Automatically verifying reachability and well-formedness in P4 Networks

Nuno P. Lopes
Microsoft Research

Nikolaj Bjørner
Microsoft Research

Nick McKeown
Stanford University

Andrey Rybalchenko
Microsoft Research

Dan Talayco
Ketos Inc.

George Varghese
Microsoft Research

ABSTRACT

P4 allows a new level of dynamism for routers beyond OpenFlow 1.4 by allowing headers and tables to be modified by software in the field. Without care, P4 can unleash a new wave of software bugs. Existing tools (e.g., VeriFlow, NetPlumber, Hassel, NoD) cannot model changes to forwarding behaviors without reprogramming tool internals or having users *manually* add new forwarding models. Further, a P4 network can introduce a new class of bugs (not tested for by existing tools) wherein the P4 network creates *malformed* packets.

To attack these two problems, we provide an operational semantics for P4 constructs and use it to compile P4 to Datalog so that the verification model can be *automatically* updated as the network changes. We demonstrate this vision by compiling the mTag example in the P4 specification (and a new sTag security example) on a sample network and by automatically detecting forwarding bugs. Efficiently verifying (across all table entries and packet headers) that a P4 network only delivers *well-formed* packets takes a few seconds.

1. INTRODUCTION

While OpenFlow limits customization of routers in the field to forwarding table entries, P4 [9] is a language to program a router data plane to express completely new forwarding behaviors. P4 allows operators to specify [9] an action language, action primitives, and control flow for each packet.

A customer who wishes to efficiently enforce a new security policy can reprogram existing P4 routers to add and process a new security tag in packets (Section 4 has a detailed example) without being limited by existing ACLs, and without years of standardization or private arrangements with router vendors. Other customers can implement different headers (e.g., larger or hierarchical tags) and monitoring features. Innovation is enabled.

But there is a fly in the ointment – a wasp lurking in the rose – that is easy to ignore in the headlong rush towards flexibility. Since P4 allows router forwarding modifications at run-time, P4 can unleash a new wave of software bugs. While one may complain the inflexibility of existing router forwarding software, such software has at least been field-tested for many years. While SDN allows programming of table entries, P4 ups the ante by also allowing table schemas and forwarding behaviors to be changed.

Network Verification: In this paper, we seek to catch bugs in P4 networks using static checking as in Anteater [24], Header Space Analysis (HSA [21], VeriFlow [22], and NetPlumber [20]. However, *these tools cannot handle changes to forwarding behaviors without reprogramming tool internals.* For instance, each tool would need to be reprogrammed to support a new security tag. Network Optimized Datalog, NoD [23], can allow users to add new rules corresponding to reprogrammed routers but these rules must be created manually, an error-prone and time consuming process.

While it is possible to add new header formats for each new customer as with OpenFlow, this accretes complexity. Further, some customers may keep internal header formats private, which then requires customers to modify verification tools. One could rewrite tools like Hassel or VeriFlow with a new level of indirection through an abstract forwarding layer which can then be instantiated by a customer. However, the P4 specification already provides such an abstract forwarding layer. Rather than repeat this task, it seems prudent to compile from P4 to an existing tool language.

Our approach is to compile P4 specifications to Datalog instead of to VeriFlow or to Hassel. VeriFlow and Hassel are fairly low level and do not have the expressivity of Datalog (negations, classes, recursion) that are useful to verify more complex properties [23], and to compactly model the full range of forwarding behaviors expressible in P4. For example, suppose a P4 program chooses to test if two fields $F1$ and $F2$ are equal or satisfy some arithmetic relation such as $F1 < F2 + 10$. While this can be modeled as sets of wildcard expressions in Header Space Analysis [21], it is unclear how to *compactly* represent these sets for large header spaces.

Assume a set of properties (e.g., reachability, no black

holes) specified in Datalog as listed in [23]. Assume the customer starts with a standard IP network specified using P4. Our P4 compiler can compile this P4 description to a set of Datalog rules and can check for the specified properties. Next, assume the customer adds security tags (Section 4) by reprogramming their P4 routers. Our tool then compiles the new P4 specification to create a new set of Datalog rules. If the specification stays the same, (e.g., can A talk to B, where A and B are IP addresses that remain the same after the change), then the verification model can automatically be updated.

In addition to classical reachability bugs considered in past work (e.g., black holes, loops), a P4 network can introduce a whole new class of what we call *well-formedness bugs* because of the ability of a P4 router to add new headers and write fields.

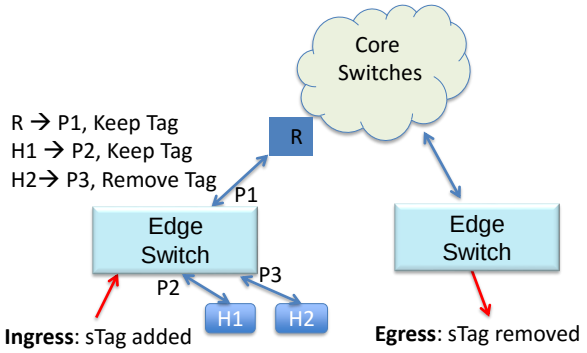

Figure 1: Example of a well-formedness bug created by not stripping a tag before leaving the network

Suppose that the routers in Figure 1 internally add a security tag (Section 4) that is carried internally and stripped before leaving the network. Suppose a forwarding table entry for host $H1$ at the leftmost edge router fails to specify (Figure 1) that the security tag is not stripped (which it should not for router R but should for host $H2$). Then host $H1$ will receive a malformed packet containing a security tag it does not understand. In the best case, this packet is dropped by the host; in the worst case it crashes host software.

Such a bug will not be caught by existing networking tools (e.g., [21–23]) as they check if a packet reaches a destination according to specification, *but do not check that if a packet p reaches, then p is well-formed.* Unlike reachability checks that require the operator to enter reachability specifications or beliefs [23], such well-formedness checks can be automated given a set of P4 programs and routers. We check whether any packets output by the P4 network at the egress edge can be parsed at the input edge. Such a bug may be hard to discover because it may only be caused by a single forwarding entry that applies to a small subset of the header space. The contributions of this paper are:

1. Automatic Verification (Section 7): We show how to automatically verify P4 networks from the P4 code at routers: as routers are reconfigured in the field, verification regression test suites stay unchanged gaining time (for manual update) and fidelity (no divergence between manual model and router dataplane).

2. Verifying Well-formedness (Section 5.4): Well-formedness is a new class of bug that differs from existing bug classes (such as reachability, loops, and slicing) that is not addressed by existing tools. While PIC [28] *tests* for protocol interoperability between peers, we do *verification* across all packet headers.

3. Verifying Compilation (Section 8.2): P4 also introduces the possibility of compiler errors. We show how to verify compilation using Datalog program equivalence.

4. Semantics for P4 (Section 5.2): Our operational semantics for P4 makes precise subtler aspects of P4.

The paper is organized as follows. We introduce P4 in Section 2, a load balancing Tag example in Section 3 and a security tag example in Section 4. We describe compilation of P4 to Datalog via an operational semantics in Section 5, and then describe our entire checking system in Section 6. We describe experiments in Section 7, extensions in Section 8, and related work in Section 9.

2. INTRODUCTION TO P4

P4 is a language for programming routers first described in [9]; the full specification is available at [3] and an open source implementation and simulator is on `github.com/p4lang`.

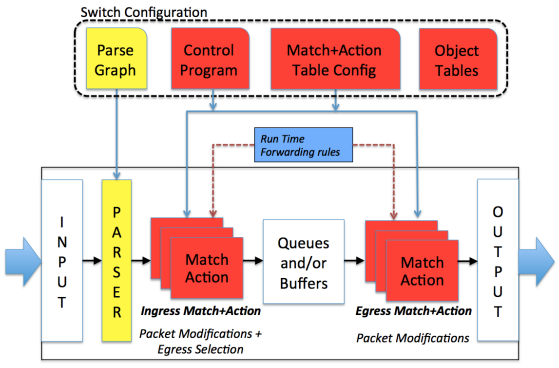

Figure 2: The P4 Forwarding Model.

P4 uses a simple abstract forwarding model[1] as shown in Figure 2. Processing is divided between ingress and egress with an opaque block for queues/buffers in between. A packet arrives on a port, is parsed and passed

[1]There has been a recent move to change this model to allow vendor-specific components which we discuss in the conclusion

to the ingress match+action tables. Ingress processing determines the set of egress packets to be sent to the egress pipeline where they may be processed further. A P4 program has the following components:

Header Types and Instances The layout of each packet header is given as a declaration. Metadata is represented in the same way as packet headers.

Parser The parser logic is expressed as a state machine which can examine each byte of an incoming packet.

Actions The actions that transform the packet and switch state are given as functions based on a set of primitive actions.

Tables Table declarations indicate the fields in packets that are matched and the possible actions executed when a table is *applied* to a packet. For example, a forwarding table might examine the L3 destination address of a packet and have "set-output-port" as an available action.

Control Flow Finally, the P4 program defines a function giving the order and conditions under which tables are be applied to a packet.

Table 1 explains P4 keywords used in this paper.

P4 Keyword	Usage
`metadata`	Declares a container for metadata by header type.
`standard_ metadata`	P4 defined metadata; ingress port, egress port, etc.
`extract`	In the parser, copies and formats packet data to a header instance.
`select`	Chooses the next parser state by selecting from a list of cases.
`action`	Define an action that can be executed by a table. It is a composition of primitive actions.
`modify_field`	A primitive action that changes a field in a header instance.
`add_header`	A primitive action which sets a specific header instance as valid.
`remove_header`	A primitive action which sets a specific header as invalid.
`drop`	A primitive action which marks the packet to be dropped. If done in the ingress pipeline, the packet will continue to the TM.
`table`	Start the declaration of a table.
`reads`	Lists the packet headers to be matched by a table.
`actions`	Lists the action functions that may be run by the table.
`control`	Defines a control function to give the order of application of tables to a packet during processing.
`apply`	Used in a control function to apply the named table to a packet. The action used by the table can affect further control flow.

Table 1: Selected P4 Keywords

3. THE mTag EXAMPLE

The `mtag` example from the original P4 paper [9] is a simple source-routing program that provides efficient load balancing (compared with ECMP or MPLS today) with simple hardware for core and POD routers.

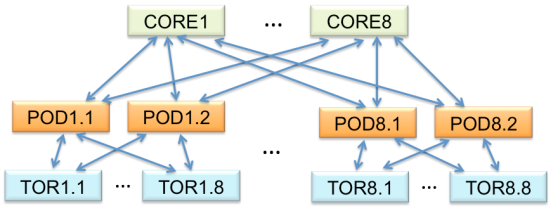

Figure 3: A Sample mTag Topology.

The topology of a network that might deploy `mtag` is shown in Figure 3 consists of top-of-rack (TOR) switches which insert the tag, a layer of aggregation switches called POD switches and a layer of core switches. In our examples, we use 8 racks in a POD, each POD has two POD switches and the core has 8 switches.

The `mtag` header includes four 1-byte routing fields and an Ethertype. The first byte describes the first POD (up-1) and the second byte (up-2) describes the incoming port on the core router; the third byte describes the outgoing port on the core router, and the last byte describes the outgoing port on the downstream POD router.

The routing fields are interpreted as port numbers by the POD and core switches. Thus the core and POD routers need no lookup tables. Note that the first TOR can easily implement load balancing by inserting different MTAGs for different flows. Details are in [9].

4. THE STAG EXAMPLE

We provide a simple, self-contained example of a P4 program that allows detailed analysis of our verification methods and showcases the ability of P4 to add useful new network functionality.

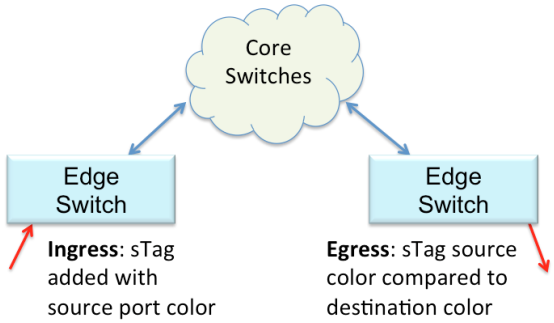

Figure 4: A Sample sTag Deployment.

Our `stag` example, as shown in Figure 4 is based on

an edge/core topology similar to `mtag`. We assume a forwarding database in the core whose details we ignore (in our experiments we ran `stag` on top of `mtag`).

We wish to enforce a flexible security model. Ideally, a security policy could be specified for every source/destination pair of hosts in the network, possibly even at the protocol (TCP Port) granularity.

Unfortunately, the policy requirements for such a model would grow with the square of the number of hosts in the entire network, requiring a prohibitive number of ACLs using today's technology.

In many deployments, are divided into groups and policy applied according to the groups to which the source and destination belong. We call these groups "colors" . We assume the security policy may be expressed in terms of the source and destination colors of the packet. This is conceptually similar to Cisco's TrustSec and Security Group Tagging [4,5]. Most cloud data center providers have similar abstractions. These are referred to as Security Groups by both OpenStack [2] and Amazon Web Services [1].

While Cisco has implemented source groups in hardware in some routers, it may be useful for other P4 routers to add a basic version of source group functionality. Note that while it is easy to add security tags *in software* at end hosts, many enterprises prefer to have the edge router add tags to guard against laptops entering the organization whose host software is not under the control of the enterprise.

For simplicity, we map ports to colors (though we could use hosts). An `stag` holds the color of the source port and is placed on the packet at the ingress of an edge switch. The tag is maintained but ignored as the packet traverses the core.

At the egress edge, a destination port is identified and the egress edge switch looks up its color. A security table then applies a policy based on examining the source color (from the `stag` field) and the destination color (from the local lookup).

If the packet is permitted to pass, the `stag` is stripped. The security table exposes an interface allowing run-time programming of specific policy to determine exactly which groups are allowed to communicate.

4.1 sTag in P4

We present the `stag` protocol as a self-contained P4 example. It will be used to illustrate our compiler and analysis.

Even a simple protocol such as this is challenging to deploy in practice. Our experience in defining `stag` examples quickly revealed how important such tools are to verifying configurations and obtaining confidence that the intent of the programmer was fulfilled.

Below we annotate much of the P4 code for the `stag` example.

Headers: Figure 5 shows the headers used by the `stag` protocol. Recall that the source color of the packet needs to be communicated from the ingress switch to the egress switch. This is done by the addition of an `stag` header which simply has one field for the source color.

The comparison of source and destination colors is done at the egress edge switch. A metadata header, `local_md` holds the source and destination colors simultaneously to decouple that process from packet tagging. Note that we use "md" for metadata throughout the example.

In order to differentiate local ports (which are assigned a color) and ports connected to the core, a `parser_value_set` called `host_ports` is used. This is just a set of numbers that can be managed at run time. It holds the port numbers of the local ports. This set can be examined by the parser to determine whether an `stag` should be expected on the packet.

Parser: The parser is shown in Figure 6. It always starts with an Ethernet header. It then checks whether the ingress port is a host (local) port by examining the value set `host_ports` as explained in the previous paragraph. If the port is **not** a host port, then an `stag` is parsed. Thereafter, parsing resumes as normal, in this case with the parsing of an IPv4 header.

The instruction `return ingress` indicates that parsing of the packet is complete and the ingress control flow should be executed.

Tables and Actions: The tables and actions for `stag` are shown in Figures 7 and 8 respectively. The first table, `resolve_source_color`, is used to determine the source port color by examining the ingress port. Recall that P4 only defines the format of the tables and a run-time interface will populate entries that specify exactly what color is associated to each port.

The `forward` table examines the IPv4 destination address and determines whether the destination is local or remote, calling the appropriate action (see below).

Finally, on an egress edge switch, a color check is performed by the `color_check` table. This has the option of dropping the packet (and implicitly may allow the packet to pass untouched).

The interesting actions are those called by the `forward` table. Packets transiting from the core to a local port will have `set_local_dest` called which will strip the existing `stag` from the packet. Packets from a local port that must be sent to the core will have `set_remote_dest` called which will add an `stag` to the packet. Recall that in this case the source color was set by the `resolve_source_color` table, so we just need to copy that value from metadata to the `stag` field.

Finally, to support the simple switching requirements of the core, the action `core_pass_through` is provided.

```
header_type ethernet_t {
    fields {
        dst_addr       : 48;
        src_addr       : 48;
        ethertype      : 16;
    }
}
header_type stag_t {
    // The stag holds the original source port color
    fields {
        source_color   : 8;
    }
}
header_type ipv4_t {
    fields {
        version        : 4;
        src_addr       : 32;
        dst_addr       : 32;
    }
}
header_type local_md_t {
    fields {
        src_port_color : 8; // Mapped at ingress or from stag
        dst_port_color : 8; // Mapped at egress
   }
}

// Header instances declared based on types
header ethernet_t ethernet;
header stag_t     stag;
header ipv4_t     ipv4;
metadata local_md_t local_md;
parser_value_set host_ports;
```

Figure 5: Headers

```
parser start { // Always start with an Ethernet header
    return ethernet;
}
parser ethernet {
    extract(ethernet);
    return select(std_md.ingress_port) {
        // Is ingress port a local host port?
        // If so, no stag is present
        host_ports : post_ethernet;

        // Otherwise, packet has an stag
        default : stag;
    }
}
parser stag {
    extract(stag);  // Get the stag from the packet
    // Save source port color in md
    set_metadata(local_md.src_port_color,
                 stag.source_color);
    return post_ethernet;
}
parser post_ethernet { // After optional stag
    return select(ethernet.ethertype) {
        0x800:      ipv4;
        default:    ingress;
    }
}
parser ipv4 {
    extract(ipv4);
    return ingress;
}
```

Figure 6: Parser

This leaves the stag untouched.

Control Flow: The control flow for the `stag` example is shown in Figure 9. If the `stag` header is not present, the packet arrived on a host port. In this case, the source color is found by applying the `get_source_color` table. In all cases, then, the `forward` table is applied to derive the egress port. At the same time, the switch can determine if the packet is destined for a local port. If so, it can apply the security policy by applying the `color_check` table to the packet.

5. COMPILING P4 TO DATALOG

Figure 10 shows an overview of our system that compiles a P4 specification of a set of routers and a topology and converts it into a system of Datalog rules that define the network, and to which the user can pose queries.

First, the user specifies parameters to a *Topology generator* that generates a network topology (currently a fat tree as in Figure 3 with parameters such as as number of TORs, core routers etc.). Each router has a unique `router.addr` and each interface has a unique `port.addr`. The topology is specified by using an `Egress` table that map each output interface at a router to the input interface and router that it is connected to via that output interface.

Second, the user specifies parameters to a *Table Generator* that generates a set of forwarding tables (not necessarily IP forwarding but all the tables specified by the P4 program) at each router called Table Data in Figure 10. The generator also fills in the table entries; in P4 terminology this generator both generates and populates the tables.

Next, the user supplies the P4 program at each router to a compiler that replaces each router by a set of Datalog rules. Finally, the user specifies queries using a *Query Generator* that translates human readable queries (i.e., can IP address A reach B) to queries for the Datalog engine.

When the programs, tables and topology information are converted into Datalog, they result in a predicate `reach` that describes the set of reachable configurations in a network as described in Section 5.1. To compile P4 programs into Datalog we rely on a big-step structural operational semantics (or Natural Semantics [19] introduced as a variant of small step semantics [29]) of P4, described next in Section 5.2. This is adequate at the level of P4, which supports parallelism, but not fine-grained concurrency as in NetCore [30]. Our compiler handles all aspects covered by Table 1 which is adequate for reachability queries. It omits handling current and proposed features of P4 for computing checksums, Quality of Service, monitoring and broadcasting. Section 5.3 shows that the SOS rules translate fairly directly to Datalog rules. Section 5.5 describes an in-

```
// Map port number to color
table resolve_source_color {
    reads {
        std_md.ingress_port : exact;
    }
    actions {
        set_source_color;
    }
}
// Map packet dest addr to exit port
// Add/remove stag if necessary
table forward {
    reads {
        ipv4.dst_addr : ternary;
    }
    actions {
        set_local_dest;    // Used by edge only, to local
        set_remote_dest;   // Used by edge only, to core
        core_pass_through; // Used by core only
    }
}
// Apply security policy based on src/dst port colors
table color_check {
    reads {
        local_md.dst_port_color : exact;
        local_md.src_port_color : exact;
    }
    actions {
        drop;
    }
}
```

Figure 7: Tables

```
action set_source_color(color) {
    modify_field(local_md.src_port_color, color);
}
// Set the egress port and remove stag, forwarding local
action set_local_dest(egr_port, color) {
    modify_field(std_md.egress_spec, egr_port);
    modify_field(local_md.dst_port_color, color);
    remove_header(stag);
}
// Set the egress port and add stag, forwarding to core
action set_remote_dest(egr_port) {
    modify_field(std_md.egress_spec, egr_port);
    add_header(stag);
    modify_field(stag.source_color,
                 local_md.src_port_color);
}
// The core (non-edge) switches do not need to modify the
// stag; just set the egress port
action core_pass_through(egr_port) {
    modify_field(standard_metadata.egress_spec, egr_port);
}
```

Figure 8: Actions

```
control ingress {
    if (not valid(stag)) {
        // Packet from local port, map its color
        apply(get_source_color);
    }
    // Where does the packet go? Local or core?
    apply(forward) {
        // Here, forward table used "local"...
        set_local_dest {
            // ... so apply the security policy
            apply(color_check);
        }
    }
}
```

Figure 9: Control Flow

lining optimization that substantially reduces memory overhead during analysis.

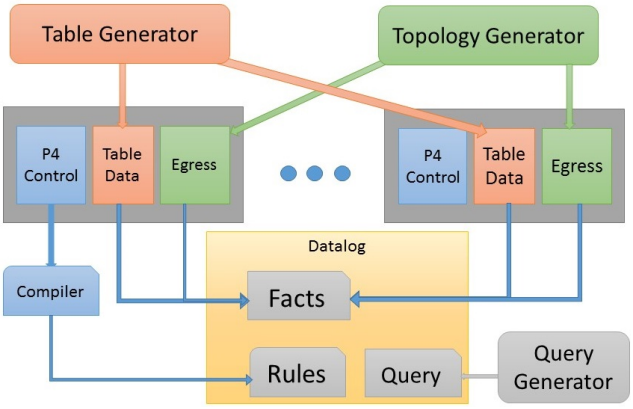

Figure 10: P4 analysis framework

5.1 Reachable States

The main ingredient for our Datalog queries is a predicate `reach` that captures the set of reachable states. Recall that a state is a combination of a packet header and an interface address, together with other meta-data.

Intuitively, to find out which packets can reach which interfaces, we have to pass packets through the router processing state machines of all the routers in the topology. Thus, to define `reach`, we need to define the state transformations performed by routers which we call `router_processing`.

The `router_processing` predicate has three main stages. The first stage ensures that meta-data fields are reset using the predicate `reset_local_data`. The second stage, `start`(S, S'), corresponds to the transition relation corresponding to the P4 program configured on the router, where S, S' are the states before and after the P4 program processes a packet. The last stage applies the `egress` table that models the links that connect routers. Using the `egress` mapping, the packet is sent to the next router `Next` and interface `Port`. In Datalog:

```
reach(S') :-
 reach(S),
 router_processing(S, S').

router_processing(S, S') :-
  reset_local_data(S, S0),
  start(S0, S1),
  egress(S1.local.addr, S1.std_md.egress_spec, Next, Port),
  S' = { S1 with std_md.ingress_port = Port, local.addr = Next }.

reset_local_data(S, S') :-
  S' = { S with local_md = 0, std_md = 0, parsed = 0 }.
```

We read a Datalog rule like

```
reach(S') :- reach(S), router_processing(S, S')
```

as follows: if the two predicates on the right of the `:-` hold, then the predicate on the left holds as well.

5.2 An Operational Semantics of P4

P4 looks simple, but there are a number of subtleties that need to be carefully specified that took us several weeks of discussion with the writers of the P4 spec. Examples include:

Control Flow: In the `stag` example in Figure 9 when we invoke *apply* on the `forward` table, the next action to be performed is specified by the matching entry in the IP4v4 forwarding table. In other words, control passes to the next action and not back to the caller, often referred to as "continuation semantics".

Parameter Passing: The `stag` program shows that the `color` parameter in Figure 8 comes sometimes from a passed parameter and sometimes from meta-data. At other times, it comes from table matches.

Update Semantics: Section 8.2.1 of the P4 specification [9] specifies that fields are updated in parallel within a table and serially between tables.

Drop Semantics: When a packet is dropped using the `drop` keyword, should the remainder of the P4 program be invoked (which can update counters and other state)?

For all these reasons, it is worthwhile to specify P4 semantics precisely. The primary payoff in our case is that once this is done, the translation to Datalog is fairly straightforward and we use the Datalog engine for calculating the meaning of a set of scenarios.

We choose to use a standard, "big-step" operational semantics for P4 statements. The semantics defines a relation, by induction on the syntax tree of P4 programs. The relation is of the form $S, \mathcal{E} \xrightarrow{stmt} S'$ where the pre-state S and the post-state S' summarize the states before and after execution of the program statement *stmt* in the environment $\mathcal{E}$ that provides values for local variables. The operational semantics rules define by induction predicates of the form $stmt(\mathcal{E}, S, S')$.

Rules, are as usual written as inferences where a horizontal line separates premises (above) from the conclusion (below). As we will later observe, each such definition corresponds to a Datalog rule.

The operational semantics manipulate a state variable S with 4 components:

1. Header fields of a packet processed by a router.

2. Internal meta-data, such as `standard_metadata` values and other auxiliary meta-data that is defined in the P4 program. This includes `local_metadata` used in our `stag` example.

3. Compiler generated meta-data corresponding to state that is maintained implicitly by the P4 runtime. For example, for each header the runtime maintains a *valid* field to indicate if the header is in the packet, and a field *parsed* to track if it is parsed.

4. An address field that is the unique identifier of the router or end-host receiving and processing the packet.

States are treated as records. Records are updated using the syntax $\{S \text{ with } \mathit{fld} = \mathit{val}\}$, meaning that the resulting record is the same as S, except the value for the field *fld* is set to *val*, regardless of the value of $S.\mathit{fld}$.

In addition, the semantics of a P4 statement depends on the environment variable $\mathcal{E}$. If one thinks of the packet headers and metadata as state, other variables such as passed parameters are part of the "environment".

The rules for most of the P4 statements either rewrite S by updating a field or they invoke a control state in the parser, table, or action component. Let us first describe selected semantic rules for actions. The effect of actions is a combination of field updates, adding and removing headers, and calling other actions recursively.

$$S, \mathcal{E} \xrightarrow{\texttt{remove\_header}(hdr)} \{S \text{ with } \mathit{valid.hdr} = \mathit{false}\}$$

$$S, \mathcal{E} \xrightarrow{\texttt{add\_header}(hdr)} \{S \text{ with } \mathit{valid.hdr} = \mathit{true}\}$$

The rule for `modify_field` is more complicated and is motivated by the two statements that modify the `color` field in Figure 8. The first sets the field using a passed parameter (from the environment) and the last statement sets `color` based on a field in the metadata. Which one should be used? To answer this question, the semantics assumes a function *eval* that evaluates the expression representing the modified field environment under the joint environment $\mathcal{E}S$. It behaves as a stack: to find the value of a field *fld*, first examine $\mathcal{E}$ if it defines a value, then examine S. This corresponds to standard programming intuition where one first checks passed parameters and then elsewhere (at the headers or metadata).

$$\frac{eval(\mathcal{E}S, exp) = val}{S, \mathcal{E} \xrightarrow{\texttt{modify\_field}(fld, exp)} \{S \text{ with } fld = val\}}$$

$$\frac{\begin{array}{c}\texttt{action}\ act(params)\{stmt\} \in Prog \\ eval(\mathcal{E}S, exps) = vals \\ S, [params \mapsto vals]\mathcal{E} \xrightarrow{stmt} S'\end{array}}{S, \mathcal{E} \xrightarrow{act(exps)} S'}$$

Actions that only modify fields are combined in parallel, other actions are combined sequentially (P4 specification [9], page 46) . We can capture both styles of semantics below.

Parallel composition is the more intricate semantics; we freeze the current state S and add a copy of S into the environment $\mathcal{E}$ to ensure the old state is used when evaluating the expressions passed in as the second arguments to `modify_field` and not the updated state.

$$\textsf{sequential ;}\ \frac{S, \mathcal{E} \xrightarrow{stmt_1} S_1 \qquad S_1, \mathcal{E} \xrightarrow{stmt_2} S'}{S, \mathcal{E} \xrightarrow{stmt_1; stmt_2} S'}$$

$$\textsf{parallel ;}\ \frac{S, \mathcal{E}S \xrightarrow{stmt_1} S_1 \ \ldots\ S_{k-1}, \mathcal{E}S \xrightarrow{stmt_k} S_k}{S, \mathcal{E} \xrightarrow{stmt_1; \ldots; stmt_k} S_k}$$

We now turn to probably the most interesting semantics, that of tables invoked from control blocks. For example, consider the `forward` table invocation in Figure 9. To understand its meaning, we must first examine the `table` declaration in Figure 7 which specifies that this table is indexed by the IP v4 header and may specify an action to `set_local_dest` and (in some cases) a set of arguments from the match.

However, when we invoke `forward` in Figure 9, it also checks whether the result of the match specifies `set_local_dest` (has the packet reached the remote edge?) in which case it applies the security check `apply(color_check)`.

In general, when the control program calls `apply`(*table*), it uses the table declaration to find the list of fields read and actions to be executed. It examines the fields mentioned in *reads* for a matching entry. These fields are mapped (via the table lookup) to values that determine which action to take, as well as any additional arguments passed to actions.

We use *vals* to refer to these additional values, and assume that can be computed by calling a function `add_entry_`*`table_action`*, which maps *reads* to arguments that get supplied to the named *action*.

As we have seen, the P4 program may specify that the continuation of a table action is another action. Indeed, Figure 9, specifies that if the `forward` table uses the `set_local_dest` action, then the tables associated with color checking are applied.

We encode this by passing the optional continuation statement to the rules that process actions and completing the action with the statement. Thus, the table rule applies the action *act* followed by the control statement *stmt*.

When reading the semantics, keep in mind the example of the `forward` table invocation in Figure 9 with `set_local_dest` as *act* and `apply(color_check)` as *stmt*. Starting at the top, we first read the table declaration to determine the fields read and the actions, then we find the extra arguments, if any, supplied by the match of the fields read. Then we apply the action *act* using the arguments supplied; then, and only then, do we execute the *stmt*.

The behavior of table application is a no-op when there are no keys that match any of the *reads* fields. P4 contains a custom continuation for no matches called miss whose semantics are very similar except that there are no arguments to gather and no *act* to execute.

$$\textsf{apply}\ \frac{\begin{array}{c}\texttt{table}\ table\ \{reads\ actions\} \in Prog \\ act \in actions \\ vals = add\_entry\_table\_act(S.reads) \\ S, \mathcal{E} \xrightarrow{act(vals)} S' \\ S', \mathcal{E} \xrightarrow{stmt} S''\end{array}}{S, \mathcal{E} \xrightarrow{\texttt{apply}(table)\{act\ \{stmt\}\}} S''}$$

For simplicity in this description we have omitted several details of the full semantics, such as programs where a table application can have multiple continuations, as well as the priorities that P4 specifies for entries. We also omit the mostly straight-forward rules for giving semantics for the parser phase.

5.3 From SOS to Datalog

It is straight-forward to extract Datalog definitions from operational semantics rules. For each program statement and declaration in a P4 program, the corresponding SOS rule from the previous section is directly converted to a Datalog definition of a predicate named by a unique identifier for the statement.

For example, the SOS rules for tables indicate how to compile tables, such as `get_source_color` to Datalog rules. We use the shorthand `gsc_scc` to identify the table entries associated with the action `set_source_color` in the table `get_source_color`.

There are two components to the transition relation: upon successful lookup in the `gsc_scc` table entries, apply the `set_source_color` action with the supplied color. On a miss, the table is a no-op, encoded using the last two rules.

```
get_source_color(S, S') :-
   add_entry_gsc_ssc(
     S.local.addr, S.std_md.ingress_port, Color),
```

```
   set_source_color(Color, S, S').

has_entry_gsc_scc(Addr, Port) :-
   add_entry_gsc_scc(Addr, Port, Color).

get_source_color(S, S) :-
   !has_entry_gsc_scc(
      S.local.addr, S.std_md.ingress_port),
   set_source_color(Color, S, S).
```

Not all P4 constructs are relevant when it comes to analyzing forwarding behavior. For example, P4 contains support for features such as metering and computing checksums of headers. We ignore these operations in our Datalog compiler.

5.4 Verifying well-formedness

We define well-formedness of a P4 program as the verification of protocol interoperability at the level of packet acceptance. That is, we verify whether the packets produced by an implementation of a protocol are always accepted by a different implementation and vice-versa. In addition to checking packet acceptance, we can also verify that the header fields sent in a packet always have the same value when decoded by the receiver's grammar. In summary, well-formedness ensures that the process of serializing and deserializing packets is consistent across multiple implementations say from different vendors.

Multiple tools have been developed in the past (e.g., PIC [28]) to test different implementations of a given protocol. We focus on verification, rather than testing.

Our approach is as follows: Given an edge in a deployment between two routers $R1$ and $R2$, let `p1(S,S')` and `p2(S,S')` be the two relations between input packet states `S` and output packet states `S'`. These are extracted from compiling the respective P4 programs at $R1$ and $R2$ and their configurations into Datalog relations and rules. We then create the relation `in2` and query as follows:

```
in2(S) :- p2(S,S').
?- p1(S,S'), !in2(S').
```

The query encodes the set of packet states `S` that that are accepted and transformed by `p1` into `S'` only to be rejected by `p2`.

Well-formedness errors arise because P4 allows user-defined packet formats; packet formats were thus far fixed by vendors. Moreover, the order in which the protocol headers are stacked is also user-defined.

Note that many bugs in this class can also be found using standard pairwise reachability, since if two implementations do not interoperate properly, then chances are some packets will be lost. However, as shown in Section 7.3, verifying well-formedness is much cheaper than checking all-pairs reachability. Second, a separate interoperability check makes it easier to localize the cause of the error. Third, this class of bugs does not require a user to write a specification, while to achieve full grammar coverage, the user would have to run multiple reachability verification tasks and write appropriate specifications. Fourth, as shown in Figure 1, it is possible to violate well-formedness without violating reachability.

5.5 Optimizations

Compiling P4 programs into *binary* relations is very convenient because we can define the semantics in a compositional way. For the runtime backend, however, this takes a large amount of memory (in the worst-case a cross-product of states when packets are rewritten) that slows down the execution.

Our compiler therefore contains an inlining phase that propagates the reachability relation over auxiliary binary relations. The result is a Datalog program that only uses *unary* auxiliary relations over states.

For example, `set_source_color`:

```
set_source_color(S, S', Color) :-
  S' = { S with local_md.src_port_color = Color }.
```

is inlined into the definition of `get_source_color`:

```
get_source_color_post(S') :-
  get_source_color_pre(S),
  add_entry_gsc_ssc(
     S.std_md.ingress_port, Color),
  S' = { S with local_md.src_port_color = Color }.
```

where `get_source_color_pre` summarizes the states entering the `get_source_color` table (found in a single place in the `ingress` control flow). The inlining transformation operates purely on Datalog and does not depend on P4. It is a special case of the fold/unfold transformation [11].

6. IMPLEMENTATION

We implemented the P4 to Datalog compiler in OCaml, in about 2,900 lines of code (excluding the parser for P4). We chose OCaml because its rich datatypes simplified the development of the compilers.

The topology generator (which generates topologies of the kind shown in Figure 3) has an additional 200 lines in OCaml. This module generates the egress tables (i.e., the wiring between router ports), as well as all necessary tables for each router (e.g., TOR local forwarding by the IP destination, mTag load balancing tables, etc).

The test generator module has about 100 lines, plus an additional 200 lines of tests for each program (mTag and sTag). Tests were written in a mini DSL we designed for this specific purpose. This language is based on first order logic, with built-in predicates for network-related properties.

The compiler works as follows. First, it converts the P4 program (as an AST) to a set of logic formulas (as described in Section 5). Then the compiler performs multiple transformations on the formulas to reduce their size and to make them more amenable for verification.

The set of transformations include SSA construction (to simplify subsequent transformations), equality propagation (to remove useless variables), constant propagation (to reduce size of formulas), rule inlining (to reduce number of rules and to enable further transformations), and a final rewrite into DNF (since our Datalog solver requires clauses to be in this form).

7. EVALUATION

We ran multiple experiments on both mTag and sTag programs with the sample topology shown in Figure 3. In summary, there were 8 core routers, 16 PODs, and 64 TORs, totaling 88 routers overall. Each TOR was connected to 62 hosts, each with 2 IP distinct addresses, totaling 3968 hosts and 7936 distinct IP address in the data center. In this topology, the number of hops between two hosts is always four, with the exception of hosts located in the same TOR, which are routed with just one hop.

For the experimental evaluation, we used Z3 4.5 (64 bits) coupled with its NoD (network optimized Datalog) engine [14,17,23], and OCaml 4.01. Tests were run on a computer with an Intel Xeon 2.53 GHz CPU and running Windows 10.

7.1 mTag

The compilation time for mTag was 1.8 seconds, including Datalog generation for the program, topology, tables, and test generation. The following running times given for each test only include the solving time (i.e., running time of Z3), since compilation time is amortized across all tests.

The first simple test we did was checking for black holes, i.e., checking that each host can reach all other hosts within the data center. This query took 3.2 seconds for a pair of hosts located in different TORs, with a peak memory usage of 63 MB. Repeating the experiment but with hosts within the same TOR reduced the running time to 1.7 seconds, and peak memory to 47 MB (reductions of 47% and 25%, respectively).

The second experiment checked for symmetric reachability, i.e., if host A can reach host B, then B should also be able reach A. This is a belief that should hold in networks [23]. This test, more complex than the first, took 3.7 seconds for a single pair of hosts, with a peak memory usage of 64 MB.

The third experiment introduced a bug in a TOR in the table that contains values for the mTag fields for an outgoing packet given its IP destination address. The down2 field was set in a way that the packet would loop between a core router and a POD. This bug broke symmetric reachability for one pair of hosts. Our tool found the problem in 1.7 seconds (half the time it takes with a successful query) and reported the path taken by the packet until the loop.

The bottom line is that queries are comparable in cost to that in Network Optimized Datalog [23]. Query times are slower than say Hassel [21] but provide more functionality: automated verification and well-formedness checks.

7.2 sTag

We ran two similar experiments for the sTag program. For these experiments, we took the mTag program and added the sTag header. Therefore, no changes were required in the aggregation layer just on the TORs in order to add/remove sTag headers and enforce color policy. The sTag program took 11.6 seconds to compile.

The first experiment was detecting black holes. For both reachable and unreachable hosts, the query took 31 seconds, with a peak memory usage of 317 MB. For symmetric reachability, the running time was 34 seconds, with similar memory usage.

For the second experiment, we checked for color consistency, i.e., if hosts A and B have the same color, then they should reach the same set of hosts. Checking this property for a pair of hosts that satisfy the property took 10 minutes, with a peak memory usage of 516 MB. We then introduced a bug in one of the hosts such that it would sometimes tag packets with the incorrect color and checked for consistency again. The solver took approximately the same time to find all the inconsistencies.

7.3 Well-formedness

We ran a third set of experiments to verify well-formedness as described in Section 5.4.

In the first experiment, we verified our sTag program against itself, and the solver took 2.2 seconds to verify the consistency. Then, we introduced a bug in the TOR's program to make it send packets to the POD with incorrectly ordered mTag and sTag headers in some cases. The solver took 0.7 seconds to find the bug.

An interesting outcome of this experiment was that we caught a problem in our code. A TOR always adds an mTag header when forwarding a packet to a POD. Hence, if a packet already had an mTag header, it would be forwarded with two such headers. This packet is then be dropped by the receiver TOR since our grammar only allows one mTag header per packet (only IPv4 is allowed after the mTag header).

The property verified in this experiment is comparable to one required to catch the bug depicted in Fig. 1. To catch that bug, for every edge from a router R to a host, we would check for interoperability between the edge router R and a dummy router that only contains the P4 program at an edge router needed to parse packets sent *from* a host. Verifying every edge beween internal routers (and between routers and hosts) at a cost of

a few seconds for each test seems like a cheap piece of insurance when a new protocol is added.

7.4 Experience

During the development of our tool, we inadvertently introduced multiple bugs including in the topology generator, the configuration tables generator, and the newly developed sTag program. Our limited experience shows that the flexibility of P4 makes it easy to introduce subtle bugs on the network that, e.g., create loops or disable or perturb otherwise valid paths.

For example, one of the bugs we introduced was the one described in mTag's experiment 3. The convention we were using was that the first 8 ports of PODs were connected to the core routers, and the remaining 8 were connected to TORs. However, the program that creates the tables for the TORs that, given a destination IP, returns the mTag up1/up2/down1/down2 fields, did not take into account the port numbering scheme correctly. The end result was that the down2 fields were all incorrect, and would make the PODs forward the packets back to the core routers, and thus creating a loop and breaking connectivity for all hosts. It is easy to imagine similar operational bugs caused by incorrect switch wiring.

A second bug we had was on the same TOR table. We had a convention that each TOR was assigned a /24 IP range. However, some of these IP addresses were not assigned to any host. The correct behavior for a TOR receiving a packet to a non-existing host was to drop the packet. In practice, TORs were not dropping these packets, but instead creating a loop (adding mTag fields, sending them across the network, and receiving them back again). If this bug happened in a real deployment, it could be used for a DoS attack against the data center operator. The cause of the problem was that the tables created for the TORs included mTag entries for the assigned IP range of the TOR. So, if the IP destination did not match any of the hosts connected to the TOR, it would fall back to adding an mTag header.

A third bug was in the topology generator. This program generates a table that states how router ports are connected to each other. The program had a bug that affected connectivity between TORs and PODs. The deployment equivalent would be a misconnection of cables between routers.

8. EXTENSIONS

Our compilation steps thus far allow us to check properties of P4 programs over a fixed number of packet headers, a fixed topology and a fixed configuration. Our methodology is not intrinsically limited to this setting. For example, it is possible to encode stateful [26, 27] middleboxes, as implemented using P4's `send_to_cpu` flag, by introducing additional state bits. It is also possible to reason about non-fixed, parametric topologies as in Vericon [8] using Datalog. We do not have space to describe these extensions; instead two extensions that have less in common with prior work.

8.1 Dynamic Headers

An MPLS network may stack multiple vlan headers on top of another, but packet processing need only consider a single vlan header. We can model the packet state by adding multiple versions of a header type. For example, we may have three vlan headers:

```
ethernet_t  ethernet;
vlan_t      vlan;
vlan_t      vlan2;
vlan_t      vlan3;
ipv4_t      ipv4;
```

The compiler to Datalog then produces three sets of rules depending on whether the current vlan header on the top of the stack refers to `vlan`, `vlan2` or `vlan3`.

8.2 Verifying P4 compilation

A more novel use of our tool is to verify P4 optimizations as equivalence between P4 programs. This is useful with the *microcosm* within routers, that consists of many pipeline stages [10] becoming as complicated as the *macrocosm* between routers. It can catch mistakes in tools for compiling P4 programs that break a large P4 table into seprate tables per hardware pipeline stage [18].

An appealing feature of P4 is that it allows specifying tables at different granularities of abstraction, so each hardware pipeline stage and its tables [18] can be modelled as separate P4 programs, while the complete router is specified by a single P4 program with say a forwarding table as in the IPv4 `forward` table in Figure 7.

As a simple example, assume that the P4 compiler decomposes the forwarding table into two tables: an exact match table in SRAM for 32-bit matches and an LPM table for other prefixes. Call these tables `forward_exact` and `forward_lpm`.

Recall that in the unoptimized P4 program (Figure 9) , after ingress control applies the original `forward` table, if it determines the destination packet to be for a local destination, it applies a security policy by first determining an exact color match; in case this match fails, it checks that source and destination colors are equal.

The decomposition of the `forward` table requires rewriting the ingress control using the new two tables. Whether performed manually or using a compiler, optimizations are a fertile ground for introducing bugs. Below we provide an optimized `ingress` control pipeline which places the color checks in an erroneous location. The correct location is indicated by a comment.

```
control ingress_opt {
 if (not valid(stag)) {
```

```
    apply(get_source_color);
  }
  apply(forward_exact) {
  miss {
    apply(forward_lpm) {
      // buggy location
      if (not valid(stag)) {
        apply(color_check);
  }}

    // * correct location *
}}}
```

We prototyped this check by representing the original P4 program and the bogus and corrected optimized P4 programs as logical formulas; we encoded the assumptions for the tables as side conditions. We then used Z3 to check equivalence. Z3 produced a counter-example for the erroneous location, and confirmed that the corrected version was equivalent. Note that our Datalog programs are over fixed finite domains so we do not need to solve an undecidable query containment problem [13].

9. RELATED WORK

Anteater [24], Hassel [21], VeriFlow [22], NetPlumber [20], Network Optimized Datalog [23] and VERMONT [6] do data plane verification do not support automatic verification from P4 though they could be used as backends. Dobrescu [15] breaks new ground by verifying the actual code of a Click router (and not a model) but only for 1-hop properties not path properties and only for software routers.

Data plane verifiers differ in their expressive power. Hassel acts as a library: new network formats and queries are encoded by programming on top of the library. Other systems handle incrementality and provide a language abstraction based on regular expressions [20], fixed-point logic [6], or Datalog [23]. Datalog, the most expressive, allowed modeling P4 programs in this paper.

Several tools do symbolic testing of SDN controller programs that are less relevant to our work, notable among them being NICE [12]. Flowlog [26] does use a Datalog variant for both synthesis and verification of controllers but would still require a compiler from P4 to Flowlog. Related to our checks for protocol interoperability [28] proposes *joint symbolic execution* using SMT solving. Engines based on symbolic exeuction check feasibility of one control path at a time; in contrast, for large header spaces, NoD allows to check interoperability exhaustively and provide all counter-examples.

NetKAT takes an algebraic/automata-theoretic approach proposing Kleene Algebras with tests as a foundation for modeling routers [7]. More recently, a reasoning tool was developed for NetKAT [16] and applied to the Stanford Campus Network. The tool relies on translating propagation over header spaces as a preprocessing step, which can be done in a single pass [31] for the case of forwarding tables that don't perform packet rewriting. Concurrent NetCore [30] is a language for packet programming. It is inspired by typed functional programming. It uses fine-grained operational, denotational and linguistic models to establish properties of the language and a compiler into the RMT model. Margrave [25] provides a modeling language based on Alloy and SAT solving for analyzing network configurations.

10. CONCLUSION

P4 lowers the barrier for network programming, but elevates the need for CAD tools for checking. Prior work requires ad-hoc modifications to deal with changing forwarding behaviors. By giving an operational semantics (useful in its own right), our compiler translates (almost) arbitrary P4 programs to Datalog. We showed how this technology can catch bugs when adding load balancing (MTag) or security (stag).

In creating synthetic bugs to evaluate our tool, we programmed a large number of actual bugs, strengthening our conviction that verification tools are essential for this mlieu. P4 also allows a new class of bugs by allowing the delivery of malformed packets. Our tool can verify syntactic interoperability between pairs of routers. In general, compilation to Datalog allows a simple framework to add new checks whether for interoperability or to verify compiler optimizations. While we currently verify a fixed set of populated tables in a fixed network, a Datalog as backend allows us to do *symbolic reasoning* across sets of networks and tables that satisfy some properties.

A recent movement proposes splitting a router P4 specification into vendor-independent and vendor-dependent portions, where vendors can add proprietary features such as monitoring. This paper's methodology can still be used if vendors supply an abstract P4 program for the complete router (which is needed anyway for behavioral simulation of P4 networks).

While regression testing is standard practice today, exhaustive and automatic verification of interoperability and reachability invariants will, we believe, become standard practice for future SDN networks.

11. REFERENCES

[1] AWS Security Groups. http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_SecurityGroups.html. Accessed: 2015-09-22.

[2] OpenStack Security Groups. http://docs.openstack.org/openstack-ops/content/security_groups.html. Accessed: 2015-09-22.

[3] The P4 Language Specification. http://www.p4.org. Accessed: 2015-09-22.

[4] Source Group Tag Exchange Protocol. https://datatracker.ietf.org/doc/draft-smith-kandula-sxp/. Accessed: 2015-09-22.

[5] TrustSec. http://www.cisco.com/c/dam/en/us/solutions/collateral/enterprise-networks/trustsec/trustsec_aag.pdf. Accessed: 2015-09-22.

[6] S. V. Altukhov, E. V. Chemeritskiy, V. V. Podymov, and V. A. Zakharov. VERMONT - a toolset for checking SDN packet forwarding policies on-line. In *MoNeTec*, 2014.

[7] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: semantic foundations for networks. In *POPL*, 2014.

[8] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: towards verifying controller programs in software-defined networks. In *PLDI*, page 31, 2014.

[9] 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, July 2014.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, 2013.

[11] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.

[12] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *NSDI*, 2012.

[13] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *CCC*, 1997.

[14] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[15] M. Dobrescu and K. J. Argyraki. Software dataplane verification. In *NSDI*, 2014.

[16] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for NetKAT. In *POPL*, 2015.

[17] K. Hoder, N. Bjørner, and L. De Moura. μZ: an efficient engine for fixed points with constraints. In *CAV*, 2011.

[18] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, 2015.

[19] G. Kahn. Natural semantics. In *STACS*, 1987.

[20] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[21] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[22] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *NSDI*, 2013.

[23] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.

[24] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[25] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *LISA*, 2010.

[26] T. Nelson, A. Ferguson, M. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2015.

[27] O. Padon, N. Immerman, A. Karbyshev, O. Lahav, M. Sagiv, and S. Shoham. Decentralizing SDN policies. In *POPL*, 2015.

[28] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, M. R, and T. Millstein. Analyzing protocol implementations for interoperability. In *NSDI*, 2015.

[29] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, 1981. Reprinted with corrections in J. Log. Algebr. Program. 60-61: 17-139 (2004).

[30] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: from policies to pipelines. In *ICFP*, 2014.

[31] H. Yang and S. Lam. Real-time verification of network properties using atomic predicates. In *ICNP*, 2013.