... 17
3.2.k Build system ... 18
3.2.l Summary ... 19
3.3 Overview of syntaxes ... 22
3.3.a Traditional programming languages ... 23
3.3.b Digital hardware description languages ... 25
3.3.c Analog simulation languages ... 25
3.3.d Summary ... 25
3.4 Comparison of existing programming ecosystems ... 25
3.5 Analysis of programming paradigms ... 25
3.5.a Imperative programming ... 25
3.5.b Object-oriented programming ... 25
3.5.c Functional programming ... 25
3.5.d Logic programming ... 25
3.5.e Dataflow programming ... 26
3.6 Existing framework ... 26
3.7 Long term compatibility and cross-vendor support ... 26
3.8 Summary ... 26
4 Translation of intent ... 27
5 The PHÔS programming language ... 28
5.1 PHÔS: an initial specification ... 28
5.2 Standard library ... 28
5.3 Compiler architecture ... 28
5.3.a Lexing ... 28
5.3.b Parsing ... 28
5.3.c The abstract syntax tree ... 28
5.3.d Desugaring ... 28
5.3.e AST to high-level intermediary representation ... 28
5.3.f HIR to medium-level intermediary representation ... 28
5.3.g MIR to bytecode ... 28
5.4 Virtual machine ... 28
5.5 Execution artefacts ... 28
5.6 Marshalling library ... 28
5.6.a Moving data around ... 28
5.6.b Modularity ... 28
5.7 Adopting PHÔS ... 28

6 Examples of photonic circuit programming 29
6.1 Using traditional programming languages 29
7 Extending PHÔS to generic circuit design 30
8 Simulation in PHÔS 31
8.1 Co-simulation with digital electronic 31
8.2 Towards co-simulation with analog electronic 31
9 Future work 32
10 Conclusion 33

Glossary

2x2 tunable coupler	*A tunable coupler with two inputs and two outputs* 6
API	*Application Programming Interface* ..... IX, 11, 14, 16, 18, 19, 20, 22
ASIC	*Application Specific Integrated Circuit* 16
CPLD	*Complex Programmable Logic Device* 4
DRY	*Don't Repeat Yourself* 18
DSL	*Domain Specific Language* 9, 11, 22, 25
DSP	*Digital Signal Processor* .....
FIR	*Finite Impulse Response* 6, 7
FPGA	*Field Programmable Gate Array* 2, 4, 9, 16, 22, 23
FPPGA	*Field Programmable Photonic Gate Array* 4
GPL-3.0	*GNU General Public License version 3.0* 21
HDL	*Hardware Description Language* 9, 10, 11, 12, 16, 17, 22, 23
HLS	*High Level Synthesis* 22, 23
HTTP	*Hypertext Transfer Protocol -- the protocol used for web navigation* 16
IDE	*Integrated Development Environment* 15
IIR	*Infinite Impulse Response* 6, 7
IP	*Intellectual Property* 17
JTAG	*Joint Test Action Group - A standard for testing integrated circuits* 13
LSP	*Language Server Protocol* 15
MEMS	*Microelectromechanical Systems* 6
PHÔS	*Photonic Hardware Description Language* 19, 21
PIC	*Photonic Integrated Circuit* ..... IX, 2, 3, 4, 5, 16
PRG	*Photonics Research Group* .....
RF	*Radio Frequency* 2
RTL	*Register Transfer Level* 9
SPICE	*Simulation Program with Integrated Circuit Emphasis* 2, 3, 9, 16, 23
SQL	*Structured Query Language* 9
TDD	*Test Driven Development* 15
VHDL	*VHSIC (*Very High Speed Integrated Circuit*) Hardware Description Language* 10, 21, 22
VHSIC	*Very High Speed Integrated Circuit* ..... VII, 10
Verilog-A	*A continuous-time subset of Verilog-AMS (*Verilog for Analog and Mixed Signal*)* 3
Verilog-AMS	*Verilog for Analog and Mixed Signal* ..... VII, 3, 9, 21, 22, 23

List of figures

Figure 1: A black box representation of a ring resonator. 7

List of tables

Table 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). 5

Table 2: Different states of a 2x2 optical coupler, (a) a simplified coupler, (b) in “bar” mode, (c) in “cross” mode, (d) in “partial” mode. 6

Table 3: Comparison of programming ecosystem components and their importance for API (*Application Programming Interface*) development and language design. 20

Table 4: This table compares the ecosystems of different programming and hardware description languages. 21

List of listings

Listing 1: Simple function that prints `"Hello, world!"`, in *Rust*. 19
Listing 2: Function that prints `"Hello, {name}!"` with a custom name, in *Rust*. 19
Listing 3: *FizzBuzz* implemented in *C*, based on the *Rosetta Code* project [1]. 24
Listing 4: *FizzBuzz* implemented in *Rust*, based on the *Rosetta Code* project [1] 24
Listing 5: *FizzBuzz* implemented in *Python*, based on the *Rosetta Code* project [1]. 25

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*) [2] 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 [3]. 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 [4].

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) [4]. 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 [5]. 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 [5, 6], which has led to efforts to simulate photonic circuit using SPICE (*Simulation Program with Integrated Circuit Emphasis*) [7]. 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 [8].

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*) [8].

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 [9]. Therefore, it is of interest to the user to co-simulate [5, 10] their photonic circuits with the other components of their systems. This is a problem that is partly addressed using SPICE simulation [7]. However, especially with regards to digital co-

simulation, SPICE tools are often lacking, making the process difficult [11], 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*) [12].

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 [13]. 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*)[2] 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 Table 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)

TABLE 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.b.

This question, which will be the driving factor behind this first section, will be answered in Section 2.2.b. 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 Table 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)

TABLE 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 [2, 13, 14].

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 [2], 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. [2, 13, 14]

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 [15], 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[14], 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 Bi-directional systems

2.2.b 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 [16]. 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 [5].

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 1.

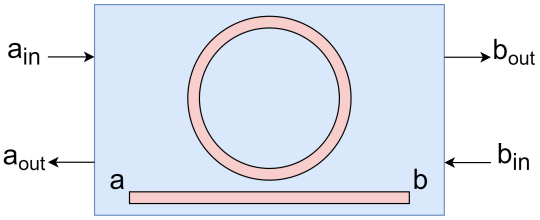

Figure 1 A black box representation of a ring resonator. The input and output ports are labeled as $a_{\text{in}}, b_{\text{in}}$ and $a_{\text{out}}, b_{\text{out}}$ respectively.",

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 3.2 – 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 or the steps needed to perform this computation. 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 [4]

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 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 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 (*Very High Speed Integrated Circuit*) Hardware Description Language *)*: an HDL (*Hardware Description Language*) that is used to describe digital hardware;
- *Verilog-AMS (*Verilog for Analog and Mixed Signal*)*: an analog simulation language that has been used to describe photonic circuits [17];

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.

3.2.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 [18]

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 [18]

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 [19].

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 [20].

3.2.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 [21]

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 [22], 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 [22] 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 [22]. 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 [23]. 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 [24]. 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.

3.2.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 [25]

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

Adapted from [25]

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.

3.2.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 [21]

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 [26, 27, 28], 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".

3.2.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* [29] and *GNU style* [30].

Adapted from [31]

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 [32]. 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.

3.2.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 [31]

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 [33].

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` [33]. This rules suggests that functions that are too complex as defined in the literature [34] should be either reworked or split into smaller, more readable code units.

3.2.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 [35]

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 features 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* [36]. 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 [37]. 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 [37].

3.2.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 [38, 39]

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*) [40]. Additionally, external software can provide metrics such as *code coverage* that inform the user of how much of their code is being tested [41].

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* [42].

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 [38, 39]

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 [7].

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 [5, 6]. 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.

3.2.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 [43]

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 [44].

3.2.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.

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 [36], it is essential to take this need from users into account.

3.2.k Build system

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

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.

3.2.1 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 3, 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 3, 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 3.8, 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 run-time.	~	~
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 3 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 4 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 4, most other languages don't come with first party support fot these tools either. However, as will be discussed in Section 3.3, 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 is has a thriving ecosystem of many libraries, colloquially called *crates*, fit for this purpose.

One can also see from Table 4, 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 5.7.

Components	Traditional languages			Hardware description & simulation languages	
	C	Rust	Python	Verilog-AMS	VHDL
Language specification	✔ [45]	✘ [20]	✘ [46]	✔ [47]	✔ [48]
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 4 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* 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.

3.3 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 syntaxes 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 [49], but is also being used for novel use cases such as quantum programming [50].
- *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 [51].
- *Python* is a common language that is used by a wide proportion of researchers and engineers [36, 52], which makes it a great candidate as the starting point of any language development. It is also used for some HDL development [53] 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 [54].
- *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* [17].
- *SPICE*: a language that is used for the simulation of analog electronics. *SPICE* as seen use in the development of photonic simulation [7].

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.

3.3.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* [55]. *C* has had a large influence on languages such as *Python* [56] and *Rust* [20] – 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 [57].

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 redable 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 linespace sensitive and statement can span multiple lines;

- *C* uses a preprocessor to perform test 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* needs a `main` function to be defined, which is the entry point of the program.

```rust
fn main() {
    for i in 1..=100 {
        print!("{i}\r");
        if i % 3 == 0 {
            print!("Fizz");
        }
        if i % 5 == 0 {
            print!("Buzz");
        }
        println!();
    }
}
```

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

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

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

3.3.b Digital hardware description languages

This are a mistake.

3.3.c Analog simulation languages

3.3.d Summary

3.4 Comparison of existing programming ecosystems

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 processor, special care needs to be taken to ensure high usability and productivity through the 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 the development of 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 Analysis of programming paradigms

3.5.a Imperative programming

3.5.b Object-oriented programming

3.5.c Functional programming

3.5.d Logic programming

3.5.e Dataflow programming

3.6 Existing framework

3.7 Long term compatibility and cross-vendor support

3.8 Summary

After comparing existing ecosystems as well as the existing framework used to program photonic processors, further discussion on ...

Should a new programming language be created for photonic processors?

- What type of programming paradigm should be used?
- What features should be included in the language?
- What tools should be created to aid in the development?
- What existing programming languages, if any, should be used as a base?

4.

Translation of intent

5.
The PHÔS programming language

5.1 PHÔS: an initial specification

5.2 Standard library

5.3 Compiler architecture

5.3.a Lexing

5.3.b Parsing

5.3.c The abstract syntax tree

5.3.d Desugaring

5.3.e AST to high-level intermediary representation

5.3.f HIR to medium-level intermediary representation

5.3.g MIR to bytecode

5.4 Virtual machine

5.5 Execution artefacts

5.6 Marshalling library

5.6.a Moving data around

5.6.b Modularity

5.7 Adopting PHÔS

6.

Examples of photonic circuit programming

6.1 Using traditional programming languages

7.

EXTENDING PHÔS TO GENERIC CIRCUIT DESIGN

8.
Simulation in PHÔS

8.1 Co-simulation with digital electronic

8.2 Towards co-simulation with analog electronic

9.
Future work

10.
Conclusion

Bibliography

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

[2] 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

[3] 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

[4] "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/

[5] 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

[6] 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

[7] 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/

[8] 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/

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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.

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

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

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

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

[21] 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.

[22] 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

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

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

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

[26] 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/

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

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

[29] 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).

[30] 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).

[31] 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

[32] 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

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

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

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

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

[37] 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

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

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

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

[41] 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

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

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

[44] 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

[45] 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.

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

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

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

[49] 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/

[50] 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

[51] 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).

[52] 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).

[53] 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/

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

[55] 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

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

[57] 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/