P4K: A Formal Semantics of P4 and Applications

Ali Kheradmand
University of Illinois at Urbana-Champaign
kheradm2@illinois.edu

Grigore Rosu
University of Illinois at Urbana-Champaign
grosu@illinois.edu

ABSTRACT

Programmable packet processors and P4 as a programming language for such devices have gained significant interest, because their flexibility enables rapid development of a diverse set of applications that work at line rate. However, this flexibility, combined with the complexity of devices and networks, increases the chance of introducing subtle bugs that are hard to discover manually. Worse, this is a domain where bugs can have catastrophic consequences, yet formal analysis tools for P4 programs / networks are missing.

We argue that formal analysis tools must be based on a formal semantics of the target language, rather than on its informal specification. To this end, we provide an executable formal semantics of the P4 language in the K framework. Based on this semantics, K provides an interpreter and various analysis tools including a symbolic model checker and a deductive program verifier for P4.

This paper overviews our formal K semantics of P4, as well as several P4 language design issues that we found during our formalization process. We also discuss some applications resulting from the tools provided by K for P4 programmers and network administrators as well as language designers and compiler developers, such as detection of unportable code, state space exploration of P4 programs and of networks, bug finding using symbolic execution, data plane verification, program verification, and translation validation.

1 INTRODUCTION

As an increasingly important part of our life and society, computer networks have grown significantly in size and complexity. Traditionally, to handle the network scale, the networking hardware has been hard coded with well-established network protocols needed to run and manage the network. However, doing so has the downside of not being able to cope with the speed of innovation that is necessary to satisfy the diverse and growing set of user demands, because the process of modifying networking equipment tends to be slow and expensive. This has ignited a line of research whose goal is to make networks more programmable.

One of the most recent developments in this line of research, P4 [9], is a high level declarative programming language for programming packet processors. P4 allows the developers to specify how a packet processor should process its incoming packets. A P4 compiler then translates the P4 program into an instruction set understandable by the target hardware. The examples of targets include software switches, high performance ASICs, FPGAs, and programmable NICs.

Since its introduction in 2014, P4 has attracted significant interest because the flexibility that it provides enables rapid development of a diverse set of applications that can potentially work at line rate, such as In-Band Network Telemetry [48] and switch based implementation of Paxos [19]. However, this flexibility, combined with the complexity of networks and networking hardware, increases the chance of introducing subtle bugs that are very hard to discover manually, yet can have catastrophic effects, from service disruptions to security vulnerabilities.

Even without P4, answering the simplest questions about the correctness of a network (e.g., what kind of packets can reach node B from node A) has become manually prohibitive when the scale and complexity of networks is taken into account. Subsequently, a large body of research has recently focused on automating the process of network verification [33–35, 47]. However, most of these works assume a simple fixed structure for the packet processors and, as a result, may miss many details. P4 makes manual verification even harder, if not impossible. Consequently, there is a big need for automated tools to analyze P4 programs or networks of nodes programmed using P4.

We adhere to [18] that analysis tools for any programming language must be based on the formal semantics of that language rather than on its informal specification. Informal semantics are subject to interpretation by different tool developers and usually there is no guarantee that these interpretations are consistent with the specification or with each other. As shown in [18], state-of-the-art program analysis tools based on informal language specifications "prove" incorrect properties or fail to prove correct properties of programs due to their misinterpretation of the semantics of the target programming language. Moreover, the informal language specification itself might have problems, such as ambiguities, inconsistencies, or even parts of the language not defined at all. This is particularly relevant for new languages, like P4, whose design has not matured yet.

It is therefore important to develop a formal semantics for P4. Furthermore, to build confidence in the adequacy of a formal semantics, we believe it should be: (1) *executable*, so it can be rigorously tested against potentially hundreds of programs; (2) *compact and human readable*, so it can be easily inspected and ultimately trusted by everyone. Finally it must be (3) *modular*, so new language features can be formalized without the need to change the previously formalized features.

To this end, we have developed P4K, an executable formal semantics of P4 based on the official P4 language specification [15]. P4K faithfully formalizes all of the language features mentioned in the P4 specification, with a few exceptions corresponding to features whose meaning was ambiguous or incorrect or under specified and we did not find any satisfactory way to correct it. We have reported some of these issues to the P4 language designers [36–44] and are working on a modified version of the specification [46] addressing the issues. We validated P4K by executing 40 test cases provided by one of the official compiler front-ends of P4 [16], a manually crafted test suite of 30 tests, and by formally analyzing several programs.

We chose the K framework [56] for our P4 formalization effort. It has several advantages that make it a suitable choice. First, a language defined in K enjoys all three properties mentioned above.

Second, once a programming language semantics is given, K automatically provides various tools for the language, including an interpreter and a symbolic model checker, at no additional effort. Finally, K has already been successfully used to formalize the semantics of major programming languages such and C [23], Java [8], JavaScript [53], etc.

The focus of this paper is the P4K formalization of P4, but we also show how P4K and the tools provided by K can be used beyond just a reference model for the language. We discuss several applications useful for P4 programmers, language designers, and compiler developers, such as: detection of unportable code, state space exploration of P4 programs and of networks, bug finding using symbolic execution, data plane verification, deductive verification, and translation validation. Specifically, we make the following contributions:

- P4K: the most complete formal semantics of P4, based on the official specification of $P4_{14}$ v. 1.0.4.
- A collection of P4 formal analysis tools for the networking domain, automatically derived from the semantics.

The paper is organized as follows. Section 2 overviews P4 and K, as well as the challenges in defining a semantics for P4. Section 3 describes P4K, our K semantics of P4, and discusses some of problems that we identified in the language specification. Section 4 evaluates our semantics. In Section 5 some of the applications of the semantics are discussed. Section 6 reviews related work and Section 7 concludes.

2 BACKGROUND AND CHALLENGES

Here we give background on P4 and K. We also discuss some of the challenges that we faced in formalizing P4.

2.1 Software Defined Networks

Control plane is the part of the network responsible for making packet forwarding decisions by running computations (e.g. routing algorithms) based on the network state. *Data plane* is the collection of forwarding devices (or packet processors) that actually carry the network packets and execute the forwarding decisions. Traditionally, each device had its own vendor-provided control plane hard-coded on the device. The need for rapid innovation has sparked interest in Software Defined Networks (SDNs). SDN is a modern architecture in which the control plane is physically separated from the data plane. In this architecture, one controller can *program* a set of forwarding devices through open, vendor-agnostic interfaces such as OpenFlow [52].

In OpenFlow, each device processes the packets according to the contents of one or more *flow tables*. Each table will contain a set of *flow entries*. Abstractly, each entry is a *(match, action)* tuple. *Match* provides values for specific fields in the packet header, and *action* denotes the action to be performed if the packet header matches the respective values in match. Possible actions include dropping, modifying, or forwarding the packet. The controller programs the data plane through installation and modification of flow entries.

OpenFlow assumes a fixed structure for the forwarding devices. It explicitly specifies the set of protocol headers on which it operates, the structure of the flow tables, the set of possible actions, etc. Modification to any of these features requires an update to the OpenFlow specification. Over the course of 4 years since the initial version of OpenFlow, the number of supported header fields in its specification has been more than tripled [9].

```
header_type h_t { fields { f1 : 8; f2 : 8; } }
header h_t h1;
parser start { extract(h1); return ingress; }
action a(n) {
  modify_field(h1.f2, n);
  modify_field(standard_metadata.egress_spec, 1);}
action b() {
  modify_field(standard_metadata.egress_spec, 2);}
table t {
  reads { h1.f1 : exact;} actions { a; b; }}
control ingress { apply(t); }
```

Figure 1: A very simple P4 program

2.2 P4

The limitations of OpenFlow and the need for expressiveness has lead to the introduction of P4, a high level declarative programming language for expressing the behavior of packet processors. In P4, one can program a custom parser for their own protocol header, define flow tables with customized structure, define the control flow between the tables, and define custom actions. Hence, P4 allows the developers to specify how a packet processor should process its incoming packets (note, however, that P4 does *not* provide a mechanism to populate the flow table entries; this is done by the controller). A P4 compiler then translates the P4 program into the instruction set of the hardware of the packet processor on which the program is installed (the *target*).

We briefly describe the basic notions of the language here. Section 3 discusses the language construct in more details. A P4 program specifies at least the following components [15]. *Header types:* Each specifies the format (the set and size of fields) of a custom header within a packet. *Instances:* Each is an instance of a header type. *Parser:* A state machine describing how the input packet is parsed into header instances. *Tables:* Each specifies a set of fields that the table entries can match on and a set of possible actions that can be taken. *Actions:* Each (compound action) is composed of a set of primitive actions which can modify packets and state. *Control flow:* Describes the custom conditional chaining of tables within the packet processor's pipeline.

For example, in Figure 1, `h_t` is a header type consisting of two 8 bit fields `f1` and `f2`. Header `h1` is an instantiation of `h_t`. The parser consists of a single state `start` which *extracts* `h1` from the input packet. There are two compound actions `a` and `b` in the program. The actions use the `modify_field` primitive action. The entries in table `t` match on field `f1` in `h1` and if applied, may call actions `a` or `b`. An entry calling `a` must provide a value for `n`. The `ingress` pipeline in this program only consists of applying table `t`.

P4 programs operate according to the abstract forwarding model illustrated in Figure 2. For each packet, the parser produces a *parsed representation* comprised of header instances and sends it to the *ingress* match+action pipeline. Ingress, among other things, may set the *egress specification* which determines the output port(s). Ingress then submits the packet to the queuing mechanism the specification of which is out of the scope of P4. The packet may also go through the optional *egress* pipeline which may make further modifications to the packet. Finally (if not dropped) the parsed representation will be *deparsed* (i.e serialized) into the packet which will be sent out.

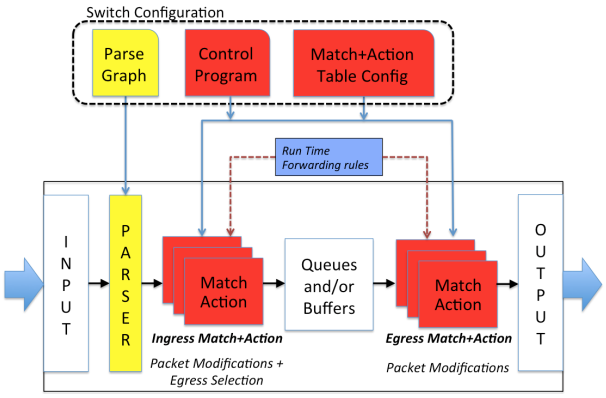

Figure 2: P4$_{14}$ Abstract Forwarding Model [15]

P4 also supports re-circulation (looping packets inside the device) and cloning of packets.

The P4 Language Consortium (http://p4.org) provides the official specification of the language, as well as various other tools including compiler front ends, software interpreters, runtime and testing infrastructure, etc. There are two versions of P4 in current use, P4$_{14}$ and P4$_{16}$. P4$_{16}$ has been released by the consortium in May 2017 [14] and it is much simpler and cleaner than P4$_{14}$, at the cost of deliberately breaking backwards compatibility with P4$_{14}$. There are, however, important P4$_{14}$ program which have not been translated to P4$_{16}$, and, indeed, the P4$_{14}$-to-P4$_{16}$ translator provided by the consortium is not semantics-preserving [27]. Ideally, we would like to prove the translator correct, but for that we need formal semantics of both P4$_{14}$ and P4$_{16}$. In this paper we only discuss our formal semantics of P4$_{14}$, leaving that of P4$_{16}$ and the correctness of the translator as future work. Throughout the paper, we refer to P4$_{14}$ simply as P4.

2.3 Challenges in Formalizing P4

P4 has several characteristics that make it a challenging target for formalization. Here we discuss some of this challenges and the way we dealt with them.

Unstable language: P4 is a relatively young language and it takes time for the community to reach consensus on its design. When we started, the only publicly available version of P4 was P4$_{14}$ v. 1.1.0. That version soon was deprecated and replaced with v. 1.0.3 which we initially used to develop our semantics. In the middle of our formalization effort, P4$_{16}$ v. 1.0.0 as well as a minor revision of P4$_{14}$ (v. 1.0.4) were released. Thanks to K's support for modular definitions and reuse, we were able to rapidly adapt to changes and finalize our semantics according to P4$_{14}$ v. 1.0.4. Through continuous discussions with the P4 designers, we hope to help them reach more stable versions sooner.

Imprecise specification: Since P4 is a newly developed language its specification is not free of problems. There are many inconsistencies and corner cases which are not discussed (Section 3). One of the important contributions of this paper is the identification of these problems through rigorous formalization. We have reported some of the problems to the community. We are also working on a modified version of the specification [46] addressing the issues we found.

No comprehensive test suite: Similar formalization efforts for other languages (e.g. [23, 32, 53]) rely heavily on official test suites. Unfortunately, there is no official test suite for P4. To alleviate this problem, in addition to testing our semantics against a test suite we obtained from a P4 compiler, which only covers about half of our semantic rules (Section 4), we hand-crafted a test suite that gives a complete coverage of our semantic rules.

Unconventional input: The input to a P4 program is different from that of conventional programming languages. P4 has two sources of input. One is the stream of incoming packets that the device running the P4 program needs to process. The other is the table entries and configurations that are installed by the controller at runtime. The mechanism by which the controller interacts with the target at runtime is device-specific and is therefore out of the scope of the language specification. Still, to be able to execute and analyze P4 programs, for the target-specific language features we tried to provide the most unrestricted executable semantics; for example, if the order of some operations was unspecified then we chose a non-deterministic semantics, so we can still explore the entire state-space of behaviors using the K tools.

We also grouped most of the target specific semantic rules in a separate semantic module. This way, the semantics is parametric on the target specific details. One can provide a new target specific module to change the target specific behavior, without the need to touch the rest of the semantics. We have already used this feature when we were testing our semantics against the p4c test suit as it contained target specific features and assumptions (Section 4).

2.4 The K Framework

K [56] is a programming language semantics engineering framework based on term rewriting. Its underlying philosophy is that tools for a language can and should be automatically derived from the formal semantics of that language. Indeed, K provides an actively growing set of language-independent tools, i.e., tools which are not specific to any language but apply to any language which has a K formal semantics. These include a parser, an interpreter, a symbolic model checker, a sound and (relatively) complete deductive program verifier, and, more recently, a cross language program equivalence checker and a semantic-based compiler. Some of the tools are useful during the formalization process itself, the most important of which is the interpreter. Using the interpreter, the semantics can be tested against potentially many programs to gain confidence in its correctness.

To define a programming language in K, one needs to define its syntax and its semantics. Syntax is defined using BNF grammars annotated with semantic attributes. Semantics is given using rewrite rules (also called semantic rules) over configurations. A configuration is a set of potentially nested cells that hold the program and its context. Each cell contains a piece of semantic information of the input program such as the its state, environment, storage, etc. Semantic rules are transitions between configurations: if parts of the configuration match its left hand side, rewrite those parts as specified by the right hand side. For example, this is a rule taken

from P4K (modified for presentation) concerning reading the value of field F from instance I:

$$\left\langle \frac{I.F}{V} \cdots \right\rangle_{\text{k}} \ \langle \langle I \rangle_{\text{name}} \ \langle true \rangle_{\text{valid}} \ \langle \cdots F \mapsto V \cdots \rangle_{\text{fieldVals}} \cdots \rangle_{\text{instance}}$$

The contents inside each matching pair of angle braces constitutes a cell, with the cell name as subscript. The k cell contains the list of computations to be executed. The fragment of computation at the front of the list (the left most) is executed first. There are multiple instance cells each corresponding to a header instance. name contains the name of the instance and valid keeps its validity state. fieldVals is a map from each field name to the value stored in the field in the given instance. The ellipsis are part of the syntax of K and denote contents irrelevant to the rule. The horizontal line denotes a rewrite. If the configuration matches the pattern, the part of the configuration above the line will be replaced by the content below the line. The rest of the configuration remain intact. A rule may contain multiple rewrites at different positions of the configuration. In that case, all rewrites will be applied in one step.

This example illustrates two properties of K that makes it suitable for giving semantics specially to evolving programming languages like P4. First, note that the actual configuration contains many more cells and each cell may contain multiple elements, but the rule only mentions the cells that are relevant. The *configuration abstraction* feature of K automatically infers what the rest of cells should be. Second, note that rewrites are local. There is no need to rewrite the whole configuration. These two features make K rules succinct and human readable. More importantly, they enable modular development of the semantics: if the language specification adds or modifies a language feature the rules irrelevant to that feature do not need to be modified.

3 P4K

P4K is the most complete executable formal semantics of $P4_{14}$. It is based on the latest official language specification (v. 1.0.4 [15]) and on discussions with the language designers. Our work is open source and is available online [45]. The formalization process took 6 months to complete by a PhD student with some familiarity with the K framework. Most of the time was spent learning K and understanding the details of the P4 specification, including its problems. P4K contains more than 100 cells in the configuration, 400 semantic rules, 200 syntax productions, and 2000 lines.

3.1 Syntax

The language specification provides a BNF grammar, whose conversion to K was straightforward. We mostly copy-pasted the grammar and made a few minor modifications to make it compatible with K.

During this process, in addition to minor problems, we identified [36] an ambiguity in the syntax between the minus sign in a constant value (for specifying negative constant values) and the unary negation operator. This ambiguity has important semantic effects. In P4, all the field values have a bit width associated with them. According to the specification "For positive [constant] values the inferred width is the smallest number of bits required to contain the value". Also "For negative [constant] values the inferred width is one more than the smallest number of bits required to contain the positive value" [15]. So for example −5 interpreted as a negative constant would yield a 4 bit value while if interpreted as negation of a positive constant would yield a 3 bit result. Used in an expression with other operators, this difference may affect whether the expression overflows or not, which subsequently may affect the final result.

3.2 Configuration

The configuration contains more than 100 cells. Figure 3 shows part of it, featuring more important cells. All of the language constructs including headers, instances, parser states, actions, tables, control flows, etc have respective cells in the configuration containing their static information and/or runtime state. For example the tables cell will contain a set of table cells ("*" denotes multiple cells with the same name). Each of the table cells contains a table's static information such as its name, the fields to match (reads), and possible actions (acts). It also contains runtime information such as the entries installed in the table.

Some cells contain the execution context. For example during the execution of an action, the stackFrame cell holds a stack of maps from each formal parameter of the executing action to the respective argument values passed to the action.

The cells in and out contain the input and output packet stream from/to all ports respectively. packetin contains the current packet being processed and packout contains the packet being serialized.

Cells are populated or modified by processing the input P4 program before execution, during the initialization, or during the execution as discussed next.

3.3 Semantics

After parsing, the P4 program populates the k cell and is executed with the semantics rules.

3.3.1 Execution Phases. The rules describe the P4 program execution, in three phases.

Preprocessing: In this phase, P4K iterates over all the declarations in the input P4 program (in the k cell), creating and populating the corresponding cells and preparing the configuration for execution. In some cases auxiliary information is pre-computed for the execution phase. An important such computation is the inference of the order of packet headers for deparsing. Details will be discussed in Section 3.3.2.

Initialization: There is an optional initialization phase after preprocessing. It is used primarily to prepopulate the tables and packet buffers before the execution in certain analysis such as symbolic execution. The tables and packet buffers can also be populated at runtime in normal execution.

Runtime: The actual execution of a P4 program happens during this phase. It implements the abstract forwarding model. Packets are taken from the input packet stream and processed using the entries installed in the match+action tables by going through the ingress and potentially the egress pipelines. The output packets are appended to the output packet stream. This phase never terminates.

3.3.2 Language Constructs and Semantics. We briefly describe the language constructs and primarily focus on interesting findings and relevant semantics.

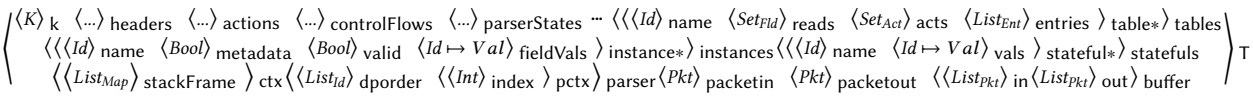

Figure 3: Part of the P4K configuration. The ellipsis symbols indicate omitted cells.

Header types: Each header type is a named declaration that includes an ordered list of fields and their attributes (e.g. field width and signedness). P4 also allows declaration of variable length headers. During our formalization, we found corner cases (e.g. [40]) in which the semantics of such headers are not completely clear.

Instances: Instances may be referenced in various runtime stages including parsing, table matching, and action execution. Some instances, called *header instances* (although the naming is not consistent throughout the specification [41]), keep the parsed representation of the respective packet headers (i.e., the packet header is extracted into the header instance). Other instances, called *metadata*, keep arbitrary per packet state throughout the pipeline. For example `h1` in Figure 1 is a header instance and `meta` in Figure 5 is a metadata. In our semantics, both types of instances are kept as `instance` cells, distinguished by their `metadata` cells (Figure 3).

It is also possible to declare fixed size, one dimensional array instances (called *header stacks*), as sequences of adjacent headers (e.g. to support MPLS [20]). We keep array elements as separate header instances, with special names that include their index. Otherwise, the elements are treated same as other instances.

Header instances are invalid (uninitialized) until validated in parsing or by specific primitive actions in match+action processing. According to the specification, reading an invalid header results in an undefined value, whose behavior is target dependent. We model this using a special value `@undef`. Use of `@undef` in an expression or action call causes the execution to get stuck by default. We use this feature to detect unportable code (Section 5.1).

Hash generators: The ability to calculate a hash value for a stream of bytes has various uses in networking. P4 provides the ability to declare hash generators (called *field list calculations*). The developer provides a list of values (declared using a *field list* struct) and selects a hash generation algorithm. The hash generator computes the hash of the bitstream generated from the list. In the example below `ipv4_checksum` is a hash generator for the `ipv4_checksum_list` field list (declaration omitted) with the IPv4 checksum algorithm (`csum16`).

```
field_list_calculation ipv4_checksum {
    input {ipv4_checksum_list;}
    algorithm : csum16; output_width : 16; }
```

The language specification identifies a set of well known hash generation algorithms (e.g., IPv4 checksum and CRC). In our semantics, we treat hash generation as a black box; K allows us to "hook" library function calls that implement the desired functionality. It is possible to also directly specify the algorithms using K rules inside the semantics, but we did not find any compelling reason to do so.

We found a problem [43] with the specification during the formalization of field lists. Each element of a field list can refer to a field in an instance, an instance itself (when all the fields in that instance are used), another field list (when all the fields identified by that list are used), a constant value, or the keyword `payload`. According to the specification "payload indicates that the contents of the packet following the header of the previously mentioned field is included in the field list" [15]. However, "previously mentioned field" is ambiguous. For instance, below it is not clear if payload refers to f1 or f2.

```
field_list fl1 { h.f1; }
field_list fl2 { h.f2; fl1; payload; }
```

Thus we do not provide semantics for payload. $P4_{16}$ has replaced field lists with a C-like *struct* construct, disallowing the payload keyword.

Checksums: A field in a header instance can be declared to be a *calculated field*, indicating that it carries a checksum. The developer provides a hash generator for verification of the checksum at the end of parsing, and/or an update of the checksum during deparsing. For example, below, the field `hdrChecksum` in the header instance `ipv4` is declared to be a calculated field which uses the hash generator `ipv4_checksum` for its verification and update.

```
calculated_field ipv4.hdrChecksum  {
    verify ipv4_checksum; update ipv4_checksum;}
```

The P4 specification leaves undefined the order in which the calculated fields must be updated or verified. For verification, the order can matter depending on the target. For update, the order can matter in cases where the field list calculation of a calculated field includes another calculated field. After discussing [44] with the language designers, to obtain the most general behavior, we decided to choose a non-deterministic order for update and verify. K provides a search tool which one can use to explore all possible non-deterministic outcomes to check whether they differ. (Section 5.2).

Parser: The user can define a parser to deserialize the input packet into header instances (the *parsed representation*). The parser is defined as a state machine. In each state, it is possible to *extract* header instances (i.e., copy the data from the packet at the current offset into respective field values for the given instances) and to modify metadata. Then, it is possible to conditionally transition to another state, to end the parsing, or to throw an (explicit) exception. For example, in state `parse_ethernet` in Figure 4, after extracting the `ethernet` header, based on the value of the `etherType` field, the parser may transition to the parser state `parse_ipv4` or end the parsing and start the ingress pipeline.

Exception Handlers: P4 allows us to declare exception handlers for implicit or explicit parser exceptions. In case an exception occurs, parsing is terminated and the relevant handler is invoked. Each handler can either modify metadata and continue to ingress or immediately drop the packet. There is a default handler that drops the packet. For example in Figure 1 if a packet is too short for the extraction of `h1`, an implicit exception is thrown and the default handler drops the packet.

Deparsing: This is the opposite of parsing. At egress, the (potentially modified) valid header instances are serialized into a stream of bytes to be sent. An important question in deparsing is the order in which the header instances should be serialized. The parsing order is not enough to find the deparse order, since header instances might also be added (validated) or removed (invalidated) in the

match+action pipeline. According to the specification "[A]ny format which should be generated on egress should be represented by the parser used on ingress" [15] and the order of deparsing should be inferred from the parse graph.

If the parse graph is acyclic, a topological order can be used as the deparsing order. However, in general the graph might be cyclic as there may be recursion in parsing. While in simple cases an order can still be inferred (e.g., cases where recursion is used for the extraction of header stacks), there are cases in which a meaningful order can not be inferred. This is a well known problem [11]. In our semantics, we support simple cases of cyclic parse graphs. All of the practical examples we have seen so far can be handled by our semantics.

$P4_{16}$ has switched to an approach in which the deparse order is explicitly defined by the programmer.

Stateful Elements: P4 supports *stateful* language constructs that can hold state for longer than one packet, as opposed to per packet state in instances. *Counters* count packets or bytes, *meters* measure data rates, and *registers* are general purpose stateful elements. The declaration of each of these elements creates an array of memory units. The units may be *direct*ly bound to the table entries. In that case, the (counter and meter) units will automatically be updated when the corresponding entry is matched in match+action. The units may alternatively be *static*; then they should explicitly be accessed or updated via special primitive actions. For example, in Figure 5, `reg` is a static register with a single 8 bit memory cell.

In our definition we unified all these elements as instances of the `stateful` cell (Figure 3) which can be accessed like registers. Other operations are defined as functions which read and manipulate the registers. Each `stateful` cell in the configuration has a map from an index to a value. The index is either a table entry id (for direct) or an array index (for static). The mechanism of updating meters is target specific (not part of the language specification). Subsequently we do not perform any action in case a meter is updated. If needed, one can add a mechanism in our target specific module.

The specification does not specify [38] the initial value of the stateful elements. It is sensible to assume that the initial value of counters is 0, and similarly for meters. For registers, by default we initialize the registers to `@undef`. Moreover, the specification is inconsistent [39] about whether direct meters are allowed to be explicitly updated by table actions. To be consistent with counters and registers, we assume they are allowed.

Finally, if multiple counters/meters are directly bound to the same table, the specification does not state [44] the order in which the elements must be updated when an entry in that table is matched. The order can affect the outcome in multi-threaded packet processors (Section 3.4) as there may be data races over stateful elements. Again, we choose a non-deterministic order for updating the counters/meters, so we can systematically explore it using K's search.

Actions: *Compound* actions are user defined imperative functions that can take arguments and if called, perform a sequence of calls to other compound actions or built-in *primitive actions*. Primitive actions provide various functionality including arithmetic, addition/removal/modification of instances and header stacks, access/-modification of stateful elements, cloning, re-circulation, dropping the packet, etc. Actions are executed as a result of table matches.

We formalized all the primitive actions (see Section 3.5 for limitations on clone primitive actions). The specification does not specify the behavior of some corner cases, such as shift with negative shift amount. We intentionally do not provide semantics for such cases to detect unportable code.

The previous version of the specification [12] stated that all primitive actions resulting from a table match execute in parallel, making it unclear what the meaning of the following:

```
action a() {
  modify_field(h.f, 1); modify_field(h.f, 2); }
```

`modify_field(f,v)` is a primitive action that updates field `f` with `v`. The latest revision (1.0.4) switched to sequential semantics, so we do not have to deal with this case anymore.

Tables: Tables will be populated at runtime by the controller. Each entry provides values for the fields that are specified in the declaration, an action that should be executed if the entry is selected, and arguments to be passed to the action.

The interaction mechanism between the controller and P4 target is out of the scope of the specification. Hence, the answer to questions such as what happens if a table is modified by the controller while it is being applied on a packet is target dependent. We currently assume that modification and application of the same table are mutually exclusive.

P4 provides various matching modes per each field. For example, *exact* matches exact numbers and *ternary* matches ternary bit vectors. It is also possible to associate priorities with table entries. In case more than one table entry is applicable, the rule with the highest priority will be selected. *Longest Prefix Match (LPM)* is a special kind of ternary match useful for IP prefixes. The specification specifies how the relative priority of an entry with LPM match can be inferred bases on the corresponding match value of the entry. However, it does not specify how the priority should be decided in cases where there are more than one field with LPM match type [37]. We assume all entries have explicit (unique) priorities regardless of their match types. We keep the entries sorted in their descending order of priority. To apply a table on a packet, we iterate over the entries in order and select the first matching entry.

Control Flow: User defines the order and the conditions under which various tables are applied to a packet using *control functions*. The body of a control function is a control block consisting of a sequence of control statements. A statement might apply a table, call a control function, or conditionally select a control block. Ingress is a special control function that is automatically called after (successful) parsing. Egress is another (optional) special control function. If defined, it will automatically be called when the queuing mechanism takes the packet to be sent out.

Other contstructs: We omit the discussion of *value sets*, *action profiles*, and *action selectors* as well as many details of the discussed constructs. Interested readers can refer to the semantics [45] for more details.

3.4 Concurrency Model

Real world high performance packet processors have multiple threads of execution. The specification is silent about the concurrency model. As a result, what constitutes a thread depends on the target hardware. In our semantics, we support a multithreading model in which each thread individually does all of what a single

threaded program does by addition of a few more cells and rules[1] (similar to Section 3.6). The input/out packet streams, the tables, and the stateful elements constitute the shared memory between the threads.

3.5 Limitations

P4 provides four primitive actions for cloning a packet under process from ingress/egress to ingress/egress. The actions that clone a packet into the egress put the clones in the queue between ingress and egress pipelines. Since we currently do not model the ingress and egress pipelines as separate threads, we only support a single packet in the queue between the two. Therefore, we do not directly support clone into egress. Instead, we treat such clones as new incoming packets with auxiliary flags to skip the ingress pipeline.

3.6 Network Semantics

It is useful to be able to simulate or analyze a network of P4 programs rather than just a single program (Sections 5.3.2 and 5.2). In order to do so, we need the semantics of the network. Thanks to the modularity of K, we easily modeled the semantics of a P4 network without changing the P4 language semantics. We only needed to add a few more cells and preprocessing rules. We added a root `nodes` cell containing multiple `node` cells each containing the configuration of a P4 program plus a `nodeId` cell. We also added a `topology` cell which holds the connection between the nodes.

To model the network links, we added a single rule that takes a packet from the end of the output stream of one node and puts it at the beginning of the input stream of the node it is connected to. If needed, one can also model packet loss in the links by a single additional rule. Note that here we have multiple threads of concurrent execution, whose interleaving is non-deterministic. The thread interleaving space can be explored using the K search mode (Section 5.2).

4 EVALUATION

K provides us with an interpreter derived automatically from the semantics, enabling us to test our semantics. Official conformance test suits are an ideal target for testing executable semantics. Unfortunately, P4 does not have such a test suite. A new official P4 compiler front end (p4c [16]) has a limited set of tests for $P4_{14}$, which we used in our evaluation.

Generally, it is non-trivial to port tests across different implementations of P4, as its IO is not specified (Section 2.3). Fortunately, the p4c tests were easy to adopt. Each test, along with the P4 program under test, contains an STF file. The file describes table entries, input packets, and the expected output packets. We systematically converted the STF files into our test format.

The suite contains tests with minor issues including use of deprecated syntax, unspecified constructs, or unspecified primitive actions. We fixed the issues by slightly modifying to the corresponding P4 programs or test files, and implementing the primitive actions in a target specific module for the tests. Moreover, the tests assume undefined[2] egress specification leads to packet drop. The specification does not specify the behavior in this case, so it is target dependent. In our semantics, by default the execution gets stuck in such cases. In our target dependent module for the tests, we added a rule to drop the packet in such cases.

The tests also helped us identify a few problems in P4K. For example, we found that we had misunderstood the semantics of a primitive action (pop). Note that push and pop have rather an unusual semantics in P4 [42].

After fixing the problems with the tests and our semantics, P4K passed 39 out of the 40 test. The failing test[3] has multiple inferable deparsing orders. The order chosen in our execution happens to be different from the order the test expects. We verified that both orders are possible.

Inspired by [53], we measured the percentage of the semantics rules exercised by the tests (the *semantic coverage* of the tests). The tests cover under 54% of the semantics and miss many of the semantic features. We have also manually developed 30 tests during our formalization process. Together, these 70 tests cover almost all the semantic rules.

Each test took 19.5s (± 3.2s) on average with the maximum of 125s.[4] We note that approx. 10s out of this time is the startup time of K and is not related to execution. We also note that K has multiple backends. We use an open source backend [18] which is relatively very poor in terms of performance. We expect the runtime to improve by orders of magnitude on performant commercial backends (e.g. [30]).

5 APPLICATIONS

Besides defining a formal semantics for P4 and thus helping make the P4 specification more precise, a secondary objective of our effort was to make use of the various tools that K provides. We demonstrate how the tools can be useful for the P4 developers and network administrators, as well as for the P4 language designers and compiler developers.

5.1 Detecting Unportable Code

As seen above, in some cases the P4 specification does not provide the expected behavior of the program. P4 programs exhibiting such unspecified behavior may not be portable among different targets and compilers. It is not wise to solely rely on the expertise of P4 developers in the low level details of the specification to check if their code is portable. It is desirable to have tools that automate this check. For simple cases, such behavior may be detectable by syntactic checks. In general, unspecified behavior may depend on the input.

By default, we do not provide semantics for cases which are not covered by the language specification. If the execution of a program reaches a point with unspecified behavior, the execution gets stuck. Avoiding over-specification therefore allows us to check for unspecified behavior in P4 programs. This is done simply by running the program and checking whether it reaches a state in which it gets

[1] In all of our experiments we used only a single thread for each P4 program. Throughout the paper we assume executions are single threaded.

[2] According to the specification egress specification is undefined unless set explicitly. We model this using the `@undef` value.

[3] Namely `parser_dc_full.stf`.

[4] All experiments are run on a machine with Intel Xenon CPU ES-1660 3.30GHz and 32GB DDR3 1333MHz RAM.

stuck or not. The check can be performed using either concrete or symbolic inputs. We show a symbolic example in Section 5.3.

To tune the semantics for a specific target, one can provide custom semantics for cases with unspecified behavior in the target specific module.

5.2 State Space Exploration

K provides a *search* execution mode which allows us to explore all possible execution traces when non-determinism is present. In K, non-determinism occurs when more that one rewrite rule is applicable, or the same rule is applicable at multiple positions in the configuration. In normal execution mode, only one of the applicable rules is (non-deterministically) selected. In the search mode, all the applicable rules are explored. Moreover, the user can explicitly control the points in which non-determinism is explored. This allows one to focus on exploration of one or more specific sources of non-determinism and ignore the rest.

There are two sources of non-determinism in P4K. The first is due to our approach to model the most general behavior. Examples are order of deparsing, order of update and/or verification of calculated fields, and order of update of direct stateful elements. The second is due to the existence of multiple threads of execution. These include the threads of execution inside a single P4 program, as well as the execution of multiple nodes in a P4 network. Both sources can be explored using the search mode. We have already shown in Section 4 how exploration of the order of deparsing can be useful. The benefits of exhaustive analysis of thread interleavings in concurrency analysis are well known.

5.3 Symbolic Execution

K allows the configuration to be symbolic – i.e., to contain mathematical variables and logical constraints over them. During execution with a symbolic configuration, K accumulates and checks (using Z3 [21]) all the logical constraints over the execution path – i.e the conditions under which the rules are applicable to respective states. Under the hood, there is no difference between symbolic and concrete execution. Symbolic execution powers some of the other K tools such as the program verifier (Section 5.4) and the equivalence checker (Section 5.5). It can also be useful on its own, say, to search for bugs in P4 programs and data planes.

5.3.1 Search for Bugs. To illustrate one application, we choose a community provided sample P4 program which defines a very basic L3 router [4]. Using symbolic execution, we find input packets for which the program fails to specify the egress specification, leading to unspecified behavior.

To do so, we prepopulate the tables with entries from the unit test provided along with the program. We then simply start the program with a single symbolic packet (P) from a symbolic port in the input packet stream (the `in` cell). Our goal is to find an input packet that leads the program to a state in which neither packet is dropped, nor its egress specification is set. We run the program in the (symbolic) search mode. The search returns multiple inputs which can lead to undefined egress specification. Here we only discuss one of the more interesting ones: the search result suggests that if "P has ethernet as its first header and ethernet.etherType != 0x0800", then the program will end up with an undefined egress.

```
...
parser start {return parse_ethernet;}
parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    0x0800 : parse_ipv4;
    default: ingress;
  }}
control ingress { if (valid(ipv4)) { ... } }
```

Figure 4: Part of a basic L3 router [4]

Figure 4 shows the relevant snippet of the program. A simple manual inspection confirms the finding. The parser extracts the `ethernet` header and checks `etherType`. If it is equal to 0x0800 (i.e the IPv4 ether type), the parser then proceeds to extracting the `ipv4` header (not shown). Otherwise, instead of, say, dropping the packet, the program starts the ingress pipeline. At the beginning of the ingress, the program checks the validity of the `ipv4` header. If valid, the pipeline applies a sequence of tables that may set the egress specification (not shown). Otherwise the program does not apply any tables and the egress specification remains undefined. Thus, under the given constraints, packet is not dropped, `ipv4` is invalid, and the egress specification is undefined.

5.3.2 Data Plane Verification. There is a growing interest towards *data plane verification* tools such as [33–35, 47, 59]. These tools analyze the table entries in a snapshot of the data plane and look for violation of properties of interest. The verification of these properties usually requires answer to queries of the following form: *What kind of packets from node A will reach node B*? While using various smart ideas to achieve better performance, all these tool are based on the same basic idea: symbolic reasoning over the space of packet headers.

Using our semantics, we can answer such queries by inserting a symbolic packet at, say, node A and using symbolic execution to find the constraints on the packets that end up at node B. The tools mentioned above use simplified hardcoded/adhoc models of packet processors in their analysis and miss the internal details of such devices. They need to be re-engineered to change their model of packet processors. There is no such need in our case. Moreover, as will be shown in the next section, these tools can verify a very restricted class of properties. We eliminate these limitations.

5.4 Program Verification

K features a language independent program verification infrastructure based on Reachability Logic [18]. It can be instantiated with the semantics of a programming language such as P4 to automatically provide a sound and relatively complete program verifier for that language. In this system, properties to be verified are given using a set of reachability assertions, where each reachability assertion is written as a rewrite rule. A reachability assertion asserts that starting from any configuration matching the left hand side of the assertion, by execution using the input semantics, one will either eventually reach a configuration that matches the right hand side of the assertion or never terminate.

The standard pre/post conditions and loop invariants used in Hoare style program verification can be encoded as reachability assertions. Intuitively, a Hoare triple $\{P\}C\{Q\}$ becomes "$C \wedge P$ *rewrites to* $. \wedge Q$" where "." is the empty program [32].

5.4.1 The Load Balancer Program. To showcase the use of the program verifier, we provide a simple P4 program and verify a simple property about it. The program above is meant to balance

```
header_type meta_t { fields { reg_val : 8; } }
metadata meta_t meta;
parser start{ return ingress; }
register reg { width: 8; instance_count: 1; }
action read_reg(){
  register_read(meta.reg_val,reg,0); }
table read_reg_table{
  reads{ meta.valid : exact; }
  actions{ read_reg; } }
action balance(port,val){
  modify_field(standard_metadata.egress_spec,port);
  register_write(reg, 0, val);}
table balance_table{
  reads{ meta.reg_val : exact; }
  actions{ balance; } }
control ingress{ apply(read_reg_table); apply(
    balance_table); }
```

Figure 5: A simple load balancer

its incoming packets (from any port) between two output ports. This is done using a register whose value alternates between 0 and 1 across incoming packets. The program features a single register, a metadata instance, and two tables. The parser starts ingress without extracting anything. We install a single entry in `read_reg_table` to call action `read_reg`. The action copies the register value (at index 0) into `meta.reg_val` [5]. We install two rules in `balance_table`. One rule matches if `meta.reg_val = 1` and calls `balance(1,0)`. The other rule matches if `meta.reg_val = 0` and calls `balance(0,1)`. `balance(p,v)` modifies the register (at index 0) with value `v` and effectively sends the packet to port `p`. Our goal is to prove that this program (along with its table entries) correctly balances the load. Specifically, we want to prove the following:

Property: For any input stream of packets, after processing all the packets, no packet is dropped and no new packet is added; all the packets in the output are either sent to port 0 or port 1; and the absolute difference between the number of packets sent to ports 0 and 1 is less than or equal to 1. Albeit simple, none of the data plane verification tools mentioned above are capable of proving it. They lack either support for stateful data plane elements or support for reasoning over an unbounded (i.e symbolic) stream of packets.

In K, the property is captured by the following reachability assertion. For presentation purposes we have omitted the less relevant, mostly static parts of the specification which hold the program and the table entries. The full specification can be found in [45].

$$\left\langle \frac{@execute}{@end} \right\rangle_{\text{k}} \left\langle \left\langle reg \right\rangle_{\text{name}} \left\langle 0 \mapsto \frac{0}{\_} \right\rangle_{\text{vals}} \right\rangle_{\text{stateful}} \left\langle \frac{I}{\cdot} \right\rangle_{\text{in}}$$

$$\left\langle \frac{\cdot}{?O} \right\rangle_{\text{out}} \quad \text{ensures} \quad \begin{array}{l} \|onPort(?O,0) - onPort(?O,1))\| \leq 1 \\ \wedge onPort(?O,0) + onPort(?O,1)) = size(?O) \\ \wedge size(?O) = size(I) \end{array}$$

In the specification above, `@excute` is the program state right before the execution starts. `@end` is state after all the input packets are processed[6]. I is a (universal) symbolic variable representing the input packet stream and $?O$ is an existential symbolic variable representing the output stream. Symbol "." in both `in` and `out` cells represents an empty packet stream. The rewrite in the `vals` cell says that the value of the register `reg` (at index 0) is 0 at start[7], and its value at the end is not relevant to the assertion. The keyword `ensures` adds logical constraints on the right hand side of the assertion (i.e the post condition). Function `onPort(s,p)` returns the the number of packets in stream s belonging to port p. Function `size(s)` returns the length of stream s.

Our semantics of P4 contains a main loop over the stream of input packets. Since in our property the input is a symbolic list with an unbounded length, similar to Hoare logic, we need a loop invariant. To prove our property, we provide the loop invariant as the following reachability assertion:

$$\left\langle \frac{@nextPacket}{@end} \right\rangle_{\text{k}} \left\langle \left\langle reg \right\rangle_{\text{name}} \left\langle 0 \mapsto \frac{R}{\_} \right\rangle_{\text{vals}} \right\rangle_{\text{stateful}} \left\langle \frac{I}{\cdot} \right\rangle_{\text{in}}$$

$$\left\langle \frac{O1}{?O2} \right\rangle_{\text{out}} \quad \begin{array}{ll} \text{requires} & \begin{array}{l} (R = 1 \wedge onPort(O1,0) = onPort(O1,1)) + 1 \vee \\ R = 0 \wedge onPort(O1,0) = onPort(O1,1))) \\ \wedge onPort(O1,0) + onPort(O1,1) = size(O1) \end{array} \\ \text{ensures} & \begin{array}{l} \|onPort(?O2,0) - onPort(?O2,1))\| \leq 1 \\ \wedge onPort(?O2,0) + onPort(?O2,1) = size(?O2) \\ \wedge size(I) + size(O1) = size(?O2) \end{array} \end{array}$$

Here, `@nextPacket` is the head of the main loop over the input packet stream. Keyword `requires` puts logical constraints on the left hand side of the assertion (i.e the precondition). The assertion reads as: starting from the head of the main loop, given the constraints in `requires` are satisfied, if the program terminates, it will reach an `@end` state that satisfies the constraints in `ensures`.

We gave the two assertions to the K's program verifier instantiated with our P4 semantics. The verifier successfully proved the loop invariant and the first reachability assertion (i.e., the desired property). The verification took about 80s.

In this example, we used concrete table entries as the entries are part of the functionality that we aimed to verify. In general, depending on the property, the tables – as well as anything else – can be symbolic. We show an example in the next section.

5.5 Translation Validation

P4 programs eventually need to be compiled into the instruction set (i.e the language) of the target hardware for execution. With any compilation, there is the question of whether or not the semantics of the input program is preserved by the compiler. Currently the compilers usually lack formal semantic preservation guarantees since providing such guarantees requires a significant effort. The issue is even more pronounced when sophisticated compiler optimizations are involved. A promising alternative approach is to verify each instance of compilation instead of the whole compiler. This approach, known as *tranlation validation* [54], aims to verify the semantic equivalence of a program and its compiled counterpart, potentially using hints from the compiler.

Recently, K has introduced a prototype tool (named KEQ [57]) for cross language program equivalence checking using a generalized notion of bisimulation. The notion enables us to mask irrelevant intermediate states and consider only the relevant states in comparing two program executions. KEQ takes the K semantics of the

[5] This is done because register values can not directly be matched in tables.

[6] This state is only added due to technical reasons for verification purposes, as actual P4 programs never terminate. We add a rule causing the program to jump to this state once the input packet stream becomes empty.

[7] We made the assumption that registers are initialized to 0.

two programming languages, two input programs written in the respective languages, and a set of synchronization points as input, and checks whether or not the two programs are equivalent.

Each synchronization point is a pair of symbolic states (called *cuts*) in the two input programs. The meaning of synchronization is defined by the user as a logical constraint over the given pair of symbolic states. It usually consists of checking the equality of certain relevant values. Each cut in the pair is essentially a pattern over the configurations of the semantics of the respective languages (similar to the right or left hand side of rewrite rules). The user labels one or more synchronization points as *trusted*. These points are assumed to already be bisimilar. Usually one (and the only one) such point is the end of the two programs and the constraint is the equality of the respective output values.

For the rest of the synchronization points, the equivalence checker checks whether the given points are bisimilar. It basically means that starting from the two cuts in a synchronization point, using the semantics of the respective languages, all reachable synchronization points are respectively bisimilar. Normally one such point is the start of the two programs. The constraint is the equality of the respective input values. Additional synchronization points may be needed as well, such as the beginning of unbounded loops. We refer the interested readers to [57] for more details on KEQ.

5.5.1 P4 → IMP+ Translation Validation. We illustrate KEQ through a small example. We check the equivalence of a simple P4 program with a program written in another language. For this purpose, we developed a very simple imperative language called IMP+. The language syntactically resembles C, although semantically it is much simpler. We also developed the semantics of IMP+ in K. We provide a set of API functions for the language to send and receive packets, read tables, etc. The name of these functions are prefixed with the "#" symbol. For simplicity, we directly provide semantics to such functions in our semantics. We chose the simple P4 program in Figure 1 for translation. We manually translate it into IMP+ as follows:[8]

```
int h1_f1; int h1_f2; bool h1_valid;
int sm_egress_spec;
bool parse(){ return start(); }
bool start(){
  if (! #has_next(8)){ return false; }
  h1_f1 = #extract_next(8, false);
  if (! #has_next(8)){ return false;}
  h1_f2 = #extract_next(8, false);
  h1_valid = true;
  return true;}
void a(int n){
  h1_f2 = n; sm_egress_spec = 1; }
void b(){ sm_egress_spec = 2; }
void apply_t(){
  //p2
  while (#get_next_entry()) {
    if (#entry_matches(h1_f1)){
      #call_entry_action(); return; }}
  if (#has_default_action()){
    #call_default_action(); }}
bool process_packet(){
  #reset();
  sm_egress_spec = -1;
  h1_valid = false;
  if (! parse()){ return false; }
  if (sm_egress_spec == -1){return false;}
  return true;}
void deparse(){
  #emit(h1_f1); #emit(h1_f2); #add_payload(); }
void main(){ //p0
  //p1
  while (#get_next_packet()){
    if (!process_packet()){ #drop(); }
    else{ deparse(); #output_packet();}}}
//p3 [trusted]
```

[8]We assume packets with undefined egress specification will be dropped.

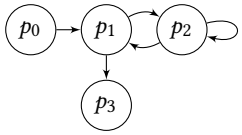

Figure 6: Abstract transition relation between $p_0 \ldots p_3$.

The goal is to prove the equivalence of the two programs. Our notion of equivalence is defined as follows: for any input stream of packets, and for any table entries in table t, at the end of processing all the input packets, the two programs generate the same output stream of packets. To do so, we manually provide a few synchronization points. We have annotated the IMP+ program with the points. Next we informally describe the points and their constraints. The full specification can be found in [45].

p_0 is the start of the two programs. The condition associated with this points is the equality of the respective input streams, table entries, and table default actions.

p_1 is the main loop over the input packets. Its condition is same as p_0's condition plus the equality of the respective current output packet streams.

p_2 is the loop over table entries. The condition is p_1's condition plus the equality of the field values in the parsed representation of the P4 program and the corresponding variables in the IMP+ program, plus the equality of the index of iteration of the tables entries[9], and the equality of the current packet payloads.

p_3 is the end of execution[10]. The condition is the equality of the respective output packet streams. p_3 is *trusted*.

Figure 6 illustrates the abstract transition relation between the points. Each arrow represents multiple rewrite steps in each program, ignoring the irrelevant (possibly non-equivalent) intermediate states of the programs. Note how this abstraction enables us to establish the equivalence even though the programs are written in two quite different programming languages.

Given the synchronization points, KEQ was able to prove the equivalence. Although the program is very simple, we believe it captures the essence of many of the programs that are used in practice. In addition, note that we provided the synchronization points by hand. In practice, the compiler can automatically provide this information as it has enough knowledge during the translation.

[9]Note that in our semantics of both P4 and IMP+, the table entries are sorted in the descending order of their priority.

[10]For technical reasons we assume both programs terminate once the input packet stream becomes empty.

6 RELATED WORK

6.1 Semantics of Programming Languages in K

The K framework has been used to provide complete executable semantics for several programming languages. Here we briefly overview the more relevant work.

KCC [23] formalizes the semantics of C11, passing 99.2% of GCC torture test suit, more than what the GCC and Clang could pass. Later work [31] develops the "negative" semantics of C11 and is able to identify programs with undefined behavior. In our work, we identify unspecified behavior by lack of semantics.

K-Java [8] formalizes Java 1.4 and follows a *test-driven methodology* to manually provide a suite of 800+ tests.

KJS [53] provides semantics for JavaScript passing all 2700+ tests in a conformance test suite [22]. The authors introduce the notion of semantic coverage for test suites which has inspired our work.

In these works, the language design predates the formalization effort by several years. Consequently, although more complex, these languages are quite stable. P4 is still at the early stages of the language design process and is relatively unstable. This made the formalization effort challenging.

Recently, KEVM [32] formalizes the Etherium Virtual Machine [58], successfully passing a suite of 40K official tests. Like P4K, KEVM targets a new language and reveals problems in its specification.

These works (except K-Java) rely heavily on existing tests to provide semantics. In our case, such a comprehensive test suite still does not exist. The only test suite that we found covers less than 54% of our semantics.

6.2 Network Verification

Bugs happen frequently in networks and lead to performance problems, service disruptions, security vulnerabilities, etc. Scale and complexity of networks make answering even the simplest functional correctness queries prohibitively hard to answer manually. This has ignited research into automating the process of *network verification* which can be broadly categorized as follows:

Data plane verification reactively checks network wide properties in a data plane by analyzing snapshots of it. We have already discussed the work in this area in Section 5. None of these tools readily supports P4. An exception is [50] which will be discussed in the next section.

Control plane verification proactively ensures a network is free of latent bugs by analyzing its control plane logic. The literature targets both traditional and SDN control planes. The techniques include static analysis (e.g. [26]), simulation and emulation (e.g. [28, 49]), model checking (e.g. [55]), SMT solving (e.g. [6]), testing (e.g. [10, 24]), control plane abstractions (e.g. [25, 29]), and deductive verification (e.g. [3]). All of these works (except [49]) assume a fixed and simple model for the data plane elements which misses the internal details of the devices.

Although control plane is out of the scope of this work, it it worth mentioning that SDN controllers are usually written using mainstream programming languages [11] for which K semantics exist (e.g. ONOS [7] is in Java and P4Runtime [17] is in C). By combining the K semantics of controller programs with our semantics of P4 data planes, we can analyze a complete model of the whole network. We leave this as future work.

[11] Though there is a recent progress towards high level programming languages for controllers [1, 2, 5].

6.3 Semantics and Analysis of P4

We are not aware of any extensive efforts to formalize the P4 language. P4NOD [50] (on paper) provides a big step operational semantics of a subset of the P4 language. The authors use the semantics to provide a translator from P4 to Datalog. The result is used in P4 data plane verification using a Datalog engine optimized for this purpose [51]. The authors also use the tool to catch a class of bugs, called well-formedness bugs, that are unique to P4 networks. Finally, the authors show an example of P4 to P4 equivalence check.

The primary focus of our work is the language itself and its problems. We provide a modular small step operational semantics for all features of P4. Using K tools, among other things, we too are able to perform data plane verification and detect well-formedness bugs. We have also shown an example of translation validation between P4 and other languages defined in K. In P4NOD the trust base consists of the semantics, the Datalog engine, and the P4 to Datalog translator. In P4K, it consists of the semantics and the K framework. We leave a quantitative comparison of the two works as a future work.

7 CONCLUSION, DISCUSSION, FUTURE WORK

We have presented P4K, the first complete semantics of $P4_{14}$. Through our formalization process, we have identified many problems with the language specification. We automatically provide a suite of analysis tools derived directly from our semantics. We have discussed and demonstrated the applications of some of the tools for P4 developers and designers.

With the introduction of $P4_{16}$, $P4_{14}$ may sooner or later be deprecated, especially because $P4_{16}$ addresses many of $P4_{14}$'s issues through backwards-incompatible changes. Nevertheless, we think that formalizing $P4_{14}$ was a worthwhile effort. There are still important applications written in $P4_{14}$ (e.g. [13]) that do not have a $P4_{16}$ equivalent. The language consortium provides a translator from $P4_{14}$ to $P4_{16}$. However, without a clear semantics of $P4_{14}$, the translation itself might be problematic. We are aware of at least one instance [27] in which the translator's $P4_{16}$ output is not equivalent to its $P4_{14}$ input.

We plan to formalize $P4_{16}$ in near future. We believe transition to $P4_{16}$ will be straight forward. $P4_{16}$ has actually a smaller core language compared to $P4_{14}$.

Beside other future directions discussed throughout the paper, we also plan to use our semantics to analyze real world P4 programs (specially [13, 19]) and networks.

Another interesting use case of our semantics which we leave as a future work is to automatically generate a test suite that covers all of our semantic rules. Such a test suite can be very useful for P4 compiler developers, because they can regenerate the tests each time the language changes.

We conclude by a lesson we learned. It is relatively easy and extremely beneficial to rapidly apply formal methods at the early stages of a language design process. Not only it helps the designers

quickly identify problems in their design and produce more robust languages, but also (in case a framework like K is employed) it can save time and effort by automatically providing various useful tools for the language.

ACKNOWLEDGMENTS

The authors would like to thank the K development team (specially Daejun Park) for their help with the K framework. We would also like to thank Nate Foster and the members of P4 Language Consortium for their support. We thank Brighten Godfrey, Farnaz Jahanbakhsh, and Alex Horn for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1421575.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*. 113–126.
[2] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *SIGCOMM*. 29–43.
[3] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. 2014. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *PLDI*.
[4] Antonin Bas. 2016. Basic Routing Example. https://github.com/p4lang/p4factory/tree/master/targets/basic_routing. (2016).
[5] Ryan Beckett, Michael Greenberg, and David Walker. 2016. Temporal NetKAT. In *PLDI*. 386–401.
[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *SIGCOMM*. 155–168.
[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. 2014. ONOS: towards an open, distributed SDN OS. In *HotSDN*. ACM, 1–6.
[8] Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *POPL*. ACM, 445–456.
[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[10] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test Openflow Applications. In *NSDI*.
[11] The P4 Language Consortium. 2016. $P4_{14}$ Language Specification Version 1.0.3. https://p4lang.github.io/p4-spec/p4-14/v1.1.0/tex/p4.pdf. (January 2016).
[12] The P4 Language Consortium. 2016. $P4_{14}$ Language Specification Version 1.0.3. https://p4lang.github.io/p4-spec/p4-14/v1.0.3/tex/p4.pdf. (November 2016).
[13] The P4 Language Consortium. 2016. switch.p4. https://github.com/p4lang/switch. (2016).
[14] The P4 Language Consortium. 2017. $P4_{16}$ Language Specification Version 1.0.0. https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf. (May 2017).
[15] The P4 Language Consortium. 2017. $P4_{14}$ Language Specification Version 1.0.4. https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf. (May 2017).
[16] The P4 Language Consortium. 2017. P4 Reference Compiler (p4c). https://github.com/p4lang/p4c. (2017).
[17] The P4 Language Consortium. 2017. P4 Runtime. https://github.com/p4lang/pi. (2017).
[18] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *OOPSLA*. ACM, 74–91.
[19] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review* 46, 1 (2016), 18–24.
[20] Bruce S Davie and Yakov Rekhter. 2000. *MPLS: technology and applications*. Morgan Kaufmann Publishers Inc.
[21] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.
[22] Ecma TC39. 2011. Standard ECMA-262 ECMAScript Language Specification Edition 5.1. (June 2011).
[23] Chucky Ellison and Grigore Roşu. 2012. An Executable Formal Semantics of C with Applications. In *POPL*. ACM, 533–544.
[24] Seyed K. Fayaz and Vyas Sekar. 2014. Testing Stateful and Dynamic Data Planes with FlowTest. In *HotSDN*.
[25] Seyed Kaveh Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd D Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation.. In *OSDI*. 217–232.
[26] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP configuration faults with static analysis. In *NSDI*. 43–56.
[27] Andy Fingerhut. 2017. Operations on header stacks in $P4_{14}$, $P4_{16}$, and bmv2. https://github.com/jafingerhut/p4-guide/blob/master/README-header-stacks.md. (2017).
[28] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *NSDI*.
[29] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast control plane analysis using an abstract representation. In *SIGCOMM*. 300–313.
[30] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roşu. 2016. RV-Match: Practical Semantics-Based Program Analysis. In *CAV (LNCS)*, Vol. 9779. Springer, 447–453.
[31] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *PLDI*. ACM, 336–345.
[32] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Dwight Guth, Philip Daian, and Grigore Roşu. 2017. *KEVM: A Complete Semantics of the Ethereum Virtual Machine*. Technical Report. University of Illinois at Urbana-Champaign.
[33] Alex Horn, Ali Kheradmand, and Mukul R Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *NSDI*. 735–749.
[34] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*. 99–111.
[35] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *NSDI*, Vol. 12. 113–126.
[36] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #398. https://github.com/p4lang/p4-spec/issues/398. (2017).
[37] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #411. https://github.com/p4lang/p4-spec/issues/411. (2017).
[38] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #412. https://github.com/p4lang/p4-spec/issues/412. (2017).
[39] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #414. https://github.com/p4lang/p4-spec/issues/414. (2017).
[40] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #429. https://github.com/p4lang/p4-spec/issues/429. (2017).
[41] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #430. https://github.com/p4lang/p4-spec/issues/430. (2017).
[42] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #431. https://github.com/p4lang/p4-spec/issues/431. (2017).
[43] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #433. https://github.com/p4lang/p4-spec/issues/433. (2017).
[44] Ali Kheradmand. 2017. P4 Lang. Spec. Repository Issue #442. https://github.com/p4lang/p4-spec/issues/442. (2017).
[45] Ali Kheradmand. 2017. P4K. https://github.com/kframework/p4-semantics. (2017).
[46] Ali Kheradmand. 2017. Suggested Revision for $P4_{14}$ Language Specification. https://github.com/kheradmand/p4-spec. (2017).
[47] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*.
[48] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*.
[49] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. Crystalnet: Faithfully emulating large production networks. In *SOSP*. 599–613.
[50] N Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. 2016. *Automatically verifying reachability and well-formedness in P4 Networks*. Technical Report. Microsoft Research.
[51] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *NSDI*.
[52] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.
[53] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *PLDI*. ACM, 346–356.
[54] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *TACAS*. 151–166.
[55] Santhosh Prabhu, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2017. Predicting Network Futures with Plankton. In *APNet*. ACM, 92–98.
[56] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. http://kframework.org/.

[57] The K Framework Development Team. 2017. KEQ. https://github.com/kframework/k/tree/keq. (2017).
[58] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. (2014).
[59] Hongkun Yang and Simon S. Lam. 2013. Real-Time Verification of Network Properties Using Atomic Predicates. In *ICNP*.