P4 Compiler & Interpreter: A Survey

Henning Stubbe
Betreuer: Sebastian Gallenmüller, Dominik Scholz
Seminar Future Internet WS2016–2017
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: stubbe@in.tum.de

ABSTRACT
Software-Defined-Networking (SDN) provides new possibilities to configure and manage large scale networks. However, most SDN control protocols limit the possibilities of this approach since they only provide a limited set of protocols and are understood only by a fraction of the available hardware. The Domain-Specific Language (DSL) P4 was developed to mitigate this issue. Designed with the intent to allow a brief description of a switch's behavior, the aim is to enable network owners to develop their own application oriented software for a programmable switch. To be able to execute the developed description on the hardware either a compiler or an interpreter is required, which translates P4 into an executable program. In the past two years, since P4 has been introduced, different tools emerged. This paper will conduct a survey on the different options available.

Keywords
Compiler, Domain-Specific Languages, P4, SDN

1. INTRODUCTION
With increasing size of network structures the necessity to simplify the management in these networks increases. The introduction of Software Defined Networking (SDN), that is the separation of the function a switch provides into the forwarding and control plane, worked towards this need. The forwarding plane, responsible for deciding which rules are applicable to a packet and then acting accordingly, remains part of the switch. Contrary to that, the control plane, responsible for describing and populating the rules the forwarding plane uses to decide, can be outsourced to a central point. This enables simpler configuration options, since modification of the desired network behaviour requires only the controller to be updated. The control plane then makes the change available in the whole network by deploying them to the forwarding plane. Due to the centralized approach previously costly problems, e.g. the calculation of a spanning tree for the network, can be solved more efficiently since the control plane is aware of the network structure.

A protocol that is used to configure SDN-enable devices is OpenFlow [11]. This open standard describes a set of properties the forwarding and the control plane must have as well as how they should interact with each other. Hence, it is possible to combine arbitrary hardware adhering to this standard. Thus network providers gain more flexibility when combining different hardware into their new centralized managed network. The catch on this approach is that current OpenFlow standard — the OpenFlow Switch Specification 1.5.1 [11] — only describes a fixed set of protocols for which rules the forwarding plane has to obey can be expressed. Those protocols are implicitly defined by the list of supported flow match fields as described in Section 7.2.3.7 of the specification [11]. This introduces the problem that custom inhouse solutions that might be desired by network owners cannot be realized since either a protocol used is not available or the custom protocol stack is not supported by OpenFlow.

Bosshart et al. [4] propose to enable network operators to describe the structure of packets that will traverse their network as well as which rules can be applied to them. This allows using separated forwarding and control planes while overcoming the issue of limited protocol support. As the language P4 which Bosshart et al. [4] introduce is a high-level language, tools are required producing a program executable by the hardware target that should later execute it. Since those tools have to abstract from the specific hardware to the hardware independent P4 language they have to cope with this by either being intended for specific hardware or other means. In Section 2 this paper will provide an introduction to the concepts and ideas of P4. Subsequently available solutions with their take on the challenges and implications of the abstraction done by P4 will be addressed in Section 3. Finally, in Section 4, the paper summarizes the current state in its conclusion.

2. BACKGROUND: P4
The Programming Protocol-Independent Packet Processors language — abbreviated P4 — is a Domain-Specific Language (DSL) intended to be used for description of the package processing capabilities of programmable switches. As a DSL focuses on describing rules that can be applied to packets. This allows P4 programs to be more concise when specifying the behavior of a switch. Targeting network operators, Bosshart et al. phrased the design goals of P4:

1. *Reconfigurability in the field.* Modification of the behavior of a switch must remain possible once the switch is permanently installed. I.e. switch behavior changes, e.g. by publishing new match-action rule entries as allowed by OpenFlow must be possible.

2. *Protocol independence.* P4 attempts to make no assumption on which protocols might be used in which combination. In fact P4 tries to create the possibility

to define and integrate new protocols formats whenever desired.

3. *Target independence.* A program written in P4 cannot require features of special hardware, but instead is usable on any hardware for which a runnable translation of P4 can be created. While P4 imposes requirements on the capabilities of the hardware in general, it does not demand the presence of e.g. a fixed instruction set.

Even though P4 is motivated by the inability of OpenFlow to use custom protocols, it borrows a few concepts to describe the packet processing. (Bosshart et al. [4] suggest, that P4 might even be used as proposal how the next version of OpenFlow could look like.) Processing of packets is done by performing actions on the packets based on values of the header fields. As in OpenFlow the mapping from header field value tuples to actions to perform is created during runtime by a control plane, not at compile time. The processing of packets in P4 can be divided into four major phases:

1. *Parsing of the packet.* When receiving a packet, it first must be translated into a representation that can be processed in the next phases. The parser is based on a finite state machine generated from the underlying P4 program.

2. *Apply match-action table to ingress.* The representation of the received object now entered the ingress pipeline. In this phase it is possible to match on the different header fields of the received packet and execute almost arbitrary rules on it. Additionally, the switch can decide which egress pipeline should later process the packet. For this the P4 program can access additional information — metadata — e.g. on which hardware port the packet arrived. If necessary, the potentially modified packet can be resubmitted to enter the ingress pipeline again.

3. *Apply match-action table to egress.* Similar to the ingress pipeline the egress pipeline allows execution of rules based how the parsed header fields. Note that neither submission to another egress pipeline nor resubmits can be used here.

4. *Deparsing.* To be able to send the packet to the wire it has to be deparsed based on its current state. In P4$_{14}$ the deparser is generated automatically from the parsed object.

Therefore, in order to describe the behavior of a switch in P4 each of these phases must be described. Statements of a P4 programs influence different parts of the processing pipeline. Which program parts impacts which phase is depicted in Figure 1. The parse graph required to specify the parser and deparser in P4 is described by a set of headers that might occur, a header to start parsing with and conditions that express which header is expected next in which case. Figure 2 shows a P4 (version 1.0.3) program which will serve as example in this section. It first describes the fictional header type `ether_t` which consists of an 48 bit long field called `src` — the source of the packet — and a second field

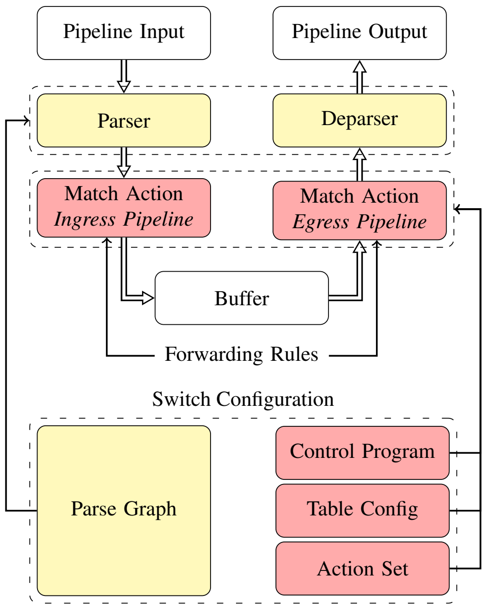

Figure 1: Packet processing as done in P4.

with the same size `dst` — the destination. Then the header named `ether` is declared to be of type `ether_t`. The `parser` lines subsequently instruct the switch to read the outermost header as `ether` and treat the rest of the packet as payload that does not require parsing.

Finally, the packet is added to the ingress pipeline. The remaining part of the program is used to describe the available actions, tables where they occur and which header fields these tables depend on, as well as which tables are part of which pipeline. The field `standard_metadata.egress_spec` describes the port on which the packet should be sent out after deparsing.

Translation tools are required to execute the program, since P4 as a high-level language is not executable. Two options for such a translation exist, either the program is compiled once or interpreted on every execution. While the former method allows a broader range of optimizations the latter

```
header_type ether_t {fields {src: 48; dst: 48;}}
header ether_t ether;
parser start {return ether;}
parser ether {extract(ether); return ingress;}
action forward_ether(out) {
 modify_field(standard_metadata.egress_spec, out);
}
table forward {
 reads {ether.src: exact; ether.dst: exact;}
 actions {forward_ether; drop; no_op;}
}
control ingress {apply(forward);}
control egress {}
```

Figure 2: Discard or forward packet to chosen outgoing port based on source and destination address.

can reduce the time between development and runtime. This makes interpretation more suitable for the development process, and compiling the program preferred for deployment respectively.

The P4 language specification has different versions, which are names with a three digit, dot separated version identifier. Increments in the rightmost digit indicate an update or addition to the last standard. An increase in the middle digit indicates a larger language change. All currently known language version identifier have one as the leftmost digit. Currently the standards 1.0.3 (released 2016–11–03) is said to be the most common, but the standard 1.1.0 (released 2016–01–27) is also available. The main features introduced in 1.1.0 were support for types, strong typing and the ability to state an order in which actions should be executed [17, sec. 17.2]. The P4 version 1.2 has currently draft status. As the current versioning scheme of P4 has caused confusion the versions P4 1.0.0 up to 1.0.3 are referred to as P4$_{14}$, since P4 1.0.0 was released 2014–09–08. The draft P4 1.2 is known as P4$_{16}$.

In the slides of Budiu [6] the differences between P4$_{14}$ and the P4$_{16}$ draft are listed. Additionally the P4$_{16}$ specification draft [18, sec. 3] briefly outlines which change will occur. A major change is that, while P4$_{14}$ specified that the deparser constructed from the given parser, P4$_{16}$ drops this idea and requires the programmer to specify how the object shall be deparsed. This change is motivated by the fact, that there might exist multiple possibilities how the object can be deparsed, e.g. IPv4 in IPv6 or vice versa [17, sec. 6]. A second major difference is that in P4$_{16}$ more targets specific information into P4 by suggestion that a separate file should be used which lists functions provided by the target. Also, one or more core files are envisioned, that shall provide common routines and thus can serve as library for other programs. P4$_{16}$ will also add annotation support and reduce the number of keywords from more than 70 to less than 40. According to aforementioned presentation [6] it will be possible to convert a P4$_{14}$ to a comparable P4$_{16}$ program.

3. COMPILER & INTERPRETER

To receive an executable file from a P4 program either a compiler or an interpreter is required. The next subsections will discuss open source options available wrapped up by a subsection dedicated to proprietary solutions.

3.1 Reference Implementation

To make a reference translator available, the P4 Language Consortium published the P4 compiler `p4c-behavioral` [14]. It will be discussed in the next section in more detail. For a couple of reasons `p4c-behavioral` is now replaced by the "behavioral model" — `bmv2` [13]. The motivation for this change as well as the properties of `bmv2` then follow subsequently.

3.1.1 *p4c-behavioral*

Developed as first reference compiler for P4 `p4c-behavioral` which is written in C implements all features of the P4 specification. It takes a P4 program as input and generates a valid C program based on this input. To do so it depends on `p4-hlir` — a program to create a High-Level Intermediate Representation of P4 according to the source code given. p4c-hlir [15] is written in Python and provides a target independent P4 parser. On successful parse the result is accessible as Python object hierarchy. By utilizing `p4-hlir`, `p4c-behavioral` can focus on generating correct C code. This code can then be complied for the intended target. As a result, additional optimization by the C compiler is possible. The Apache 2 licensed source code can be found at [14].

After completion of the implementation of `p4c-behavioral` a few issues regarding its design were discovered. It was seen as hassle to be required to compile twice (from P4 to C and from C to binary) in order to produce an executable. Further the generated C code from `p4c-behavioral` was deemed to be hard to understand and thus discouraging people from looking at it. Finally, the underlying switch model in the old compiler is fixed and assumes the presence of two pipelines: one ingress and one egress pipeline. This assumption stands in contrast to the paradigm of P4 not to require any hardware properties. Hence it motivated a change in order to fully support P4.

3.1.2 *Behavioral Model*

Addressing some issues with `p4c-behavioral` the new so called behavioral model [13], also called `bmv2`, was developed. The behavioral model is, in contrast to the previously introduced `p4c-behavioral` an interpreter. To run a P4 program with `bmv2`, the P4 source code must first be compiled to a JSON file which then, combined with the P4 file, serves as input for the interpreter. The JSON output is generated by `p4c-bm` [16] which also can generate program dependent C++ code. This program dependent code is either an Apache Thrift [1] or an nanomsg [21] based mechanism to enable communication between the control plane and the forwarding plane on the switch.

As `p4c-behavioral`, the new interpreter is released under the Apache 2 license. It fully supports the P4 specification and can be integrated into mininet [12].

3.1.3 *P4C*

The language P4 is developing and the next version P4$_{16}$ will be released in the future. It is written in C++ and provides different backends. Depending on the P4 input the program can either be compiled to be used with the `bmv2`, or as input for a compiler that can compile C code to eBNF — extended Berkeley Packet Filter. The code released under Apache 2 of [19] is currently rated as alpha quality and thus at the moment is not ready for production.

3.2 P4@ELTE

Similar to the reference implementation of the P4 Language Consortium the compiler P4@ELTE by Laki et al. [9] is not focused on a specific hardware target. Instead, the aim is to provide a target independent compiler. To be able to do so they identified which functions required for P4 are hardware dependent and hardware independent respectively. Providing the independent functions with their compiler the requirements to successfully implement a compiler for a new target is reduced to describe the hardware dependent functionality. This hardware dependent implementation is also called Hardware Abstraction Layer (HAL). [9] currently provides one implementation of such a Hardware Abstraction Layer that makes use of Intel's DPDK [7] and thus can be used with any network interface which supports C code compatible with said library[1]. For input parsing P4@ELTE makes use of `p4-hlir`. The compilation steps are summarized in Figure 3.

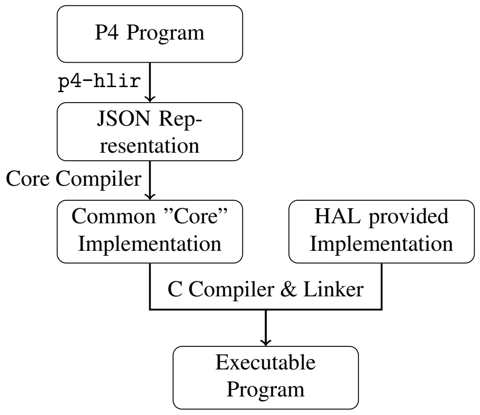

Figure 3: Compilation process of P4@ELTE.

As Laki et al. [8] describe, the approach taken by P4@ELTE to offload the hardware specific details into a library has the benefit that the actual compiler implementation is simpler, as determining properties of the target is not necessary. They argue that the modularity introduced by this separation of concerns increases the maintainability of the compiler. In addition, the fact that the Hardware Abstraction Layer is independent of the P4 programs compiled and thus does not require recompilation if the program changes, is mentioned as advantage of P4@ELTE.

Reduced performance compared to a hardware dependent compiler which can make use of hardware specific optimizations is seen as drawback. In fact, it is difficult to implement any optimization which introduces a constraint on the protocol or hardware used since it may not hold in all cases. The author concludes with the note that implementing the Hardware Abstraction Layer introduces the difficulty to find a suitable abstraction that includes as many hardware targets as possible while avoiding reducing the number of applicable optimizations. The implementation of P4@ELTE was released under the Apache 2 license [10].

3.3 P4FPGA

As the name already suggest P4FPGA [24] differs from the programs mentioned so far regarding the target hardware. The idea of P4FPGA is to provide a tool that eases the development of FPGA based switches. This is done by translating from P4 to Bluespec System Verilog, a hardware description language. For this to work P4FPGA integrates as backend into `p4c`, thus receiving the intermediate representation generated from P4 program parsed by the compiler.

While the full P4 language is supported by P4FPGA, the language P4 itself does not suffice to fully describe how the resulting system should behave. Wang et al. [24] note, the runtime implementation is out of scope for the P4 specification and hence has to be described by other means. This implies that e.g. the description of the Memory Management Unit must be done by other means. The source code of P4FPGA, which is released under the Apache 2 license can be found at [20].

3.4 PISCES

While all previous projects focused on hardware targets, the work of Shahbaz et al. [22] — PISCES — focuses on software switches. Bringing to mind that switching between virtual machines hosted on the same physical machine often also involves new or custom protocols, [22] argue that there is a need for an inexpensive way to implement these protocols. Inexpensive in CPU cycles when executed as well as in time to develop and maintain. To address this Shahbaz et al. modified Open vSwitch, introduced by Pfaff et al. [3], to be usable in conjunction with a P4 program. An important aspect in these changes is the effort to remove any protocol dependency — all conclusions based on assumptions how the arriving packet will look like — from the Open vSwitch implementation.

One implication of this is that the concept of micro-/megaflow caches in Open vSwitch has to be revisited. Micro-/megaflow caches utilize the idea that flows of packets resemble each other in certain field, e.g. source and destination address, ports, etc. Since it is no longer ensured that IP addresses or ports fields are present in the header to match on, the justification of this caching mechanism must be verified. After reviewing Shahbaz et al. [22] come to the conclusion, that this matching algorithm does not have to be modified. It is enough to enable the control plane to manage which fields of the header shall be used to match on.

As they were not required for the supported protocols, not all primitives P4 offers were implemented in Open vSwitch and hence had to be added:

- Since protocols described can have any desired header format Shahbaz et al. [22] enhanced Open vSwitch with functions that allow prepending and removal of a header.
- The incremental checksum modification of Open vSwitch was augmented to also support explicit checksum computation, if desired by the programmer.
- Comparison of header fields must be expressed as bitwise equality test, in Open vSwitch. On the other hand P4 enables the programmer to request such tests for fields of arbitrary length. Hence an implementation of that feature, utilizing the available comparison function was implemented to provide that feature.

Some of these additions, e.g. the question of how the checksum should be computed or if inline editing should be used, require knowledge about the protocol which the compiler does not have. To overcome this problem Shahbaz et al. [22] introduced new annotations that allow the developer in these situations to inform the compiler which action should be performed.

[1]A list of supported network interface controller is maintained at `http://www.dpdk.org/doc/nics`.

Equipped with the modified code they wrote a compiler to generate C code, that makes use of Open vSwitchs functionality, from a P4 program. Compiling those two sources together results in an Open vSwitch derivate with support for an arbitrary protocol stack.

Removing the protocol dependent optimizations from Open vSwitch causes in average a longer execution time per packet, when implementing the same protocol stack as the original version supported. Trying to be competitive with the original Open vSwitch version Shahbaz et al. [22] implemented optimizations to increase the throughput of PISCES:

- With the annotations to the P4 program the compiler can now decide when to compute header checksums or if the use of incremental checksum computation is advised. By delaying or using an incremental computation the compiler can potentially reduce the processing time for the packet.
- When modifying header fields, there are two options. Either the packet is modified inline before all rules are applied, or after all rules are applied post-pipeline. If the modification of the header fields involves changes of the header size, then it might be beneficial to delay the modification to avoid superfluous memory operations. With annotations to the P4 program a developer can influence when PISCES applies modifications to packets.
- The PISCES parser makes use of the knowledge which fields must be parsed to be able to make a forwarding decision and thus is able to reduce parsing time.
- When modifying fields, Open vSwitch can make use of the protocol dependence, i.e. it is known which fields are supposed to remain unchanged and thus do not need to be checked before applying an action. The P4 compiler does not have this knowledge, but can deduce from the P4 program which header fields might have changed and thus need to be checked. This reduction of checks increases the processing speed.
- By analyzing the program the compiler might be able to combine certain modification rules into one.
- Open vSwitch uses its domain specific knowledge to divide its lookup into stages, each stage uses an additional layer of the ISO/OSI stack for the match. While the ordering of layers is not inferable from the program, annotations were introduced that allow to specify an ordering.

When comparing PISCES to Open vSwitch in an extensive benchmarking, Shahbaz et al. [22] come to the conclusion that an implementation of the features of Open vSwitch in P4 was about 40 times shorter (measured in lines of code) than the original implementation with almost the same performance. In their conclusion the authors note, that P4 features that imply state on the target are not implemented, e.g. counter, since this would require a larger modification of Open vSwitchs caching model.

3.5 Proprietary Solutions

Even though P4 is a comparably new language, a couple of companies exist that offer hardware which can be used in combination with P4. The lack of accompanying papers makes it difficult to compare these to the scientific work presented in the previous sections. Nevertheless, it is worthwhile to mention them here, since they as well provide a possibility to use P4 on different targets.

3.5.1 Xilinx SDNet

The company Xilinx, Inc. provides SDNet [5], a development environment which can be used to manage hardware sold by them. To compile a P4 program for their hardware they utilize the `p4-hlir` to parse the source code and make use of the returned python object. A mapper built by them then constructs Xilinx PX code from the result of the previous step, which finally is compiled to firmware by Xilinx SDNet. Xilinx PX is a forwarding plane programming language which, similar to P4 describes the packet processing parsing, one match-action table and deparsing.

3.5.2 Netronome SDK

Netronome indicates that they implemented a compiler that allows execution of programs written in P4 1.0 on Netronome iNIC devices [23]. Extending this compiler to be P4 1.1 compatible is anticipated as effortless by Netronome, since expected syntax changes do not influence the optimization process performed on an intermediate representation of P4.

3.5.3 Barefoot Capilano

A third company providing a P4 compiler for their hardware, called Barefoot Tofino, is Barefoot Networks, Inc. [2]. Barefoot Capilano creates an Integrated Desktop Environment that includes a P4 compiler to create firmware for their hardware.

4. CONCLUSION

In this paper a survey on different methods to translate programs written in P4, a DSL to describe the behavior of programmable switches, was conducted. Each of the programs was discussed regarding the translation process, the translation target as well as unique properties.

Most of the programs available are compilers, but an interpreter intended for development of P4 programs exists. The focus regarding the translation targets are programmable hardware switches with PISCES as notable exception that compiles to a customized Open vSwitch. All implementations support P4 1.0 — the current released version of P4. Apart from `p4c` which is intended to be the reference compiler for the next major version of P4.

It is interesting to see that many of the available tools use the frontend provided by the P4 Language Consortium and hence are able to reduce the effort associated with writing the compiler and interpreter respectively. Providing said frontend might reduces the effort required by developers to support new P4 versions, since provided that the abstract representation remains the same no changes to the backend are required. The question how the different projects cope with the change and how they develop over time might motivate another survey in the future.

5. REFERENCES

[1] Apache Software Foundation. Apache Thrift. https://thrift.apache.org/, 2015. visited 2016–12–21.

[2] Barefoot Networks. The World's Fastest & Most Programmable Networks. Whitepaper, Barefoot Networks, https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf, 2016. visited 2016–12–21.

[3] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA, 2015. USENIX Association.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014.

[5] G. Brebner. Programmable Target Architectures for P4. 2nd P4 Workshop by Stanford/ONRC, 2015. http://sched.co/4eqF. visited 2016–12–21.

[6] M. Budiu. Migration guide P4. Technical report, Barefoot Networks, 2016. https://github.com/p4lang/p4c/blob/master/docs/migration-guide.pptx. visited 2016–12–21.

[7] Intel Corporation. Intel Data Plane Development Kit (DPDK), 2016. http://dpdk.org/. visited 2016–12–21.

[8] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. High-Speed Forwarding: A P4 Compiler with a Hardware Abstraction Library for Intel DPDK. http://p4.elte.hu/publications/p4-ws-2016.pdf, 2016. visited 2016–12–21.

[9] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 629–630, New York, NY, USA, 2016. ACM.

[10] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. Retargetable compiler for the P4 language. https://github.com/P4ELTE/p4c, 2016. visited 2016–12–21.

[11] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.5.1*, 2015.

[12] Open Networking Laboratory. Mininet: An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/, 2015. visited 2016–12–21.

[13] P4 Language Consortium. Behavioral Model (bmv2). https://github.com/p4lang/behavioral-model, 2014. visited 2016–12–21.

[14] P4 Language Consortium. p4c-behavioral. https://github.com/p4lang/p4c-behavioral, 2015. visited 2016–12–21.

[15] P4 Language Consortium. p4c-hlir. https://github.com/p4lang/p4-hlir, 2015. visited 2016–12–21.

[16] P4 Language Consortium. Preprocessor for the P4 behavioral model. https://github.com/p4lang/p4c-bm, 2015. visited 2016–12–21.

[17] P4 Language Consortium. *The P4 Language Specification, Version 1.1.0*, 2016.

[18] P4 Language Consortium. *The P4 Language Specification, Version 16 (Draft 2016–12–16)*, 2016.

[19] P4 Language Consortium. p4c. https://github.com/p4lang/p4c, 2016. visited 2016–12–21.

[20] P4FPGA Project. P4 Bluespec Compiler. https://github.com/hanw/p4fpga, 2016. visited 2016–12–21.

[21] A. Roussel, A. Fabijanic, A. Brem, A. Jonsson, A. Starks, A. Santogidis, A. Degtiarov, B. McCroskey, B. Zentner, B. Mitchener, B. Bigras, C. Salzenberg, D. Beck, D. Ochtman, D. Fang, D. Crawford, D. Socolobsky, E. Chevalier, E. R. Berthing, E. Wies, F. S. Mathieu, G. Roberts, G. D'Amore, G. Diethelm, G. Gupta, H. Saito, H. Lieberman-Berg, I. Weber, I. Pechorin, I. Vachkov, J. R. Dunaway, J. Foster, J. Ammous, K. Schiess, K. Lein-Mathisen, L. Barbato, M. Mendez, M. Ellzey, M. Sustrik, M. Howlett, M. Drechsler, M. John, M. Koppanen, N. Desaulniers, N. Hillegeer, N. Soffer, Örjan Persson, O. Timperi, P. Colomiets, P. Kapyshin, R. Brunno, R. Sciuk, R. Killea, R. G. Jakabosky, S. Avseyev, S. Kovalevich, S. Nikulov, S. Velmurugan, S. Strandgaard, S. Mihai, S. Atkins, S. McKay, S. Wallace, T. Besset, T. Peters, V. Guerra, Y. Luo, and Z. Boszormenyi. nanomsg. http://nanomsg.org/, 2016. visited 2016–12–21.

[22] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.

[23] J. Tönsing. P4/PIF + C Programmable Intelligent NICs: Requirements and Implementation Notes. 2nd P4 Workshop by Stanford/ONRC, 2015. http://sched.co/4epr. visited 2016–12–21.

[24] H. Wang, K. S. Lee, V. Shrivastav, and H. Weatherspoon. P4FPGA: High-Level Synthesis for Networking. 2016.