UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME TORRESAN BAZZO

NESOI: A Framework for Building Distributed Intelligent Applications in Programmable Data Planes

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Weverton Luis da Costa
Cordeiro

Porto Alegre
October 2022

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Helena Lucas Pranke
Pró-Reitoria de Ensino (Graduação e Pós-Graduação): Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Marcelo Walter
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

AGRADECIMENTOS

Agradeço ao Prof. Dr. Weverton Cordeiro, meu orientador neste trabalho, pelas reuniões semanais e a nossa constante troca de ideias para desenvolver este trabalho. Ao Willian Tomaz, por ter feito parte deste projeto e contribuído no desenvolvimento deste trabalho. Agradeço à minha família, por todo o apoio antes mesmo desta jornada começar, obrigado por sempre estarem comigo e por terem tornado essa jornada possível. A todos meus amigos e colegas de curso que estiveram presentes nestes últimos anos, em especial ao Guilherme Malta e Nícolas Duranti, a quem não consigo agradeçer o suficiente por todas as conversas, troca de ideias, e trabalhos em grupo. A Natália, por sua companhia mesmo nas noites em claro, por suas palavras de motivação nos momentos mais difíceis, e pelos momentos de descontração e recarrega que encontrávamos no meio de tanto caos, sejam eles construídos pelas séries que maratonávamos juntos, nossos jogos de cartas ou pelo simples toque suave na mão. A todos que não citei mas conviveram comigo nessa trajetória, deixo meus sinceros agradecimentos por terem feito a diferença nessa construção.

ABSTRACT

Software-defined networking and data plane programmability are relatively new concepts that enable researchers and network operators to develop network applications to run directly on the data plane. More recently, studies proved that the data plane can run intelligent applications implementing techniques of artificial intelligence and machine learning. However, building complex intelligent applications that run distributedly in the data plane remains challenging. On one hand, network developers must write source code for each switch in which the intelligent application will run. On the other hand, there might have dozens to hundreds of switches in the network in which the distributed intelligent network application might need to operate in the programmable data plane. To tackle this problem, we propose in this work NESOI, framework for building distributed intelligent applications in the programmable data plane (PDP). We developed a compiler and a programming language that, based on a network specification, generates switch code on multiple target languages (P4 and NPL), implementing the distributed logic for such applications. The ultimate goal is having a framework that works like a TensorFlow for programmable forwarding planes. Our results indicate that NESOI simplifies the development process of intelligent distributed applications by using templates as the basis for the code generation process, presenting a flexible approach to define neural networks and their potential to generate target programs on multiple languages and switch architectures.

Keywords: Programmable Data Plane. Neural Networks. In-Network Intelligence. Compiler. Programming Language. P4. NPL.

RESUMO

Rede definida por software e a programabilidade do plano de dados são conceitos relativamente novos que permitiram que pesquisadores e operadores de rede desenvolvam aplicações de rede para serem executados diretamente no plano de dados. Mais recentemente, estudos comprovaram que o plano de dados pode executar aplicações inteligentes implementando técnicas de inteligência artificial e aprendizado de máquina. No entanto, a construção de aplicações inteligentes complexas que executem de forma distribuída no plano de dados é um desafio. Por um lado, os desenvolvedores de rede devem escrever código-fonte para cada switch na qual a aplicação inteligente será executada. Por outro lado, pode haver dezenas a centenas de switches na rede em que a aplicação de rede inteligente distribuída pode precisar operar no plano de dados programável. Para resolver este problema, propõe-se neste trabalho NESOI, um framework para construção de aplicações inteligentes distribuídas para plano de dados programáveis (PDP). Como parte do trabalho, foi desenvolvido um compilador e uma linguagem de programação que, com base em uma especificação de rede, gera código de dispositivos de rede em várias linguagens de destino (P4 e NPL), implementando a lógica distribuída para tais aplicações. Assim, tem-se um framework que funciona de certa forma como um TensorFlow para planos de dados programáveis. Nossos resultados indicam que o framework proposto simplifica o processo de desenvolvimento de aplicações inteligentes distribuídas com o uso de templates como base para o processo de geração de código, apresentando uma abordagem flexível para definir redes neurais e seu potencial para gerar os programas alvo em múltiplas linguagens e arquitetura de dos dispotivos de rede.

Palavras-chave: Plano de Dados Programável, Redes Neurais, Inteligência na Rede, Compilador, Linguagem de Programação, P4, NPL.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
AI	Artificial Intelligence
ANN	Artificial Neural Network
BNN	Binary Neural Network
DNN	Dense Neural Network
HLIR	High-Level Intermediate Representation
JSON	JavaScript Object Notation
LOC	Line Of Code
ML	Machine Learning
MAT	Match-Action Table
NOS	Network Operating System
NN	Neural Network
NPL	Network Programming Language
PDP	Programmable Data Plane
P4	Protocol-Independent Packet Processing
POF	Protocol Oblivious Forwarding
SNMP	Simple Network Management Protocol
SDN	Software Defined Networking
TOML	Tom's Obvious Minimal Language
TCP	Transmission Control Protocol
WSL	Windows Subsystem for Linux

CONTENTS

1 INTRODUCTION

Software-Defined Networking (SDN) has long been a reality in computer networking, separating the control plane from the data plane (forwarding plane) and enabling a centralized manager (running as software-centric services on top of a network operating system) to control multiple data plane devices. The control plane handles the decision-making on operating the network traffic, and the data plane forwards traffic based on those decisions (FEAMSTER; REXFORD; ZEGURA, 2014).

More recently, a novel set of SDN-related solutions (*e.g.* P4 (BOSSHART et al., 2014)) revived the concept of *data plane programmability*. These programming languages, along with the concept of reconfigurable match-action tables (BOSSHART et al., 2013), enabled researchers and network practitioners to develop novel protocols and services that run directly on the data plane, while reprogramming forwarding devices and customizing the network behavior (CORDEIRO; MARQUES; GASPARY, 2017).

In this context, there has been a substantial effort by the industry and research community to explore the capabilities of programmable data planes to deliver innovative services directly in network switches, trend which was coined with *in-network computing* (SAPIO et al., 2017). Examples of applications and services that explore the capabilities of programmable data plane include load balancing (KATTA et al., 2016; NKOSI; LYSKO; DLAMINI, 2018), intrusion detection (SHAGHAGHI; KAAFAR; JHA, 2017; LIN et al., 2015), firewall (KRONGBARAMEE; SOMCHIT, 2018; CAPROLU; RAPONI; PIETRO, 2019), and adaptive routing mechanisms (PIZZUTTI; SCHAEFFER-FILHO, 2019). This trend has also been explored in the intersection of Networking and Artificial Intelligence, with researchers coming up with novel set of *intelligent* applications (*i.e.* that implements artificial intelligence (AI) or machine learning (ML) techniques) to handle network traffic being developed directly on the forwarding plane. Before the emergence of PDPs, these intelligent applications were constrained to the control plane, thus making it difficult for them to keep pace with the ever-increasing speed of network links (which now may easily go beyond 100G).

To cite a few examples: Xiong and Zilberman (2019) explored packet classification using in-network supervised and unsupervised machine learning algorithms. The works of N2Net (SIRACUSANO; BIFULCO, 2018) and BaNaNa (SANVITO; SIRACUSANO; BIFULCO, 2018) introduced the initial implementation of binary neurons in network devices. And Luizelli et al. (2021a) provided empirical evidence of an intelligent

distributed data plane. Although these investigations provide evidence that running distributed intelligent applications on the data plane is feasible, writing complex intelligent applications for programmable data planes remains a challenging task – particularly if the application must run *distributedly* in the data plane, which is the case of neural networks running distributedly in the data plane (LUIZELLI et al., 2021b; SAQUETTI et al., 2021).

To bridge this gap, we introduce NESOI[1], a framework for writing/building distributed intelligent applications in programmable data planes. In summary, from a higher level specification of an intelligent application, NESOI generates the source code implementation to be deployed in each of the switches that will run the intelligent distributed application. We develop a transpiler and an auxiliary programming language that, based on the high-level specification given by the network developer, implements the distributed logic of applications to run on multiple network devices, and generates the code for each switch that will be part of this intelligent network. Based on the network topology and the switch architecture given as input to the transpiler, it generates source code matching the target architecture of each switch in the network – to be finally compiled by the respective switch compiler and deployed onto it. We provide evidence, through a series of experiments, that NESOI is able to significantly reduce the burden in the development process of distributed intelligent applications for the forwarding plane, while minimizing bugs often associated to such complex and distributed deployments.

This work is structured as follows: in Chapter 2, the theoretical background needed to understand the proposal is explained. Chapter 3 presents an analysis of the related works, discussing the most common approaches to in-network programs and how to build applications in network switches, highlighting the relevance of the present study. In Chapter 4, we present the nominated requirements for this work, and describe the transpiler's architecture and execution steps are to fulfill the requirements. Chapter 5 we detail the language features and its basic types, giving a few code examples. Chapter 6 gives a more in-depth explanation of the parsing and templating features on the language. Chapter 7 details the results of the experiments, highlighting its main strengths. Finally, Chapter 8 presents the conclusion from our findings in this study and a brief discussion of future steps for NESOI.

[1]Nesoi, in Greek Mythology, is the goddess of the islands. We thus named our framework after Nesoi as an analogy to the archipelago that is an intelligent application that runs distributedly in the data plane, formed by parts of the program (islands) running in the switches across the network.

2 BACKGROUND

This work introduces a compiler and auxiliary programming language to build distributed intelligent applications in the programmable data plane. This chapter introduces essential concepts for a better understanding of the development of this work. We present the concepts of software-defined networking and data plane programmability to understand better the networking architecture on which the generated programs will execute. Then, we briefly describe how artificial neural networks function, focusing our discussion on multilayer perception. Lastly, we discuss the main concepts of a compiler, the classic phases of compilation, and how it handles code analysis.

2.1 Software-Defined Networking

Software-Defined Networking (SDN) is a paradigm designed to simplify the programming of network devices. SDN architecture separates the control plane from the data plane, allowing the administrator to configure the network hardware directly from a centralized controller. Because of this centralized management, the network becomes more flexible and enables the abstraction of the infrastructure from applications and network services (FIROUZI; RAHMANI, 2022).

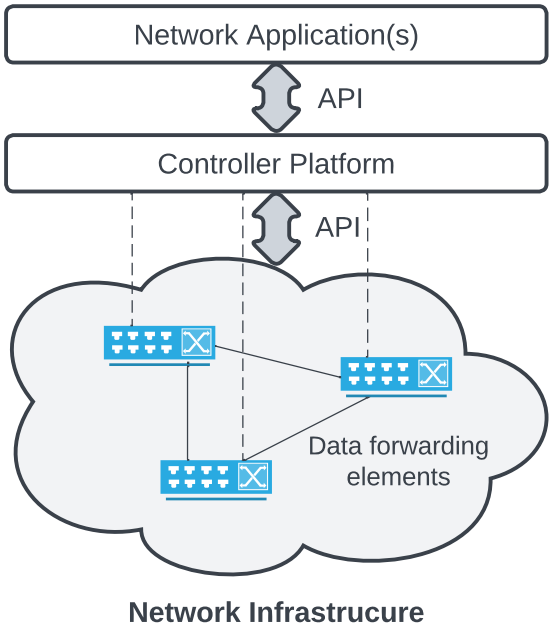
With decoupling control and forwarding layers, the control functionality is removed from network devices and implemented in a logically centralized controller or Network Operating System (NOS). This new organization simplifies policy enforcement and network (re)configuration and evolution (KIM; FEAMSTER, 2013). Figure 2.1 shows a simplified view of this architecture.

The communication between the control plane and data plane is handled by a well-defined Application Programming Interface (API). The most well-known example of API in this context is OpenFlow. OpenFlow proposes the *OpenFlow Protocol* as a standard way of communication between the SDN controller (or NOS) with a network device (MCKEOWN et al., 2008).

OpenFlow also proposes the *OpenFlow Switch*, consisting of a *Flow Table*, which indicates to the switch how to process the flow based on a table of actions associated to a flow entry, a *Secure Channel* to send commands and packets to a *controller*, and the *OpenFlow Protocol* (MCKEOWN et al., 2008).

In SDN, the forwarding decisions are flow based instead of destination based.

Figure 2.1: Simplified view of an SDN architecture



Source: Kreutz et al. (2014)

Flows do not have a well-established definition and are only constrained by the capabilities of the implementation of the specific *Flow Table*. For example, a flow entry can be defined as a TCP (Transmission Control Protocol) connection or all packets from the same switch port.

When a packet arrives at an OpenFlow Switch, it looks up a flow entry in the Flow Table and applies the corresponding set of associated instructions or actions. These actions can be: (i) forward the packet to a given port (or ports), (ii) send it to the controller via the Secure Channel, drop it, send it to the regular processing pipeline, or (iii) send it to the following flow table or to special tables.

In the context of OpenFlow Switches, OpenFlow Protocol handles the communication between the switch and the SDN controller (or NOS). The controller enables the development of forwarding devices based on a logically centralized, abstract network view by providing the necessary tools and abstractions to the network operators.

Network operators are set above the control plane, which can be seen as a management plane. It is in the management plane where the control plane defines and enforces network policy. It includes software services such as the Network Configuration Protocol (NETCONF) (ENNS, 2006) or Simple Network Management Protocol (SNMP) (CASE et al., 1989) to handle the configuration of network devices and can be used for routing,

firewalls, and load balancers. There has also been a proposal from the IRTF Software-Defined Networking Research Group (SDNRG) (HALEPLIDIS et al., 2015) to maintain the management plane at the same level as the control plane.

Finally, SDN decoupling can be seen as an easier way to program applications due to the separation of concerns between the definition of network policies, their implementation in switching hardware, and the forwarding traffic. This created flexibility makes it easier to introduce new abstractions in networking, simplifying network management and assisting network evolution and innovation progress.

2.2 Data Plane Programmability

As described in Section 2.1, Software Defined Networking decouples the control plane from the data (forwarding) plane, in which the decisions of packet forwarding are made in a (logically) centralized controller. The controller uses standard protocols like OpenFlow (MCKEOWN et al., 2008) to configure switches on the data plane with rules on how to forward data flows.

More recently, a newer generation of SDN-relates solutions introduced the notion of *data plane programmability*. The ability to program the data plane makes it possible to change the network behavior and to make networking more secure, improving its reliability, availability, and integrity (AVIZIENIS et al., 2004). Additionally, it may enable significant control plane and protocol modifications for forwarding devices without needing hardware upgrades (CORDEIRO; MARQUES; GASPARY, 2017).

Data plane programmability is made possible by programming languages designed specifically to work on the data plane. The most well-known examples of programming languages are POF (SONG, 2013) and P4 (BOSSHART et al., 2014), but other examples such as NPL [1], Domino (SIVARAMAN et al., 2016), and SNAP (ARASHLOO et al., 2016) are also good alternatives.

Protocol-Oblivious Forwarding (POF) is an OpenFlow (MCKEOWN et al., 2008) extension that enables greater flexibility in packet processing definition. POF includes extended versions of OpenFlow instructions and actions to achieve this goal. Programming Protocol-Independent Packet Processing (P4) is a target-independent abstract model for packet processing. Its code organizes the program as sections of data declaration, parser logic, and match/action tables. P4 can also specify processing primitives beyond those

[1] NPL website: <https://nplang.org/>

supported by OpenFlow. Still, it differs from POF by providing an adequate abstraction level for generic packet parsing and processing and being OpenFlow-independent.

The network infrastructure has been going through significant changes in what was once described as "ossified" (MCKEOWN et al., 2008; TURNER; TAYLOR, 2005). SDN allowed the decoupling of the control and forwarding layers, and now, the concept of data plane programmability brings flexibility through home-brewed network protocols and customizes control plane apps.

2.3 Artificial Neural Networks

Artificial Neural Networks (ANNs), also known as Neural Networks (NNs), are computational tools used for decades to solve complex problems, such as classification, data analysis, function estimation, pattern recognition, etc. ANNs can solve problems by first learning from examples. They also take inspiration from studies of the mechanisms for information processing in biological nervous systems (BISHOP, 1994).

ANNs can be viewed as neuron (nodes) connections in a directed graph layout. The edges between neurons are weighted to decide whether a neuron should activate or not (JAIN; MAO; MOHIUDDIN, 1996). The network typically learns the connection's weights from training data.

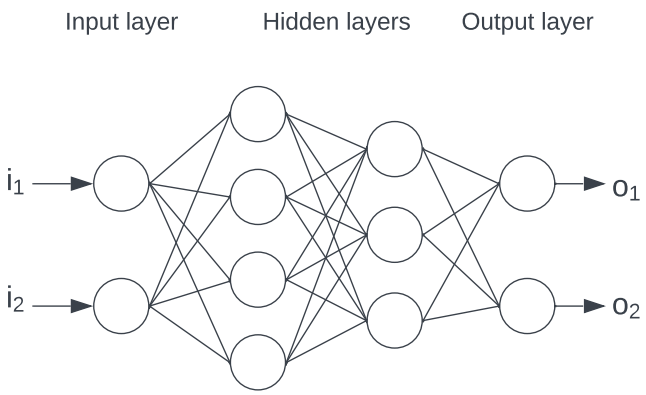
ANNs can be divided into two groups based on their connection pattern:

- *feed-forward* networks, in which the flow of the network only goes in one direction (no loops),
- *recurrent* networks, in which feedback connections resulting in loops are present.

Feed-forward networks produce one set of output values from a given input. They also do not have an awareness of *memory*, as their response to a piece of information is not dependent on the previous state of the network. On the other hand, recurrent networks are dynamic systems, meaning that their input states change based on the outputs of their neurons (JAIN; MAO; MOHIUDDIN, 1996).

The multilayer perceptron is one of the most used types of ANNs. It consists of a system of interconnected neurons organized in layers. An example of a multilayer perceptron structure is shown in Figure 2.2, containing two input neurons in its input layer, two hidden layers with four and three neurons, respectively (multilayer perceptrons can have an arbitrary number of hidden layers), and two neurons in its output layer.

Figure 2.2: Multilayer perceptron structure with two hidden layers



Source: Author

Multilayer perceptrons are described as fully connected, where each node is connected to every node in the previous and subsequent layers. Selecting a suitable set of connecting weights and activation functions has demonstrated that the multilayer perceptron can approximate any smooth, measurable function (HORNIK; STINCHCOMBE; WHITE, 1989).

Multilayer perceptrons can learn through training. Multilayer perceptrons learn in a supervised manner, where training data is supplied to the training algorithm (*e.g.* Backpropagation algorithm (RUMELHART; HINTON; WILLIAMS, 1985)). The training data contains training values and their corresponding expected output. It modifies the network's weights until the desired input-output mapping is achieved (GARDNER; DORLING, 1998).

2.4 Compilers

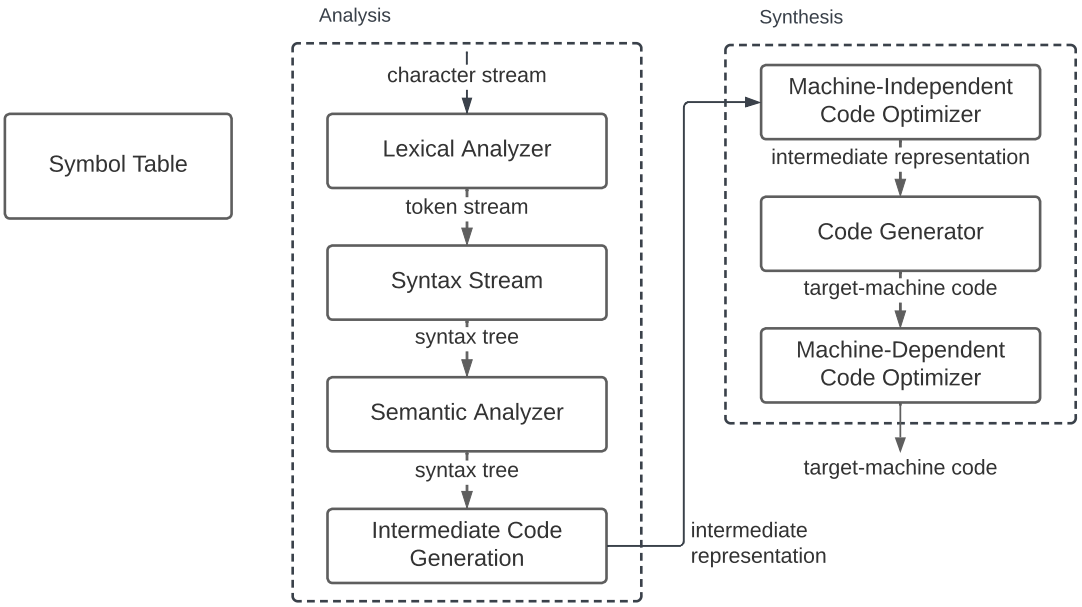
According to Aho et al. (2007), a compiler is *a program that can read a program in one language — the source language — and translate it into an equivalent program in another language — the target language*.

The compilation process is comprised of two main parts: analysis and synthesis. The *analysis* is responsible for the syntactical and semantic validation, informing any error or warning to the user so they can take corrective action. It is also responsible for creating a *symbol table*, which is a data structure that holds information such as the declared position, type, and scope of an identifier, and an intermediate code representa-

tion of the source program. The *synthetics* part constructs the target program from the intermediate code representation and the symbol table, generated in the analysis part.

The processes of analysis and synthesis may be broken down into a more specific sequence of phases. A typical series of phases for a compiler is shown in Figure 2.3, but although it is represented sequentially, many phases may be grouped as one.

Figure 2.3: Phases of a compiler



Source: Adapted from Aho et al. (2007)

Even though a compiler's main usage is generating a target program in machine code, a compiler can also generate a program from a high-level source language into another high-level target language. This type of compiler is known as a *transpiler*, or *source-to-source* compiler.

Lexical Analysis. Also known as *scanning*, the *lexical analysis* is responsible for creating *lexemes* from the source program. A lexeme is a unit of lexical meaning considering the grammar of the source-language program. It then enhances each lexeme with relevant information and passes it to the subsequent phase, syntax analysis.

Syntax Analysis. Also known as *parsing*, the *syntax analysis* creates a *syntax tree* representation of the source-program from the tokens generated on the lexical analysis.

Semantic Analysis. This step checks whether the source program is semantic consistent with the language definition. It gathers type information and saves it into the syntax tree or symbol table. In this phase, it performes *type checking*.

Intermediate Code Generation. This step typically generates a low-level rep-

resentation of the source program. The main characteristic of an intermediate code representation is to be easy to produce and translate into the target language. A typical intermediate representation is the *three address-code* representation.

Code Generation. This step takes the intermediate code representation and maps it into the target language.

3 RELATED WORK

In this chapter, we analyze the related works that approach intelligent applications running in the programmable data plane and frameworks that enable the development of such applications in a distributed matter, utilizing a high-level definition. Our goal is to summarize the methodologies and results found by other researchers and highlight the lack of a tool for generating code for switches that implement intelligent distributed applications that run on the data plane.

3.1 Intelligent Data Planes

The work of Luizelli et al. (2021b) analyses the fundamental concepts of in-network neural networks and the technical difficulties of implementing machine learning methods solely in the forwarding plane. They discuss the potentialities of ANN towards in-network intelligence, and present recent research opportunities on distributed in-networking ANN seeking to achieve self-driven networks.

Sapio et al. (2017) took the first steps toward offloading computations (*e.g.*, MapReduce) to the data plane. The authors introduced DAIET, a system capable of aggregating data on routing devices. DAIET reduces network congestion while improving overall application performance. Their P4 prototype provides a data reduction ratio of around 85 percent and a similar decrease in computing time. In a follow-up work (SAPIO et al., 2021), the authors provided evidence of the feasibility of speeding up deep neural network (DNN) training by minimizing communication overhead at single-rack scale.

Li et al. (2019) presented iSwitch, an in-switch acceleration solution for offloading the distributed training of reinforcement learning. iSwitch moves the gradient aggregation from server nodes into the network switches, reducing the training overhead considerably. Wu et al. (2019) introduced Dejavu, a system that efficiently performs network function chaining using programmable data planes. Dejavu merges multiple functions into a single monolithic application, optimizing the packet forwarding task within a single pipeline. The key idea behind merging multiple applications is to minimize recirculations required to implement a given chain, at the expense of reducing throughput super-linearly and increasing packet processing latency.

The work of Siracusano and Bifulco (2018) introduces N2Net, a simplified model of Artificial Neural Networks (ANNs), such as Binary Neural Networks (BNNs) designed

for embedded applications running on devices with constrained resources. It implements the forwarding task of a BNN network and assumes that the BNN activations, which represent the information computed by the neurons, are encoded in a portion of the packet header, which will be processed by a pipeline implementing match-action tables (MATs). This work also provides a compiler that produces the switch configuration implementing a given neural network (NN) model.

The work of Sanvito, Siracusano and Bifulco (2018) designs BaNaNa Split, a solution that uses the N2Net method to conduct NN model quantization and run such quantized models on programmable switches. A quantized model is smaller than its original version since it uses fewer bits to express a NN's activations and parameters.

The work of Swamy et al. (2022) introduces Taurus, a per-packet machine learning (ML) architecture to run on the data plane. As packets enter a switch, they are parsed to extract header-level features (such as connection duration and protocol and service types) to execute preprocessing MAT to handle data validations on a packet's fields. Then, it runs its inference module based on the extracted features in the preprocessing step and generates a numeric result that will be used on a post-processing MAT to handle the packet-forwarding decision.

The work of Gobatto et al. (2022) formalizes an optimization model for neuron placement and chaining problem, proposes programmable data plane constructs for performing neuron computation, and customize in-band telemetry for neuron intercommunication utilizing production flows. This work also contributes to the first open-source implementation of a distributed ANN on programmable data planes.

3.2 High-Level Compilers for Programmable Data Planes

The work of Gao et al. (2020b) presents a compiler named Chipmunk to transform high-level programs to switch machine code. This work uses Domino (SIVARAMAN et al., 2016) as the input program. The feature in which this compiler resembles our project is its capability to translate the source program into the P4 (BOSSHART et al., 2014) programming language. They also use a template system containing *holes* (sections) that will receive the translated code. It differs from our project by its constraint to be supported by the Tofino switch compiler, while our project permit specification of the output switch architecture to generate code based on its limitations, and the authors do not mention any possibility of developing programs implementing distributed applications.

In Sonchack et al. (2021), the authors introduce Lucid, a high-level programming language for implementing control applications in data planes based on event-driven programming. Lucid's event carries user-specific data that will trigger a handler to perform an action. Lucid's compiler translates its programs into P4 optimized for Intel Tofino. Although it can be used for programming one switch or many switches distributed across the network, it does not present a straightforward way to define distributed intelligent applications.

We also highlight the study of Gao et al. (2020a), which presents Lyra, a cross-platform language and compiler for the data plane. Lyra aims to provide a simple way to develop data plane programs for programmable data center networks. It offers the programmer an *one-big-pipeline* abstraction, which increases flexibility without placing itself too close to the hardware. It compiles programs that run in a distributed matter and in both P4 and NPL languages but does not present a straightforward way to define distributed intelligent applications.

4 NESOI: DISTRIBUTED INTELLIGENT APPS FOR PDPS

In this chapter, we describe NESOI. We first enumerate the requirements that drove our research. Then we review the methodology considered in the design of NESOI, illustrating the compilation process and describing how each process functions to solve the exposed requirements.

4.1 Requirements

The first step of this research project was to nominate the essential requirements that the compiler should satisfy. The requirements are presented below:

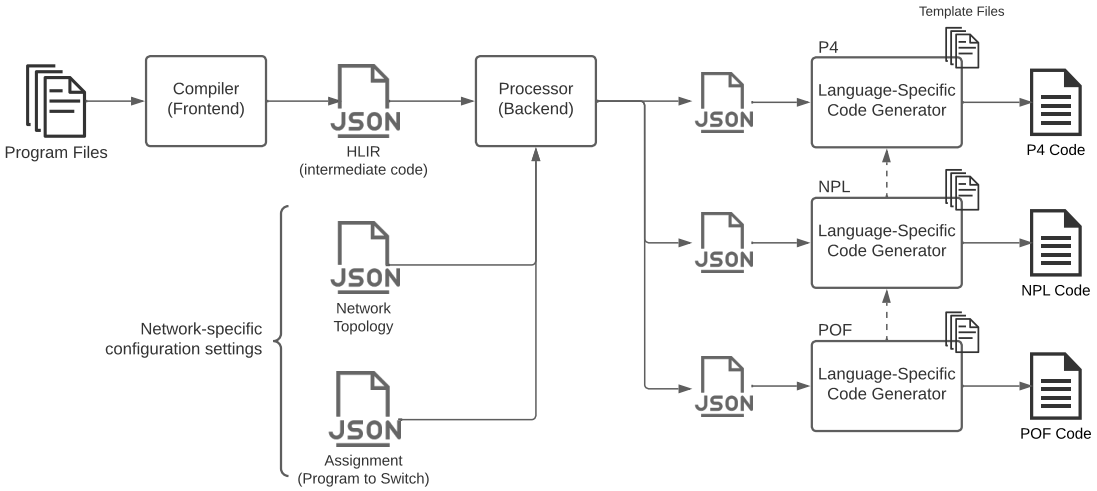
- The compiler needs be able to generate code, from a high-level specification, that can be compiled on switches that are part of a network,
- The compiler should generate code that is valid for the switches architecture,
- The compiler needs be able to distribute the program in the data plane based on the network topology provided,
- The compiler needs to be able to allow developers to extend the base library with new implementations of their own programs,
- The packet parser definition should be flexible, allowing the operator to specify which level of parsing should be extracted in each layer of a described network

4.2 Methodology

The transpiler's architecture was designed to be flexible and extendable from the compiler's developer perspective. The proposed architecture is built as a series of processing components organized so that each component's output serves as the subsequent component's input.

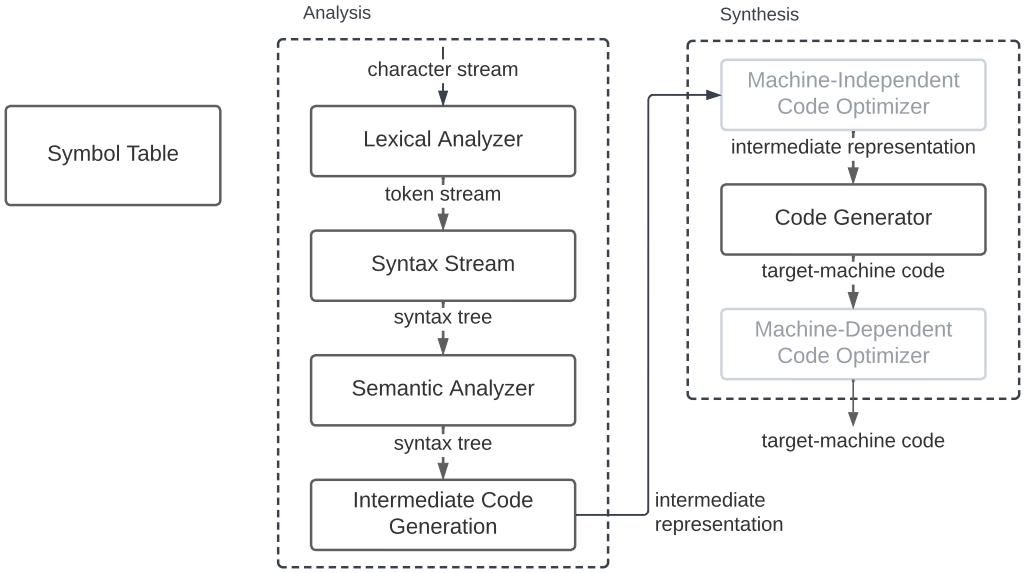
Figure 4.1 shows the designed processing pipeline. There are three main processes: *Frontend*, *Backend*, and *Language-Specific Code Generator*. The *Frontend* and *Backend* processes are target-language independent. In contrast, the *Language-Specific Code Generator* is built specifically for each supported target language, allowing additional control on the switch architecture in which the program will execute.

Figure 4.1: NESOI compilation processing pipeline



The compilation process can be seen as a series of phases, as was discussed in Section 2.4. Our compiler follows the phases of processing close to the typical form, as shown in Figure 4.2, aside from the code optimization phases as of this initial version.

Figure 4.2: NESOI compilation phases



4.2.1 Frontend Process

The *Frontend* is responsible for validating the source code's syntax and semantics and creating an intermediate code representation to be used in forthcoming steps. This

step performs the three main phases of code analysis: lexical, syntax, and semantic analysis, used for validating the program's meaning and checking if it is well-constructed.

The lexical analysis phase creates tokens from meaningful sequences of characters, called lexemes, which will then be used in the syntax analysis to generate the syntax tree representation of the program. It is also the lexical analysis's responsibility to identify syntactically ill-formed programs, inform the user of the error, and indicate its location.

The semantic analysis performs an analysis of the syntax tree, validating the semantic consistency of a program. One of the primary analyses performed is type checking, where the compiler verifies that each operator's operand matches. In case of inconsistency, this phase also informs the user's issue, indicating the error's location and a message describing it.

This project focus on creating a network model to execute as part of the network infrastructure. The frontend has a specialized module to handle the ANN definition. After the code analysis phases are finished, the network module extracts all network-related code definitions and creates a representation of the network as relationships between nodes. This intermediate network representation will be used during the generation of the intermediate code to determine how many switches will be required, how they should connect to each other, and which type of node they represent in the network.

The network module utilizes a configuration file to specify each ANN's connectivity behavior. The configuration file informs the network module how a layer (or node) should connect to the nodes that already are part of the network. In order for this to be done, the compiler implements a small set of functions to handle the expected cases of connection (*e.g.* a "Dense" layer connecting to another in a Neural Network). This module searches the program for any network model and utilizes the configuration files to identify the model and which methods were applied on the source files.

The frontend process generates a High-Level Intermediate Representation (HLIR) of the source program as output. It is a high-level representation due to its high similarity to the source-language program. The HLIR representation does not adopt any common intermediate representation (*e.g.* three address code representation), and it was built according to the project's needs.

Due to its high flexibility, the HLIR is expressed as a JSON (JavaScript Object Notation) object. It holds a list of objects containing all the information acquired from the source files required to generate the target program. For example, ANN node connectivity, the relative position of nodes on the ANN model, the activation function defined for the

ANN, the parsing structure, and other program-specific definitions.

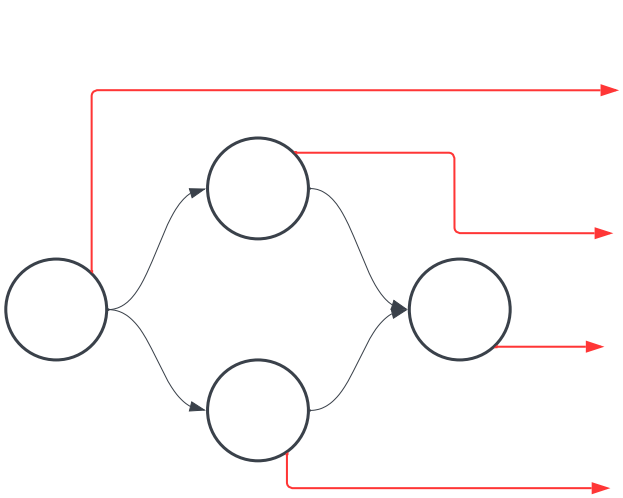
4.2.2 Backend Process

The *Backend* step is responsible for preparing the input file for executing the Language-Specific Code Generator. It receives three input files: the HLIR generated in the *Frontend* step, a JSON file representing the network topology, and a JSON file to assist the assignment of a program to its corresponding switch.

This process generates n files, where n is the number of nodes in the ANN specified on the source program. Each file contains the information of the source program, its network-related model data, and the corresponding switch information gathered from the topology and assignment files.

The assignment of a neuron to a switch is done by specifying the connection between a generated program to a switch on the topology file. This process is demonstrated in Figure 4.3, exemplifying the operation for a Neural Network of four nodes: one input, two nodes on the first hidden layer, and one output. The backend reads every program in order of layer and its relative position on the layer and from the assignment file, sets which switch will run the neuron.

Figure 4.3: Assignment of Neural Network's nodes to switches based on topology file.



```
"switches": [
  {
    "switch": "switch1",
    "arch": "tofino",
    "lang": "p4"
  },
  {
    "switch": "switch2",
    "arch": "tofino",
    "lang": "p4"
  },
  {
    "switch": "switch3",
    "arch": "trident-4",
    "lang": "npl"
  },
  {
    "switch": "switch4",
    "arch": "tofino",
    "lang": "p4"
  }
]
```

4.2.3 Language-Specific Code Generator Process

The *Language-Specific Code Generator* is responsible for generating the target program. To provide easy extendability from the compiler's developer perspective, this phase is divided into multiple individual self-contained programs. Each program handles the translation from the intermediate representation to a target language, as shown in Figure 4.2. For the first version of the transpiler, only P4 and NPL will be supported.

To enhance control on the translation process, each program may use the switch architecture to specialize the target program output and handle any misused feature on a specific language/architecture. In each target program, the developer can specialize a class definition and create a new object that will be used to override general functions and to specify the expected behavior for each allowed architecture. This enables a finer control of a program translation by generating the code accordingly to the switch's architecture, but may also help to trigger errors in case a code section cannot be converted into an architecture structure. Although not developed in this project, the switch architecture may be used to generate machine-dependent optimizations on the target program.

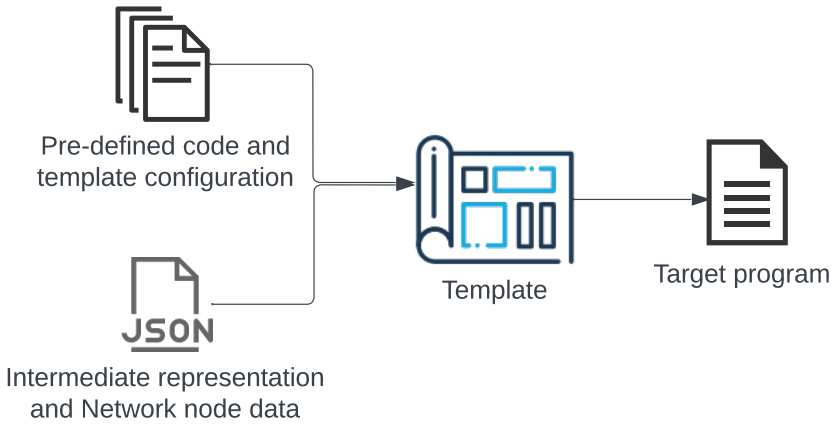
As input data, this process receives the intermediate representation and the ANN information collected from the frontend step and a series of templates and pre-defined code samples that will be used to assemble the target program. While the code generation phase in a typical compiler translates the source program to a semantically equivalent target program, this transpiler generates several programs built on top of templates, using the source program definitions and the other resources the user provides.

A template is a file in the target language containing sections where the transpiler may insert code. These available sections receive code from two different sources: the intermediate representation and the configuration files. The code that is already placed on the template will be shared among all generated programs in one layer.

Figure 4.4 shows this process of building a program based on a template. When injecting code to the destined section, the program translates the intermediate representation to the target language or directly inserts the code from the configuration.

The code generator step searches the template for any available section and then adds the corresponding code. The transpiler takes various data sources to retrieve the right segment to select the correct code segment to add to a template. For example, target language, ANN type, relative position in the ANN model, and others. In this version, only three possible relative positions are considered in a Network model: input node,

Figure 4.4: Building the target program based on a template



hidden-layer node, and output node.

Placeholders in the template file indicate a section that new data will replace. A placeholder is a string known by the compiler or defined by the user. There is a collection of pre-defined placeholders with known behavior to the compiler, but users may add as many placeholders as they need.

For example, there is a pre-defined placeholder named *__HEADERS__*. This placeholder indicates that its position on the template will receive the headers and structs defined in the source program. There are also pre-defined placeholders for the parser, deparser, and activation functions.

On the user's side, they create the placeholders. One example of a placeholder name is *__P4_TOFINO_MYEGRESS__*, which might indicate that it will receive the code for P4 language, architecture Tofino, and that the code section is placed on the Egress function. As the user creates this placeholder, they also need to specify the code segment that the compiler will use to insert into the template. In this case, the configuration file should be placed in the correct directory (as the compiler builds the path to access this data), following the language-architecture-layer folder organization.

Finally, the code generator step is also prepared to generate auxiliary files for the target program execution on a switch. For example, we generate a *runtime.json* file for each P4 program. These files can set the control plane rules based on match-action tables. Although these files are not being largely used as of now, they may be required for setting the values for connections between neurons in the future.

5 LANGUAGE SPECIFICATION

The language for this project was highly inspired by the syntax of the P4 language. Its resemblance to P4 makes it an easy transaction for those who already program in that language. This language also has similarities to languages such as Java or C# to declare classes, and methods and use them in an Object-Oriented way. In this chapter we introduce the basic data types to develop in this language, as well as how to define statements for classes and parsers.

5.1 Data Types

The developed language is statically-typed, which means that all variables must be declared with their type and name before they can be used. It provides several base types and type operators that construct derived types. There has been no definition of default values for this initial version, either for the base types or for derived types, so each variable or struct is expected to be correctly initialized.

5.1.1 Base Types and Derived Types

The language supports three build-in types. These types are the base for the language, and can be used to define derived types such as headers and structs.

Bit-string: Inspired by P4, the bit-string type (**`bit<>`**) is an unsigned integer with arbitrary width, expressed in bits. A bit-string of width **`W`** is declared as: **`bit<W>`**, where the width **`W`** must be known at compile-time and be a non-negative integer. Bit-strings with width 0 are allowed.

Boolean: This type contains just two possible values: **`true`** and **`false`**.

Integer: The Integer type holds constant integer values. By default, integer values are represented in base 10, but may also hold hexadecimal values (with prefix `0x`).

Derived types, in turn, are compositions of base types and other derived types. Three types of constructors can be used to derive additional types: **`header`**, **`struct`**, and **`parser`**. The types **`header`**, **`struct`**, and **`parser`** can only be used in type declarations where a new name for the type is introduced. This identifier can then be used

to refer to the type.

`Header`: The declaration of a **`header`** type is shown in Appendix-A. Each member from a header is restricted to be of type bit-string. Unlike P4, a header does not include a **`boolean validity`** field, nor any methods to manipulate this field. Headers are allowed to be declared empty (with no members).

`Struct`: The declaration of a **`header`** type is shown in Appendix-A. Each member from a **`struct`** is restricted to be of type bit-string, **`header`**, or other **`struct`** types. Structs are allowed to be declared empty (with no members).

5.2 Typedef

A **`typedef`** declaration can be used to give an alternative name to a type. Note that it does not create a new type, it works as an alias for a type that is already defined. The typedef notation is shown in Appendix-A.

5.3 Parser

A **`parser`** definition creates a parsing state for processing packets. A parsing state contains its identification token (name), a set of parameters, its relative states of parsing (defining the flow of processing), and its inner processing states. The grammar for parsers is shown in Appendix-A.

A **`parser`** parameter can be of two types: **`header`** or **`struct`**. Each parameter can also describe its *direction*. There are three types of direction: **`in`**, **`out`**, or **`inout`**. The direction information is used as an annotation to our language. Still, its value is passed on to the target language, guaranteeing that they follow their correct policies.

As the parser is built as a state machine (as will be described in Section 6.1) the user can specify which states may come before a specific parsing state.

The body of a parser contains inner states of validation that can be found. It is required that each parser includes at least one inner state named **`start`** . Inside each state, packets can be extracted and/or transit to another inner state or one of the possible outcomes of a parser: **`accept`** or **`deny`**. Figure 5.1 shows an example of parser definition to validate IPV4 (parsing through Ethernet parser initially).

Figure 5.1: Example of parser definition for IPV4 and Ethernet headers.

```
parser(ethernet, headers out hdr, metadata inout meta) {
    start {
      extract(hdr.ethernet);
      transition accept;
    }
}

parser(ipv4, headers out hdr, metadata inout meta)
          : ethernet (hdr.ethernet.etherType) {
  start {
    transition(super) {
      0x800: parse_ipv4;
      default: accept;
    }
  }
  parse_ipv4 {
    extract(hdr.ipv4);
    transition accept;
  }
}
```

5.4 Classes

The **`class`** type is used to define objects in the language. Inside the class definition, a constructor and methods to work with the class object can also be defined. Class methods can be overloaded, meaning a class can have multiple methods with the same name but different function signatures. The grammar for classes and its methods are shown in Appendix-A. Figure 5.2 shows an example of how classes are defined in our language. A class definition may declare its methods inside the class body, but there is currently no way of specifying their behavior on our language.

Currently, the **`class`** definition is used as a *placeholder* definition. This means that its definition holds no value to the program other than defining the classes and methods the user can use. The primary use of classes and its methods is to give the user the ability to specify an ANN model that will be later translated as a distributed application.

The creation of new classes may be done by the user, but as they can't specify any functionality by themselves, it holds no actual value to the user right away. A further explanation of how the compiler uses classes is described in Section 4.2.1. The classes files should be placed alongside the NESOI script, to better organize the framework resources.

5.5 Comments

The language supports two kinds of comments: single and multi-line comments. The double forward slash // introduces the single-line comment and spans to the end of

Figure 5.2: Base class definition to build Neural Network.

```
class Dense {
   func Dense(int)
}

class NeuralNetwork {
   func NeuralNetwork(ActivationFunction)
   func input(int)
   func add(Dense)
}
```

the line. The multi-line comment is enclosed by the characters $/*$ and $*/$.

Note that comments are treated as token separators and are not allowed within a token. For example, $pars/**/er$ is parsed as two tokens, $pars$ and er, and not the single token $parser$.

5.6 Scope

As of this initial version, the program's scope is of function and global scope. A function scope binds the declared variables and parameters only inside the scope of a function. It is important to point out that currently, the language does not directly support the implementation of functions on the program, so this kind of scope is used mainly on the parser declaration and is considered while defining methods inside a class.

The global scope binds a name to the entire program scope. This is usually seen as a bad practice, where a term used for a value cannot be used in another file, for example. In the context of this initial version, this is a simple issue to resolve for the developer. A better approach would be using a file or module scope, but this is one of the improvements that will be handled in future versions of the language.

6 DISCUSSION

This section presents a more in-depth discussion of some of the transpiler's elements. We detail how the parser works and how it builds a state machine parsing representation in our language and describe how it can be used in our framework. We also describe in more detail the templating process by giving a more in-depth explanation of how the user can add their own programs and configure them based on placeholders and how to set the code snippets correctly on the system.

6.1 Parsers

The parser describes the allowed sequence of headers received from packets and which headers and fields should be extracted from them. Parsers are defined as a state machine, where each state validates its correspondent header and may transition to another state or finish the parsing with one of two possible outcomes: **`accept`** (indicating successful parsing) or **`deny`** (indicating a parsing failure).

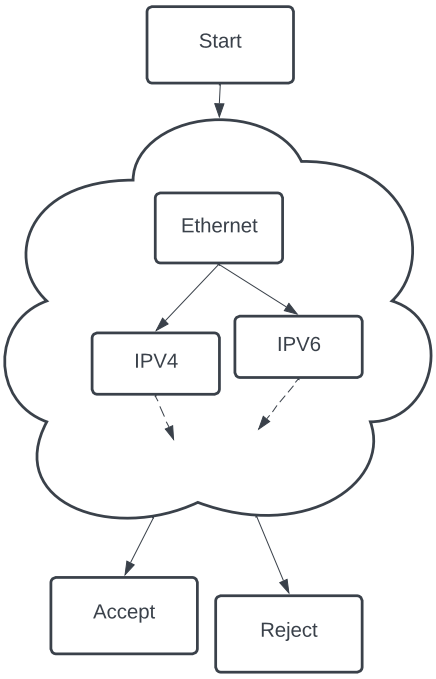
The state machine definition allows the program to build a tree-like structure of the parsing flow. Users can define the parser structure, its headers, and extracting rules and utilize these definitions in every target program. Figure 6.1 illustrates the general structure of a parser state machine, where the parser definition is shown inside of the cloud, which will be wrapped by the **`start`** state on the code generation process, and the two possible resulting states of **`accept`** and **`reject`**.

The main advantage of this design is that it allows the user to specify at which state the program should stop evaluating a packet for each layer. This can be achieved by specifying the state on the configuration file for each layer, and the compiler will translate the parser accordingly. This becomes especially helpful in a situation in which an ANN evaluates a packet down to IPV4 in the input layer but adds a new header in the packet to control the state in the following layers in the network, for example.

The user specifies the state machine during development by expressing the parser as relationships between parsing states. The relationships are expressed by indicating the possible previous states that can lead to a given state. This specification format allows each state to be defined as a finishing step in the parsing, simplifying the parser's translation to the target language.

As the state machine is assembled bottom-up, one possible issue for the compiler

Figure 6.1: Parsing state machine structure.



is not knowing how to create a state with the correct validation of packet headers. In our language, the user must specify the headers to be evaluated from each of its previous states. Consider the Transmission Control Protocol (TCP), which comes after IPV4 or IPV6 headers. In both cases, we need to specify which header should be evaluated from IPV4 and IPV6 to transit to the TCP parsing state. Figure 6.2 shows a code snippet to exemplify this situation.

In this example, we are showing the parser states for Ethernet, IPV4 and TCP, and we can see that there is no direct link between IPV4 to TCP, even less describing how a translation between one another should be constructed. The parser for IPV6 has been left aside as it would look very similar to IPV4. When a configuration file specifies that one layer will use the TCP parser state, it sets a connection on the state machine between those states. When the translation process generates the parsing code, a new parsing state is created based on the IPV4 state. Then, it evaluates the headers indicated to be required to transit to the TCP state and performs the transition.

The state machine created for Figure 6.2 is shown in the top half of Figure 6.3. By having this state machine organization, the user can specify which state it will set for each layer of the network. In this case, there are four options (Ethernet, IPV4, IPV6, or TCP), which are represented in the bottom half of the same image.

Figure 6.2: Example of parser states for IPV4 and TCP.

```
parser(ethernet, headers out hdr, metadata inout meta) {
  start {
    extract(hdr.ethernet);
    transition accept;
  }
}

parser(ipv4, headers out hdr, metadata inout meta) : ethernet (hdr.ethernet.etherType) {
  start {
    transition(super) {
      TYPE_IPV4: parse_ipv4;
      default: accept;
    }
  }
  parse_ipv4 {
    extract(hdr.ipv4);
    transition accept;
  }
}

parser(tcp, headers out hdr, metadata inout meta) :
       ipv4 (hdr.ipv4.protocol), ipv6 (hdr.ipv6.protocol) {
  start {
    transition(super) {
      TYPE_TCP: parse_tcp;
      default: accept;
    }
  }
  parse_tcp {
    extract(hdr.tcp);
    transition accept;
  }
}
```

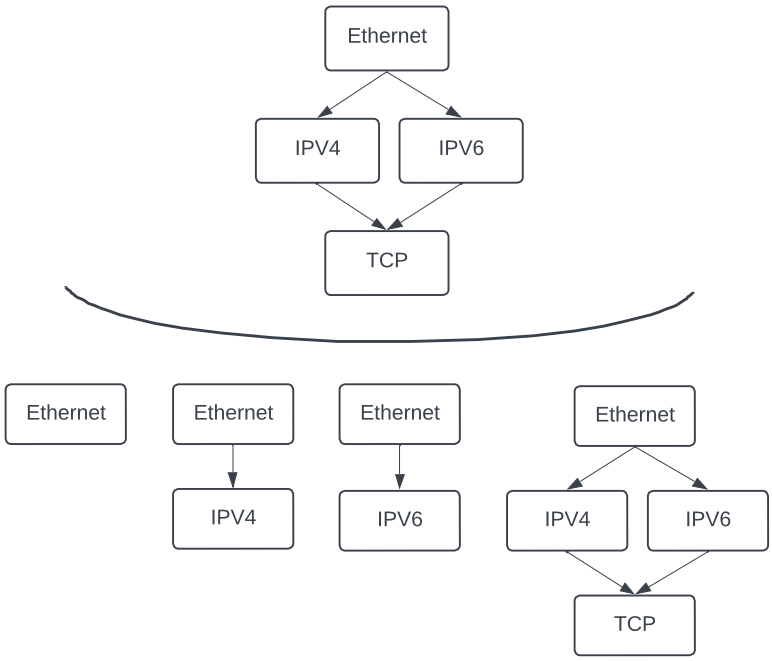
Another possible issue is when a state transitions to multiple states. The Ethernet protocol, for example, may transition to two different states, IPV4 or IPV6. To handle this situation, the translation process generates a helper state that handles this multiple state-transition. This new state creates a transition statement (similar to transition-select in P4) that uses the specified header (on the relationship from the IPV4 and IPV6 states to the Ethernet) to check its value and transit to the correct state. In this scenario, the specified headers in IPV4 and IPV6 must be equal.

6.2 Templating

As it was described in section 4.2.3, the transpiler is highly based on the usage of templates. A template is a scheme used to assist the code generation process. They are scaffolded directly on the target languages required and have sections indicating where the compiler should supply code in the translation process. It also works as a way to define shared code for every target program.

On the template, a *section* is defined by using a placeholder string. This helps

Figure 6.3: State machine arrangement for Figure 6.2 and its potential sub-compositions to be used on each layer.



to distinguish a placeholder from the rest of the program and to manage them in the template file. The recommended notation for the user's defined placeholders is using the "__" (*double underscore*) as a prefix and suffix for the string and giving it a meaningful name based on the target code, architecture, and code section.

Aside from the default placeholders provided by the transpiler, the user can define and handle as many placeholders as they need. The placeholders should be informed to the compiler by the configuration file.

The Language-Specific Code Generator process consumes an input file providing the folder location where the user's placeholder definitions and other general data for the compiler resides. This file uses the TOML[1] (Tom's Obvious Minimal Language) file format due to its user-friendly syntax.

There are four sub-folders in this provided location. Each sub-folder has a specific purpose for the compiler. There are the following:

- `auxiliary_data/`: This folder contains data for the *frontend* step. Currently, it holds the configuration on how to generate an ANN representation of a network as neurons and layers relationships, based on a set of predefined functions in the network module, as discussed in Section 4.2.1 (from the classes definitions),

[1]TOML website: <https://toml.io/en/>

- `implementation/`: This folder contains the placeholders defined by the user. They are represented in a JSON file where the key is the placeholder string, and the value can be a simple string to be replaced on the template or a path to a file, which might contain a longer text to be replaced (*e.g.*, code snippet). This is also where the user indicates some other configuration for each layer. For example, the parser state that the switch should evaluate for and up to which header it should emit on the deparsing step (considering a P4 target program),
- `target/`: This folder contains pre-defined code in each supported target language for the implementation of the activation functions. This is primarily configured by the compiler developer, but can be extended by the user,
- `templates/`: This folder provides the templates that will be used for each target language and its supported architecture. Each template is build in the target program language and is handled by the user.

The `implementation/` and `templates/` folders have their unique sub-folder structure to represent specialization of information. The `implementation/` sub-folder nesting have the following structure:

```
implementation/
└─ network model/
   └─ target language/
      └─ target language architecture/
```

The `templates/` folder follow a similar structure, but it disregards the `network model` folder nesting step.

This folder structure gives the user greater control of the value to be replaced in the template. The key-value inputs are placed on a JSON file for each network layer representation. In a situation where the same placeholder value may be used on more than one target language architecture (as of the same layer), the configuration file can be placed directly in the target language folder. This shares the value among all architectures and removes the need to duplicate values on separate folders.

7 EXPERIMENTAL EVALUATION

We built NESOI using C++. It utilized Flex e Bison (LEVINE, 2009) to handle the tokenization and parsing of the grammar. It also used external open-source libraries such as *JSON for Modern C++*[1] to handle JSON objects, *toml++*[2] to handle TOML objects, *Lyra*[3] for line argument parsing, and *Spdlog*[4] for logging. By the end of this initial implementation, the source code base of NESOI contained more than 3,800 LOC.

The primary focus of our evaluation of NESOI are the framework requirements we presented in Section 4.1. These requirements describe the expected functionalities for the compiler and the auxiliary programming language. To recapitulate, in Section 4.2.3, we presented how the compiler generates code for each switch and how it utilizes the architecture for this process. In Section 4.2.2, we indicated how the compiler distributes each node of the ANN to a switch in the network based on a topology file. We also described how to extend the library using templates in Section 6.2, and how the parsers are defined and set for the target program in Section 6.1.

To assess the potentialities of NESOI as a framework for building large-scale distributed intelligent applications for the forwarding plane, we consider two metrics in our evaluation: (i) a comparison between the number of lines of code (LOCs) required to develop a distributed program using NESOI, compared to an approximation of developing the distributed application on the target languages by hand, and (ii) the compilation time to generate the distributed program using NESOI.

To evaluate the number of LOCs, we created 9 test cases. Each test contains several neurons following the powers of two series (2^n), starting with n equals 2. To build the neural network, we use one input neuron, one output neuron, and we create *n* hidden-layers based on the log2 of the number of neurons ($log_2(neurons)$), and we distribute the remaining neurons between the hidden-layers roughly by the same amount.

Table 7.1 presents the LOCs for both situations. We count all lines used to specify the neural network, parser, and required headers and structs on the source files. The predefined templates and code implementation, topology layout, and assignment scheme are not considered lines of code.

We count all lines in the generated code for the target files to represent the approx-

[1] Json for C++ website: <https://json.nlohmann.me/>
[2] TOML for C++ website: <https://marzer.github.io/tomlplusplus/>
[3] Lyra for C++ website: <https://www.bfgroup.xyz/Lyra/>
[4] Spdlog for C++ website: <https://github.com/gabime/spdlog>

Figure 7.1: Example of describing a Neural Network instance in NESOI

```
NeuralNetwork nn = NeuralNetwork(relu)
nn.input(1)
nn.add(Dense(2))
nn.add(Dense(1))
```

imate workload required to develop each file by the user. To approximate the number of LOCs for the target program, we utilized a sample distributed application that we turned into a template.

Table 7.1: Comparison between LOCs between NESOI files and the target program files.

Neurons	Hidden Layers	LOCs NESOI	Approx. LOCs Target
4	2	84	1,000
8	3	85	2,000
16	4	86	4,000
32	5	87	8,000
64	6	88	16,000
128	7	89	31,800
256	8	90	63,800
512	9	91	127,500
1,024	10	92	255,000

We can see that in NESOI, the number of LOCs increase according to the number of hidden layers. This happens because NESOI tries to provide a simple way to define ANNs. A code example of a neural network specification for the first test case is shown in Figure 7.1. In the following tests, we adapt this code sample adding new layers to the ANN model and adjusting the number of neurons in each layer.

For the target programs analysis, the number of LOCs increases rapidly according to the number of neurons in the network. This happens because the compiler generates one file for its correspondent switch in the network.

We also show the execution time to compile and generate the target programs in Table 7.2. It shows an average of 10 executions for each test case from Table 7.1, running on a six-core Intel i7-10850H with 16GB RAM, in a Windows Subsystem for Linux (WSL) operating Ubuntu 20.04 distribution. Note that the compilation time grows according to the number of neurons it needs to generate. Each Language-Specific Code Generator is triggered to run simultaneously, which speeds up the process when we run a few neurons but consumes a lot of computational resources when there is a high number of neurons, which we found to be the bottleneck of execution.

It is important to mention that the process of developing code by hand for switches can be a complex task, one in which the user might need to structure programs using mul-

Table 7.2: Compilation time of NESOI based on the number of neurons and layers

Neurons	Hidden Layers	Compilation Time
4	2	0m0.281s
8	3	0m0.354s
16	4	0m0.555s
32	5	0m1.125s
64	6	0m2.486s
128	7	0m7.489s
256	8	0m26.978s
512	9	1m35.762s
1,024	10	6m18.752s

tiple target languages and switch architecture in the same networking, not to mention the need to develop such programs as a distributed intelligent application in the networking infrastructure. Such issues may have an impact on the *cognitive load* for the users. Helgesson et al. (2019) identified that there are direct aspects of the tools used in development that cause cognitive load, for example, using a tool outside its intended purpose, or being unintuitive or lacking functionalities.

8 FINAL CONSIDERATIONS

This work proposed a compiler and an auxiliary programming language, NESOI, to specify an intelligent program to run on multiple network devices. We developed a pipeline for processing a high-level specification of an artificial neural network program that generates code for switches implementing distributed logic for intelligent applications to run on the network. Previous works approaching the development of intelligent applications in-network have proven that this class of programs can be constructed and executed in the network infrastructure. However, they lack a developer kit to facilitate the development of these programs. Nonetheless, works of code generation for the data plane are also promising, specially Lyra (GAO et al., 2020a), which generates code for both P4 and NPL, but there is still no support for building distributed intelligent applications on the data plane environment. This work combines those two features into the same program. Using templates, it generates distributed applications based on a neural network description and generates code to be executed on network switches.

NESOI's first version only has support for the Neural Network model. The language is minimal and has no means for defining the activation function, aggregation function or normalization function directly on the specified program. The network model input is currently limited to the headers and structs defined on the program. The communication between neurons has not been implemented yet.

It is important to emphasize that this work represents a first attempt to develop intelligent applications for the data plane. The proposed compiler and auxiliary programming language, NESOI, enabled us to demonstrate that building intelligent applications is feasible and can be achieved with small programs. As of now, there is no complete template for a working neural network available on NESOI, and even though the generated code passed on syntax analysis for both P4 and NPL, the generated programs are not yet ready to be executed due to their current limitation.

For future work, we plan on expanding the grammar of NESOI, making it possible to develop the activation function and other useful helper functions for ANNs, such as aggregation and normalization functions directly on the source program. Extend the supported target languages, complete the auxiliary file(s) that are generated alongside every program file (e.g. runtime in P4), add support for other ANNs (*e.g.* Graph Neural Network). We will also be working on the communication between neurons, and enabling a file with pre-trained data as input for the framework, so we can set the weights and bias

values for the neurons' connections of an ANN based on a pre-trained model.

REFERENCES

AHO, A. V. et al. **Compilers: principles, techniques, & tools**. [S.l.]: Pearson Education India, 2007.

ARASHLOO, M. T. et al. Snap: Stateful network-wide abstractions for packet processing. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.: s.n.], 2016. p. 29–43.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE transactions on dependable and secure computing**, IEEE, v. 1, n. 1, p. 11–33, 2004.

BISHOP, C. Neural networks and their applications. **Review of Scientific Instruments**, v. 65, p. 1803 – 1832, 07 1994.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 44, n. 3, p. 87–95, 2014.

BOSSHART, P. et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 43, n. 4, p. 99–110, 2013.

CAPROLU, M.; RAPONI, S.; PIETRO, R. D. Fortress: an efficient and distributed firewall for stateful data plane sdn. **Security and Communication Networks**, Hindawi, v. 2019, 2019.

CASE, J. D. et al. **Simple network management protocol (SNMP)**. [S.l.], 1989.

CORDEIRO, W. L. da C.; MARQUES, J. A.; GASPARY, L. P. Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management. **Journal of Network and Systems Management**, Springer, v. 25, n. 4, p. 784–818, 2017.

ENNS, R. **NETCONF configuration protocol**. [S.l.], 2006.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: An intellectual history of programmable networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 2, p. 87–98, apr 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2602204.2602219>.

FIROUZI, R.; RAHMANI, R. A distributed sdn controller for distributed iot. **IEEE Access**, v. 10, p. 42873–42882, 2022.

GAO, J. et al. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 435–450.

GAO, X. et al. Switch code generation using program synthesis. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 44–61.

GARDNER, M. W.; DORLING, S. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. **Atmospheric environment**, Elsevier, v. 32, n. 14-15, p. 2627–2636, 1998.

GOBATTO, L. et al. Improving content-aware video streaming in congested networks with in-network computing. **arXiv preprint arXiv:2202.04703**, 2022.

HALEPLIDIS, E. et al. **Software-defined networking (SDN): Layers and architecture terminology**. [S.l.], 2015.

HELGESSON, D. et al. Cognitive load drivers in large scale software development. In: IEEE. **2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)**. [S.l.], 2019. p. 91–94.

HORNIK, K.; STINCHCOMBE, M.; WHITE, H. Multilayer feedforward networks are universal approximators. **Neural networks**, Elsevier, v. 2, n. 5, p. 359–366, 1989.

JAIN, A. K.; MAO, J.; MOHIUDDIN, K. M. Artificial neural networks: A tutorial. **Computer**, IEEE, v. 29, n. 3, p. 31–44, 1996.

KATTA, N. et al. Hula: Scalable load balancing using programmable data planes. In: **Proceedings of the Symposium on SDN Research**. [S.l.: s.n.], 2016. p. 1–12.

KIM, H.; FEAMSTER, N. Improving network management with software defined networking. **IEEE Communications Magazine**, v. 51, n. 2, p. 114–119, 2013.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, Ieee, v. 103, n. 1, p. 14–76, 2014.

KRONGBARAMEE, P.; SOMCHIT, Y. Implementation of sdn stateful firewall on data plane using open vswitch. In: IEEE. **2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)**. [S.l.], 2018. p. 1–5.

LEVINE, J. **Flex & Bison: Text Processing Tools**. [S.l.]: " O'Reilly Media, Inc.", 2009.

LI, Y. et al. Accelerating distributed reinforcement learning with in-switch computing. In: IEEE. **2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.], 2019. p. 279–291.

LIN, Y.-D. et al. An extended sdn architecture for network function virtualization with a case study on intrusion prevention. **IEEE Network**, IEEE, v. 29, n. 3, p. 48–53, 2015.

LUIZELLI, M. C. et al. In-network neural networks: challenges and opportunities for innovation. **IEEE Network**, IEEE, v. 35, n. 6, p. 68–74, 2021.

LUIZELLI, M. C. et al. In-network neural networks: challenges and opportunities for innovation. **IEEE Network**, IEEE, v. 35, n. 6, p. 68–74, 2021.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 38, n. 2, p. 69–74, 2008.

NKOSI, M. C.; LYSKO, A. A.; DLAMINI, S. Multi-path load balancing for sdn data plane. In: IEEE. **2018 International Conference on Intelligent and Innovative Computing Applications (ICONIC)**. [S.l.], 2018. p. 1–6.

PIZZUTTI, M.; SCHAEFFER-FILHO, A. E. Adaptive multipath routing based on hybrid data and control plane operation. In: IEEE. **IEEE INFOCOM 2019-IEEE Conference on Computer Communications**. [S.l.], 2019. p. 730–738.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. **Learning internal representations by error propagation**. [S.l.], 1985.

SANVITO, D.; SIRACUSANO, G.; BIFULCO, R. Can the network be the ai accelerator? In: **Proceedings of the 2018 Morning Workshop on In-Network Computing**. [S.l.: s.n.], 2018. p. 20–25.

SAPIO, A. et al. In-network computation is a dumb idea whose time has come. In: **Proceedings of the 16th ACM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2017. p. 150–156.

SAPIO, A. et al. Scaling distributed machine learning with {In-Network} aggregation. In: **18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)**. [S.l.: s.n.], 2021. p. 785–808.

SAQUETTI, M. et al. Toward in-network intelligence: running distributed artificial neural networks in the data plane. **IEEE Communications Letters**, IEEE, v. 25, n. 11, p. 3551–3555, 2021.

SHAGHAGHI, A.; KAAFAR, M. A.; JHA, S. Wedgetail: An intrusion prevention system for the data plane of software defined networks. In: **Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security**. [S.l.: s.n.], 2017. p. 849–861.

SIRACUSANO, G.; BIFULCO, R. In-network neural networks. **arXiv preprint arXiv:1801.05731**, 2018.

SIVARAMAN, A. et al. Packet transactions: High-level programming for line-rate switches. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.: s.n.], 2016. p. 15–28.

SONCHACK, J. et al. Lucid: A language for control in the data plane. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. [S.l.: s.n.], 2021. p. 731–747.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: **Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking**. [S.l.: s.n.], 2013. p. 127–132.

SWAMY, T. et al. Taurus: a data plane architecture for per-packet ml. In: **Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems**. [S.l.: s.n.], 2022. p. 1099–1114.

TURNER, J. S.; TAYLOR, D. E. Diversifying the internet. In: IEEE. **GLOBECOM'05. IEEE Global Telecommunications Conference, 2005.** [S.l.], 2005. v. 2, p. 6–pp.

WU, D. et al. Accelerated service chaining on a single switch asic. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2019. p. 141–149.

XIONG, Z.; ZILBERMAN, N. Do switches dream of machine learning? toward in-network classification. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2019. (HotNets '19), p. 25–33. ISBN 9781450370202. Available from Internet: <https://doi.org/10.1145/3365609.3365864>.

APPENDIX A — LANGUAGE GRAMMAR

```
members_list ::= member[] | null
member ::= vType id;
header ::= HEADER id { members_list }
struct ::= STRUCT id { members_list }
typedef ::= TYPEDEF vType id;
class_def ::= id { class_in_def }
class_in_def ::= function_def class_in_def | null
function_def ::= FUNC id ( parameters_list )
parameters_list ::= parameter[] | null
parameter ::= vType

parser ::= PARSER ( id parser_param_list )
   ↪ parser_dependency { parser_steps_list }
parser_dependency ::= dependency_id[] | null
parser_param_list ::= parser_param[] | null
parser_param ::= vTypeid direction id
direction ::= in | out | inout | null

parser_steps_list ::= parser_step
   | parser_step parser_steps_list
parser_step ::= id { packet_extraction transition }
packet_extraction ::= EXTRACT ( expression ) ; | null
transition ::= TRANSITION tr_state_or_select
tr_state_or_select ::= tr_state ; | transition_select
transition_select ::= ( expression ) { tr_select_case_lst }
   | ( TK_SUPER ) { tr_select_case_lst }
tr_select_case_lst ::= select_case tr_select_case_lst
   | select_case tr_default_case
select_case ::= id : tr_state ;
   | hex : tr_state ;
tr_default_case ::= default : tr_state; | null
tr_state ::= accept | deny | id
```

```
dependency_id ::= id opt_elemAccessDep;
opt_elemAccessDep ::= ( expression ) | null
```