Program Synthesis for Programmable Data Planes

Generating P4 code by input-output examples

Timotei-Cornelis Jugariu

TU Delft

Program Synthesis for Programmable Data Planes

Generating P4 code by input-output examples

Thesis report

by

Timotei-Cornelis Jugariu

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on the 22nd of August 2023

Thesis committee:	Prof. dr. ir. F.A. Kuipers	Responsible Professor
	Dr. S. Dumančić	Associate Professor
	Chenxing Ji	Daily Supervisor
Project Duration:	August, 2022 - August, 2023	
Master programme:	Embedded Systems	
Specialization:	Software & Networking	
Student number:	4595165	

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Faculty of Electrical Engineering, Mathematics and Computer Science · Delft University of Technology

Preface

An all-important step in the ambitious pursuit towards autonomous networks has been the introduction of Software Defined Networking which has advocated the concept of separating a network's control plane from the data plane and creating a programmable controller with a wider view of the network. This innovation proved to be very promising, but the non-programmable data plane quickly became a limitation. The next step was brought by the emergence of the programmable switch architecture and the P4 language.

During the last year, I have researched ways to make another modest step in the right direction which I believe to be synthesising code for programmable data planes. This report represents the primary output of my efforts and the last stride in my time being a student at the Delft University of Technology. In spite of the unfortunate delays and the setbacks that I experienced throughout my education, I graduate as a master of my field, a versatile computer scientist, well-versed in the domain of Embedded Systems.

There are many people to thank for my accomplishments, not the least of all my responsible professor Dr. Fernando Kuipers. He has helped me find a fitting thesis subject in his area of expertise and has guided my work together with my daily supervisor Chenxing Ji, often called Gabe. I have come in contact with them after taking their course on High Performance Data Networking which I have genuinely enjoyed.

The subject of my thesis lies at the intersection of two fields, namely computer networks and program synthesis. As an expert on the latter, Dr. Sebastijan Dumančić has given me valuable research ideas and feedback on my work which I am very grateful for. In fact, Dr. Dumančić has recommended me the original research on Probe which I improved upon and used for my final solution.

Ever since I was a kid I have had the ambition to study abroad at a reputable university. With the graduation date in sight, I would like to thank my uncle, Marco, for allowing me to live in his house for so many years and for helping me so often that it is difficult to count. I would also like to thank my cousin for always bringing my favorite food and my friends and brothers for keeping me company. Last but not least, I am very grateful for my parents, Daniel and Emma, whose support throughout this endeavour has been unconditional and without limits.

Timo Jugariu
Delft, June 2023

Contents

List of Figures vi

1 Introduction 1
- 1.1 Problem Definition . . . 1
- 1.2 Research Question . . . 2
- 1.3 Thesis outline . . . 3

2 Background and Context 4
- 2.1 Software Defined Networking . . . 4
- 2.2 Programmable data-planes and P4 . . . 6
- 2.3 Program Synthesis . . . 7
  - 2.3.1 Programming By Examples . . . 8
  - 2.3.2 Syntax Guided Enumerative Synthesis . . . 9
- 2.4 Relevant prior research . . . 10
  - 2.4.1 Intent-based networking . . . 10
  - 2.4.2 Program Synthesis in computer networks . . . 11
  - 2.4.3 P4 code testing and validation . . . 12

3 The Probe-cpp Synthesizer 14
- 3.1 The choice for Probe . . . 14
- 3.2 Original algorithm . . . 15
  - 3.2.1 Probe's guarantees . . . 17
  - 3.2.2 Cost-based versus height-based enumeration . . . 17
  - 3.2.3 Just-in-time learning . . . 17
- 3.3 Improved implementation . . . 18
  - 3.3.1 Differences to the original algorithm . . . 18
  - 3.3.2 Architecture . . . 19
- 3.4 Evaluation . . . 23

4 Synthesizing P4 code 27
- 4.1 Architecture . . . 27
  - 4.1.1 Input-output format . . . 27
  - 4.1.2 Evaluating candidate P4 programs . . . 28
  - 4.1.3 Recomputing subtrees . . . 29
  - 4.1.4 Reconciling subtrees . . . 32
  - 4.1.5 Intermediate grammar . . . 34
  - 4.1.6 Reducing the hypothesis space . . . 36
- 4.2 User provided code snippets . . . 37
- 4.3 Enumerating the rules for the match-action tables . . . 38

5 G4BE's evaluation 40
- 5.1 Selected benchmarks . . . 40
- 5.2 Runtime performance and memory usage . . . 41
- 5.3 Observations . . . 43

6 Closure 44
- 6.1 Discussion . . . 44
- 6.2 Limitations and future work . . . 45

References 48

A Probe-cpp technical details 49

B Detailed benchmarks **51**

Nomenclature

List of Abbreviations

ANN	Artificial Neural Network
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BGP	Border Gateway Protocol
DSL	Domain Specific Language
IBN	Intent Based Networking
IDL	Intent Definition Language
IP	Internet Protocol
MLP	MultiLayer Perceptron
NF	Network Function
NLP	Natural Language Processing
OSPF	Open Shortest Path First
PBE	Programming By Examples
PCFG	Probabilistic Context-Free Grammar
PISA	Protocol Independent Switch Architecture
PSA	Portable Switch Architecture
RAW	Read-After-Write
SDN	Software Defined Networking
SMT	Satisfiability Modulo Theory
SyGuS	Syntax-Guided Synthesis
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
VN	Virtual Network
WAW	Write-After-Write

List of Figures

2.1 Illustration of simple interaction in a network supporting OpenFlow 5

3.1 Abstract syntax tree for a program that repeats a string four times 15
3.2 Observationally equivalent program for the program in figure 3.1 16
3.3 Components that read and parse the grammar and problem specification 20
3.4 The representation of a program in Probe-cpp 21
3.5 Data structures for storing the enumerated programs and their evaluations 22
3.6 The components of Probe-cpp responsible for enumerating programs 22
3.7 Runtime performance comparison for the eight benchmarks 24
3.8 Execution times for the original Probe implementation for eight benchmarks 25
3.9 Execution times for Probe-cpp for eight benchmarks 25
3.10 Average number of AST nodes of the final solution for Probe and Probe-cpp 26
3.11 Memory usage comparison between the original Probe and Probe-cpp 26

4.1 Data structures used to represent the execution of a P4 program 30
4.2 Recomputation of an example AST 31
4.3 Conceptual AST of the program in listing 4.3 32
4.4 Difference in AST brought by bottom-up enumeration paired with observational equivalence 33
4.5 AST of the code in listing 4.6 35
4.6 Example of a Write-After-Write hazard 36

5.1 Three topologies used for the basic benchmark 41
5.2 Runtime performance of G4BE in milliseconds for the various benchmarks 42
5.3 G4BE's memory usage in MB for the various benchmarks 42
5.4 Number of programs synthesized in time 43

A.1 UML diagram of Probe-cpp's architecture 50

1

Introduction

Recent years have seen a major push within the networking domain to make computer networks self-configurable, self-optimizing and self-healing in an attempt to relieve some of the burden usually carried by the network administrators. Aside from the rapid growth of devices requiring connectivity, networks also have to deal with higher demands in terms of bandwidth, latency and availability. As the costs associated with building and maintaining such networks increase, network service providers are in a constant search to make their networks more autonomous. Most notably, the field of mobile networks has seen research on this topic that dates back to the development of the Long Term Evolution standard (LTE) and it has continued to be an important area of research ever since [1].

An autonomous network is more efficient both in terms of resource utilization and cost-effectiveness. With such significant benefits, there is much interest around this topic both from the industry as well as from the academic environment. Unfortunately, one cannot make a network autonomous if most of its infrastructure is based on fixed hardware solutions, collectively called Application Specific Integrated Circuits (ASICs). Traditionally network devices were designed as ASICs by various manufacturers, and they carried out fixed functionalities which was also reflected in their names, such as switch, bridge, repeater, firewall, router, etc. For a long time, the performance benefits of using ASICs outweighed their inflexibility but they proved to be a barrier in the pursuit for programmable networks.

A significant step forward was made with the invention of the Software Defined Network accompanied by the introduction of the OpenFlow protocol [2]. The SDN concept separated the control-plane from the data plane, often referred to as the forwarding plane. The responsibilities of the control-plane were assigned to a central controller with a wide view of the network. The data plane would be relieved of many of the more complex tasks allowing it to focus on its core strength, namely forwarding packets at line rate.

With a solution found for the previous issue, the road to programmable networks was still blocked by the fixed data plane. Progress was made by Bosshart et al. [3] who designed the Reconfigurable Match Table (RMT) architecture which was later generalised to the Protocol Independent Switch Architecture (PISA) [4] and which, in turn, evolved into the Portable Switch Architecture (PSA) [5].

Accompanying these advances in chip architectures, came P4 [6], a language specifically designed for defining the behaviour of programmable network devices. Despite not being Turing complete, it is a remarkably powerful language that allows the software developer to define almost any packet-processing program or any forwarding behavior. It abstracts away from the specifics of the hardware architecture through a compiler that takes the high-level code and transforms it into a configuration specific to the target platform.

P4 brought a disruption in the interplay between network providers and hardware vendors. Previously, the network behavior was constrained by the fixed functionality of the hardware and, to make matters worse, change cycles took far too long. Empowered by P4, SDNs enable network administrators to define the network behavior in a top-down manner since both the controller plane and the forwarding plane can be adapted to fit the high-level desired behavior.

1.1. Problem Definition

Despite its many benefits, P4 brings with it an additional layer of complexity for the network administrators. As networks continue to grow in complexity, effective management becomes increasingly more challenging.

Network administrators may find themselves overwhelmed by having the task of learning a new programming language, added to their growing list of responsibilities. Moreover, fixed hardware solutions were extensively tested by their vendors. One cannot expect a network administrator to do the same amount of testing on their own P4 code.

To mitigate the aforementioned problem, researchers have proposed the concept of Intent-Based Networking which allows the developer to provide his/her intent to a system which then automatically generates a complying program. In this report, we describe a tool that is aligned with the goals of Intent-Based Networking with the distinguishing feature that the intent is given in the form of input-output examples.

We firmly believe that defining a program by means of input-output examples is highly intuitive for human beings. Furthermore, this form of program specification fits our problem domain very well as we shall explain later.

1.2. Research Question

Considering the problem described above, the main aim of this work is stated as follows:

Research Objective

Build a proof-of-concept that is capable of synthesizing P4 code for programmable switches using example traces of input-output packets.

As shall be explained in chapter 2, there are multiple ways to express a program specification and input-output examples is one such method that seems very suitable for the problem at hand. The first research question explores the correctness of our intuition.

Research Question 1

Are traces of input packets and their corresponding output packets, an effective way to specify what a P4 program should do?

The second research question is self-explanatory and directly concerns the stated objective:

Research Question 2

Given a program specification in the form of a set of input-output packets, can we synthesize a P4 program in reasonable time that, when run on a programmable switch, will comply with the specification?

Suppose a solution is found but it is a lot longer or less efficient than a solution written by a software engineer who is proficient in the P4 language. In that case, the network administrator may still refuse to deploy it in a real-world environment. The third research question aims to evaluate the solutions generated by such a P4 synthesizer.

Research Question 3

If a complying P4 program is found, how does it compare to a solution written by an expert?

The thesis resides at the juncture of two separate areas of research: computer networks and program synthesis. The field of program synthesis concerns itself with the automatic generation of programs that are provably compliant with a pre-defined specification. To this end, it often restricts itself to domain specific languages (DSL) that are conceptual or simplified in some capacity. Research question 4 tries to bridge the gap between program synthesis and real-world by using a state-of-the-art enumerative synthesizer to generate programs in a DSL that is actually used in practice. Compared to widely used programming languages such as C/C++ and Python, the P4 language is rather limited, making it an easier candidate to make such an assessment.

Research Question 4

Can the state-of-the-art enumerative program synthesizer be used for generating solutions in a DSL used in the real world?

1.3. Thesis outline

The remainder of this report is structured in five main chapters which are briefly described below.

- **Chapter 2. Background and context** provides the needed contextual information for the readers that are less familiar with the topics of SDN and/or program synthesis. The chapter also includes a brief review of some of the preliminary research that is relevant to the main topic of the thesis. Specifically, the literature review shall provide a quick dive into the novel concept of intent-based networking before describing prior attempts at P4 code generation followed up by a brief discussion on the topic of P4 testing and verification.
- **Chapter 3. The Probe-cpp synthesizer** describes the first major component of the final solution. Probe-cpp is a improved version of the Probe synthesizer, originally proposed by Barke et al. [7]. The original algorithm is discussed, followed by a detailed description of the improved implementation. The chapter ends with a performance comparison between the two implementations.
- **Chapter 4. Synthesizing P4 code** delves into all the relevant details of the main deliverable of this project. The main concept and its implementation are discussed at length, including the main challenges that arose from adapting Probe-cpp to synthesize programs in the quite complex P4 language. The final solution is entitled G4BE which is an abbreviation of **G**enerating P**4** **B**y **E**xamples.
- **Chapter 5. G4BE's evaluation** presents the approach taken to evaluate the final solution. The measurements from the evaluation are plotted and interpreted to determine G4BE's viability and performance in terms of execution time and memory usage.
- **Chapter 6. Closure** ends this report with a comprehensive discussion on the various aspects of G4BE, including an overview of its limitations and suggestions for future improvements.

2

Background and Context

Before discussing the details of our own contributions, it is important to clarify the context and provide a brief review on the relevant research that has been conducted in the past. In addition, it is important to understand what the state-of-the-art is and identify some of the research gaps that should be explored further. This chapter is divided in four sections with the first three providing basic background information on software defined networking, programmable data-planes, and program synthesis respectively. The last section takes a closer look at the more recent publications that are directly relevant to our research questions.

2.1. Software Defined Networking

It is impossible to get a precise view of the internet, but it is widely known that current networks follow the TCP/IP model which splits the responsibility of the network onto four layers: link, internet, transport, and application [8]. In spite of this logical separation of concerns, current networks do not make an explicit distinction between two types of responsibilities which have minimal overlap. Computer networks adhere to a distributed design in which network devices are responsible for both the quick forwarding of packets and the coordination & processing necessary for implementing the overall network functionality that the network operator requires. In other words, the network functionality is split among the independent network devices which are also concurrently responsible for packet forwarding.

In a common network that is tasked with routing packets from any two endpoints, network devices forward packets based on routing tables that are populated and maintained by the devices themselves using algorithms such as Open Shortest Path First (OSPF), Border Gateway Protocol (BGP), Intermediate System to Intermediate System (IS-IS) or others. These algorithms are distributed, causing significant communication overhead that is needed to ensure that all participants exchange information (such as link-state updates or distance vectors) and eventually end up with a consistent view of the network. In link-state routing algorithms such as OSPF, the link-state updates are flooded all over the network and devices accumulate all the information and apply a variant of the Dijkstra's algorithm [9] to find the shortest path to any other point. Since every device executes this shortest path algorithm locally but on slightly different input, there is quite some redundant computation. To make matters worse, changes in the topology and traffic patterns cause the network to adapt slowly which can create temporary disruptions of connectivity. This problem is especially relevant for distance vector routing protocols which suffer from the so-called counting to infinity problem [10].

Attributing routing algorithms to one of the four layers is at times a matter of debate. Some people consider such algorithms to be part of the application layer [11] while others argue that they modify the routing behavior and thus should be included in the internet layer [12]. Furthermore, different algorithms that accomplish similar goals may be attributed to different layers depending on the required protocols stack (e.g., BGP runs on top of TCP).

SDN represents an architectural shift relative to the traditional network, an approach that explicitly separates the control-plane from the data-plane. The control-plane is run on a central programmable controller, and it is in charge of all the decision making while the data-plane is simply responsible for fast packet forwarding. For instance, the controller will maintain a global view of the network and will install the forwarding rules

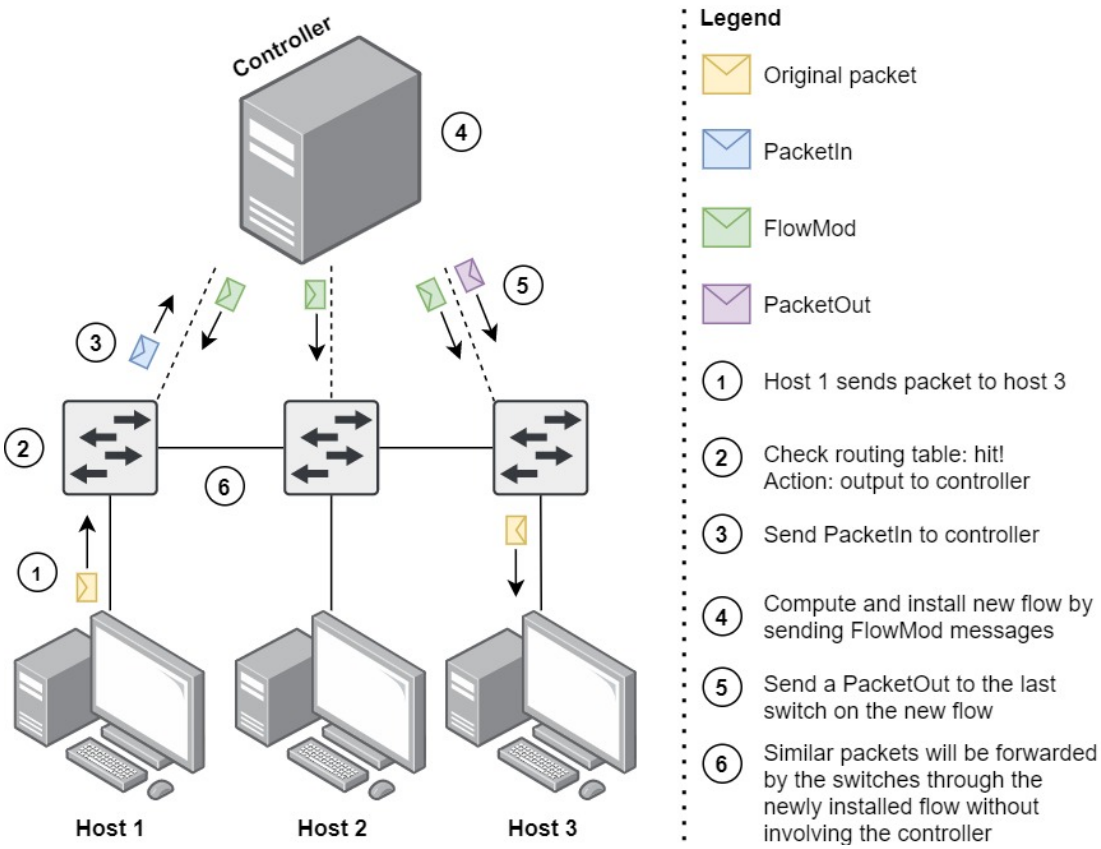


Figure 2.1: Illustration of simple interaction in a network supporting OpenFlow

in the switches (i.e., the devices forming the data-plane), removing the communication and computation overheads mentioned earlier.

With the network functionality more centralized, modifications are easier to deploy since they can be done on the controller alone unlike for the traditional networks which require most devices to be modified. Hardware interoperability also becomes less of a challenge since network devices on the data-plane do not need to communicate with each other but only with the orchestrating controller. This is in contrast to traditional networks, in which devices from different vendors are often incompatible, constraining network operators into situations known as vendor lock-in.

OpenFlow [2] is the de-facto standard protocol used to facilitate communication between the controller and each switch. At start up, switches may be configured to forward to the controller any packets that do not match any other entries from the routing table. The switch will wrap the original packet into a *PacketIn* message and send it to the controller. Based on this message, the controller may decide to install a new flow/path in the network by sending *FlowMod* messages to all switches involved. Afterwards, the controller forwards the packet back to the last switch on the installed path through a *PacketOut* message. In the future, similar packets that match the same chosen criteria are going to be forwarded through the new flow without involving the controller. This behavior is illustrated in figure 2.1 and it is just one of the many possible functionalities that one could implement with OpenFlow.

Another benefit of SDN is that it makes the task of monitoring the network traffic a lot easier. In a network that supports OpenFlow, the controller could, as an example, send a *OFPFlowStatsRequest* to one of the switches in order to receive specific flow statistics such as byte count, packet count, duration, match criteria and so on. Another example use case could be installing a flow that begins and ends in the controller and then sending out a probe packet to measure the delay along that path.

As to be expected, SDN also has some drawbacks, not the least of which is the introduction of a single point of failure. This single controller can also become a performance bottleneck for large networks leading

to reduced scalability. These issues have been researched [13] and the proposed solution is to use multiple cooperating controllers, although that comes with its own set of drawbacks. A distributed controller is more difficult to implement since it has to solve the same problems faced by any distributed system: coordination, reaching consensus, synchronization, etc.

2.2. Programmable data-planes and P4

As Bosshart et al. [6] point out, a concerning problem with OpenFlow is the ever-increasing number of header fields as a consequence of the increased requirements coming from the hardware vendors. These vendors want the OpenFlow protocol to support features that enable the capabilities of their latest switches. However, changes to the protocol are implemented slowly and must first be agreed upon by a consortium of users. Moreover, extending the protocol repeatedly is not a sustainable solution.

The programmable controller has many benefits, but the added flexibility is hampered by the rigidity of the data-plane. Switches implemented solely on hardware have a fixed functionality and although they operate only within the data-plane, they constrain the functionality that the controller can implement and by extension, the overall behavior of the network.

Instead of continuously extending OpenFlow, Bosshart et al. researched the possibility of making the data-plane programmable by first designing the Reconfigurable Match Tables architecture [3]. RMT is a flexible design for switching chips that is capable of changing the forwarding behavior without modifying the hardware and without conceding any performance or energy usage. Making forwarding devices flexible has always been possible and in fact, software implementations of switches running on top of general-purpose CPUs [14] do exist. However, these solutions have always struggled to match the throughput and latency of a switch implemented in hardware (i.e., an ASIC). This consideration is important since forwarding devices should not become a bottleneck and cause congestion by operating at a lower rate than the maximum data transfer rate of its physical interfaces, also known as line rate.

Alongside the advances in switching chip designs, the P4 language was developed that would allow network operators to program the behavior of any P4-enabled switch. The first version of P4 ($P4_{14}$) compiled to a configuration for a specific architecture (a generalization of RMT called PISA) but the second version ($P4_{16}$) was designed to target a variety of different architectures. The authors had three main objectives during the design of the P4 language: reconfigurability, protocol independence and target independence.

Reconfigurability implies that changes to the switch behavior should be possible upon request. Protocol independence ensures that P4 remains agnostic to any specific header structure. Lastly, P4 should abstract away from the hardware details of the target platform, a concept referred to as target independence. In essence, this means that the network operator can focus on implementing the desired functionality in P4, offloading the task of generating platform-specific code/configurations to the compiler.

Legacy switches had to perform many functions to accommodate every possible use case, leading to designs that were too complex for what they were ultimately used for. On the other hand, programmable switches can be programmed to contain only the required functionalities together with the minimal set of header definitions that are needed to perform the said functionalities.

Throughout this project, we only use the v1model architecture[1]. Each P4 program for the v1model consists of the following six programmable blocks in the order of their execution: parser, checksum verification, ingress processing, egress processing, checksum computation and deparser. P4 also allows to define custom data types, custom metadata structures and, most importantly, custom header fields and header structures. Having custom headers and being able to read, match and modify them as if they were standard headers, guarantees that P4 fulfills the protocol independence objective.

The parser block is essentially a state machine that is capable of interpreting the header structure of the received packet. The two checksum blocks allow the developer to control which header fields should be considered for the checksum computation (either for verifying it or updating it) and which hashing algorithm should be used. The ingress and egress blocks are the control components that are responsible for most of the packet processing as implemented by the developer. Lastly, the deparser is responsible for emitting the headers and in the right order.

[1] The v1model can be found at https://github.com/p4lang/p4c

Within the ingress and egress blocks the developer can define actions (similar to functions in other languages), match-action tables, registers, meters & counters coupled to entries in the tables, etc. These must all be defined before the *apply* block which contains the actual packet processing logic. This inner block may contain assignment instructions, conditional instructions, instructions that call actions or apply tables and so on.

Kfoury et al. [15] provides an exhaustive survey of all the applications of P4-enabled networks that have been researched in the recent years. One of the most prominent applications is In-Band Network Telemetry (INT) [16] which allows for fine-grained telemetry measurements through customer headers that are added by the switches themselves.

In-network computation is another emerging concept that proposes offloading some of the upper-layer logic from the end hosts to a network of P4 switches. This concept alone can be used for a variety of applications, ranging from reaching consensus in a distributed system [17] to machine learning [18].

The authors mention many more uses of P4-enabled switches, but the list is too long to describe thoroughly. Other notable examples included in this list are pub/sub systems without brokers, packet aggregation for Internet-of-Things (IoT) devices, heavy hitter detection, cryptography at line rate, etc.

2.3. Program Synthesis

Considered to be a major challenge within the field of Computer Science, program synthesis is a branch of Artificial Intelligence that concerns itself with the automatic generation of executable code that is provably compliant with a given high-level specification. The specification can take many forms and it is a representation of the user's intent.

One of the reasons why program synthesis is such a daunting task, is the fact that for any non-trivial language, the space of possible solutions grows exponentially with the size of the solutions. Therefore, if one wants the guarantee that a solution will be found (if one exists), one needs to perform an exhaustive search on the solution space and accept that the synthesizer will have an exponential time-complexity in the size of the smallest solution.

Another reason that makes program synthesis difficult, is brought by the ambiguity of the user's intent. High-level specifications may be too comprehensive, allowing for very different programs to be considered as valid solutions. At the same time, specifications may be incomplete, not fully grasping the intended behavior of the desired program, due to its high complexity. Writing a specification that covers all the implementation details may require the same amount of effort as writing the program itself.

Throughout the literature, there are two recurring criteria that are frequently used to categorise synthesizers: the format of the specification and the technique deployed to generate programs. Gulwani et al. [19] distinguish four main state-of-the-art techniques for program synthesis: enumerative search, constraint solving, stochastic search and deduction-based programming. Subsection 2.3.2 covers enumerative search in more detail while the others are briefly explained below.

Constraint-solving is a common technique used to find concrete values for the free variables in a formula such that the formula becomes true. In the context of program synthesis, the user's intent is expressed in terms of mathematical constraints and added to a formula together with other free variables that encode the structure of the solution. One notable synthesizer that uses this technique is SKETCH [20], a tool that takes a partial program (i.e., a sketch) written by the developer and resolves all so-called holes such that the final program satisfies some specification.

Stochastic search is a diverse category of strategies that leverage randomness to guide the search process towards regions of the solutions space that are more likely to contain programs that comply with the given specification. Often times, they offer no guarantees in terms of finding a solution, if one exists, but they are relatively fast and work surprisingly well.

Genetic programming [21] is one such example of a stochastic synthesizer which uses concepts inspired from the theory of evolution. It finds the solution to a problem by creating an initial population of random programs which evolve genetically from generation to generation to better suit the environment. The program specification is encoded into a fitness measure that quantifies how well a candidate algorithm fits the user's intent. New algorithms are formed through operations such as reproduction, crossover, and

mutation. The fitter the algorithm, the higher the chance it gets to produce offsprings or to reproduce to the next generation.

An interesting division of stochastic synthesizers is formed by algorithms that incorporate machine learning techniques in their approach. One instance of this is DeepCoder [22], a synthesizer that trains two neural networks using a large set of programs and corresponding input-output examples. After being trained, these neural networks find a probability distribution over the set of possible instructions, conditioned on the given program specification. These probabilities are used to steer the enumerative search while clever heuristics are being applied to prune away faulty and redundant candidates.

As the name suggests, deductive search makes use of deductive reasoning to transform in an iterative fashion, the given logical specification into a valid solution. The proof of correctness lies in the very manner in which the programs are constructed with every iterative step following strict deductive rules [23]. It is often paired with more formal specification formats such as logical models containing clauses that cover the desired properties.

There are many different ways to capture the user's intent into a program specification and each has an influence on the kind of synthesizers that can be used. Some of the formats used by previous researchers have been: input-output examples, models containing logical formulas, descriptions in informal natural language, sketches and impartial programs. Some methods are more formal than others and require more effort from the user. On the other hand, a format as informal as natural language adds a lot of complexity to the synthesizer as it is far more difficult to parse natural language due to its expressiveness, redundancy, and overall lack of structure. Some synthesizers allow users to articulate their intent through an intermediary language that is more restricted and structured than natural language but it is still intuitive and easy to use.

2.3.1. Programming By Examples

The user's intent needs to be provided at a level that is detailed enough to define the behavior accurately while also being abstract enough to require as little effort as possible from the user. Programming-By-Examples (PBE) is an intuitive concept that enables the user to define the behavior of a program by means of input-output pairs. This approach comes as very natural to us humans, as we use it instinctively for a variety of reasons, from explaining mathematical concepts to describing the functionality of a product and everything in between.

The PBE concept is akin to a team leader using input-output examples to communicate and assign a programming task to a software developer. The team leader will give enough examples to cover the desired functionality while avoiding examples that are completely redundant as they do not narrow down the possible interpretations of his/her intent. If the software developer writes an implementation that is correct with respect to the initial specification but incorrect for some unseen input-output examples, the team leader will ask the developer to refine the implementation and additionally, he/she will provide a counter-example that clearly points to a difference of functionality between the desired and the actual implementation. In the context of program synthesis, this idea is referred to as Counter Example Guided Inductive Synthesis (CEGIS) as introduced by Solar-Lezama in [20].

In the analogy above, if the task description can be interpreted in multiple ways, the software developer will often ask the team lead for further clarifications by providing an input that exposes the ambiguity and results in different outputs based on the different interpretations. In the field of program synthesis, this idea is known as the distinguishing inputs technique (see [24]) and it proposes a closer human-computer interaction where the human is queried interactively at various steps in the synthesis process. Aside from resolving ambiguities very efficiently, this approach brings the additional benefit that the user gains confidence in the synthesizer and the obtained solution.

PBE also allows for dividing the specification in parts by having the user provide separate sets of input-output examples for the different components. Furthermore, if the program has a specific case for which it must behave differently, it is far easier to incorporate that corner case in the specification with a single example than to explain it using other formats.

One of the important advantages of PBE is the fact that it simplifies the data processing pipeline. PBE requires little to no data pre-processing to transform it into a clean and structured format that can easily be consumed by the synthesizer [19]. Evaluating how well a program complies with a given specification is as

trivial as counting the number of inputs for which it produces the correct output. For synthesizers that rely on machine learning models, input-output examples are very common form of providing training data in the context of supervised learning.

Extracting the examples from other data sources may pose some difficulty depending on the domain in which the synthesizer is used. While in some cases, input-output examples may be easy to obtain, in other cases users may prefer to convey their intent by describing in natural language, the iterative transformations that the program should perform. Extracting input-output examples from such descriptions, adds another processing step and it may not be worth the effort. Furthermore, if the user's requirements change, it is far easier to change a brief informal description than to come up with an entire new set of input-output examples.

In the domain of programmable data-planes, PBE is a rather obvious choice for specifying what a network device should do. Routers and switches often work on a packet-basis meaning that their functionality can almost entirely be described by the changes that a packet undergoes from the moment it is received to the moment that it is sent back out. Irrespective of the format used for the program specification, the most straightforward way to validate a P4 program is to install it on a P4 switch, sample input packets and count how many times the program outputs the expected packet. This validation approach goes hand in hand with PBE, unlike other specification formats such as natural language which would require a packet generator that can capture the user's intent. Other validation techniques for P4 exist and are discussed in subsection 2.4.3.

Still, there is one roadblock to consider. Most network devices also store state for the purpose of creating and maintaining connections between endpoints, shaping and policing network traffic, computing statistics and so on. This apparent mismatch with the PBE paradigm can be alleviated by considering state information as part of the input and output. The drawback of this change is that it complicates the process of extracting input-output examples for stateful network functionalities. P4 switches in particular have counters, meters and registers as part of their state. The entries inside the match-action tables can also be considered as part of the state but they typically do not change from packet to packet and thus, they can be assumed to be fixed.

To add to the list of inconveniences, the behavior of a P4 program can depend not just on the content of the packets but also on metadata such as packet arrival time, length of the queue, ingress port and queuing delay. Depending on the desired functionality, these factors may also need to be included in the input-output examples.

2.3.2. Syntax Guided Enumerative Synthesis

Enumerative synthesizers perform an exhaustive search over the space of possible solutions by systematically trying all possible combinations of components and evaluating them against a given specification. The important trick is to structure the hypothesis space in such a way that only promising candidates are considered. The order of enumeration is another important performance factor since programs that are more likely to satisfy the given specification, can be enumerated earlier.

Syntax Guided Synthesis (SyGuS) [25] is a technique to restrict the possible programs enumerated by a synthesizer to only the ones that make sense from a syntactic point of view. SyGuS relies on the existence of a context free grammar (CFG) that describes the DSL that should be used by the candidate programs.

There are two common approaches to enumerative search: top-down and bottom-up. In top-down enumeration, the synthesizer first constructs a general skeleton before resolving the gaps recursively with the same top-down approach. With each iteration, more detail is added to the templates obtained earlier to obtain concrete solutions or more complete templates that can be refined further in the next iterations. In contrast, bottom-up enumeration first obtains a few primitive programs and then it enters a recursion that constructs larger programs by re-using the smaller programs enumerated in the previous rounds of the recursion. Probe [7] is an example of such a bottom-up synthesizer that shall be explained in detail in chapter 3.

Enumerative search is a rather simple concept to understand but quite powerful for relatively small search spaces. Many variants exist as a consequence of its flexibility and extensibility. It can be used in combination with other techniques and it is agnostic on the specification format. Domain-specific knowledge can easily be integrated into the synthesis process thereby filtering on semantics the programs that are

being considered. As an example, if the specification describes a transformation of the input that one knows it is only obtainable by applying a specific pattern of instructions, one can limit the synthesizer to only consider candidates with those particular instructions. On the flip-side, without the use of effective heuristics and pruning techniques, enumerative search can suffer from poor scalability. A bare-bones enumerative synthesizer will have poor performance for programs larger than a few instructions for any non-trivial language.

2.4. Relevant prior research

In this section, we shall explore some of the research conducted in the recent past that directly concerns our stated objective.

2.4.1. Intent-based networking

A concept that is gaining momentum in the networking field is called Intent-Based Networking (IBN). IBN enables administrators to specify policies, configurations settings, and desired network functionality through intents which are often defined as business-level goals expressed in natural language, abstracting away from the nitty-gritty details of implementation [26]. The network adapts itself to find the right parameters such that the network satisfies the user's desired functionality.

Han et al. [26] apply the IBN philosophy in the context of network virtualization in an effort to create a platform that allows tenants of the physical network to express high-level intents which are subsequently used by the platform to automatically manage their respective virtual networks (VN). The proposed solution performs the functions of a SDN controller and simultaneously the functions of a network hypervisor such as VN provisioning, modification, and removal. The authors do not describe in details how intents are represented but they do point out that an accurate intent contains four contextual attributes: resources (i.e., the type and number of requested resources), conditions (i.e., the criteria that must be fulfilled for activating the intent), priority and instructions (i.e., the actions are that should be applied to the packets that satisfy the conditions).

Jacobs et al. [27] describe the design of a system capable of turning user utterances in natural language into configurations commands for the SONATA NFV platform [28]. The system is composed out of three components: a chatbot based on Google's DialogFlow[2], an intent translator and an intent deployer. The first component processes the user's intent and extracts a set of entities that are then passed on to the second component. The second component consists of a pre-trained sequence-to-sequence algorithm [29] that outputs an intent expressed in the *Nile*, a new high-level yet non-ambiguous intent definition language. The benefit of this approach is that intents are much easier to parse and deploy but they can still be read and understood by the untrained user. This consideration is important since the next step taken by the proposed solution is to show the processed intent to the user and ask for confirmation. If the user confirms that the processed intent is equivalent with the original intent, the last component will deploy it by creating commands for the SONATA NFV platform.

The authors manage to obtain near perfect translation accuracies with training datasets of few thousand entries. Furthermore, the authors show empirically that incorporating the user's feedback leads to a considerable improvement in accuracy.

P4I/O [30] is an IBN framework that builds on top of Nile [27] to create an extensible intent-definition language (IDL) that allows the user to import custom actions into the intent and apply them. More importantly, P4I/O is capable of transforming the user's intent, expressed in the extended IDL, by rendering and merging templates taken from its own repository of P4 code. If users modify their intents, P4I/O will re-parse and install them dynamically on the switch without affecting its prior state and with minimal disruptions to the traffic flow [30].

The drawbacks of the approach proposed by P4I/O is that the set of intents that can be realized is limited by the set of templates stored in the repository and by the actions that can be imported. Furthermore, while Nile makes it easier for the user to express his/her intent, it is still a structured language that the user must learn as opposed to natural language.

[2] https://cloud.google.com/dialogflow

2.4.2. Program Synthesis in computer networks

This work tackles a relatively novel concept but in spite of that, there are quite a few researchers that have explored similar ideas in the past that could prove insightful for our stated objectives. The most comparable research to ours is represented by the work of Riftadi et al. [31] who proposed a system called GP4P4 which applies the concepts of genetic programming to the field of programmable data-planes to obtain P4 programs that comply with the user's intents. The program specification in GP4P4 is expressed in a structured format consisting of behavioral rules on the attributes of the packet.

Given a program specification and a set of primitive blocks, GP4P4 creates an initial population of solutions and evaluates them using a simulator to obtain the fitness value for each candidate. Afterwards, two tournaments are held and the winners are used to create offsprings which replace the losing programs. Mechanisms of genetic programming, namely crossover and mutation, are applied to the tournament winners in order to create the new generation.

Deriving the fitness value of a program is done by generating traces of input packets, simulating the P4 code on the generated input and then counting how many rules from the program specification hold. In terms of performance, GP4P4 is capable of synthesizing small programs of a few instructions in a matter of seconds to minutes.

Although our goal is quite similar to that of GP4P4, the format of the user's intent is very different and so is the method used for generating new programs, as shall be explained in chapter 3. To the best of our knowledge, we are the first to attempt synthesizing P4 programs using only examples of input-output packets while offering the guarantee that if a solution exist, it will be found by our system.

Gao et al. [32] explores the same overlapping space between program synthesis and programmable networks but for different reasons. Traditional compilers translate higher-level code into lower-level code through a static set of rules. In contrast, Geo et al. propose a novel compiler called Chipmunk that applies program synthesis to generate the machine code while the original high-level code acts as the program specification. To understand the advantages of this approach, consider a program that cannot be compiled due to resource constraints or computational limits such as the incapability of processing packets at line speed. Although the rule-based compiler may fail to compile, Chipmunk may find an equivalent solution that is more efficient and does not suffer from the same limitations. Therefore, Chipmunk will find valid compilations more often than the rule-based compilers.

Chipmunk uses the SKETCH engine [20] to synthesize the machine code and a clever idea to speed up the compilation time, called slicing. With slicing, the synthesis problem for generating code for the switch pipeline is decomposed into a collection of independently synthesizable subproblems called slices. In each slice, Chipmunk synthesizes a sub-implementation that has the same behavior as the specification, but just on a single packet field or state variable from the specification. These sub-implementations can be directly "stacked" on top of each other to form the full pipelined implementation.

More recently, Beurer-Kellner et al. [33] point out that researched methods to synthesize network configuration parameters have mostly relied on Satisfiability Modulo Theory (SMT) solvers which are slow and suffer from scalability issues. Instead, the authors propose relaxing the correctness guarantees and implement a system that generates approximate configurations that are very useful even when incomplete or incorrect. The users provide the specification through a set of requirements in the form of logic predicates which are added to a fact base that also contains predicates about the topology of the network. The parameters that need to be synthesized are also included in the fact base as "holes".

A generated configuration is tested against the specification and its consistency is measured as the number of requirements that hold. The system generates configurations that are roughly 92% consistent with the specifications while being orders of magnitude faster than SMT-based solvers. Under the hood, the proposed system follows the neural algorithmic reasoning concept [34], consisting of two graph neural networks which encode and process the fact base and multiple perceptron models that are responsible for predicting the values of the unknown parameters.

As mentioned earlier, one drawback of programmable networks is the fact that network operators must learn new languages to implement the desired network functionalities. Pereira et al. [35] make the same observation and they additionally point out that each programmable platform comes with its own programming language and hardware features which the network operator is expected to master. Their

solution is called SyNAPSE, a platform to implement network functions (NF) in C which are then used to synthesize programs for both the controller and the switch.

The key idea behind SyNAPSE is the components API which defines an abstract interface that the users can use to define network functions. Behind this API, there is a pool of implementations for the exposed data structures and functions implemented by platform experts and the hardware vendors. Multiple implementations can exist for the same abstraction, and they may differ in how they choose to divide the workload between the control-plane and data-plane. Moreover, there should be implementations for all types of switch architectures as well as general-purpose CPUs.

A developer will write an NF in C using the SyNAPSE component API and additionally, he/she will specify a performance objective (minimize resource usage, limit CPU usage, etc). Afterwards, SyNAPSE will synthesize the programs for both planes, choosing the right implementations such that the performance objectives are met. SyNAPSE accomplishes this by first using symbolic analysis to obtain a tree of code paths and then relying on heuristics and static rules to gradually replace the nodes in that tree with the right components while taking in consideration the target platforms and the chosen performance objective.

In the research paper authored by Angi et al. [36], a unique solution is described for configuring P4-enabled switches by providing intents in natural language. Their solution, entitled NLP4, relies on techniques from the Machine Learning (ML) field and it is composed of three components. The first component makes use of Natural Language Processing (NLP) to preprocess the user's intent into a vector of tokenized and encoded words that are all consistent with a dictionary of words. The second component passes the obtained vector through a trained MultiLayer Perceptron (MLP) whose output is another vector that encodes the action that must be performed, on what elements and how. The third component is an API that receives the output of the MLP component and converts it into configuration files for the P4-enabled switches.

The execution time of NLP4 is in the order of milliseconds even for large networks of 100 switches. Unfortunately, the authors mention very little in terms of how their machine learning algorithms were trained. It is also unclear how effective NLP4 is and in how many scenarios it can be of use to the network administrator.

2.4.3. P4 code testing and validation

The topic of verifying whether a P4 program is bug-free and exhibits the desired behavior has been extensively researched in the recent years [37]–[43]. We dedicate some attention to this topic because it is important to properly test the correctness of the synthesized solutions and to evaluate the general performance of the synthesizer.

Lopes et al. [37] propose a solution for verifying that a P4 network is free of two classes of bugs, namely reachability and well-formedness. The authors point out that the emergence of programmable switches, introduces a new class of bugs that they refer to as well-formedness bugs. These bugs occur when the switches/hosts in a P4 network have different header definitions or parse the packet differently because they expect a different order of the header. These bugs lead to packets being dropped due to errors that occur during parsing. The authors verify the well-formedness in a P4 network by testing whether any packet sent through an egress edge (an edge leaving the backbone of the P4 network) can be parsed at any of the input edges.

Their solution is composed of multiple parts: a topology generator, a table generator, a P4-to-Datalog compiler, the Datalog solver and a query generator. The user must provide parameters for the first two components which are used to generate the network topology as well as the set of forwarding tables referenced by the P4 program. The compiler transforms the P4 program that should be installed on each switch, into a set of Datalog rules. Lastly, the query generator translates human readable queries to queries for the Datalog engine. With all information converted into Datalog rules, the solver is used to verify the network for reachability and well-formedness bugs.

P4pktgen is an open-source tool proposed by Nötzli et al. [38] which can be used for automatic generation of test cases for P4 programs. P4pktgen takes as input the JSON file generated by *p4c*, the reference compiler for P4. It then finds concrete paths in the program and for every given path it generates a packet that exercises that path. To achieve this, a packet has to be crafted that triggers the correct parser transitions, the correct conditional branches and the correct table actions. Using symbolic execution,

these requirements are translated into a set of constraints for a SMT solver which determines whether the constraints can be satisfied and if so with what values. In other words, if a given path can be reached, the SMT solver will return concrete values for the symbolic inputs which can be used to construct the packet and the table entries needed to ensure that the program takes the path in question.

By using P4pktgen, test cases can be generated for each possible path that consist of pairs of input-output packets. The input packet is generated by p4pktgen while the output packet is obtained from running the P4 program on the input packet. One way to use p4pktgen is to validate that P4 programs act as intended on their target devices, by running the same test packets through a software reference implementation and the target hardware and comparing the output of the two configurations.

Freire et al. [39] propose ASSSERT-P4, a tool to annotate P4 programs with correctness and security assertions which are then verified compile time. To realize this, ASSERT-P4 introduces an intuitive language grammar that allows developers to add to their P4 programs two types of assertions: location-restricted and unrestricted invariants. A location-restricted assertion tests the value of a variable at a specific point in the execution while an unrestricted one checks whether a condition holds during the entire execution. ASSERT-P4 first translates the annotated code to the C language through a set of static translation rules. Then, it uses symbolic execution to check whether there are no assertion violations.

Without the contents of the match-action tables, the behavior of a programmable switch running a P4 program is not fully specified. The approaches described thus far circumvent the issue by relying on symbolic execution or by inferring them from the topology information that is provided by the user. Liu et al. [41] develop a verification tool for data-planes called P4V that allows the developer to define control-plane interfaces that make assumptions about the rules that may be installed on the P4 switch. If a verified P4 program is paired with a control plane that complies with these assumptions, the behavior of the resulting network will be as expected.

P4V is built to verify a few classes of properties: safety properties (checking header validity, checking overflow for arithmetic operations), architectural properties and program specific properties. Besides defining control-plane interfaces, the user can directly annotate the P4 code with assertions.

Under the hood, P4V first translates the P4 code to the Guarded Command Language [44]. Afterwards, the control-plane interfaces, which can be stored in a separate file, are weaved into the P4 program. From the outcome, P4V generates a formula and checks whether it is valid using the Z3 [45] theorem prover. If the formula is not valid, a counter-example is generated specifying a concrete trace through the program that can be used for debugging.

3

The Probe-cpp Synthesizer

Considering the end objectives set in the first chapter, a general concept was needed to describe, at an abstract level, how to go from examples of inputs and outputs to a P4 solution. A synthesizer needed to be implemented that is compatible with the problem at hand. The synthesizer is a major part of the final solution, and it impacts not just the runtime performance but also the number of compute resources needed, the amount of information needed from the user, the format of the input and outputs and everything in between. To illustrate, a synthesizer using artificial neural networks such as [22] and [46], requires a large corpus of code to train the underlying neural network. Considering that P4 is a recent development, and it is used only in the specific context of programmable switches, it would be rather challenging to obtain such a corpus.

The overall complexity of the synthesizer was also an important factor due to the fact that implementing a complex synthesizer or setting it up for our purposes could take too much time considering the scope of this project.

3.1. The choice for Probe

Developed by Barke et al., Probe [7] is a state-of-the-art synthesizer that proved to fit most of our criteria. However, during the later stages of the development, we did discover some minor incompatibilities which shall be discussed in chapter 4. In order to work, Probe needs a set of input-ouput examples and a Probabilistic Context Free Grammar (PCFG) which is a CFG for which every production rule has an assigned probability. As explained previously, the programming-by-examples paradigm suits our problem domain rather well.

Probe enumerates programs bottom-up while also dynamically learning what production rules to prioritize by modifying their assigned probabilities. Because of the bottom-up nature of the algorithm, every subprogram synthesized by Probe needs to be evaluated and thus, it must be executable and lead to an observable output. On top of that, Probe's performance is greatly increased if evaluations of subprograms can be cached and reused to determine the output of a bigger program. This consideration poses both a drawback and an opportunity.

Interpreting a synthesized P4 program can easily be the most time-consuming task in the synthesis process. This step entails running a P4 switch emulator, compiling the code and transferring it to the switch, and finally running it for all input-output pairs. Even if done efficiently, performing this step for all synthesized solutions is going to severely affect runtime performance. Furthermore, a full-fledged interpreter cannot evaluate incomplete solutions and it cannot reuse the evaluations of partial programs to efficiently evaluate the program composed of the smaller parts.

The implementation of a lightweight simulator seems to be unavoidable regardless of the choice of synthesizer. Nonetheless, Probe's performance dependency on the method used to interpret and evaluate programs provides an additional incentive to implement a simulator tailored to leverage the strengths of bottom-up enumeration.

Last but not least, Probe is rather deterministic as it performs an exhaustive search on the solution space. However, its performance can be enhanced by adapting it to make use of probabilistic components. Its performance can be improved further by incorporating domain knowledge in its internal mechanisms such

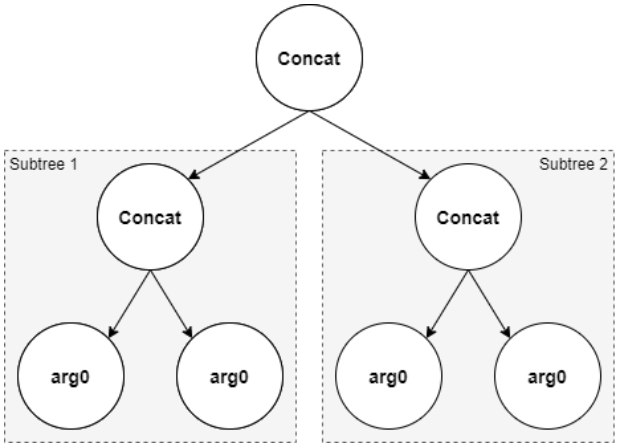


Figure 3.1: Abstract syntax tree for a program that repeats a string four times

that it leverages the particularities of the P4 environment. One example of this is modifying the initial probabilities of the production rules depending on what parts of the output packet are different from the corresponding input packet.

In summary, Probe was chosen as a synthesizer for this project because of its flexible and simple yet powerful design. It is aligned with the PBE concept and all it requires in terms of input is a set of input-output examples and the grammar of the language to be used by the synthesized programs. It is compatible with our objectives in spite of some minor conflicts that can be resolved.

3.2. Original algorithm

Bottom-up synthesizers create larger programs by combining the smaller programs that were previously enumerated. This is in contrast to top-down enumeration which first defines the outlines of the program and then divides it into smaller parts that are recursively refined. Syntax guided synthesizers represent programs as abstract syntax trees (AST) for which every node corresponds to the use of a production rule from the grammar.

Creating a larger problem is simply a matter of selecting a grammar rule, creating a corresponding root level node and finding already-enumerated sub-solutions that can act as children to the chosen production rule. Figure 3.1 illustrates this by showing a program that repeats a string four times by reusing a program that repeats it twice. The root level node uses the production rule $S \longrightarrow (concat\ S\ S)$ from the grammar shown below which is a reduced version of the full grammar shown in appendix A. This particular program is composed by applying this specific production rule and retrieving subtrees from a bank of previously synthesized programs. The element $arg0$ is a terminal and is replaced by a given program argument during the evaluation step.

S	$\longrightarrow$	arg0	the input to the program
	|	(replace *S S S*)	finds the first occurrence of S_2 in S_1 replaces it with S_3
	|	(concat S S)	concatenates two strings
	|	(ite B S S)	if B yields true, return S_1, otherwise return S_2
B	$\longrightarrow$	(contains S S)	checks whether S_1 contains S_2

A major benefit of using bottom-up enumeration is precisely the possibility of re-using sub-solutions. Unfortunately, the space of possible solutions remains exponential but composing an individual program has constant time-complexity. What is even more, subtrees one and two in figure 3.1 are, in fact, two different pointers to the same program. Because this program is already stored in memory, the larger program only needs a constant amount of space to store the information about the root node and some

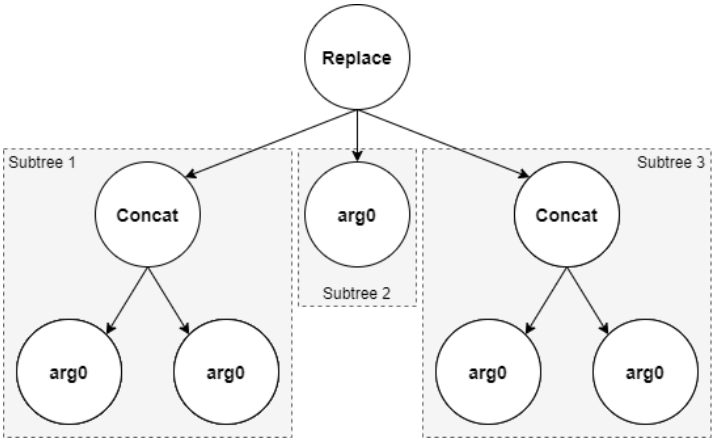


Figure 3.2: Observationally equivalent program for the program in figure 3.1

pointers to the subtrees. In other words, the space requirements of bottom-up enumeration grow linear with the number of programs.

Another benefit of bottom-up enumeration is the possibility to discard programs that are not promising. If a program is equivalent in functionality to another, but it is longer, it can and will be thrown away. In general, the problem of deciding whether two programs are equivalent is undecidable. However, if the set of possible inputs is finite, two programs can be deemed observationally equivalent relative to this set of inputs, if they lead to the same output for all inputs from the finite set. In our case this holds true because the set of inputs and outputs is given as part of the problem specification. Figure 3.2 depicts a program that is observationally equivalent to the program in figure 3.1. Depending on the order of enumeration, one of them will be discarded.

A naive implementation of Probe evaluates each synthesized program individually, interpreting each program from scratch. This leaves performance on the table since evaluations of larger programs can make use of the evaluations of their subtrees. Let us illustrate this by looking again at figure 3.1. Let us assume the program specification contains a single input-output pair: (”P4”, ”P4P4P4P4”). When fed the input, the program represented by subtrees one and two produces the output ”P4P4”. To evaluate the larger program, one merely needs to take the evaluations of the two subtrees and perform the operation that is represented by the root-level node. In this case, this boils down to concatenating two instances of the string ”P4P4” to form the desired output.

By integrating the interpreter into the grammar and assigning an executor to every production rule, we avoid the unnecessary re-computation of subtrees. As the synthesizer enumerates larger and larger programs, the time spent evaluating will remain the same. That is to say, the time complexity of the evaluation of a program is constant irrespective of the height of its AST. That is in contrast to an inefficient standalone interpreter for which larger programs take longer to execute.

At an algorithmic level, Probe starts with a PCFG that has an uniform distribution of probabilities. For every rule, a cost is derived that is inversely dependent on the rule’s probability. The cost of a program is the sum of the costs of the rules that were used to synthesize it. Probe populates a bank of synthesized solutions that is indexed by cost. This bank is filled incrementally, from the lower to the higher costs. At any point in time, Probe tries to enumerate a program of a specific cost by iterating through the production rules and for each one, searching through the sub-solutions stored in the bank. The aim is to find right subtrees such that an AST can be formed that is syntactically correct, and whose total cost adds up exactly to the target cost. If no program is found, Probe increments the target cost, and the search continues.

Probe also maintains a bank of previously seen evaluations. When a new program is synthesized, the evaluations bank is searched to check whether the program is observationally equivalent to a previously enumerated program, in which case it is discarded.

3.2.1. Probe's guarantees

The Probe synthesizer offers two strict guarantees, namely soundness and completeness. The first guarantee ensures that if a solution is found, it is correct syntax-wise and it is compliant with the program specification that was provided. The second guarantee tells us that if there exists a solution that can be expressed with the given grammar, Probe will eventually find it, or it will find an equivalent solution.

Aside from these guarantees, Probe will also find the smallest solution in terms of the total cost of the program. If the cost of every production rule is the same and it is never modified, then Probe will certainly find the smallest solution in terms of the number of nodes in the AST. However, these costs are potentially updated by the learning feature of Probe and as a consequence, longer programs can actually be considered cheaper and enumerated earlier. Still, compared to other synthesizers, Probe finds solutions of optimal or near-optimal size (see chapter six from [7]).

3.2.2. Cost-based versus height-based enumeration

Standard bottom-up synthesizers enumerate programs in order of increasing height. This means that two programs that are very different in size can be enumerated during the same step despite one being much larger than the other. In the examples shown in figures 3.1 and 3.2, the program represented by subtrees one and two is a promising sub-solution that is one layer away from the final solution. However, a height-based enumerator will first search programs of the same height but with far more nodes before finally incrementing the target height. In contrast, Probe derives a cost for each rule and enumerates programs in order of their total cost. If the PCFG has a uniform distribution of probabilities, the total cost of a program is equal to a constant (the only possible cost) multiplied with the number of nodes in its AST. The effect of this change is that Probe prioritizes programs whose ASTs are tall and narrow as opposed to short and wide. The authors show empirically that this is a more efficient approach which leads to more useful programs.

3.2.3. Just-in-time learning

An important feature of Probe, called Just-In-Time learning, is the ability to dynamically learn and adjust the probabilities of the grammar rules in order to steer the search towards regions of the solution space that seem more interesting. The search remains exhaustive but the solutions that seem promising are given priority. The general algorithm is split in two alternating phases, namely the synthesis phase and the learning phase. During the synthesis phase, Probe enumerates programs bottom-up, from lower costs to higher. Promising solutions that are unique and solve at least one of the input-output examples, are stored in the bank and also in a separate data structure. The synthesis phase ends after reaching the specified target cost.

During the learning phase, Probe loops through the promising solutions found during the synthesis phase and selects the most promising ones according to a pre-defined selection policy. The selected programs are used to update the grammar by changing the probabilities assigned to every rule and implicitly the costs. The intuition is that the rules used by the promising solutions are more useful for the problem at hand and are more likely to lead to the final solution. If no promising solutions are found then no changes are made to the grammar, the learning phase ends and the synthesis phase continues where it left off. In the event of a grammar update, the subsequent synthesis phase must reset the two banks and restart the enumeration due to the fact that changing the rule costs will change the computed costs of the programs stored in the solutions bank. This occurrence invalidates the bank's indexing and re-indexing can prove to be more expensive than starting from scratch.

To illustrate the learning feature of Probe, let us assume the program in figure 3.1 is considered a promising solution to a larger problem. The production rules that are contained in this program, namely $S \longrightarrow (concat\ S\ S)$ and $S \longrightarrow arg0$ will have their probabilities increased and thus, their costs decreased. Because the probabilities are normalised, the other rules will have their cost increased. The amount of change is relative to the number of inputs for which the selected program yields the expected output. In the following synthesis phase, the enumeration restarts, and the same selected program will be found earlier in the enumeration process and potentially used sooner to build larger solutions closer to the final solution. It is important to not allow programs which were selected in the past to further bias the PCFG. Therefore, any such program is cached and ignored in subsequent learning phases.

The authors discuss and compare three policies for selecting promising solutions: *largest subset*, *first*

cheapest and *all cheapest*. Ideally, the selection policy should select only the rules that seem necessary while simultaneously not becoming too biased towards one approach when multiple are possible. The authors show empirically that the most efficient policy is the *first cheapest* which selects the cheapest solutions that satisfy (i.e., are correct with respect to) a unique subset of the input-output examples.

3.3. Improved implementation

Although the original implementation was available online, we made the decision to implement it from scratch for reasons that are explained next. First, the original Probe uses the SyGuS language standard to express in one file, both the grammar as well as the input-output examples [47]. This format did not seem very adequate because we preferred to separate the more general grammar from the particular problem specifications which, in our case, would include not just the input-output examples but also additional information about the execution context as well as some potential parameters for the interpreter.

Second, the authors of Probe focused on writing the research paper and not so much on following good software engineering practices to deliver well-structured and clean code. Understanding it and adapting it would have taken about as much time as implementing it anew.

Third, re-implementing it gave us the opportunity to improve it and customize it to fit our objectives. Most notably, we managed to obtain significant speed-ups by parallelizing the synthesis phase while still maintaining the original guarantees explained in subsection 3.2.1.

3.3.1. Differences to the original algorithm

Probe-cpp is written with modularity in mind and does not fix the format for the grammar or problem specification. Nonetheless, it comes with one default grammar parser which expects a grammar expressed in the same form as used by the Bison parser generator. The grammar notation used by Yac/Bison is a variant of to the Backus-Naur Form, a simple and clear notation, widely used to describe context-free grammars. In fact, P4's language specification is also available in Yacc/Bison language[1].

A concept explored in the early phases of the development was to use Bison to generate an interpreter which could be used by the synthesizer to evaluate the enumerated programs. The user would only need to provide a context-free grammar with semantic actions which would then be used by both Probe-cpp and Bison. The concept was implemented and it worked but it was too slow and inefficient for reasons explained in section 3.1.

The choice was made to provide the problem specification in the ubiquitous JSON format because of its flexibility and simplicity. Since the inputs and outputs to the conceptual P4 synthesizer would contain byte strings of actual network packets, we imagined a feature for automatically transforming traces of captured packets into JSON files containing input-output examples for the synthesizer. Such a feature could be useful if one would want to port the functionality of a traditional fixed-purpose switch onto a programmable one. He/She would capture input and output packets and would use the P4 synthesizer to generate a solution that complies with the examples given in the captured traces.

Unlike the original implementation, Probe-cpp indexes the bank not just by cost but also by the start symbol (i.e., the non-terminal that is on the left-hand side of the production rule represented by root of the AST). For the small grammar shown above, the bank separates the programs by their costs and whether they result in one of the non-terminals *S* or *B*. If, for instance, the enumerator is considering a production rule which requires a sub-program of a specific cost that outputs a *B*, only the right part of bank needs to be considered. In contrast, the original implementation performs unnecessary filter operations on the list of programs of a specific cost to find suitable candidates. Similarly, equivalence between two programs only needs to be checked if the two programs have the same start symbol. For this reason, the data structure storing the evaluations in Probe-cpp is also indexed by start symbol.

Aside from code-level improvements, Probe-cpp also comes with enhancements at the algorithm level. Probe-cpp's policy for choosing promising solutions is called *Dominant Sets* because it selects the cheapest solutions that are not dominated by others. A solution is said to be dominated if there exists another solution that satisfies a proper superset of the input-output pairs satisfied by the former. If multiple solutions satisfy the same set, only the one enumerated first is selected. The intuition behind this policy is that

[1] https://p4.org/p4-spec/docs/P4-16-v1.2.4.html#sec-grammar

Probe-cpp should not reward rules that are used by sub-solutions that are objectively worse than others. Different approaches are still considered but only if it they satisfy different sets of input-output examples.

The *Dominant Sets* selection policy is similar to the *first cheapest* policy but different in one important aspect. To illustrate, assume that the list of promising solutions at the end of the synthesis phase includes program P_1 and P_2 and that the input examples are e_1, e_2 and e_3. Consider a case in which P_1 satisfies e_1 while P_2 statistifes e_1 and e_2. Unlike *Dominant Sets*, the *first cheapest* policy will select both solutions and potentially reward irrelevant rules that are used by P_1 and not P_2.

Because the two implementations have different selection policies, they often follow different execution paths and may obtain different but equivalent solutions. Furthermore, Probe-cpp tries to update the grammar much more frequently which could even lead to programs that are slightly longer but considered cheaper due to the more biased state of the grammar.

Significant speed-ups were obtained by modifying the algorithm slightly to allow for parallel execution of the synthesis phase. As explained earlier, the synthesis phase progresses in steps with each step corresponding to a target cost. At each step, the enumerator tries all possible grammar rules and all sub-solutions enumerated earlier to create programs of the desired cost before advancing to the next step by incrementing the target cost. Profiling Probe-cpp on a single thread showed that almost all of the execution time is spent inside the execution of a step and thus, dividing this workload on multiple threads would bring great benefits.

To achieve parallel execution, the key idea is to split the grammar and make each of the available thread responsible for a subset of the production rules. Take, as an example, a division of the grammar shown above where the first thread is assigned all rules with an *S* on the left-hand side. This thread will only generate programs that have *S* as a start symbol but it may need sub-solutions that have *B* as a start symbol, which is assigned to a different thread. Although this may seem like an obstacle, threads will only need sub-solutions generated by other threads that are of smaller costs than the current target cost. If threads wait for each other before incrementing the target cost, it cannot be the case that a sub-solution that is needed by one thread has not yet been enumerated by the responsible thread. In other words, the threads must synchronize at the end of every step. Aside from this constraint, the sub-solutions bank needs to be made thread-safe and shared by all threads. An alternative implementation would have each thread populate its own partial bank which would be merged into the central bank at the end of the step, reducing the need for mutual exclusion.

3.3.2. Architecture

This subsection dives a little deeper in the architecture of Probe-cpp. The building blocks of Probe-cpp can be divided into four categories depending on their responsibility: parsing the grammar & program specification, representing ASTs, program synthesis and caches for sub-solutions & evaluations. This subsection will describe each of these blocks separately. A UML diagram of the full architecture is included in appendix A.

Figure 3.3 shows the components of Probe-cpp responsible for parsing the grammar and reading the program specification. The default formats accepted is Bison for the CFG and JSON for the program specification. However, the user could extend Probe-cpp with other parsers that implement the *GrammarParser* or *IOReader* interface. Aside from the input-output pairs, the program specification may include additional literals that are not general enough to be included in the grammar file but are specific to the program that needs to be synthesized.

A solution is represented in Probe-cpp with the *AstNode* class shown in figure 3.4, together with the representation of a production rule and a PCFG. A program takes a constant amount of space regardless of the size because it stores pointers to its sub-solutions that are already saved in the bank. Each *AstNode* has a corresponding grammar rule and each grammar rule can be assigned an executor by an *ExecutorFactory* but it is not mandatory. An executor is merely a function pointer or a lambda expression that performs the operation specific to the grammar rule.

The way an synthesized program is evaluated is dependent on the interpreter. The basic interpreter used by Probe-cpp assumes that all grammar rules are executable and by extension every *AstNode*. When an instance of *AstNode* is executed, it takes the evaluations of its children and feeds them as input to the executor corresponding to the rule pointed at by *rulePtr*. Along with the outputs of the sub-solutions, the

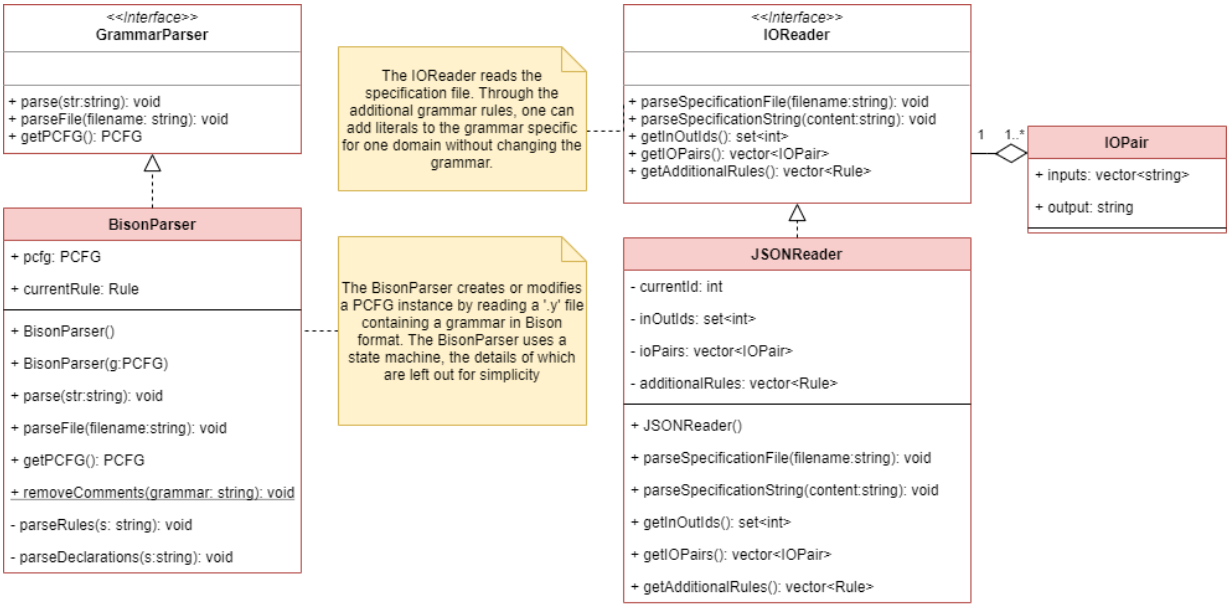


Figure 3.3: Components that read and parse the grammar and problem specification

executor also requires as input, the program arguments from the program specification (i.e., the values for *arg0*, *arg1*, etc).

For every enumerated program, a check needs to be made to verify that it is not observationally equivalent to another already found program. This check is a rather expensive operation, which makes an efficient implementation all the more important for the overall performance. Likewise, the solutions bank is iterated over during every step and potentially by multiple threads simultaneously. Choosing an inefficient underlying data structure or overusing mutual exclusion could significantly affect performance. The implementation of both data structures is sketched in figure 3.5.

The evaluations bank is implemented as a hash map with separate chaining. A custom hashing function is used to map program evaluations to their respective buckets. Unlike the original Probe, evaluations are not checked for equivalence against all other evaluations seen but only against the ones who map to the same bucket. Probe-cpp is optimized for runtime performance so it allocates more buckets than necessary, but that increases the chance that all calls to *insert* and *contains* take a constant time.

As explained earlier, the solutions bank is divided by start symbol and once more by cost. The programs belonging to the same group (i.e., having the same cost and start symbol) are stored in a separate instance of *ProgramList*. Hash maps are used to find the right instance of *ProgramList* in $O(1)$ time. To deal with the issue of concurrent access, a program list has only two modes: write-only or read-only. A program list starts in write-only mode and threads must acquire a lock in order to insert a program. Each program list has its own lock which is why it is unlikely that threads will hinder each other too much, also considering that the chance of two threads accessing the exact same program list at the same time is rather small.

After all programs of a specific cost have been enumerated, Probe-cpp moves on to the next target cost. Hence, the solutions bank can mark the old cost as ”closed” and all respective program lists as read-only. The benefit of this change is that program lists that are marked as read-only can be read concurrently without the acquisition of a lock. Concurrent access for both read and write operations would not be possible without the use of mutual exclusion. Furthermore, threads need to read from multiple program lists at once. Forcing them to acquire multiple locks could lead to the dining philosopher’s problem.

Figure 3.6 contains a UML diagram of the components responsible for enumerating programs and updating the PCFG. If a user wants to apply Probe-cpp to a custom CFG, he/she must provide an implementation to either the *ExecutorFactory* interface or to the *Interpreter* interface. The *BasicInterpreter* is a rather simple class that relies on the *ExecutorFactory* to assign executors to each grammar rule. However, one could, as an example, implement a different interpreter that retrieves the string representation of a synthesized program and passes it to an external simulator.

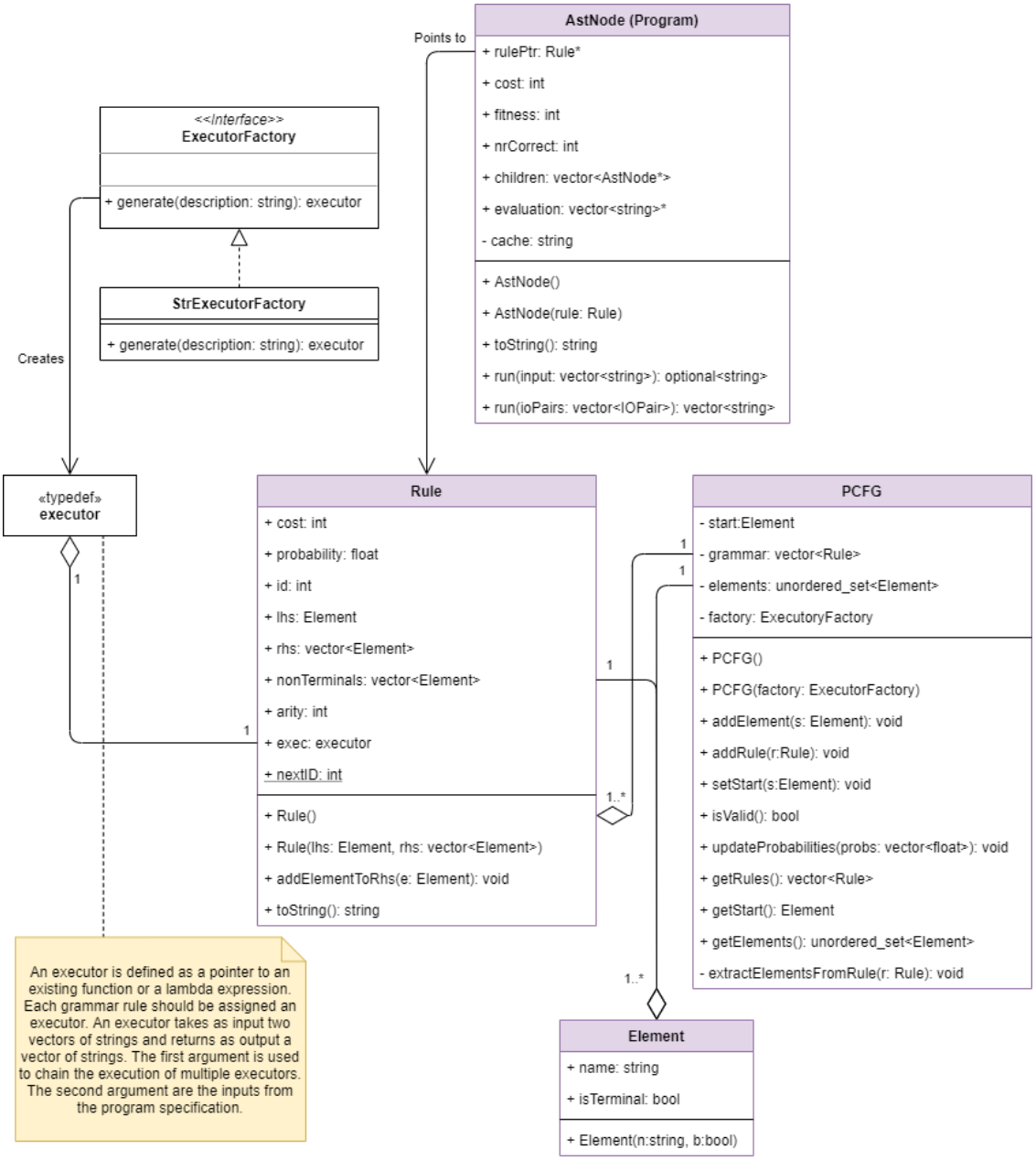


Figure 3.4: The representation of a program in Probe-cpp

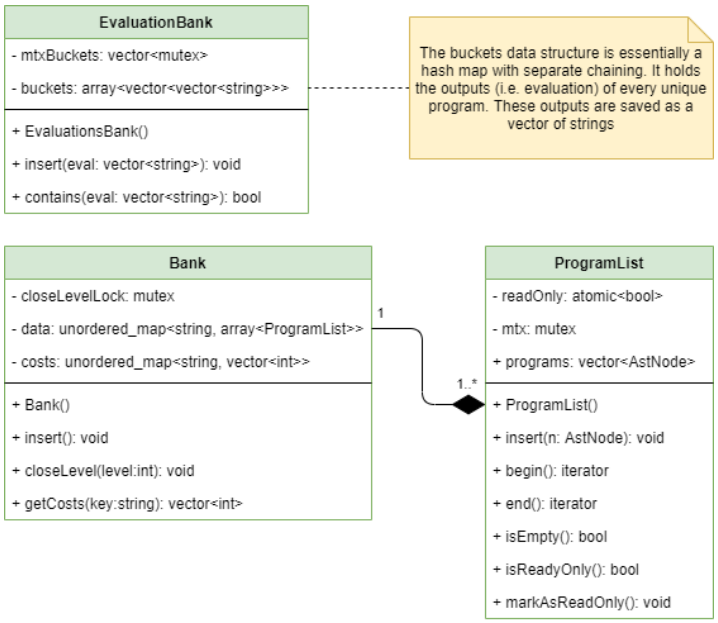


Figure 3.5: Data structures for storing the enumerated programs and their evaluations

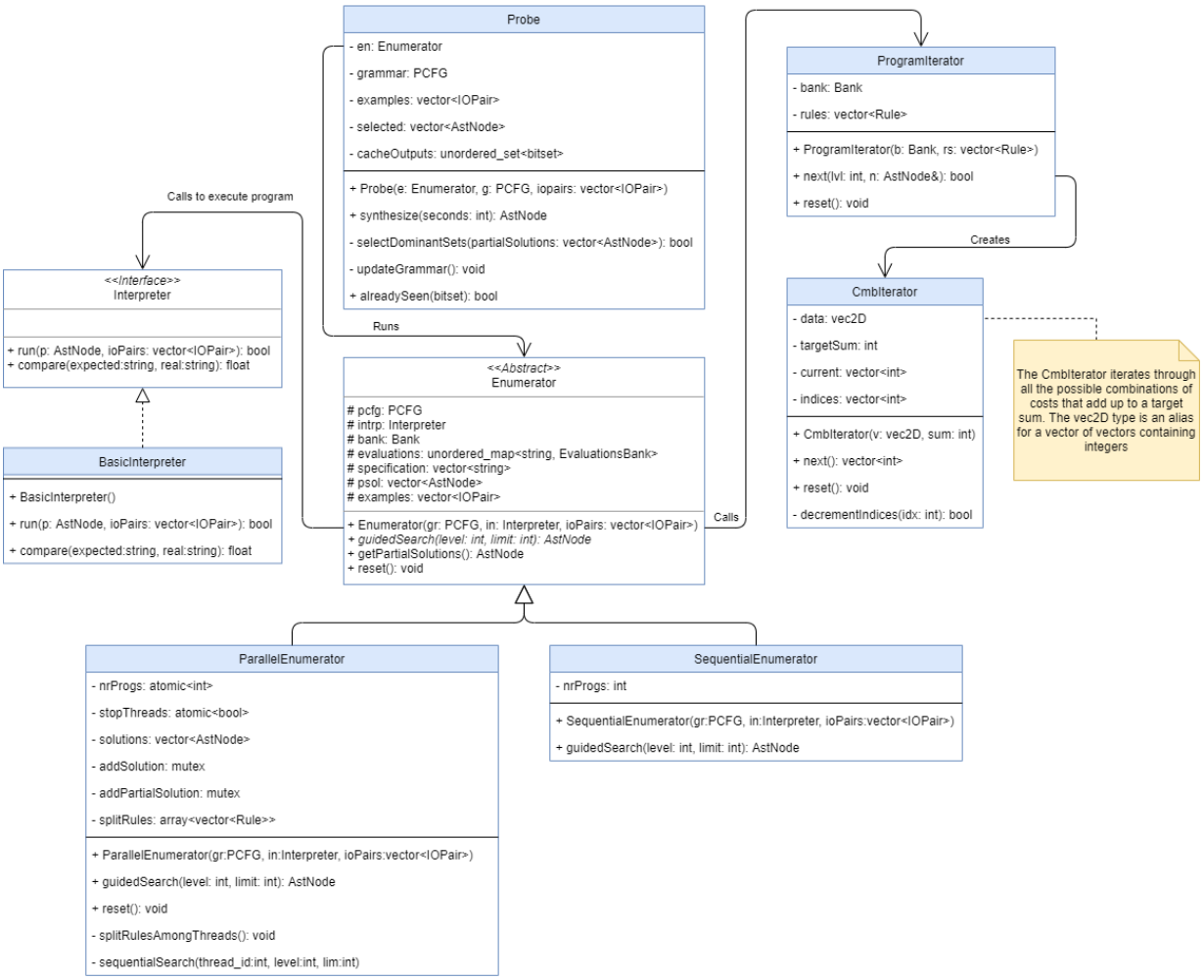


Figure 3.6: The components of Probe-cpp responsible for enumerating programs

Synthesizer	Solution
Original Probe	(substring arg0 (indexof (replace arg0 ” ” (int2str (length arg0))) ” ” 1) 4)
Probe-cpp	(substring arg0 (+ (indexof arg0 ” ” (+ (indexof arg0 ” ” 1) 1)) 1) 4)

Table 3.1: Difference in obtained solutions for the *stackoverflow10* benchmark

3.4. Evaluation

This section presents an overview of the measurements that were conducted to assess and compare the performance of Probe-cpp against the original implementation. The original Probe implementation (simply referred to as Probe) is written in Scala and it is open-source[1]. As the name suggests, Probe-cpp is implemented in C++20 and it will also be made publicly available[2].

To make an accurate comparison of the two implementations, we have chosen eight random benchmarks and three criteria: execution time, the number of AST nodes of the solution and the memory usage. The benchmarks were taken from the Probe repository which in turn, were taken from the Euphony Benchmark Suite[3]. With this goal in mind, the implementations were ran 20 times for each of the eight benchmarks. All measurements were done on the same system which runs a Linux-based operating system and has 8GB of DDR3 memory and a quad-core CPU with Hyper-Threading, each core running at a maximum frequency of 3.6 GHz.

Averaging out the observed speed-ups for all benchmarks, Probe-cpp is 8.5 times faster than the original implementation. A direct comparison of the average execution times per benchmark is shown in figure 3.7 (note the logarithmic scale). The highlighted region below and above the dotted line represents one standard deviation on each side. Figure 3.8 contains boxplots showing the execution time of the original Probe implementation for the eight benchmarks. Likewise, figure 3.9 shows Probe-cpp’s performance for the same benchmarks.

Looking at these figures, two aspects stand out. First, Probe-cpp is significantly faster for all benchmarks, and while multithreading speeds up the computation substantially, it is not the only factor. Second, there is more variation in the execution times if multithreading is used. This should come as no surprise since parallel execution is known to be prone to timing issues. However, this effect is reinforced by the fact that different runs of multithreaded synthesizer can actually take different execution paths and even lead to different solutions. At first, this may seem like a problem, but it has a rather simple explanation.

The order of execution of threads can be different from run to run and each thread is responsible for searching in one region of the solution space. Two programs that are observationally equivalent, have the same cost but are discovered by two separate threads, may be discovered in different orders across different runs. This may lead to a divergence of the execution paths. Different production rules may be considered more promising during the learning phase, perpetuating the divergence. Nevertheless, Probe’s guarantees would not be violated by such an occurrence. Both programs have an equivalent program of the same cost (approximately the same size) that can be used to build larger solutions. The search is still exhaustive, but the order of enumeration may be slightly different causing minor variations in performance.

Figure 3.10 plots the average number of AST nodes contained in the final solution of every benchmark. Once again, the multithreaded Probe-cpp shows slight variations due to the effect explained earlier. The two implementations yield solutions of the same size with the exception of two benchmarks for which the difference is small. To illustrate, table 3.1 contains two solutions synthesized for *stackoverflow10* which describes a program that can extract the year out of a string like ”February 7 1892 Verona Township”. The solution obtained by Probe-cpp is slightly longer but it is enumerated first because it is considered cheaper for reasons explained in subsection 3.3.1.

Figure 3.11 shows a comparison of the memory usage of the two implementations. However, the information contained in the figure is quite misleading. Unlike Probe-cpp, the original implementation runs on top of the Java Virtual Machine (JVM) which has its own memory footprint and does its own memory management.

[1] https://github.com/shraddhabarke/probe

[2] https://gitlab.tudelft.nl/lois/probe-cpp/

[3] https://github.com/wslee/euphony/tree/master/benchmarks

Therefore, it is difficult to accurately estimate the memory requirements and make a like-to-like comparison. It is also difficult to measure the memory overhead caused by the JVM because the JVM may allocate more memory than needed. For these reasons, the plot does not account for JVM's memory overhead which would otherwise make original Probe's measurements appear worse. Nonetheless, Probe-cpp's memory efficiency could be improved as well. It uses many instances of *std::array* whose size needs to be known at compile time, leading to frequent over-sizing.

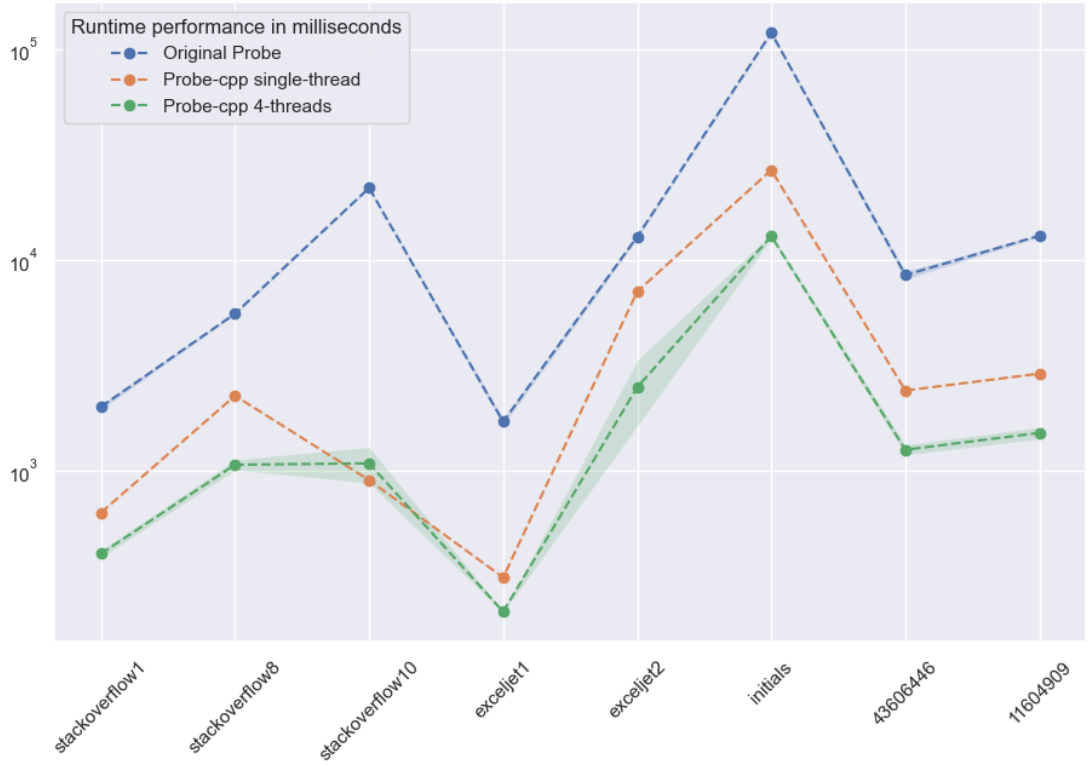


Figure 3.7: Runtime performance comparison for the eight benchmarks

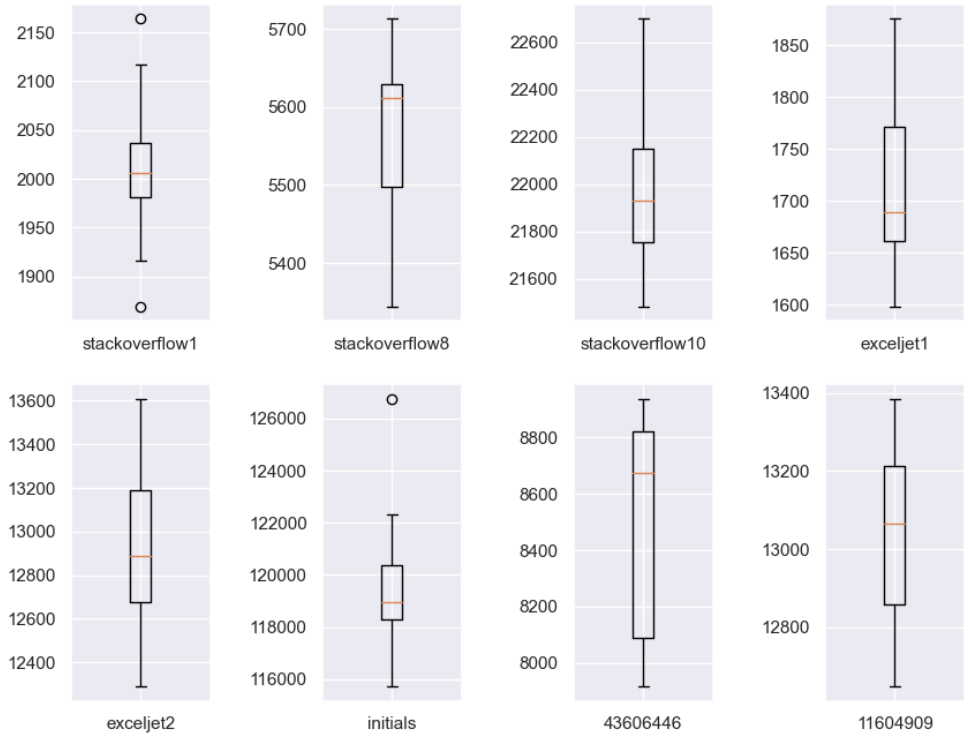


Figure 3.8: Execution times for the original Probe implementation for eight benchmarks

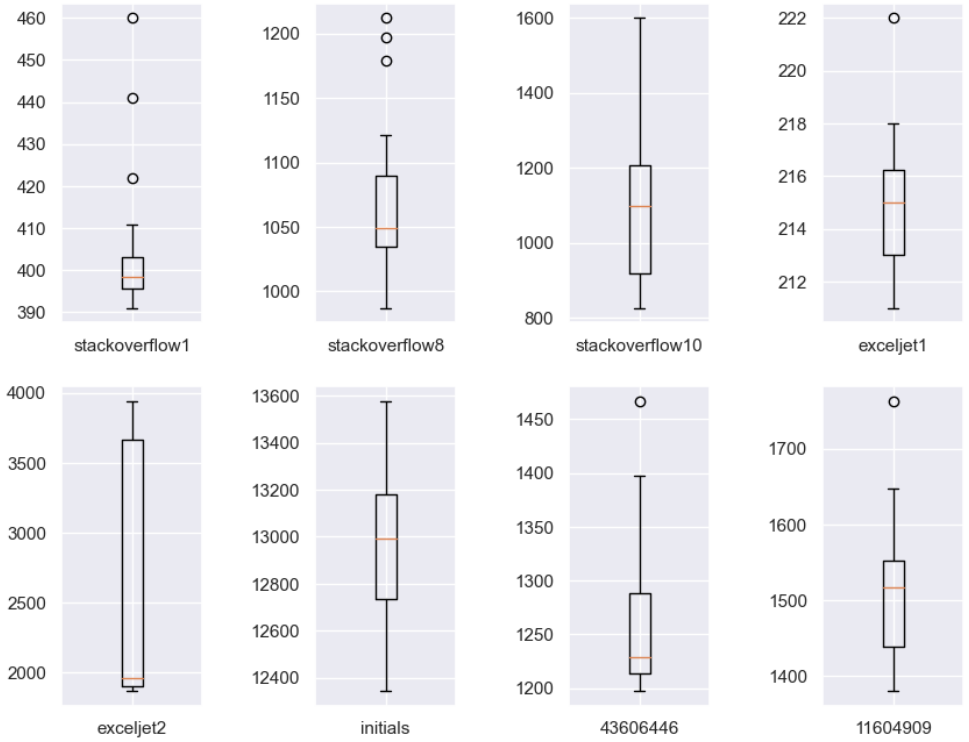


Figure 3.9: Execution times for Probe-cpp for eight benchmarks

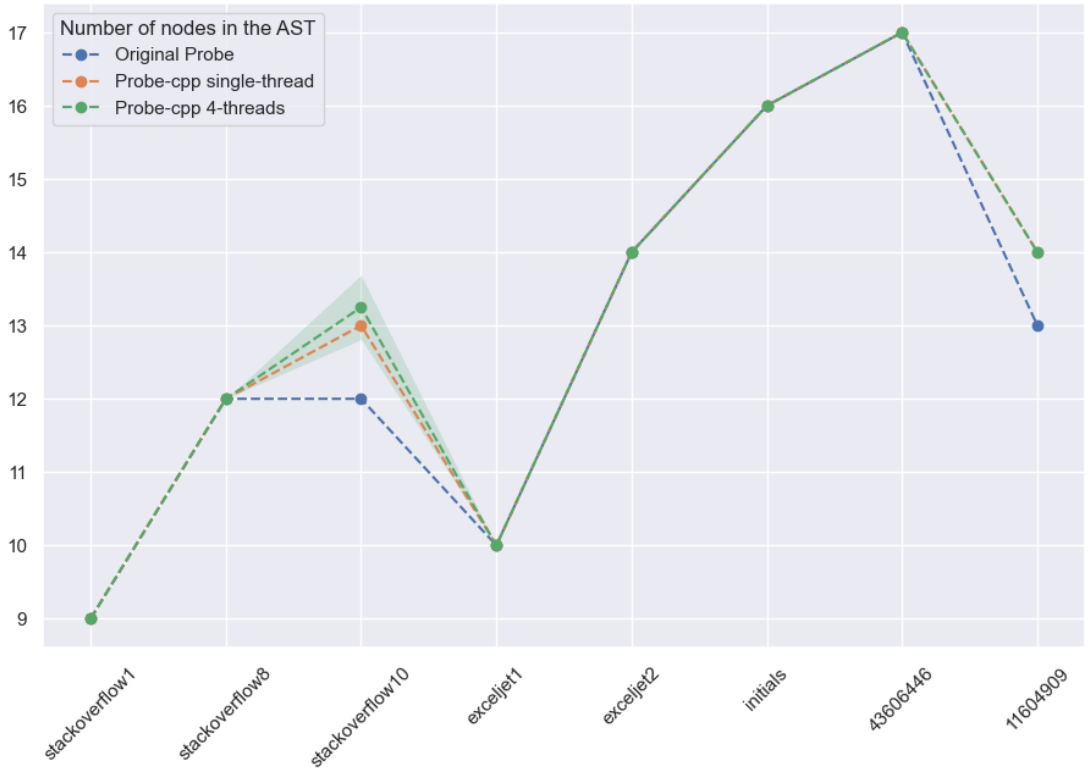


Figure 3.10: Average number of AST nodes of the final solution for Probe and Probe-cpp

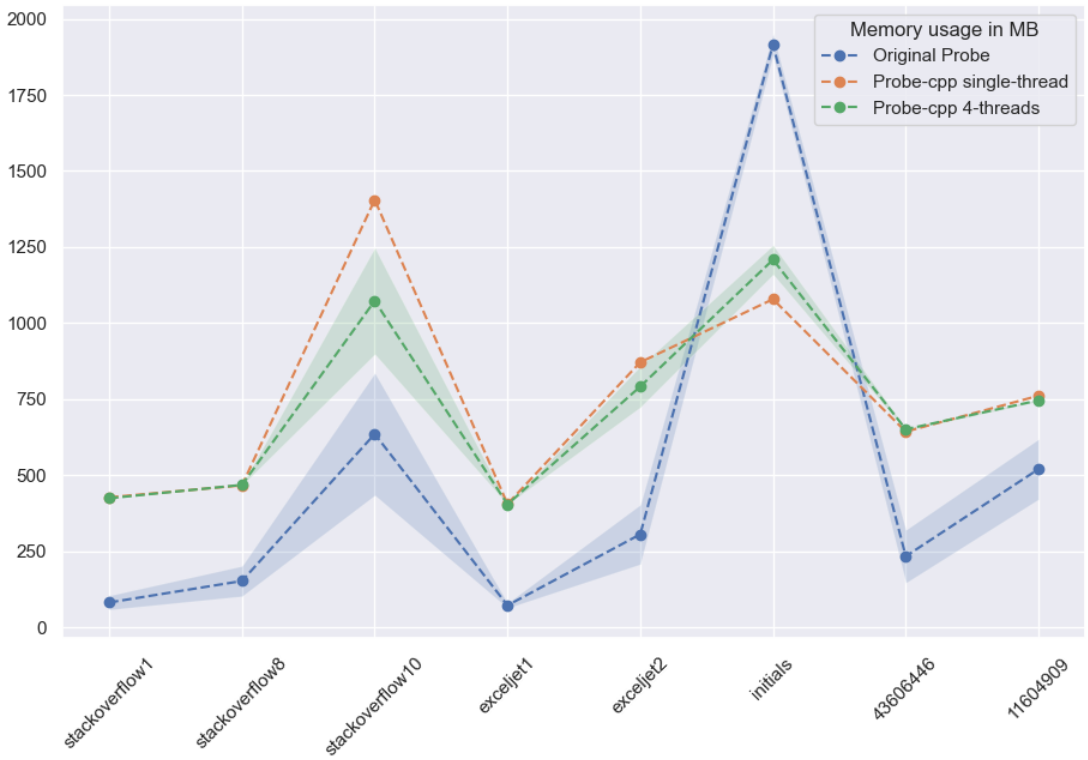


Figure 3.11: Memory usage comparison between the original Probe and Probe-cpp

4

Synthesizing P4 code

Having implemented a fast and flexible synthesizer, the challenge now becomes using it for the purpose of synthesizing code for programmable data-planes. The solution that we describe in this chapter is called G4BE[1] which stands for **G**enerating P**4** **B**y **E**xamples. Section 4.1 dives into the details of G4BE's architecture, describing at length the different obstacles that we faced during the implementation. Section 4.2 explores the idea of mining code snippets from a P4 code corpus or having the user provide them in order to enhance the performance of G4BE. Section 4.3 ends the chapter with an explanation of how the synthesized match-action tables are populated.

4.1. Architecture

Going from the idealised CFGs that are usually used in program synthesis to a DSL that is actually used in practice, requires extensive adaptations to the synthesizer in order to capture the quirks of the underlying language and its execution environment. The P4 language in particular, together with the P4 runtime environment, includes many features that set it apart from regular general-purpose programming languages.

4.1.1. Input-output format

The first important aspect of the P4 environment that G4BE must be aware of, is that the input and output to every synthesized solution is a network packet, which essentially is a byte string representing a composite object and not merely a primitive type such as a string, integer or boolean. In fact, packets can vary in size, content and even in structure. Herein lies the first hurdle on the way towards synthesizing P4 code.

The parser component of every P4 program is responsible for casting the input from an unstructured format into a structured one, where bytes are grouped into fields with each having their own separate meaning. Usually, the structured format follows one of the standardised headers described by organisations such as the Internet Engineering Task Force (IETF) through documents called Request For Comments (RFC). One of the advantages of P4 is that it allows the developer to define custom packet headers, as well as a custom state machines for the parser and deparser blocks. The way one defines these custom components is in line with his/her interpretation of what the bytes represent at the different offsets in the packet. In other words, the parser component does not limit the possible functionalities of the control block, it is merely a way of enforcing a structure on the input that is easy to understand from a human perspective. If the program specification contains only input-output pairs in the form of byte strings, it is impossible to know what the correct implementation is for the parser since it is entirely relative, and the specification format does not capture the original interpretation as intended by the user. For these reasons, we have decided to hard-code the implementation of the parser to only accept packets with the following combinations of headers: ethernet, ethernet & IPv4, ethernet & IPv4 & TCP, ethernet & IPv4 & UDP.

As mentioned in subsection 2.3.1, a P4 synthesizer needs not just the bytes in the packet but context information about the receival of the packet as well as information about the state of the device prior to the receival. For the sake of simplicity, G4BE can only synthesize programs that are not dependent on such context information.

[1] https://gitlab.tudelft.nl/lois/g4be

Unlike for Probe-cpp, two synthesized solutions may be both correct with respect to the program specification, but they may not be equivalent. A P4 program is considered correct if the right bytes are being sent via the correct output port. However, the P4 language has many instructions that do not directly lead to an observable output. Consider the two programs in listings 4.1 and 4.2. Although they lead to the same packet being sent via the same port, the two programs are not equivalent, and both should be stored in the bank as promising sub-solutions. One counts the number of packets seen while the other stores the old source MAC address in a temporary variable. Hence, both could be used in the future as subtrees of bigger programs. We call these programs weak-equivalent.

Listing 4.1: Program 1

```
1 standard_metadata.egress_spec = 1;
2 bit<48> tmp;
3 tmp = hdr.ethernet.srcAddr;
4 hdr.ethernet.srcAddr = hdr.
      ethernet.dstAddr;
```

Listing 4.2: Program 2

```
1 meta.counter = meta.counter + 1;
2 hdr.ethernet.srcAddr = hdr.
      ethernet.dstAddr;
3 standard_metadata.egress_spec = 1;
```

Just like Probe, G4BE discards programs that are not promising but only if they are strong-equivalent to an already-seen program. Strong equivalence implies the same observable behavior and the same side-effects. In our case, two P4 programs are strong-equivalent if they output the same packet via the same port and additionally, they bring the P4 device in the same state.

4.1.2. Evaluating candidate P4 programs

Since G4BE is based on Probe-cpp, all subtrees must be executable and must return values in a format that can be processed further by a larger program. Therefore, each subtree must represent a standalone program that can be integrated into a larger program. However, the P4 language has components that are mutually dependent but reside on different regions of the code. For instance, the tables being referenced in the control block are not declared in the control block but above it.

Let us assume a CFG that includes the production rules shown below. These rules should guide the synthesizer to enumerate subtrees that can later be included in the larger final solution. It is important to note that bottom-up enumeration builds subprograms without knowing the context in which they are going to be used in the future. Hence, it is impossible to enumerate subtrees that have *IngressControlBlock* as a start symbol, are independent and at the same time, apply match-action tables that are declared elsewhere. Because the subtrees must be evaluated, not just the name of a table is required but also its functionality and thus, one cannot assume the existence of a predefined number of tables with generic names that are going to be synthesized at a later stage.

IngressControlBlock	$\longrightarrow$	apply { Instructions }	The apply section of the ingress block
Instructions	$\longrightarrow$	Instruction	A single P4 instruction
	|	Instructions Instruction	A chain of instructions
Instruction	$\longrightarrow$	ConditionalStatement	If-then-else block
	|	AssignmentStatement	Assign value to a field, variable, etc

This problem also occurs for actions, variables, registers, and meters. The synthesizer builds actions by choosing an action signature at the root-level of the tree and using instruction blocks that were enumerated earlier as subtrees. At the time of their synthesis, these instruction blocks did not know about the existence of any action parameters, their type, or their concrete values. As a consequence, an intermediate grammar is needed that is equally expressive as the P4 grammar but circumvents these obstacles. Subsection 4.1.5 covers this idea in more detail.

G4BE uses a standardised object called *ExecutionState* to represent the outcome of any subtree, no matter the start symbol. This data structure is depicted in figure 4.1 along with other structures used to simulate the execution of a P4 program. Because of the issue explained in the previous subsection, this class needs to be quite verbose and must account for all possible effects a subtree may have. In the interest of simplicity, G4BE can only synthesize subtrees that return a concrete value (like the result of

an arithmetic operation), drop packets, or modify fields in the header, metadata or standard metadata structures.

At the start of G4BE's execution, each input-output pair from the program specification is transformed into a pair composed of an initial execution state and expected execution state. A synthesized program is correct if, for each of the starting execution states, it ends in an execution state that is weak-equivalent to the respective expected execution state. A candidate program is discarded if it leads to a set of execution states, one for each input from the specification, that are all strong-equivalent to the ones from a different set belonging to an already enumerated program.

In order to reduce space usage, instances of *ExecutionState* only store the values of the different fields without knowing their interpretation. It does so by maintaining instances of the *P4DataStructure* class. The information about a specific field, such as its name and bit width, is the same for all instances of that field and thus, it is stored only once using the *StructInfo* class. One notable member of the *StructInfo* class is the access variable which allows us to embed some domain knowledge into the synthesizer and steer the enumeration process to some extent. For instance, having the synthesizer attempt to figure out when and how to modify the total length field from the IPv4 header is wasteful since its derivation is fixed for all IPv4 packets and a hard-coded code snippet could be added to the solution.

More often than not, the bottom-up synthesizers researched within the program synthesis field rely on domain specific languages that do not need shared or global state. The grammar shown in appendix A is a good example since every production rule computes a single output from the outputs of a few independent subtrees. Two subtrees of the same AST operate on completely different areas of the memory and never interfere. Bottom-up synthesizers enumerate these subtrees separately and also evaluate them separately, each on the original unmodified input. However, P4 programs usually work extensively with the global state, most frequently through the header and metadata structures. This observation becomes problematic when we consider an AST with subtrees that access the same parts of the global state. Regardless of the order of execution, there is the risk of running into read-after-write hazards (RAW) where the one subtree modifies a part of the global state that the other subtree depends on, invalidating its prior evaluation. The following two subsections cover two approaches that were considered to solve this issue, namely recomputation and reconciliation.

4.1.3. Recomputing subtrees

The most straight-forward solution to the problem mentioned above is to recompute one subtree as if it was run right after the other one. Assuming left-to-right execution, the right sibling would be evaluated again but starting from the final execution state of the left sibling. It should be emphasized that this is required only in the case of a RAW hazard. If the two subtrees access different parts of the global or shared memory, their final execution states obtained at the time of their synthesis can safely be combined through superposition and there is no need for recomputation. Write-after-write (WAW) hazards also do not pose any threats since the evaluation of the right sibling does not depend on that of the left sibling whose effects are going to be overwritten anyway by the right sibling.

The same check for a RAW hazard can be applied recursively to only recompute the parts of the subtree that actually cause the RAW hazard. Figure 4.2 illustrates this idea with a conceptual AST for which some parts are not shown. It is important to note that P4 is a relatively complex language, making it nearly impossible to define a non-ambiguous grammar for it. The structure of an AST depends heavily on how the grammar was defined and on the order in which programs are synthesized. Nevertheless, the issues and approaches discussed in this subsection and the next are agnostic on those factors.

In figure 4.2, the red nodes need to be recomputed, the yellow nodes may or may not while the gray ones do not. The left-most subtree modifies the time to live (TTL) field from the IPv4 header and subtree 2 depends on it which means that a call will be made to subtree 2 to recompute based on the final execution states of subtree 1. The recomputation follows a top-down path and at each step a check is made to find potential RAW hazards. Subtree 2a does not depend on the TTL field and thus, it does not need to be executed again. In contrast, subtree 2b reads the TTL field and assuming the branch is actually taken, a recursive call will be made to it to recompute.

Subtree 3 does not depend on the TTL field, but it does depend on the metadata field which is potentially modified by subtree 2. The original evaluation of subtree 3 is checked against the final execution states

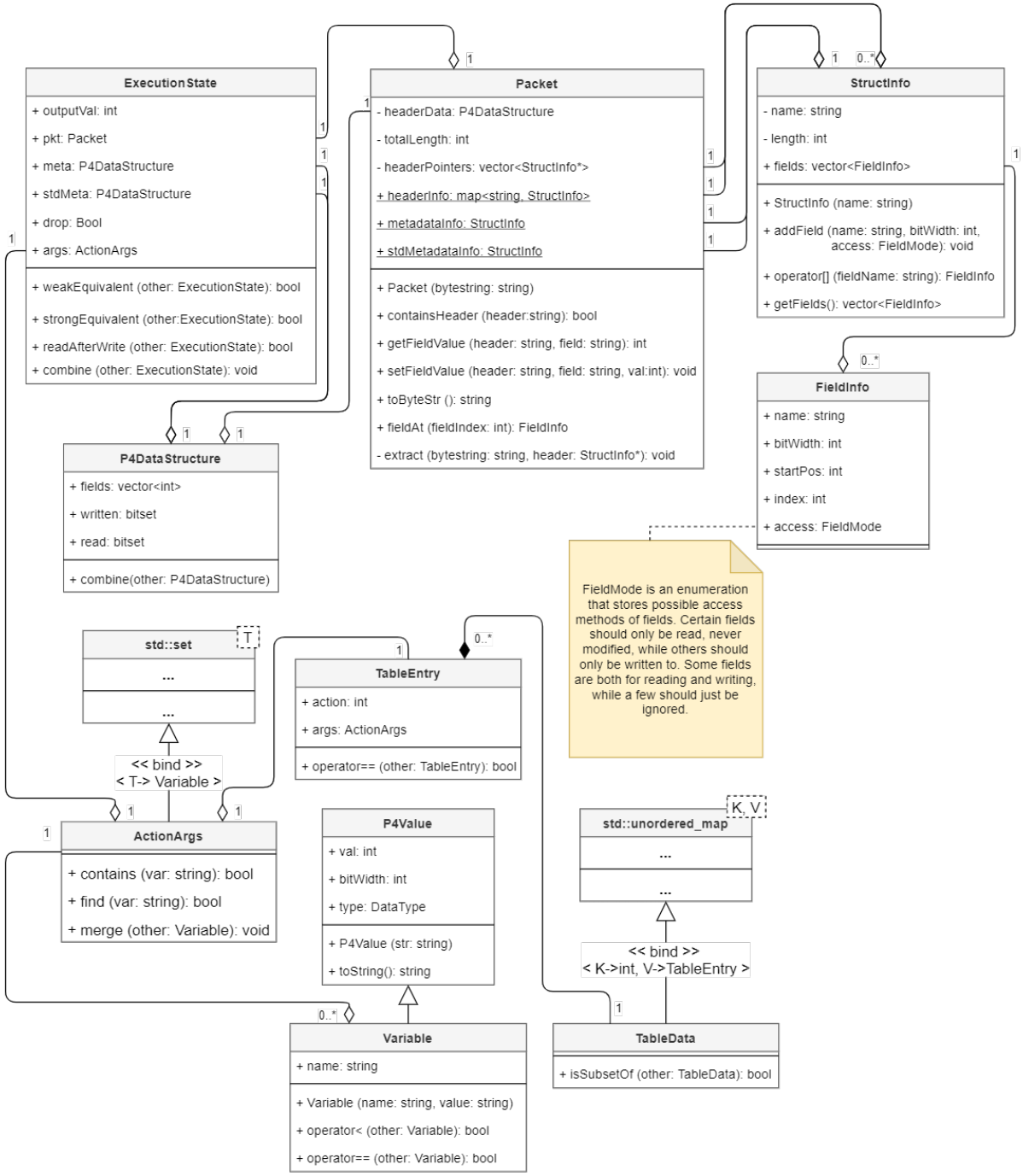


Figure 4.1: Data structures used to represent the execution of a P4 program

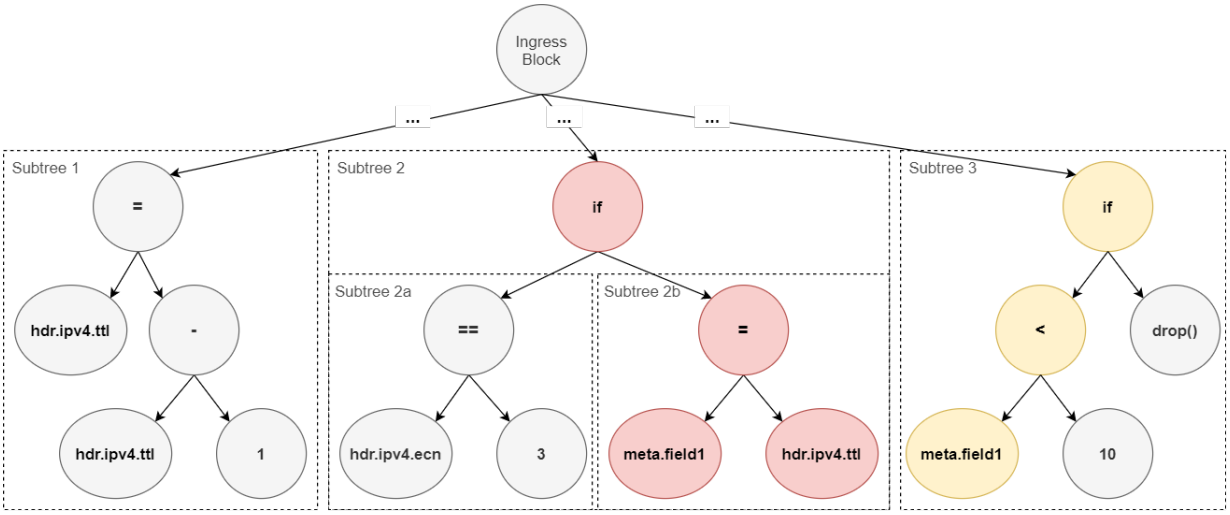


Figure 4.2: Recomputation of an example AST

obtained after the recomputation of subtree 2. Although a RAW hazard may not have existed initially, it may occur now as a consequence of the recomputation.

All in all, this efficient evaluation method alleviates much of the problem described in the previous subsection. Looking at figure 4.1, the *P4DataStructure* class enables this recomputation approach with two member variables, written and read, which are used to indicate which fields were read and/or modified during the execution.

Unfortunately, there is a serious complication with this idea that is difficult to spot. The program that is synthesized is not the program that is evaluated since the synthesizer generates a solution under the assumption that all subtrees run separately and ”from scratch”. One would question whether this has any effect, since the enumeration is still exhaustive but unfortunately that is not correct. Due to G4BE’s observational equivalence check, the existence of a subtree is conditioned on the uniqueness of its evaluation. Let us exemplify using the P4 code snippet shown in listing 4.3. Figure 4.3 depicts a sketch of the corresponding AST which is purposely left incomplete because the missing details bear no significance for this argument.

Listing 4.3: Example of an ingress control block in a P4 program

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action m_action() {
        standard_metadata.egress_spec = 1;
        meta.counter = hdr.ipv4.ttl - 1;
        ...
    }

    apply {
        m_action();
        if (meta.counter < THRESHOLD) {
            hdr.ipv4.diffserv = 0;
            ...
        } else {
            hdr.ipv4.diffserv = 10;
            ...
        }
    }
}
```

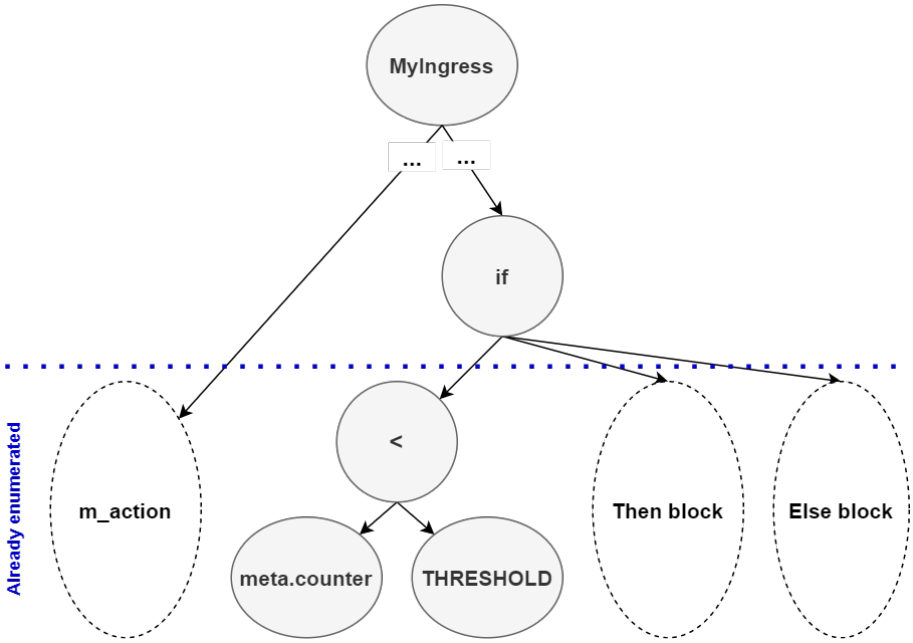


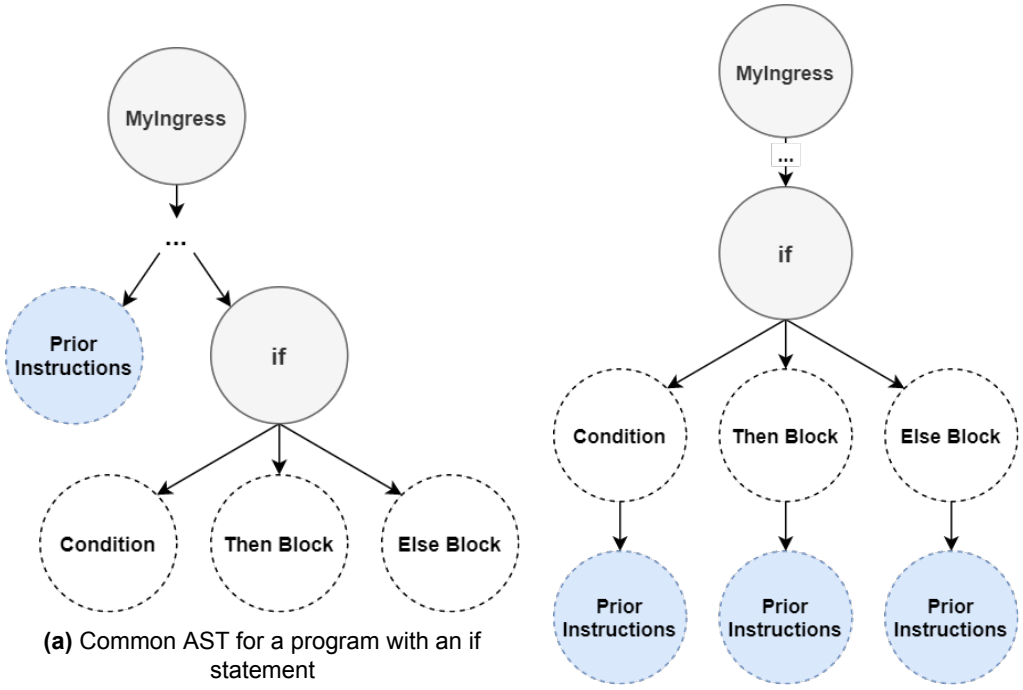
Figure 4.3: Conceptual AST of the program in listing 4.3

Let us assume the program shown is the desired solution. Recall that G4BE will enumerate programs bottom up, composing larger programs by applying all possible production rules together with the subtrees enumerated in the past. Assume that all subtrees in figure 4.3 that are below the dotted blue line have already been enumerated and are stored in the bank. G4BE now has all building blocks to create the next component of the final solution, which is the subtree with a conditional statement at its root. However, the evaluation of this subtree will be strongly equivalent to the evaluation of the *Then* block since the condition of the if statement is always true, making the *Else* block completely redundant. This occurs because the initial evaluation of any enumerated program begins ”from scratch” (from the initial execution states) and all metadata fields are set to zero by default.

The subtree at hand is discarded even though it may actually be useful in the future when a larger tree is created that actually modifies the *meta.counter* field prior to the if statement. In general, the sub-solutions that can be used to compose larger programs, are filtered based on their evaluation obtained from running them in isolation. A discarded sub-solution may still have a unique functionality, but it may only be visible once placed in a specific context. This approach invalidates Probe’s guarantee for completeness and, by extension also G4BE’s. One could question whether the test for observational equivalence can be omitted altogether and although it would solve the issue, the number of programs stored at each level would increase drastically. The search space would explode much earlier and quicker and as a consequence, the performance would degrade considerably.

4.1.4. Reconciling subtrees

The problem described earlier is part of a broader context. If the desired solution has a branch at the end of a long sequence of instructions, the program that a Probe-like synthesizer will come up with will inevitably suffer from duplication. The code snippets in listings 4.4 and 4.5 build the intuition behind this statement. The most intuitive way to represent the program on the left would be with an AST that looks like shown in figure 4.4a. Sadly, Probe may never find this AST because of the reasons mentioned in the previous subsection. Moreover, if one of those instructions prior to the if statement declares a variable that is referenced by one of the instructions in the conditional block, a synthesizer like Probe will certainly not find it. That would require the bank to store subtrees that are self-contained, yet reference variables that are not declared at the time of their synthesis, which is a contradiction.



(a) Common AST for a program with an if statement

(b) Equivalent AST enumerated by G4BE

Figure 4.4: Difference in AST brought by bottom-up enumeration paired with observational equivalence

Listing 4.4: Desired solution

```
1 ...
2 meta.foo = meta.foo - 1;
3
4 if (hdr.ipv4.protocol == 256) {
5     meta.bar = meta.foo * 2;
6 }
7 else {
8     meta.bar = meta.foo * 4;
9 }
```

Listing 4.5: Probe's version

```
1 if (hdr.ipv4.protocol == 256) {
2     ...
3     meta.foo = meta.foo - 1;
4     meta.bar = meta.foo * 2;
5 } else {
6     ...
7     meta.foo = meta.foo - 1;
8     meta.bar = meta.foo * 4;
9 }
```

A bare-bones G4BE driven only by the Probe synthesizer will eventually synthesize an equivalent program for which the instructions prior to the conditional statement are actually contained in each of the subtrees. This program is depicted in listing 4.5 and its AST looks like shown in figure 4.4b. To enable this approach, one needs to define a grammar that allows for a different form of instruction chaining compared to the way instructions are chained by the CFG shown in subsection 4.1.2. This topic is discussed in more detail in the following subsection. Note that this method may not directly degrade performance but it de-prioritizes programs that exhibit the same particular pattern of branches having a common past. The duplicated code is not synthesized multiple times, but the program stores multiple references to the same subtree increasing its total cost and lowering its priority.

Despite all of the above, two subtrees accessing the same parts of the global/shared state is a problem that remains unsolved. Reconciling subtrees implies adding, moving, and rewriting instructions according to a few static rules, such that the dependencies between the siblings are resolved without the need for recomputation. In fact, this approach accepts Probe's synthesis and evaluation model and once a solution is found it applies specific transformations to obtain the P4 code that is equivalent in functionality. These transformations are not needed during the synthesis process and thus, they do not degrade performance.

Assume two subtrees that are siblings, with the left sibling writing to a field that the right one reads. By re-using the old evaluations which are computed in isolation, G4BE essentially assumes that the two

subtrees operate on different copies of the same field. The equivalent P4 code can be obtained by creating a temporary variable that holds the original value of the said field before being modified by the left subtree. The references made by the right subtree to read the field in question need to be modified to references to the newly created variable. By doing that, a P4 program can be obtained that has the same functionality as the synthesized AST. Furthermore, if the solution obtained by G4BE has an AST similar to the AST shown in figure 4.4b, G4BE could detect the duplication and can move the common instructions to a single location.

4.1.5. Intermediate grammar

The main idea behind the intermediate grammar is to define it such that all information needed by the root of a subtree is included in that subtree. To achieve this, production rules must be written in a manner that allows any AST node representing a P4 instruction to have as a child, a subtree representing all the instructions preceding it. In this grammar, the closer a node is to the bottom of the tree, the earlier it comes in the execution. ASTs generated by this type of grammar are essentially upside-down relative to the natural top to bottom way in which programs are written.

A small part of the grammar used by G4BE is shown below with terminals being colored gray. Notice that instructions are chained through the *Operand* non-terminal which has a reference to the *Instruction* non-terminal. This construction allows G4BE to synthesize instructions (assignment instructions, boolean conditions, arithmetic instructions, etc) for which the operands are the result of other subtrees. Looking at the code in listing 4.6, the instruction on line three depends on the preceding two instructions. The AST that corresponds to this code snippet is shown in figure 4.5.

Instruction	$\longrightarrow$	(assign Field ArithmeticInstr)	assign a value to a field
	|	(if BooleanCondition Instruction Instruction)	if-then-else block
ArithmeticInstr	$\longrightarrow$	Operand	
	|	(Operand + Operand)	simple addition of operands
Operand	$\longrightarrow$	(get Field after Instruction)	get field value
	|	(get @argument after Instruction)	get action argument value
	|	(return Const after Instruction)	return constant value

Listing 4.6: Sample sequence of P4 instructions

```
1 meta.foo = hdr.ipv4.ttl - 1;
2 meta.bar = 20;
3 meta.baz = meta.foo + meta.bar;
```

The intermediate grammar must also address the issue of components being referenced in a different part of the subtree than where they are declared. There are essentially two approaches, and both are used by G4BE. The first method assumes the existence of a fixed number of instances of the needed component. For instance, a register can be assumed to be declared at the top of the ingress block. After a solution is synthesized, a check can be made to trim off the components that were declared in anticipation but are not actually referenced anywhere in the solution. This method is simple and effective but has the obvious drawback that it limits the space of possible solutions. Furthermore, it gets a little more complex when considering the different scopes that components may have, such as temporary variables.

The other approach is to consider the first reference to a component to also act as the declaration. If necessary, the information about the declared component is propagated further up the tree through the final execution state. Action arguments form a fitting example for this method. Recall that subtrees are built without knowing in which context they are going to be used later in the synthesis process. A subtree can be built that makes a symbolic reference to an action argument as if it were a regular operand. The *args* data structure from the *ExecutionState* class (see 4.1) is used to carry the information upwards about the declaration and use of a specific argument.

G4BE could enable this approach through the use of synthesizer terminals which are terminals whose value and meaning the synthesizer must resolve on the fly, in between the enumeration of the containing AST

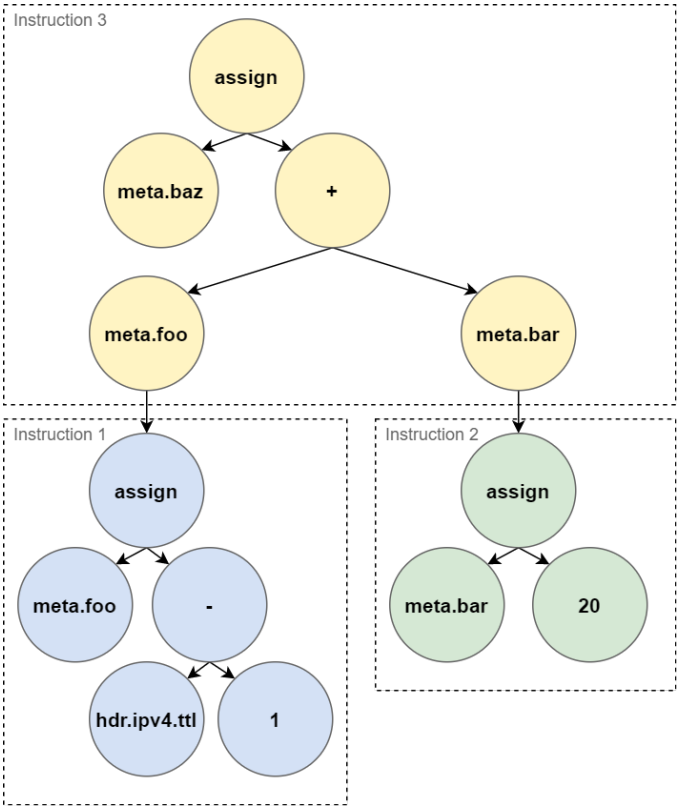


Figure 4.5: AST of the code in listing 4.6

and its evaluation. Looking at the partial grammar above, let us assume that the synthesizer constructs an AST by taking subtrees from the bank and applying the sixth rule from the top which includes the *@argument* synthesizer terminal. It then creates variations of this AST where this synthesizer terminal is replaced with a reference to either a new argument or to an argument declared previously by the subtrees. If a new argument is declared, further variations are made for each possible value that the argument can take. These potential values are taken from a set of terminals provided by the user through the program specification. All the obtained variations are then evaluated independently.

It is important to not propagate the list of action arguments outside of the scope in which they can be used. For instance, a production rule that calls an action does not not need to pass on to the encompassing AST, the list of arguments which were used by the action, because they fall out of scope.

Sadly, synthesizing actions is not currently implemented in G4BE but the method described above provides a clear idea on how to do it. Nevertheless, actions can be provided by the users as code hints, a feature that will be described in section 4.2.

Tables are defined in the same place in the tree as where they are applied. In turn, tables reference actions whose signature and functionality is fully captured by the subtrees of the AST in question. Subsection 4.3 describes in more detail how match-action tables are synthesized and populated with data.

With the described intermediate grammar, the textual representation of a candidate program is very dissimilar to the corresponding actual P4 program. Below a program is shown that decrements the *TTL* field, and then sends the packet through the port that it was received on. Static rules can be applied to transform the final solution from this format to valid P4 code. It is during this step that subtrees must be reconciled as discussed in the previous subsection.

(assign standard_metadata egress_spec (get standard_metadata ingress_port after (assign hdr ipv4 ttl ((get hdr ipv4 ttl after ()) - (return 1 after ())))))

Unfortunately, chaining instructions as shown above causes dependencies other than RAW to also be problematic. In fact, unless all siblings read without writing, reconciliation is needed. Figure 4.6 illustrates a

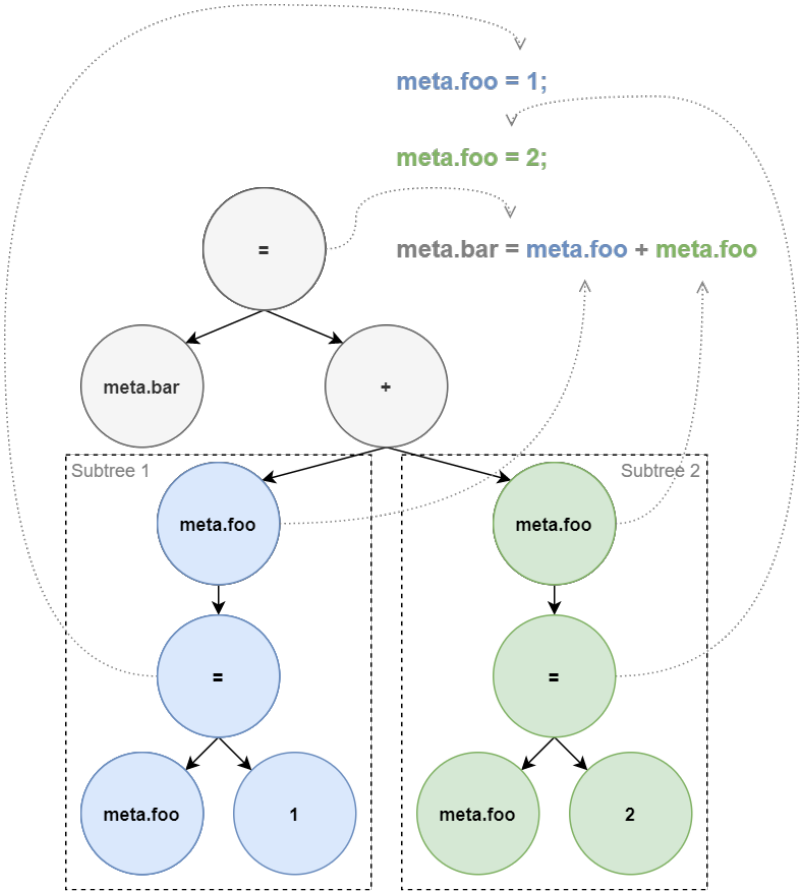


Figure 4.6: Example of a Write-After-Write hazard

scenario in which there is a WAW hazarad that must be resolved. The issue stems from the fact that the two subtrees write to the same field and end up intertwined in the final sequence of instructions. After executing the instructions shown in figure 4.6, the field *meta.bar* will hold the value 4, while in the synthesizer's evaluation model, it will be 3. Reconciliation solves this mismatch by creating separate temporary variables for both subtrees, as shown in listing 4.7.

Listing 4.7: The code represented by the AST in figure 4.6

```
1 aux1 = 1; // auxiliary variable for meta.foo for subtree 1
2 meta.foo = aux1;
3 aux2 = 2; // auxiliary variable for meta.foo for subtree 2
4 meta.foo = aux2;
5 meta.bar = aux1 + aux2;
```

4.1.6. Reducing the hypothesis space

Although it may seem insignificant at first, the layout of the grammar plays an important role in the shape and size of the hypothesis space. Assume we modify the grammar shown above by replacing the right-hand side of the first production rule, with a new non-terminal, *IfThenElse*, and we add a new production rule that allows *IfThenElse* to expand to the right-hand side that was just removed. This may seem a harmless change, but it adds to the cost of the programs that use if statements. The cost of a program is not an indication of the time it takes to synthesize it, but rather the place it takes on the enumeration order. Increasing the costs of all production rules simultaneously has little to no impact on the synthesizer's performance. However, changing the cost of just one instruction de-prioritizes programs that rely on that instruction relative to the programs that do not. This bias in the PCFG is greater than the bias that can be learned by the just-in-time learning feature of Probe. The grammar used by G4BE is defined with this

consideration in mind and with the goal of minimizing the number of production rules, while keeping it relatively intuitive.

Grammar ambiguity is another factor that has a significant impact on performance. The more ways there are to express the same functionality, the more equivalent programs are going to found just to be discarded by the observational equivalence check. This issue is exacerbated by the existence of any syntactic sugar in the underlying language. Making the grammar as simple as possible and pruning away redundant regions of the hypothesis space can significantly improve the performance of any algorithm performing an exhaustive search on that space. Unfortunately, this topic remains mostly unexplored with some exceptions described below.

G4BE prioritizes complete programs over longer but incomplete ones. To this end, production rules that are unavoidable are assigned a fixed minimum cost, thereby forcing the synthesizer to try them frequently to find potential solutions as early as possible. To illustrate, the default P4 switch will drop packets if the output port is not explicitly set and thus, the rule that sets the output port is unavoidable. Furthermore, the redundancy in the language is reduced by minimizing the height difference between a sequence of instructions and the complete solution containing that sequence.

Each field (be it header, metadata, or standard metadata) is assigned an access mode which restricts the list of instructions that can be synthesized and how they access the said field. As an example, no instructions will be synthesized that write to a field that is marked as read-only. A field can be marked as read-only, write-only, read-write or ignore. Some fields are assigned fixed access modes that stem from our domain knowledge. For instance, IPv4's checksum field is not usually read nor modified in the ingress and/or egress blocks of a P4 program but rather in the designated blocks, one for verifying its correctness on ingress and one for updating it at the egress. There are standard algorithms that are applied in these blocks and therefore, we decided to reduce the hypothesis space by hard-coding these blocks and ignoring the checksum field.

By inspecting the difference between the input and the output packets from the program specification, one can determine which fields need to be modified and which are likely to be read. If one knows that a field is not going to be modified because the input and output have exactly the same value for it, there is no reason to synthesize instructions that write to it. However, the packets from the program specification are mere byte strings and without explicit knowledge about their header structure, such a comparison is impossible. Therefore, G4BE makes the simplifying assumption that the header structure does not change from input to output which effectively implies that G4BE cannot synthesize programs that add new headers or remove existing ones.

4.2. User provided code snippets

Although powerful and quite versatile, enumerative synthesizers quickly suffer from the exponential size of the search space. Just a sequence of a few instructions and a small sized grammar can already yield millions of possibilities. The problem only gets worse if there are also parameters and constraints that need to be found separately such as the runtime rules described in the next section.

In order to speed-up the synthesizer, we implemented a feature that allows the user to provide code snippets (or code hints) to the synthesizer that can be parsed into an internal format and subsequently used to enumerate larger programs. The *P4Parser* component takes a string representing a snippet of P4 code and parses it into an AST using the internal P4 grammar that is also used to enumerate P4 programs. The AST is saved in the bank of promising sub-solutions, and it is assigned a fixed low cost, independent of the cost of the production rules that it consists of.

A code hint is provided by the user with the intention to be used anywhere in the solution. A user may provide as a hint, an action that is needed by a table that is applied at the end of a long sequence of instructions. The user provides the action believing it may potentially be used on top of any sequence of instructions that the synthesizer may enumerate. However, in the evaluation model discussed in subsections 4.1.4 and 4.1.5, any subtree (including the one representing the action) is self-contained and is evaluated ”from scratch”, always yielding the same output no matter the context in which it is placed. In other words, the evaluation model mentioned earlier does not match with the user's intuition when providing the code hints.

To alleviate this issue, code hints are marked with the flag *alwaysRecompute* which ensures that the corresponding subtrees are evaluated differently from the rest. Such subtrees are evaluated much like explained in subsection 4.1.3, being executed as if they were an extension of their left sibling. Since this feature does not change G4BE synthesis algorithm, it does not bring the same issues as discussed in subsection 4.1.3.

Through this feature, we envisioned the possibility of mining a corpus of P4 code for re-usable code snippets that are likely to be useful for the program specification at hand. This search could be made using stochastic tools such as artificial neural networks (ANN) that learn the correlation between certain instructions and modification patterns made to the input packet. Given the difference between the input packet and desired output packet, the ANN could predict which code snippets may be useful, essentially enhancing G4BE's runtime performance while maintaining the completeness guarantee.

4.3. Enumerating the rules for the match-action tables

As mentioned earlier, the behavior of a P4 program is not solely defined by the P4 code but also by the data contained in the used match-action tables, which we refer to as runtime rules. These table entries are provided separately through a file at start-up or, in the context of SDN, they are installed and removed by the orchestrating controller.

The established objective, as stated in chapter 1, requires that G4BE find a complete solution that is provably compliant with program specification, as given by the user. Hence, one must synthesize both the P4 code and the runtime rules if one wants to evaluate the solution and prove its correctness.

Listing 4.8 shows a JSON object that installs a rule in the *ipv4_routing* table. If the table is applied to a packet with the shown destination IP address, the action *forward* is called with 1 as a value for its *port* parameter.

Listing 4.8: Example runtime rule

```
{
  "table": "MyIngress.ipv4_routing",
  "match": {
    "hdr.ipv4.dstAddr": "10.0.1.1"
  },
  "action_name": "MyIngress.forward",
  "action_params": {
    "port": 1
  }
}
```

Let us assume a program was synthesized by applying the example production rule shown below which declares and also applies a table that has two possible actions that can be applied. The *Instruction* non-terminal on the right-hand side makes it so that instructions prior to the table are represented as subtrees of the bigger AST. This table's behavior can vary widely depending on how it is populated. Hence, G4BE must enumerate all variations of this table and evaluate each of the resulting program completely independently.

$$\text{Instruction} \longrightarrow \text{(table (match Field after Instruction) (action\_list Action Action)}$$

In essence, what must be enumerated is all possible mapping between the possible match values and the actions from the action list. One does not need to enumerate match values in a random fashion, since they can be extracted from the execution states that are obtained after evaluating the instructions just prior to applying the table.

Recall that the actions referenced by the table should already be evaluated independently and thus, the values for their arguments should already be resolved by the time the encompassing table is generated.

This is true under the assumption that G4BE synthesizes actions as described in subsection 4.1.5, which it does not in order to reduce the hypothesis space.

G4BE relies on the user to provide actions as code hints which can be used to generate tables. These code hints are recomputed depending on the context they end up in. With this approach, a specific variation of a table includes not just one concrete mapping between match values and actions but also specific values for the action parameters. The actions referenced by this specific variation are recomputed with the new arguments specific to this variation. This makes for a more intuitive synthesis process which generates a table and the data that populates it during the same step.

5

G4BE's evaluation

Four benchmarks were selected to evaluate our proposed solution and showcase the potential of using enumerative program synthesis and PBE for generating P4 code. These benchmarks are four small P4 programs that G4BE should be able to synthesize in reasonable time. The first step in our evaluation approach is writing the desired P4 solutions by hand followed by their installation on a reference P4 switch (the behavioral model 2). Scripts and terminal commands are used to generate traffic manually while a packet sniffer is used to capture both the input packet to the switch and the output packet as modified by the P4 code. The traffic consists of ICMP, TCP and UDP packets.

A script is used to find the input-output pairs from the captured traces and process them into program specifications that can be used by G4BE. To put it concisely, we aim to let G4BE synthesize a program that has the same behavior as the program that was written previously by hand.

Section 5.1 lists the benchmarks that were used while section 5.2 evaluates and plots G4BE's performance in terms of synthesis duration and memory usage. Section 5.3 ends this section with a few qualitative remarks.

5.1. Selected benchmarks

The benchmarks chosen for the evaluation of G4BE are listed in table 5.1 in increasing order of their difficulty. These were taken from public repositories with many examples of P4 programs[1, 2]. Appendix B contains a detailed description of the *basic* benchmark showing parts of the program specification, synthesized solution and runtime rules.

Benchmark	Functionality
min	Decrement the Time-to-live field and send packet though port 1
repeater	Repeat a packet on the port that it was not received on
reflector	Swap the MAC addresses and send the packet back through the same port it was received on
basic	Perform basic IPv4 routing: route the packet according to destination IP address and decrement TTL

Table 5.1: Chosen benchmarks for evaluating G4BE

The runtime rules are an important part of the solution, and they can have large effect on the performance. The higher the number of runtime rules that can be synthesized, the longer the execution. In the case of the *basic* benchmark, the number of runtime rules is dependent on the size of the network due to the fact that a larger network will have more MAC & IP addresses that must be accounted for. To evaluate the impact of the network topology on the execution duration, we let G4BE synthesize basic.p4 for the three topologies shown in figure 5.1. In all three cases, the target switch that must be programmed is *S1*. Note

[1] https://github.com/p4lang/tutorials

[2] https://github.com/nsg-ethz/p4-learning

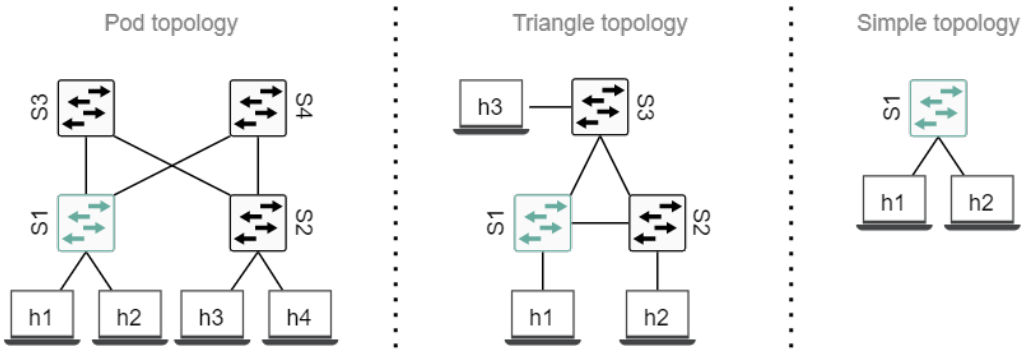


Figure 5.1: Three topologies used for the basic benchmark

that G4BE has no information about the network topology aside from the data contained in the input-output examples and the terminals included in the program specification.

5.2. Runtime performance and memory usage

In order to accurately measure G4BE's performance, we ran it 20 times for each benchmark and measured the execution time and the memory usage. The first two measurements were thrown away to account for potential warm-up effects. These experiments were performed on the same system that was described in section . The synthesis duration for the various benchmarks are plotted in figure 5.2. Likewise, the memory used by G4BE to synthesize the solutions is shown in figure 5.3.

For small benchmarks that require G4BE to synthesize a maximum of two to three instructions, the execution time is almost instantaneous. However, the execution time quickly starts to grow beyond solutions of two instructions. It is also important to note that the default P4 grammar used by G4BE is minimal, but its size can vary. As mentioned in subsection 4.1.6, certain instructions and header fields may be ignored because they may not be deemed useful. Furthermore, the default grammar is also expanded with the terminals and code hints provided through the program specification.

For the *basic* benchmark which makes use of a match-action table, the synthesis duration is really affected by the number of table entries that must be synthesized. G4BE must synthesize four table entries for the pod topology, three for the triangle topology and two for the simple topology. Looking at the boxplots in figure 5.2, one can notice that every additional runtime rule that must be synthesized increases the execution duration with one order of magnitude.

Due to the exponential nature of the solution space, G4BE has an exponential space complexity and that is consistent with figure 5.3. The memory requirement grows quickly for relatively small programs which is rather worrying. The memory efficiency of G4BE is rather poor which is part in due to the large data structure needed to store an execution state, as was explained in subsection 4.1.2. Nevertheless, the implementation itself is far from being perfect which only aggravates the problem. It should be pointed out that G4BE was meant only as a proof of concept and good coding practices were at times ignored due to the limited scope of this project.

Figure 5.4 offers more insight in how the solution space grows for the various benchmarks. The figure plots the cumulative number of programs enumerated in time, measured in the number of synthesis steps passed. Recall from chapter 3 that every synthesis step corresponds to an increment of the target program cost. However, the learning feature of Probe (and G4BE) may cause the enumeration to restart, resetting the target program cost and the number of synthesized programs back to 0. This is precisely the reason for the sudden drop in the number of enumerated programs for the *repeater*.

Looking at the *basic* benchmark, the provided code hint makes it so that a match-action table can be enumerated rather early (i.e., at a low program cost). In general, as soon as G4BE can enumerate a program with a match-action table, it will also enumerate all possible variations of that table that have the same cost. For every table variation, G4BE will also synthesize all possible ways to populate it. These two factors explain the explosion in the number of programs synthesized so early in the enumeration process.

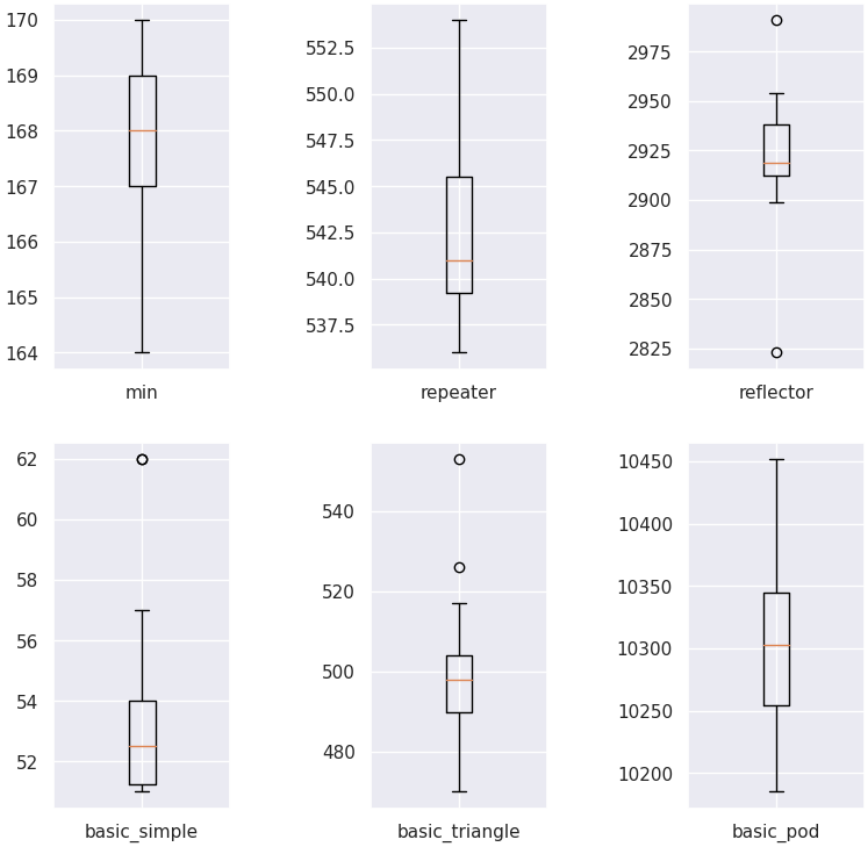


Figure 5.2: Runtime performance of G4BE in milliseconds for the various benchmarks

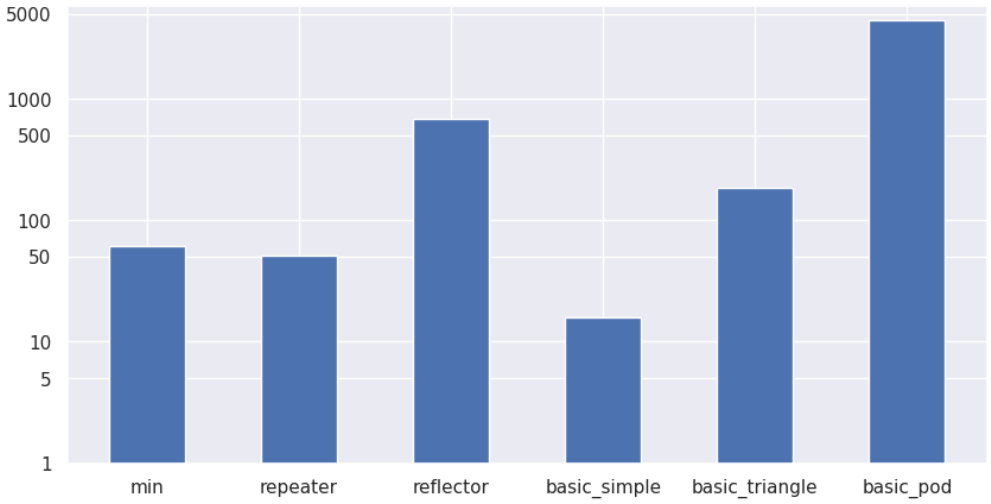


Figure 5.3: G4BE's memory usage in MB for the various benchmarks

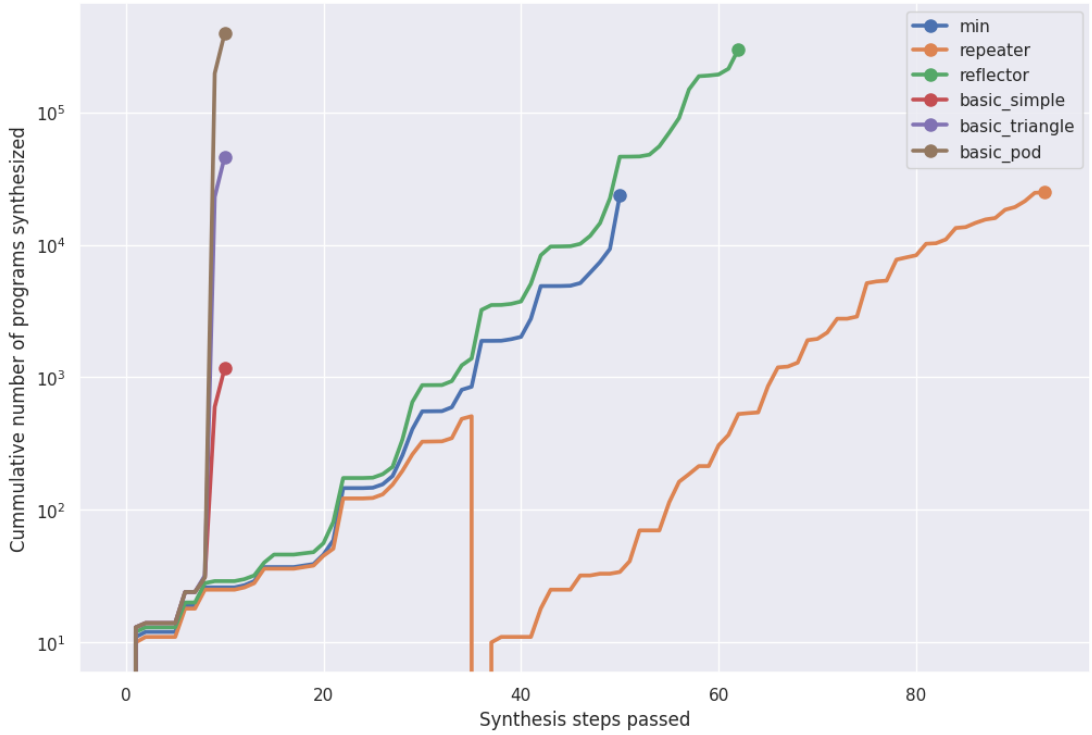


Figure 5.4: Number of programs synthesized in time

5.3. Observations

What stands out from the previous section is that G4BE is not immune to the problem of state explosion. Even though *basic.p4* is rather short in terms of lines of code, the number of programs found at each cost explode rapidly because of the many different ways to populate the match-action table. Even with a very efficient implementation and with all the simplifications, G4BE is still not immune to the problem of state explosion.

Let us illustrate with an example of a table with one possible action and four match values. The action takes two arguments which in turn, can have four possible values. Doing the math, this leads to $(4^2)^4 = 65536$ variations of this table that need to be considered. One needs to realize that for the same cost, multiple programs can be created that have the same table but with a different match field. Therefore, the number of variations derived earlier needs to be multiplied by the number of fields that can be considered for a match. P4 also allows matching on multiple fields at once but to prevent the hypothesis space from growing even further, G4BE only enumerates tables that match on a single field and only with the *exact* match type.

G4BE does not synthesize an instruction that is not needed in order to comply exactly with the program specification, even if the instruction may be crucial in some cases. The quality of the solution is directly correlated to the completeness of the set of examples. To illustrate, an if statement that checks the validity of the IPv4 header before applying a table, may be omitted, if there are no examples in the program specification without an IPv4 header. Similarly, a header field whose value is the same across all input-output examples, may be used as an operand in instructions that are semantically unrelated.

6

Closure

In this chapter, we reflect on the performance of the proposed solution in the frame of the original research questions stated in the introduction. Afterwards we discuss some limitations and some suggestions for future research.

6.1. Discussion

During the development of G4BE, we have faced unexpected challenges that resulted in repeated simplifications of the scope of the project in order to keep the initial research objective attainable. Our knowledge on the subject has changed and therefore, it is necessary to repeat the research questions stated in the introduction and try to answer them one by one.

Q1. Are traces of input packets and their corresponding output packets, an effective way to specify what a P4 program should do?

In retrospect, the format for the program specification is a very suitable choice for our problem domain, but there are some limitations to consider. P4-enabled switches can hold state and their behavior is based on both the incoming packets as well as the state of the switch at the time of receival. Therefore, input-output examples may have to also include information about the prior state of the switch. For switches that perform stateless functions, pairs of input-output packets can perfectly capture the desired functionality.

Q2. Given a program specification in the form of a set of input-output packets, can we synthesize a P4 program in reasonable time that, when run on a programmable switch, will comply with the specification?

G4BE is a proof-of-concept that fits the description above but only for a few small programs. Enumerative program synthesis is a powerful technique if applied in the right context but even state of the art variants of this synthesis method can quickly run into the problem of the exponential search space. There is certainly potential in the use of enumerative program synthesis to automatically program P4-enabled switches, but generating long and complex P4 programs using algorithms such as Probe [7] is rather unrealistic and requires further research.

These enumerative techniques can be very fast, all while offering completeness and soundness guarantees. The conditions for this to occur is that the grammar is limited in the number of production rules (unlike P4 which is an extensive DSL) and that the desired solution has a small number of instructions. We believe more promising results could be obtained if these techniques would be applied to a lesser extent such as for filling in holes left by the developer, similar to SKETCH [20]. Another idea is using stochastic approaches to mine code blocks and letting Probe enumerate all programs that can be built with these blocks.

Q3. If a complying P4 program is found, how does it compare to a solution written by an expert?

Unfortunately, synthesizing complex programs with the use of enumerative program synthesis could potentially take far longer than it would take an expert to write a solution by hand. However, small programs such as *basic.p4* may take seconds to synthesize automatically but minutes to write by hand. In terms of solution size, G4BE is built on Probe which enumerates programs in increasing order of their cost/size. However, reconciliation adds to the length of synthesized solution compared to the manually written one, but this increase is minor.

Q4. Can the state-of-the-art enumerative program synthesizer be used for generating solutions in a DSL used in the real world?

A DSL such as P4 has a grammar that is much larger than the grammars used to evaluate Probe. Unfortunately, the number of production rules in the grammar has a crucial effect on synthesizer's performance both in terms of execution duration and memory usage. However, it is not necessary for Probe to be aware of the entire DSL in order to be useful. It is crucial to minimize the size of the grammar by removing all syntactic sugar, all ambiguities and all the different ways to express the same program.

DSLs may often rely on the use of global state while synthesizers like Probe may find it difficult to enumerate such programs for the reasons explained in section 4.1. Adjusting a enumerative synthesizer to account for the inner-workings of the execution environment may prove difficult and may require workarounds.

What also must be pointed out is that throughout the research on program synthesis, the impact of interpreting and evaluating candidate programs is often overlooked. By profiling the code, we find that G4BE spends roughly 40% of the total execution time running functions that simulate the P4 programs. Executing candidate programs on full-fledged emulators would have created serious bottlenecks and the performance would have degraded immensely.

6.2. Limitations and future work

Considering the unexpected difficulties and the limited time frame of this project, some simplifying assumptions were made and some of the initial goals were left out. These limitations are listed below for reference. While some limitations are just a matter of implementation, others can serve as challenges to be tackled by future research.

- Testing and validating the final synthesized program was done by hand by running G4BE's output on a software switch. Preferably, G4BE should use more robust ways to validate the final generated programs like the ones discussed in subsection 2.4.3.
- In the interest of reducing complexity, the multi-threaded feature of Probe-cpp was not added to G4BE which essentially means that performance was left on the table.
- G4BE synthesizes the ingress control block. All other components are hard-coded.
- The grammar used by G4BE leaves out many aspects of the P4 language. As an example, G4BE cannot synthesize programs that rely on registers, counters and/or meters. Actions are not synthesized but they can be provided as code hints by the user.
- Each field is represented by a 64-bit value. Manipulating arrays of 64-bit variables is a lot faster than manipulating arrays of variable length data types. In practice, some fields (such as the address fields in IPv6) may exceed this limit.
- The structure of the packet header is assumed to remain the same from input to output. The packets use either TCP/IP or UDP/IP. Although P4 allows to add or remove headers from the packet, G4BE does not synthesize such programs. Aside from reducing the hypothesis space, this restriction speeds up the comparisons between packets and by extension, the evaluation step, and the observational equivalence check.

Bibliography

[1] S. Feng and E. Seidel, “Self-organizing networks (son) in 3 gpp long term evolution,” 2008.

[2] N. McKeown, T. Anderson, H. Balakrishnan, *et al.*, “Openflow: Enabling innovation in campus networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008. DOI: 10.1145/1355734.1355746. [Online]. Available: https://doi.org/10.1145/1355734.1355746.

[3] P. Bosshart, G. Gibb, H.-S. Kim, *et al.*, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn,” *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, 2013. DOI: 10.1145/2534169.2486011. [Online]. Available: https://doi.org/10.1145/2534169.2486011.

[4] C. Kim, *Programming the network data plane*, Accessed: 2023-07-14, 2016. [Online]. Available: https://conferences.sigcomm.org/sigcomm/2016/files/program/netpl/netpl16-kim.pdf.

[5] S. Padhi, E. Polgreen, M. Raghothaman, A. Reynolds, and A. Udupa, *Portable switch architecuture*, Accessed: 2023-07-14, 2019. [Online]. Available: https://p4.org/p4-spec/docs/PSA.pdf.

[6] P. Bosshart, D. Daly, G. Gibb, *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[7] S. Barke, H. Peleg, and N. Polikarpova, “Just-in-time learning for bottom-up enumerative synthesis,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020. DOI: 10.1145/3428295. [Online]. Available: https://doi.org/10.1145/3428295.

[8] A. S. Tanenbaum and D. Wetherall, *Computer Networks*, 5th ed. Boston: Prentice Hall, 2011. [Online]. Available: https://www.safaribooksonline.com/library/view/computer-networks-fifth/9780133485936/.

[9] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[10] S. Vutukury and J. Garcia-Luna-Aceves, “Mdva: A distance-vector multipath routing protocol,” in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, IEEE, vol. 1, 2001, pp. 557–564.

[11] F. Baker, *Rfc1812: Requirements for ip version 4 routers*, 1995.

[12] J. T. Moy, *OSPF: anatomy of an Internet routing protocol*. Addison-Wesley Professional, 1998.

[13] O. Blial, M. Ben Mamoun, R. Benaini, *et al.*, “An overview on sdn architectures with multiple controllers,” *Journal of Computer Networks and Communications*, vol. 2016, 2016.

[14] B. Pfaff, J. Pettit, T. Koponen, *et al.*, “The design and implementation of open {vswitch},” in *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, 2015, pp. 117–130.

[15] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, “An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends,” *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021. DOI: 10.1109/ACCESS.2021.3086704.

[16] L. Tan, W. Su, W. Zhang, *et al.*, “In-band network telemetry: A survey,” *Computer Networks*, vol. 186, p. 107 763, 2021.

[17] H. T. Dang, P. Bressana, H. Wang, *et al.*, “P4xos: Consensus as a network service,” *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1726–1738, 2020.

[18] A. Sapio, I. Abdelaziz, M. Canini, and P. Kalnis, “Daiet: A system for data aggregation inside the network,” in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 626–626.

[19] S. Gulwani, O. Polozov, and R. Singh, “Program synthesis,” *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017. DOI: 10.1561/2500000010. [Online]. Available: http://dx.doi.org/10.1561/2500000010.

[20] A. Solar-Lezama, “Program synthesis by sketching,” AAI3353225, Ph.D. dissertation, USA, 2008.

[21] J. R. Koza, “Genetic programming as a means for programming computers by natural selection,” *Statistics and Computing*, vol. 4, no. 2, pp. 87–112, Jun. 1994. DOI: doi:10.1007/BF00175355.

[22] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, “Deepcoder: Learning to write programs,” in *Proceedings of ICLR’17*, 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/.

[23] O. Polozov and S. Gulwani, “Flashmeta: A framework for inductive program synthesis,” *SIGPLAN Not.*, vol. 50, no. 10, pp. 107–126, Oct. 2015. DOI: 10.1145/2858965.2814310. [Online]. Available: https://doi.org/10.1145/2858965.2814310.

[24] M. Mayer, G. Soares, M. Grechkin, *et al.*, “User interaction models for disambiguation in programming by example,” in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST ’15, Charlotte, NC, USA: Association for Computing Machinery, 2015, pp. 291–301. DOI: 10.1145/2807442.2807459. [Online]. Available: https://doi.org/10.1145/2807442.2807459.

[25] R. Alur, R. Bodik, G. Juniwal, *et al.*, “Syntax-guided synthesis,” in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385.

[26] Y. Han, J. Li, D. Hoang, J.-H. Yoo, and J. W.-K. Hong, “An intent-based network virtualization platform for sdn,” in *2016 12th International Conference on Network and Service Management (CNSM)*, 2016, pp. 353–358. DOI: 10.1109/CNSM.2016.7818446.

[27] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, “Refining network intents for self-driving networks,” in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, ser. SelfDN 2018, Budapest, Hungary: Association for Computing Machinery, 2018, pp. 15–21. DOI: 10.1145/3229584.3229590. [Online]. Available: https://doi.org/10.1145/3229584.3229590.

[28] S. Dräxler, H. Karl, M. Peuster, *et al.*, “Sonata: Service programming and orchestration for virtualized software networks,” in *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2017, pp. 973–978. DOI: 10.1109/ICCW.2017.7962785.

[29] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to sequence learning with neural networks*, 2014. arXiv: 1409.3215 [cs.CL].

[30] M. Riftadi and F. Kuipers, “P4i/o: Intent-based networking with p4,” in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 438–443. DOI: 10.1109/NETSOFT.2019.8806662.

[31] M. Riftadi, J. Oostenbrink, and F. Kuipers, *Gp4p4: Enabling self-programming networks*, 2019. arXiv: 1910.00967 [cs.NI].

[32] X. Gao, T. Kim, M. Wong, *et al.*, “Switch code generation using program synthesis,” English (US), in *SIGCOMM 2020 - Proceedings of the 2020 Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM 2020 - Proceedings of the 2020 Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, Association for Computing Machinery, Jul. 2020, pp. 44–61. DOI: 10.1145/3387514.3405852.

[33] L. Beurer-Kellner, M. Vechev, L. Vanbever, and P. Veličković, “Learning to configure computer networks with neural algorithmic reasoning,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 730–742, 2022.

[34] P. Veličković and C. Blundell, “Neural algorithmic reasoning,” *Patterns*, vol. 2, no. 7, 2021.

[35] F. Pereira, G. Matos, H. Sadok, *et al.*, “Automatic generation of network function accelerators using component-based synthesis,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’22, Virtual Event: Association for Computing Machinery, 2022, pp. 89–97. DOI: 10.1145/3563647.3563656. [Online]. Available: https://doi.org/10.1145/3563647.3563656.

[36] A. Angi, A. Sacco, F. Esposito, G. Marchetto, and A. Clemm, “Nlp4: An architecture for intent-driven data plane programmability,” in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 25–30. DOI: 10.1109/NetSoft54395.2022.9844035.

[37] N. McKeown, D. E. Talayco, G. Varghese, N. P. Lopes, N. S. Bjørner, and A. Rybalchenko, “Automatically verifying reachability and well-formedness in p4 networks,” 2016.

[38] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, “P4pktgen: Automated test case generation for p4 programs,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’18, Los Angeles, CA, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3185467.3185497. [Online]. Available: https://doi.org/10.1145/3185467.3185497.

[39] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, “Uncovering bugs in p4 programs with assertion-based verification,” in *Proceedings of the Symposium on SDN Research*, 2018, pp. 1–7.

[40] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, “Debugging p4 programs with vera,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 518–532.

[41] J. Liu, W. Hallahan, C. Schlesinger, *et al.*, “P4v: Practical verification for programmable data planes,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, 2018, pp. 490–503.

[42] K. S. Kumar, R. K, P. Prashanth, M. T. Arashloo, V. U, and P. Tammana, “Dbval: Validating p4 data plane runtime behavior,” in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2021, pp. 122–134.

[43] D. Lukács, M. Tejfel, and G. Pongrácz, “Keeping p4 switches fast and fault-free through automatic verification,” *Acta Cybernetica*, vol. 24, no. 1, pp. 61–81, 2019.

[44] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. DOI: 10.1145/360933.360975. [Online]. Available: https://doi.org/10.1145/360933.360975.

[45] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[46] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, *Neuro-symbolic program synthesis*, 2016. arXiv: 1611.01855 [cs.AI].

[47] S. Padhi, E. Polgreen, M. Raghothaman, A. Reynolds, and A. Udupa, “The sygus language standard version 2.0,” 2019, Accessed: 2023-07-14. [Online]. Available: https://sygus.org/assets/pdf/SyGuS-IF_2.0.pdf.

A

Probe-cpp technical details

This appendix provides more details on the implementation of Probe-cpp. The full architecture diagram is shown in figure A.1. Below, the reader can find the string processing grammar, taken from [7], which was used to evaluate the performance of Probe-cpp and compare it to that of the original implementation. The grammar is adapted slightly for each problem specification by the addition of certain literals specific to the considered problem. These literals are included in the program specification provided by the user and are added by Probe-cpp to the grammar at runtime.

S	$\longrightarrow$	arg0 | arg1 ...	the inputs to the program
	|	(replace S S S)	finds the first occurrence of S_2 in S_1 replaces it with S_3
	|	(concat S S)	concatenates two strings
	|	(substr S I I)	returns a substring of S of length I_2, from index I_1
	|	(ite B S S)	if B yields true, return S_1, otherwise return S_2
	|	(int2str I)	cast integer I to a string
	|	(at S I)	return character from string S at index I
B	$\longrightarrow$	true	
	|	false	
	|	(= I I)	checks whether I_1 is equal to I_2
	|	(contains S S)	checks whether S_1 contains S_2
	|	(suffixof S S)	checks whether S_1 ends with S_2
	|	(prefixof S S)	checks whether S_1 starts with S_2
I	$\longrightarrow$	(str2int S)	cast string S to an integer
	|	(+ I I)	add two integers
	|	(- I I)	substract two integers
	|	(length S)	returns the length of the string
	|	(ite B I I)	if B yields true, return I_1, otherwise return I_2
	|	(indexof S S I)	returns the index of S_2 in S_1, starting at index I

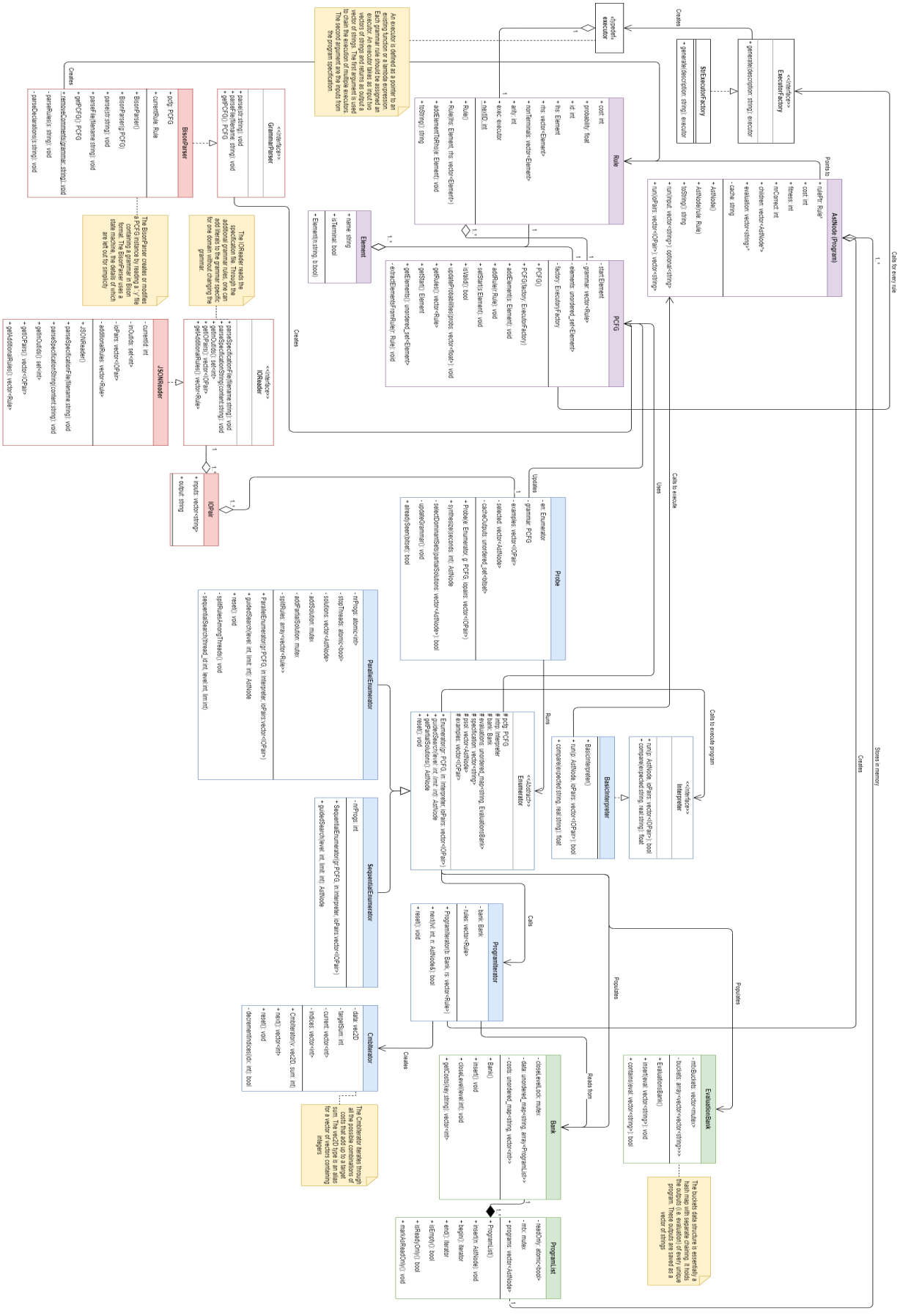


Figure A.1: UML diagram of Probe-cpp's architecture

B

Detailed benchmarks

This appendix provides more details about the *basic* benchmark. The input to G4BE is the program specification which is given in the JSON format. Listing B.1 shows a part of the program specification for the *basic* benchmark. These JSON objects include the pairs of input & output packets in the form of byte strings, terminals to add to the grammar (such as MAC addresses, port numbers, etc), and any optional code hints.

Listing B.1: Program specification for basic.p4

```
{
  "examples": [
  ...
    {
      "input": {
        "packet": "ffffffffffff080000000222080045000033000100004006 63c...",
        "port": "2"
      },
      "output": {
        "packet": "080000000111ffffffffffff080045000033000100003f0664c...",
        "port": "1"
      }
    }
  ],
  "terminals": {
    "number": [
      "1", "0", "10"
    ],
    "macAddr": [
      "08:00:00:00:01:11", "08:00:00:00:02:22",
      "08:00:00:00:03:00", "08:00:00:00:04:00"
    ],
    "port": [
      "1", "2", "3", "4"
    ]
  },
  "hints": [
    "action ipv4_forward(bit<48> macAddr, bit<9> port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = macAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }"
  ]
}
```

Given a program specification, G4BE produces two separate outputs, namely the P4 program and a JSON object containing the data that was used to populate the match-action tables. Listing B.2 shows a part of the solution generated by G4BE. Note how the code hints given as part of the program specification are included in the solution.

Listing B.2: Synthesized basic.p4

```
control MyIngress(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata) {
 action a1 (bit<48> macAddr, bit<9> port) {
  standard_metadata.egress_spec = port;
  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
  hdr.ethernet.dstAddr = macAddr;
  hdr.ipv4.ttl = hdr.ipv4.ttl-1;
 }
 table t0 {
  key = {
   hdr.ipv4.dstAddr: exact;
  }
  actions = {
   a1;
  }
  size = 1024;
 }
 apply {
  t0.apply();
 }
}
```

For the *basic* benchmark, G4BE synthesizes a match-action table together with the entries in that table. These entries are included into a JSON object which is presented as an output to the user, together with the synthesized P4 code. For the pod topology, G4BE outputs the JSON object depicted in listing B.3.

Listing B.3: Synthesized runtime rules for basic.p4

```
{
    "target": "bmv2",
    "p4info": "build/switch.p4.p4info.txt",
    "bmv2_json": "build/switch.json",
    "table_entries": [
        {
            "table": "t0",
            "match": {
                "hdr.ipv4.dstAddr": 167773188
            },
            "action_name": "MyIngress.a1",
            "action_params": {
                "macAddr": "08:00:00:00:04:00",
                "port": 4
            }
        },
        {
            "table": "t0",
            "match": {
                "hdr.ipv4.dstAddr": 167772931
            },
            "action_name": "MyIngress.a1",
            "action_params": {
```

```
                "macAddr": "08:00:00:00:03:00",
                "port": 3
            }
        },
        {
            "table": "t0",
            "match": {
                "hdr.ipv4.dstAddr": 167772674
            },
            "action_name": "MyIngress.a1",
            "action_params": {
                "macAddr": "08:00:00:00:02:22",
                "port": 2
            }
        },
        {
            "table": "t0",
            "match": {
                "hdr.ipv4.dstAddr": 167772417
            },
            "action_name": "MyIngress.a1",
            "action_params": {
                "macAddr": "08:00:00:00:01:11",
                "port": 1
            }
        }
    ]
}
```