LTL-based Specifications for P4 Program Synthesis

Theunissen, Lorenzo; Dumančić, Sebastijan; Kuipers, Fernando

DOI
10.1145/3750022.3750456

Publication date
2025

Document Version
Final published version

Published in
FMANO '25: Proceedings of the 2nd Workshop on Formal Methods Aided Network Operation

LTL-based Specifications for P4 Program Synthesis

Lorenzo Theunissen, Sebastijan Dumančić, and Fernando Kuipers
Delft University of Technology

Abstract

Programmable networks enable us to define the behaviour of a network through software. This added freedom comes with added complexity because multiple switches need to coordinate and be programmed correctly. To ease this task, we focus on intent-based networking via program synthesis. In this paper, we explain how to leverage linear temporal logic to describe the desired behaviour of a program, how to verify a P4 program against that description, and how to use the formula describing the program's behaviour to reduce the search space of the program synthesiser.

CCS Concepts

• **Networks** → **Network design and planning algorithms**; • **Software and its engineering** → **Formal software verification**.

Keywords

Program Synthesis, P4, LTL, Logic

ACM Reference Format:
Lorenzo Theunissen, Sebastijan Dumančić, and Fernando Kuipers. 2025. LTL-based Specifications for P4 Program Synthesis. In *2nd Workshop on Formal Methods Aided Network Operation (FMANO '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3750022.3750456

1 Introduction

Software-defined networks (SDNs), along with programmable data planes, have enabled network administrators to program the network control and data planes in a top-down manner, by using programming languages such as P4 [4]. As such, they can adapt their networks without being limited by device configuration options, or having to wait for new functionalities to be integrated into the hardware.

However, this benefit of flexibility comes at the cost of complexity; network administrators have to learn (yet) another programming language (e.g., P4), which adds to their already growing list of responsibilities. In addition, programming a network differs from a typical programming task, where you write a program to be run on a single machine, because it requires the programmer to deploy and coordinate multiple programs over various switches in the network.

To mitigate this complexity, we turn to the field of program synthesis to automatically synthesise programs for the network switches. Program synthesis has already been utilised to synthesise network configurations [2, 10], and to synthesise network code in NDLog [6, 7]. Synthesising P4 programs for a network comes with many challenges, such as determining switch resource allocations [14] and expressing network intent [18]. This paper starts with the latter challenge: how to specify what the programs-to-be-synthesised are supposed to do?

In program synthesis, a common way to specify the intent of a desired program is through input-output examples, or, less commonly, but in a more precise manner, through formal specification. Whilst input-output examples can be used to synthesise network code, gathering these input-output examples can be challenging. Without the assistance of a formal specification, input-output examples struggle to capture the complete desired behavior. Furthermore, these examples become cumbersome when the intent is to capture multiple different programs for a network of switches that need to be coordinated with each other. This includes both the functionality of the switches and the routing of the packets.

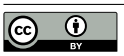
Our idea, in this paper, is based on the observation of the following property of packets: When packets traverse a network, their location in the network and the header data can change over time. Such information of a packet's location and data can be seen as a state that changes over time. We view the possible states of a packet as a model and will use linear temporal logic (LTL) to describe the desired behaviour of this model.

In this paper, we show how to describe the desired behaviour of a network consisting of P4-compatible devices as an LTL formula and how to turn a (set of) synthesised P4 program(s) into a model. We will then give the synthesised programs and LTL formulae to an LTL model checker to verify that the synthesised programs provide the desired behaviour. We further explore how behaviour formulae can be leveraged to reduce the search space, guiding the synthesiser to accelerate the synthesis process. Additionally, we demonstrate how these benefits, driven by the expressiveness of LTL formulae, contribute to performance gains.

2 Background

P4 [4] is a domain-specific language used for packet processing on programmable network hardware, such as switches. Program synthesis [13] is a branch in computer science focused on the automatic generation of provably correct programs with respect to a given specification. Commonly, such a specification is given in the form of input-output examples, and more rarely in the form of a formal specification. In addition to a specification, a synthesiser needs a definition of the program space, which is given in the form of a grammar.

What we need for our program synthesis is the following: a grammar to express our P4 program space in; a way to synthesise candidate programs from the grammar; a way to specify what the desired program should do; and a way to verify the candidate programs.



FMANO '25, Coimbra, Portugal

ACM ISBN 979-8-4007-2103-8/2025/09
https://doi.org/10.1145/3750022.3750456

Whilst P4 has its own specification and grammar[1], it involves the complex task of learning that domain-specific language, and the programs are typically composed for a singular switch. We will create a simpler intermediate language for our synthesis. Programs expressed in this higher-level language can easily be translated to standard P4, yet are easier to work with during synthesis and verification, and allow us to talk about behaviour that is not confined to a single switch. This intermediate language gives us the same expressiveness compared to using P4 directly, and imposes no loss of functionality.

A common method to describe the desired behaviour in program synthesis is through input-output examples, however, this comes with limitations. Whilst on a single switch it can be useful to describe the behaviour through input-output examples, on a network there are multiple ways to do it. To name a few, you can take input-output examples at every switch, you can take traces from the packets traversing the network, or you can take where a packet starts and ends. Whilst these methods can describe the behaviour of individual packets, more complex behaviour of the network itself becomes hard to describe via input-output examples. Instead of input-output examples, in the following section, we will use LTL for our program specification, which will allow us to describe behaviour using logical formulae.

3 LTL Verification

3.1 Preliminaries

In this section, we explain different aspects of LTL and its use for model checking [9]. We have a set of atomic propositions called Π that comprise a finite set of variables, which can be true or false. Each proposition represents a property of our packet, such as "the packet needs to end up at switch 1" or "the packet is currently at switch 2." We define an LTL formula as follows: $\phi ::= T \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \mathcal{X}\phi \mid \phi_1 \,\mathcal{U}\, \phi_2$, where T is true in each state, $p \in \Pi$ is an atomic proposition, $\neg$ and $\vee$ are standard Boolean operators, and $\mathcal{X}$ and $\mathcal{U}$ are temporal operators. $\mathcal{X}\phi$ is true if ϕ is true in the next state, and $\phi_1 \,\mathcal{U}\, \phi_2$ is true if ϕ_1 is true until ϕ_2 is true. With these fundamental operators we can construct additional operators, F and G. F is finally or eventually: $F\phi$ is true if ϕ will be true at some point from now. $G\phi$ is true if ϕ is always true from this point. We can combine these to get $FG\phi$, which means that at some point ϕ will always be true, and $GF\phi$ which means that at every state ϕ will always be true in a future state.

We use nuXmv/nuSMV [3, 5] as our model checker. A model describes a state of variables. An update or next functionality describes how a variable updates to the next state.

3.2 Simple Instructions

Let us consider a simplified packet that only holds 2 variables, Foo and Bar, and consider the following simple program that executes on it.

```
0    Foo = Foo + Bar;
1    Bar = Bar + 1;
```

Listing 1: Simple program.

[1]The specification and grammar can be found at https://p4.org/wp-content/uploads/2024/10/P4-16-spec-v1.2.5.html.

To check if this program works, we need to compare the values of Foo and Bar to their initial values, let us call those Initial_Foo and Initial_Bar. We can then create the following LTL formula to describe the desired behaviour: $F(\, G(\, Foo = Initial\_Foo + Initial\_Bar \,\wedge\, Bar = Initial\_Bar + 1))$. This formula states that at some point in time it will always be true that Foo = Initial_Foo + Initial_Bar and that Bar = Initial_Bar + 1. We need to transform our code into a model that can be checked. The simplest way to do this is to use a program pointer to execute our program one line at a time. We can now transform our code into the following model:

```
FROZENVAR
    Initial_Foo : unsigned word[32];
    Initial_Bar : unsigned word[32];

VAR
    Foo : unsigned word[32];
    Bar : unsigned word[32];
    PPointer : 0..2;

ASSIGN
    init(Foo) := Initial_Foo;
    init(Bar) := Initial_Bar;
    init(PPointer) := 0;

    next(Foo) := case
        PPointer = 0 : Foo + Bar;
        TRUE : Foo;
    esac;

    next(Bar) := case
        PPointer = 1 : Bar + 1;
        TRUE : Bar;
    esac;

    next(PPointer) := case
        PPointer < MAXLENGTH :
            PPointer + 1;
        TRUE : PPointer;
    esac;
```

Listing 2: Simple model.

If we update a variable more than once, for example:

```
Foo = Foo + Bar;
Foo = Foo + Bar;
```

Listing 3: Multiple updates code.

we get the following next function for Foo:

```
next(Foo) := case
    PPointer = 0 : Foo + Bar;
    PPointer = 1 : Foo + Bar;
    TRUE : Foo;
esac;
```

Listing 4: Multiple updates model.

and the following LTL formula:
$F(\ G(\ Foo = Initial\_Foo + 2 * Initial\_Bar))$.

3.3 Conditionals

Based on the previous section, we can now model any assignment instruction, but we would like to also be able to model conditional code, such as if, elseif, and else blocks. Consider the following code:

```
0    if(Foo > 10){
1        Foo = Foo + Bar;}
```

Listing 5: Conditional code.

This will give us the following LTL formula:
$F(\ G(\ Foo > 10 \implies Foo = Initial\_Foo + Initial\_Bar))$. To model this, we create additional variables for every condition in the code, and add an additional case to the lines of code that fall within the code block of the condition. That will give us the following next functions:

```
next(if1) := case
    PPointer = 0 : true;
    TRUE : if1;
esac;

next(Foo) := case
    PPointer = 1 : case
        if1   : Foo + Bar;
        TRUE : Foo;
    esac;
    TRUE : Foo;
esac;
```

Listing 6: Conditional model.

3.4 Forwarding

For packets to be forwarded from one switch to an adjacent switch there is a table lookup. One of the variables of the packet is used as the key, and 2 values are returned. To accommodate this, we create 2 additional variables in our model, and do the table lookup in 2 steps. We set these variables based on the value of the key, and then simply execute the forwarding function as we do other lines of code. A simplified model of the table lookup would look like the following:

```
next(ForwardingVar1) := case
    PPointer = 0 : case
        key = 0 : 3;
        key = 1 : 0;
        key = 2 : 2;
        TRUE : ForwardingVar1;
    esac;
    TRUE : ForwardingVar1;
esac;

next(ForwardingVar2) := case
    PPointer = 1 : case
        key = 0 : 15;
        key = 1 : 27;
        key = 2 : 39;
        TRUE : ForwardingVar2;
    esac;
    TRUE : ForwardingVar2;
esac;
```

Listing 7: Forwarding model.

3.5 Multiple switches

We can model multiple switches by putting all their code in one model. We can then simply use a variable to keep track of the switch we are currently in and update the program pointer to point to the appropriate section of the model. We can use a variable to indicate where the packet currently is. We can set this to the starting switch and then use the specification to say where it should end up. For example: $F(\ G(\ Switch = Switch\_3))$. We can also say that a packet must go through a certain switch by also adding the following formula: $F(Switch = Switch\_2)$.

4 Search Space Reduction

4.1 Single Switch

4.1.1 Simple Reductions. We can use the LTL formulae that describe the desired program to inform our synthesis and reduce the search space. Instead of starting with an empty program and the entire search space, we can use the information in the LTL formulae to get a better starting point and inform the synthesiser during the search. Let us take a look at the first formula: $F(\ G(\ Foo = Initial\_Foo + Initial\_Bar\ \wedge\ Bar = Initial\_Bar + 1))$. From this formula we can see that eventually Foo = Initial_Foo + Initial_Bar and Bar = Initial_Bar + 1. We could use this information to immediately create the two lines of code in listing 1. In this simple case, it is possible, since the formula only describes the end conditions, which can easily be turned into assignment statements. This does not generalise to more complex programs. For example, a program that switches two variables can not be made using two assignment statements. However, we can still use the information in the formula to know that Foo and Bar must change their values at least once. In the more general case, we can see in the formulae which variables change their values and which keep their values. This can be used to reduce the search space to programs that adhere to these constraints. More formally, every variable $v \in \mathcal{V}$ in the formula that does not equal their initial value must appear at least once in the program in the form of $v = x$, where x can be any instruction.

4.1.2 Operator Reductions. If the next operator ($\mathcal{X}$) is used to describe the state for a variable, for example $\mathcal{X}Bar = 1$, then we can say that an assignment must take place. Because only one variable can be changed at a time, if something must be true in the next state, then that must happen with the current line of code. If the until operator ($\mathcal{U}$) is used, that ϕ_1 must be true until ϕ_2 is true. This means that as long as ϕ_2 does not hold, we can not make changes such that ϕ_1 becomes false. This restricts the possible options, and reduces the search space. For example, if we have the

formula $Foo = 1\ \mathcal{U}\ Bar = 0$, we know we can not make changes to Foo unless $Bar = 0$. The global operator G states that something must always be true from this point forward. If used on a variable, we know that that variable can never change again, so it should not be changed in the code. If used on a formula, we know that formula must always be true. If this is part of a subformula, and a formula θ under G is already true, we can use the other reduction rules to ensure that θ stays true whilst pruning the search space. The combination of GF means that at every state the formula will always be true in a future state, indicating cyclical behaviour. This means that the formula under GF must be kept true, or if made false, must be made true again. In the latter case, it means that the code that makes it true must always be reachable, which limits the search space.

4.1.3 Conditional Reductions. We can also recognise conditionals and add constraints for them. If we have an implication in our formula, then we have an if statement in the code. Whilst it is possible to have an implication that would not result in an if statement in the code, this would only be the case if the implication always holds. For example, if we have the formula $Foo > 10 \implies Foo = Initial\_Foo + Initial\_Bar$ and Foo is always equal to Initial_Foo + Initial_Bar, then the formula will be true. Realistically, someone who writes the specification will use an implication for case distinction and not for it to always hold. Meaning that if there is an implication, we can add a constraint that there needs to be an if statement in the code. If we have n number of implications in the formula, that does not necessarily mean that we have n number of if statements. Two implications could be an if and else statement, or an if and elseif statement, instead of two if statements. What we can say however, is that the sum of the number of if statements, elseif statements, and else statements, must be more or equal to the number of implications in the formula. n is the minimum number of conditional statements, since the desired program can use conditional statements to produce behaviour that is not specified with implications in the formulae.

4.2 Multiple Switches

When we have multiple switches, we can still use the hints of single switches, but we need to make an additional distinction. We can now also use implications to check where we are in the network, which would not indicate an if statement in the code. We now distinguish between functional and conditional parts of the formulae. Functional formulae describe the behaviour of a single switch in the network, whilst conditional formulae describe conditions over the network or packet. We can also use these conditional formulae to reduce the search space. For example, if a packet needs to visit a certain switch, or needs to reach one switch before another, that will reduce the number of routing options, and will thus reduce the search space.

5 Results

5.1 Simple examples

We use Herb.jl [21], a library for defining and efficiently solving program synthesis tasks in Julia, to synthesise our P4 programs. We create a simple (incomplete) grammar that can synthesise simple P4 programs. A program is synthesised, turned into a model, and then given to nuXmv/nuSMV with the LTL formula of the desired program to verify if the synthesised program is correct. We can solve the following 4 simple cases:

(1) Basic: route the packet according to its destination and decrease the time-to-live counter.
(2) Min: decrease the time-to-live counter and send the packet to port 1.
(3) Reflector: swap the MAC addresses and send the packet back to the port it came from.
(4) Repeater: repeat the packet on the ports it was not received on.

Program	Verification Time
Basic	7.64s
Min	1.28s
Reflector	8.21s
Repeater	1.23s

Table 1: Given the desired program in the form of a model, the times are how long it takes nuXmv/nuSMV on average to verify the program (when run on an i7-1365U cpu and 16GB of memory).

5.2 Complicated examples

5.2.1 Single Switch. We will look at the more complicated example of a port knocking firewall. For a packet to make it through the switch, it needs to be received 3 times in a row on 3 different ports. For this we need to keep track of 5 additional variables. The three ports that the packet needs to be received on, some way to identify the packet, and the stage of the knocking. This gives us the variables Port_1, Port_2, Port_3, ID, and Stage. We have multiple stages, so we can split this into 3 formulae:
$\theta_1 = (Port = Port\_0 \ \wedge\ Stage = 0 \implies ID = Source\_Address \ \wedge\ Stage = 1)$
$\theta_2 = (ID = Source\_Address \wedge\ Port = Port\_1 \ \wedge\ Stage = 1 \implies ID = Source\_Address \ \wedge\ Stage = 1)$
$\theta_3 = (ID = Source\_Address \wedge\ Port = Port\_2 \ \wedge\ Stage = 2 \implies Forward)$.
This gives us the final formula of $F(\ G(\ \theta_1 \ \wedge\ \theta_2 \ \wedge\ \theta_3))$.

5.2.2 Multiple Switches. We will now look at the more complicated example of a simple firewall existing somewhere on the network, where the firewall protects a certain part of the network, which we will call the InNetwork.

First, our simple firewall will only allow traffic to a node in the InNetwork from outside of it, if that node messages that switch first. This will be kept track of using a register on every node for each node in the network.

We have three cases, and therefore three formulae that all need to hold. The first is if the input is from the InNetwork going to the outside. We can formulate it as such:
$\phi_1 = input \in InNetwork \wedge output \notin InNetwork \implies$
$\{\exists switch | switch[output] = 1 \wedge$
$F(CurrentSwitch = Switch)\} \wedge forward$

The other two cases are similar to each other. They are when a packet is from the outside and they differ on having received a packet from the InNetwork. This gives:
$\phi_2 = input \notin InNetwork \wedge output \in InNetwork \wedge$
$\{\exists\ Switch | Switch[output] = 1 \wedge F(CurrentSwitch = Switch)\}$
$\implies F(G(CurrentSwitch = output))$
$\phi_3 = input \notin InNetwork \wedge output \in InNetwork \wedge$
$\{\exists\ Switch | Switch[output] = 0 \wedge F(CurrentSwitch = Switch)\}$
$\implies Drop$

We can combine all of these in $\phi = \phi_1 \wedge \phi_2 \wedge \phi_3$. We used notation that is not available in LTL, but it is syntactic sugar to make it easier to read. All of this can be written in pure LTL. For example, if we have a list $b = [b_1, b_2, b_3, b_4, ..., b_n]$, we can say that $a \in b == (a = b_1 \vee a = b_2 \vee ... \vee a = b_n)$ and $a \notin b == (a \neq b_1 \wedge a \neq b_2 \wedge ... \wedge a \neq b_n)$.

6 Related Work

[8, 12, 17] utilise program synthesis in programmable switches. However, these works focus on updating switches, monitoring switches, and compiling P4 code, but not on generating P4 programs directly.

Facon [6] and NetSpec [7] show program synthesis being applied to produce network code. However, these programs are not P4 programs, but rather NDLog, a representative declarative networking language based on DataLog. Moreover, these approaches focus on input-output examples and turn those into a formal specification, whilst we take a formal specification as a direct starting point.

Traditional compilers translate higher-level code to machine-code using a rule-based approach. Chipmunk [11] uses the higher-level code of a program as its specification and then uses program synthesis to create a more optimised program.

NLP4 [1] takes natural language as an input for the specification and then translates that to configuration files for P4 switches. Whilst the use of natural language is an intuitive way for users to express their intent, it is often ambiguous to interpret. Moreover, the current use cases of this implementation are limited to traffic management.

[19, 20] implement a P4 verifier that verifies the behaviour taking not only the control block into account, but also the parser, deparser, and non-configurable parts of the switch. The behaviour is considered correct only if the configurable components are configured such that the entire switch behaves according to the specification.

[15] turns a program into a logical formula and tests with an SMT solver whether its properties hold.

Our use of logical formulae as the specification can be seen as assertions that must hold for the program. [16, 22] use an assertion-based approach to verify the behaviour of P4 code. These approaches are given a set of assertions and, if these assertions hold, the program is considered correct.

7 Conclusion and Future Work

In this paper, we have presented (1) a method to translate a P4 program into a model that can be used by an LTL model checker and (2) a novel way to express the intended behaviour of a switch or a network as an LTL formula. We have shown how insights from such formulae can be used to guide the synthesis process by decreasing the program space and hence speeding up the synthesis.

In the future, we aim to create a tool that would make it easy to go from intent to an LTL formula. The introduction of syntactic sugar for more complicated ideas makes it easier to express your intent, but leaves it still relatively complicated. We would like a tool that automatically translates your intent to a correct LTL formula that can then be used for the synthesis. Additionally, we think that at different levels of abstraction, different insights can be made to speed up the synthesis.

Acknowledgements

This research was supported by the National Growth Fund through the Dutch 6G flagship project "Future Network Services".

References

[1] Antonino Angi, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. 2022. NLP4: An Architecture for Intent-Driven Data Plane Programmability. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. 25–30. https://doi.org/10.1109/NetSoft54395.2022.9844035

[2] Ryan Beckett, Ratul Mahajan, Todd D. Millstein, Jitendra Padhye, and David Walker. 2017. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 437–451. https://doi.org/10.1145/3062341.3062367

[3] Armin Biere and Roderick Bloem (Eds.). 2014. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Lecture Notes in Computer Science, Vol. 8559. Springer.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95. https://doi.org/10.1145/2656877.2656890

[5] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *CAV*. 334–342.

[6] Haoxian Chen, Anduo Wang, and Boon Thau Loo. 2018. Towards Example-Guided Network Synthesis. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking, APNet 2018, Beijing, China, August 02-03, 2018*, Mosharaf Chowdhury and Kun Tan (Eds.). ACM, 65–71. https://doi.org/10.1145/3232565.3234462

[7] Haoxian Chen, Chenyuan Wu, Andrew Zhao, Mukund Raghothaman, Mayur Naik, and Boon Thau Loo. 2023. Synthesizing Formal Network Specifications From Input-Output Examples. *IEEE/ACM Trans. Netw.* 31, 3 (2023), 994–1009. https://doi.org/10.1109/TNET.2022.3208551

[8] Xiaoqi Chen, Andrew Johnson, Mengying Pan, and David Walker. 2022. Synthesizing state machines for data planes. In *SOSR '22: The ACM SIGCOMM Symposium on SDN Research, Virtual Event, October 19 - 20, 2022*. ACM, 81–88. https://doi.org/10.1145/3563647.3563650

[9] Edmund Clarke, Ansgar Fehnker, Sumit Kumar Jha, and Helmut Veith. 2005. *Temporal Logic Model Checking*. Birkhäuser Boston, Boston, MA, 539–558. https://doi.org/10.1007/0-8176-4404-0_23

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. 2017. Network-Wide Configuration Synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 261–281. https://doi.org/10.1007/978-3-319-63390-9_14

[11] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. 2019. Autogenerating Fast Packet-Processing Code Using Program Synthesis. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*. ACM, 150–160. https://doi.org/10.1145/3365609.3365858

[12] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch Code Generation Using Program Synthesis. In *SIGCOMM '20: 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, Virtual Event, USA, August 10-14, 2020*, Henning Schulzrinne and Vishal Misra (Eds.). ACM, 44–61. https://doi.org/10.1145/3387514.3405852

[13] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[14] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 193–207. https://www.usenix.org/conference/nsdi22/presentation/hogan

[15] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, Sergey Gorinsky and János Tapolcai (Eds.). ACM, 490–503. https://doi.org/10.1145/3230543.3230582

[16] Miguel C. Neves, Lucas Freire, Alberto E. Schaeffer Filho, and Marinho P. Barcellos. 2018. Verification of P4 programs in feasible time using assertions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, Xenofontas A. Dimitropoulos, Alberto Dainotti, Laurent Vanbever, and Theophilus Benson (Eds.). ACM, 73–85. https://doi.org/10.1145/3281411.3281421

[17] Yiming Qiu, Ryan Beckett, and Ang Chen. 2023. Synthesizing runtime programmable switch updates. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 613–628.

[18] Mohammad Riftadi and Fernando Kuipers. 2019. P4I/O: Intent-Based Networking with P4. In *2019 IEEE Conference on Network Softwarization (NetSoft)*. 438–443. https://doi.org/10.1109/NETSOFT.2019.8806662

[19] Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Lennart Beringer, and Andrew W. Appel. 2023. Foundational Verification of Stateful P4 Packet Processing. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland (LIPIcs)*, Adam Naumowicz and René Thiemann (Eds.), Vol. 268. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:20. https://doi.org/10.4230/LIPICS.ITP.2023.32

[20] Shengyi Wang, Mengying Pan, and Andrew W. Appel. 2024. Comprehensive Verification of Packet Processing. (2024). arXiv:cs.PL/2412.19908 https://arxiv.org/abs/2412.19908

[21] P. Wochner, J. de Jong, B. Swinkels, T. Hinnerichs, N. Filat, R. Gardos Reid, P. Cichoń, I. Hanou, T. Margirescu, L. Janjić, L. Theunissen, I. Bozhinov, and S. Dumancic. [n. d.]. Herb.jl. ([n. d.]). https://github.com/Herb-AI

[22] Delong Zhang, Chong Ye, and Fei He. 2024. P4Inv: Inferring Packet Invariants for Verification of Stateful P4 Programs. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications, Vancouver, BC, Canada, May 20-23, 2024*. IEEE, 2129–2138. https://doi.org/10.1109/INFOCOM52122.2024.10621366