
# Computing Precise Control Interface Specifications

ERIC HAYDEN CAMPBELL, Cornell University, USA
HOSSEIN HOJJAT, Tehran Institute for Advanced Studies, Iran
NATE FOSTER, Cornell University, USA

Verifying network programs is challenging because of how they divide labor: the control plane computes high level routes through the network and compiles them to device configurations, while the data plane uses these configurations to realize the desired forwarding behavior. In practice, the correctness of the data plane often assumes that the configurations generated by the control plane will satisfy complex specifications. Consequently, validation tools such as program verifiers, runtime monitors, fuzzers, and test-case generators must be aware of these *control interface specifications* (ci-specs) to avoid raising false alarms.

In this paper, we propose the first algorithm for computing *precise* ci-specs for network data planes. Our specifications are designed to be *efficiently monitorable*—concretely, checking that a fixed configuration satisfies a ci-spec can be done in polynomial time. Our algorithm, based on modular program instrumentation, quantifier elimination, and a path-based analysis, is more expressive than prior work, and is applicable to practical network programs. We describe an implementation and show that ci-specs computed by our tool are useful for finding real bugs in real-world data plane programs.

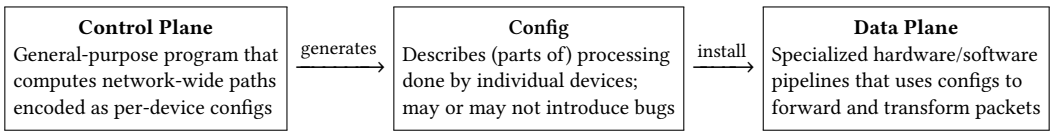
CCS Concepts: • **Theory of computation → Program specifications**; **Logic and verification**; • **Networks → Programming interfaces**.

Additional Key Words and Phrases: deductive synthesis, quantifier elimination, programmable networks

**ACM Reference Format:**
Eric Hayden Campbell, Hossein Hojjat, and Nate Foster. 2024. Computing Precise Control Interface Specifications. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 303 (October 2024), 30 pages. https://doi.org/10.1145/3689743

## 1 Introduction

Modern networks are increasingly programmable [8, 24, 26, 35]. Abstractly, network architectures can be modeled in terms of two cooperating programs: the *data plane* and the *control plane*. The *control plane* is a general-purpose program that computes forwarding paths through the network topology and generates configurations (configs) for data plane devices such as routers, switches, firewalls, etc. The *data plane* is a collection of restricted (e.g., loop-free and finite-state) programs that process packets efficiently, typically using a pipeline of configurable forwarding tables. This relationship is characterized in the schematic below:




Authors' Contact Information: Eric Hayden Campbell, Cornell University, Ithaca, USA, ehc86@cornell.edu; Hossein Hojjat, Tehran Institute for Advanced Studies, Tehran, Iran, h.hojjat@teias.institute; Nate Foster, Cornell University, Ithaca, USA, jnfoster@cs.cornell.edu.


ACM 2475-1421/2024/10-ART303
https://doi.org/10.1145/3689743

In an ideal world, data plane programs would be written to exhibit correct behavior under any possible config that might be generated by the control plane. However, due to pragmatic hardware-level concerns, programmers make simplifying assumptions about which configs its controller will generate. Unfortunately, existing data plane verification tools take an adversarial perspective, assuming that anything the control plane *can* do it *will* do. Consequently, these tools are subject to false alarms—i.e., configs that violate a given property but will never be generated [2, 32].

To address this problem, the research community has proposed several solutions. Vera uses a runtime monitor that inlines the config and re-verifies the configured data plane program every time the control plane generates a new config [40]. Intel's `p4v` tool and Google's `p4-constraints` library use first-order formulae to specify assumptions about the control plane-generated configs. These constraints are then used to rule out false alarms during verification [2, 32, 37], and to monitor the configs generated by the control plane [39]. However, re-verifying the data plane every time the config changes is expensive, and writing assumptions by hand is complicated and error-prone. How can programmers be certain the control plane will satisfy complex requirements on configs?

*Computing Interface Specifications.* A different approach is to compute a *precise* specification for the interface between the control plane and data plane. We call these descriptions *control interface specifications* (ci-specs). Rather than declaring that a data plane program is "verified" or "unverified", a ci-spec characterizes the conditions that configs must satisfy for the data plane program to satisfy its correctness properties. Hence, it shifts the onus for establishing correctness to the control plane—provided its configs satisfy the ci-spec, the data plane will behave as expected; conversely, if its configs violate the ci-spec, the data plane will be buggy. The ci-specs can be used to monitor the control plane—configs that violate the ci-spec can be logged for offline analysis or rejected outright.

*Precise and Efficient Control Interface Specifications.* In this paper, we propose Capisce, the first inference engine capable of computing *precise* and *efficiently control-monitorable* ci-specs. Informally, a *precise* ci-spec is both safe, meaning that satisfying configs trigger no bugs, and tight, meaning that violating configs have at least one packet that triggers a bug. Note that computing a *precise* ci-spec has a well-studied solution—we can compute the *weakest precondition* for the data plane and universally quantify over the variables that describe the packet state (Section 3, also vera [40], `p4v` [32]). However, checking that a config satisfies an arbitrary universally-quantified formula is expensive [30]. Instead, Capisce produces ci-specs that the control plane can monitor efficiently. We define a class of *efficiently control-monitorable sentences* (ECMS) and show that every ECMS has *polynomial* complexity. Importantly, Capisce infers *precise* ci-specs in ECMS.

To characterize the complexity of ci-spec inference, we show that it is equivalent to quantifier elimination (Qe) in the quantified theory of bitvectors (QBV). In one direction, we describe a compiler pipeline from a high-level model of pipeline programs called the guarded pipeline language (GPL) to the theory of bitvectors with uninterpreted functions (UFBV), and we show how to use Qe on specific variables to produce a precise ci-spec in ECMS. In the other direction, we show how to reduce Qe to the problem of computing ci-specs—i.e., we produce a simple GPL program whose ci-spec requires eliminating a universal quantifier.

*A Practical Implementation Based on Path-Based Heuristics.* The correspondence between ci-spec inference and Qe provides a daunting complexity challenge for the practical tractability of ci-spec inference. In particular, while Qe *can* be solved in a finite domain by enumerating the possible instantiations for the quantified variable, a strategy affectionately known as *bit-blasting*, this strategy isn't tractable for real-world data plane programs that manipulate thousands of bits.

For practical programs, however, it is often possible to side-step the worst-case complexity. We draw inspiration from two software engineering folk theorems: (1) "programs are usually correct"

and (2) "bugs have simple causes." We interpret (1) to mean that most program *paths* are correct, and the remaining paths are "buggy." Similarly, we interpret (2) to mean that among those relatively few buggy paths, it suffices to compute ci-specs for only a few of *those*.

Capisce leverages these path-based insights in its core algorithm CegQe: a counterexample-guided inductive inference (CeGIS) loop that uses counterexample paths to iteratively strengthen a candidate ci-spec until it is strong enough to prove the data plane program correct. The precision comes from ensuring that the strengthening step never "overshoots"—i.e., the candidate ci-spec $\psi$ never becomes strictly stronger than the weakest ci-spec.

We have implemented our approach in a tool called Capisce (Section 8), and used it to check a standard safety property on a collection of practical programs. Our experiments show that Capisce is able to handle real-world programs, and effectively finds bugs, while only exploring a tiny fraction of these programs' paths (e.g., for our repaired version of `fabric.p4`, only .00000000049%).

*Contributions.* Overall, this paper makes the following contributions:

- a formal model of data plane pipelines in our new language GPL, and a compiler from GPL to the quantifier-free theory of bitvectors and uninterpreted functions (QFUFBV);
- the class of *efficiently control-monitorable sentences* (ECMS) and a proof that inferring precise ci-specs in this class is *equivalent* to quantifier elimination (Qe) in the theory of bitvectors;
- an iterative-strengthening algorithm (CegQe) for computing precise ci-specs in ECMS that exploits software engineering insights;
- an implementation of Capisce in OCaml, leveraging Princess and Z3 as black-box Qe engines;
- an evaluation of Capisce on a benchmark suite of real-world data plane programs, which shows that Capisce can compute precise ci-specs for real-world P4 programs.

## 2 Background and Motivation

In a data plane program, the programmer declares a set of *match-action tables*, and then specifies a conditional *pipeline* that determines the order in which the tables are executed, or *applied*.

A table declaration comprises two components: a *key* and a set of *action*s. The *key* is a list of expressions $e_1, \ldots, e_n$ whose runtime values are used to determine which action is executed. An action is simply a function whose arguments are determined by the table itself—these arguments are called *action data*. As an example, consider the table below:

```
action nop () {}
action set_port (p) { port = p }
table fwd {
  key = { ipv4.dst }
  actions = { set_port; nop }
}
```

| ipv4.dst | **Action** |
|---|---|
| 192.0.2.47 | set_port(47) |
| 192.0.2.42 | set_port(42) |
| otherwise | set_port(DROP) |

The table `fwd`, defined in the pseudocode on the left, has a single expression as its key: the variable `ipv4.dst` that holds the IPv4 destination address. It also has two possible actions: `nop` and `set_port`. At runtime, the table's configuration (shown on the right above) will read the value of `ipv4.dst` and run either the `set_port` action, or `nop`.

As defined above, the `nop` action has no action data parameters and executes no operations, while the `set_port` action assigns its single action data parameter `p` to the `port` variable. Whenever a config indicates that the `set_port` action should be run, it must provide an argument, called *action data*, to the `set_port` function.

At runtime, a match action table is a kind of lookup table whose entries are configured by the control plane. To *apply* or *run* a table means evaluating its key expressions, finding the matching table entry, and executing the indicated action with the indicated action data. For example, the table below is to the control plane's config. This table has three entries, or *rows*. The first two execute

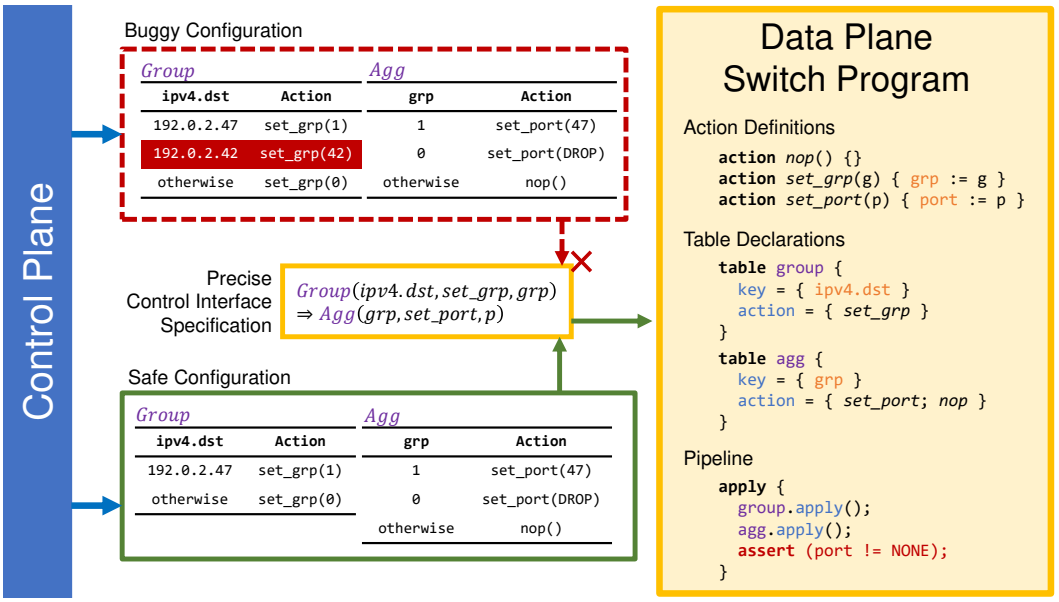



Fig. 1. An example data plane pipeline program (right) and with an asserted ci-spec (bottom right). Capisce computes a precise ci-spec (center), which ensures that the pipeline satisfies the spec. If the control plane (left) installs a bad config (top), it is rejected. Safe configs, like the one shown on the bottom, are accepted and can be safely installed into the pipeline program.

the `set_port` action with action data $n$ whenever the IPv4 destination address is `192.0.2.`$n$ for $n \in \{42, 47\}$. The final row executes `set_port` with action data `DROP` for every other packet.

These table configs are fundamental to determining the functionality of the switch. To see this, let's look at another example, shown in Figure 1. This pipeline exhibits a common pattern known as *link aggregation* [23, 42]. In this program, packet forwarding is divided into two tables: `group`, which computes a forwarding group ID for each packet; and `agg`, which maps each group ID to its forwarding port. In more detail, `group` looks up the IPv4 destination address (`ipv4.dst`) in the controller-provided config, which determines the action to be run. The `group` table only has a single allowed action, `set_group`, which assigns its action data to the `grp` field. For example, first row of the example config shown in Figure 1 for `group` assigns the `grp` field to `1` whenever `ipv4.dst` is `192.0.2.47`. Then, `agg` looks up the new `grp` in its config and either runs `nop`, which does nothing, or `set_port`, which assigns its action data `p` to the `port` field. Continuing the example, the first row of the config for `group` sets the port to `47` when `grp` is `1`. Running these configured tables in sequence has the effect of forwarding packets with `ipv4.dst` equal to `192.0.2.47` on port `47`.

The layer of indirection provided by `group` and `agg` is extremely valuable to network operators. Networks must react rapidly to hardware failures or changing service demands by forwarding packets on new routes. Unfortunately, modifying the contents of tables can incur high costs in hardware: due to the way that ternary content addressable memories (TCAMs) work, it can take minutes to process modifications that update thousands of entries [44]. The link-aggregation pattern avoids having to routinely execute minutes-long transactions by rerouting link aggregation groups. If many IP addresses map to the same link aggregation group and the adjacent link goes down, the control plane can reroute traffic for all of those IP addresses by updating a single rule.

The price for efficient reconfigurability is correctness—it is possible for the controller to introduce bugs in this program. Concretely, it can violate the so-called *determined forwarding* safety property, which asserts that every packet has a defined `port` value at the end of the pipeline. This is required because on certain hardware devices [27, 45], failing to assign a port value causes the packet to be forwarded on an *undefined port*. In building large systems of critical infrastructure (like networks), we want to avoid undefined behavior, so we classify such behavior "buggy." One config that produces undefined behavior is shown at the top of Figure 1. The `group` table maps address `192.0.2.42` to group 42, which triggers the catch-all rule in `agg` and executes `nop`. Hence, in this config, the forwarding behavior for packets with destination `192.0.2.42` is undefined.

### 2.1 Inference of Control Interface Specifications

Rather than rejecting programs for which the control configs *may* introduce buggy behavior, such as the one in Figure 1, we propose computing an interface specification $\psi$ that describes the set of configs that ensure the data plane program $p$ satisfies a given specification $\varphi$.

For instance, for the example in Figure 1, we want all configs for which the `group` table sets the group field to a value for which `agg` runs `set_port`. We call these restrictions *control interface specifications* (ci-specs). Mathematically, we can specify these specifications using first-order logic.

We can represent each table using a function symbol, *Group* for `group` and *Agg* for `agg`. Each function symbol has an argument for each key, and returns both an identifier that indicates which action will run, and the action's data. For notational elegance, when writing ci-specs, we notate these functions as relations, with the implicit understanding that they also adhere to the requisite functional dependencies and totality constraints. For instance, if we write $Agg(g, a, p)$, the variable $d$ corresponds to an input IPv4 address, then $a$ is the output action identifier (either `nop` or `set_port`), and $p$ is the output port value. Formally, a ci-spec for a pipeline program $p$ is a first-order logic formula over the functions induced by their tables.

Our goal is to compute *precise* ci-specs. A ci-spec $\psi$ is *safe* for a program $p$ and spec $\varphi$, if $p$ is guaranteed to satisfy $\varphi$ for all configs that satisfy $\psi$. Dually, a ci-spec $\psi$ is *tight* for $p$ and $\varphi$, if it is satisfied by every config for which $p$ satisfies $\varphi$. To define these notions formally, we stipulate some semantics function $[\![p]\!] : \text{Config} \to \text{Packet} \to \text{Packet}$ (see Section 3) that takes in a config $\sigma \in \text{Config}$ and produces a function on packets ($pkt \in \text{Packet}$).

*Definition 2.1 (Safe ci-spec).* Given a pipeline $p$ and specification $\varphi$, we say that a ci-spec $\psi$ is *safe* if for every config $\sigma$, we have: $\sigma \models \psi \Rightarrow \forall pkt.\ [\![p]\!]^{\sigma}\ pkt \models \varphi$

*Definition 2.2 (Tight ci-spec).* Given a pipeline $p$ and specification $\varphi$, we say that a ci-spec $\psi$ is *tight* if for every config $\sigma$, we have: $\left(\forall pkt.\ [\![p]\!]^{\sigma}\ pkt \models \varphi\right) \Rightarrow \sigma \models \psi$

Finally, we say that a ci-spec is *precise* if it is both safe and tight. For example, the ci-spec shown in the center of Figure 1 is precise. Note that a precise ci-spec has the property that for each config that does not satisfy it, there is at least one input that causes the data plane program to violate its spec. Hence, precise ci-specs can also be seen as the *weakest*—i.e., the most-permissive ci-spec.

The overall goal of this paper is to solve the following problem:

*Definition 2.3 (Problem Statement).* For a program $p$ and a spec $\varphi$, compute a precise ci-spec $\psi$.

In what follows, we will show how to produce precise ci-specs; but first, we describe previous work in this area, and elucidate why it doesn't suffice in our domain.

### 2.2 Previous Work

The general problem of synthesizing ci-specs has been studied both in and out of the networking community. The `bf4` tool uses program synthesis to infer single-table *necessary* ci-specs [14, 19],

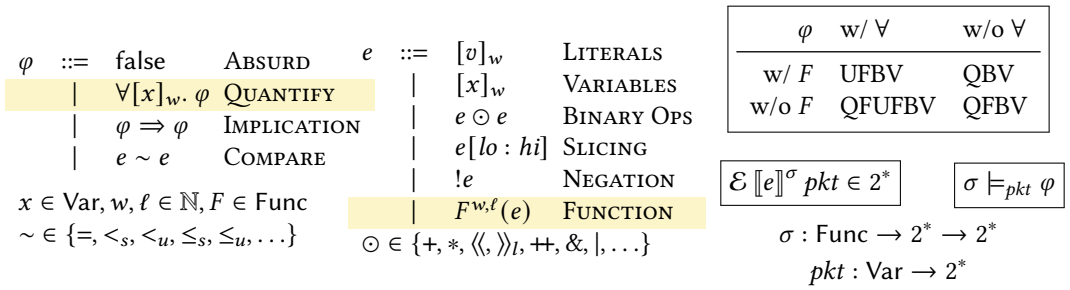



Fig. 2. Bitvector Theories. The syntax of UFBV formulae (left) and expressions (middle). The classification of bitvector theories (above right), depending on whether they allow quantifiers (∀) and/or uninterpreted functions ($F$). The semantics of bitvector expressions are standard. We stipulate the existence of expression semantics functions and boolean satisfaction relations (bottom right).

which prohibit no good runs. Formally, a ci-spec $\psi$ is *necessary* for a program $p$ and spec $\varphi$, if for every config $\sigma$ s.t. $\sigma \not\models \varphi$, every input packet causes $p$ to violate $\varphi$, that is $[\![p]\!]^\sigma\ pkt \not\models \varphi$. If `bf4` cannot infer a necessary constraint that is also sufficient, it reports the program as having *true bugs*. In Section 8.6, we compare our approach against `bf4` and find that we infer many more safe ci-specs. As an example, when provided with the example from Figure 1, `bf4` computes no ci-spec, because there is no *necessary* single-table ci-spec.

The problem of inferring interface specs (i-specs) has also been studied for general-purpose programs. The MaxSafeSpec algorithm synthesizes the weakest i-spec that is a conjunction of formulae over single function symbols [3]. In our context, this syntactic constraint is analogous to `bf4`'s single-table constraint. The difference here is that MaxSafeSpec computes *sufficient* (or *safe*) i-specs. However, the single-function restriction leads to false alarms when used with data plane programs. Returning to the example, MaxSafeSpec would compute $Agg(\texttt{grp}, a, \texttt{port}) \Rightarrow a = \texttt{set\_port}$, which would reject the sound config at the bottom of Figure 1.

So, using current approaches, a data plane engineer seeking to compute ci-specs would need to decide between a potentially-unsafe under-approximation, and an over-approximation, which can lead to false alarms. Capisce threads the needle by computing efficient and precise ci-specs, to provide a safety guarantee while minimizing false alarms.

## 3 Modeling

The remainder of this paper describes Capisce, which computes precise and efficient ci-specs. The first step is to obtain a symbolic model of the data plane. To do this, we describe a symbolic compilation pipeline from an abstract model of data planes (Section 3.2) to the theory of bitvectors and uninterpreted functions (Section 3.1). Our abstract pipeline language (Section 3.2) is called the *guarded pipeline language* (GPL), which lets us reason about branching pipelines of tables. We show we can model pipelines as programs in the assume-variant [25] of Dijkstra's guarded command language [16] by leveraging uninterpreted functions (Section 3.2). This modeling lets us employ fairly standard symbolic compilation techniques (Section 3.4) to develop a symbolic model. We use this symbolic model to compute precise and efficiently monitorable ci-specs.

### 3.1 Theories of Fixed-Width Bitvectors

Since data plane programs perform careful, bit-precise reasoning, we use the theory of bitvectors as our symbolic pipeline model, using the syntax shown in Figure 2. Semantically, let $2^*$ be the set

$$
\begin{array}{lcll}
p & \in & \mathrm{GPL}(\mathcal{T}) & \\
p & ::= & x := e & \textsc{Assignment} \\
 & | & \mathsf{asm}\,\varphi & \textsc{Assumption} \\
 & | & t(e) & \textsc{Table Application} \\
 & | & p; p & \textsc{Sequence} \\
 & | & p \;[]\; p & \textsc{Choice}
\end{array}
$$

$$x \in \mathsf{Var} \qquad t \in \mathsf{Table}$$
$$e \in \mathsf{Expr}(\mathcal{T}) \qquad \varphi \in \mathsf{Form}(\mathcal{T})$$

$$
\begin{array}{rcl}
[\![p]\!]^{\sigma} : \mathsf{Packet} \to \mathcal{P}(\mathsf{Packet}) & & \\
[\![x := e]\!]^{\sigma}\ pkt & \triangleq & \{pkt[x \mapsto \mathcal{E}\,[\![e]\!]^{\sigma}\ pkt]\} \\
[\![\mathsf{asm}\,\varphi]\!]^{\sigma}\ pkt & \triangleq & \{pkt \mid \sigma \models_{pkt} \varphi\} \\
[\![t(e)]\!]^{\sigma}\ pkt & \triangleq & [\![a_i(d)]\!]^{\sigma}\ pkt \\
\multicolumn{3}{r}{\textit{where } t : 2^n \to \{a_1, \ldots, a_i, \ldots, a_n\}} \\
\multicolumn{3}{r}{\textit{and } \langle i, d\rangle = \sigma(t)(\mathcal{E}\,[\![e]\!]^{\sigma}\ pkt)} \\
[\![p_1; p_2]\!]^{\sigma}\ pkt & \triangleq & \bigcup_{pkt' \in [\![p_1]\!]^{\sigma} pkt} [\![p_2]\!]^{\sigma}\ pkt \\
[\![p_1 \;[]\; p_2]\!]^{\sigma}\ pkt & \triangleq & [\![p_1]\!]^{\sigma}\ pkt \cup [\![p_2]\!]^{\sigma}\ pkt
\end{array}
$$

Fig. 3. Syntax (left) and semantics (right) of Guarded Pipeline Language GPL($\mathcal{T}$) over a bitvector theory $\mathcal{T}$. Highlighted variants only occur in GPL($\mathcal{T}$); the other variants are Guarded Command Language GCL($\mathcal{T}$).

of all bitvectors, and let $2^w$ be the set of bitvectors of width $w$, that is, bitvectors with precisely $w$ bits. We let $u, v \in 2^*$ range over bitvector literals. We use bracket notation, that is $[u]_w, [v]_w \in 2^w$ to precisely indicate a bitvector's width. Lower case variables $x, y, z \subseteq \mathsf{Var}$ range over first-order (bitvector) variables. In the theory of fixed-width bitvectors, each variable is equipped with a bitwidth, which we write $[x]_w$, indicating that values of $x$ must be drawn from $2^w$. The expression language defined in the theory of bitvectors could feasibly be any finite function on bitvectors, but typically we take a familiar set of core operations: addition (+), subtraction (−), multiplication ($*$), division (div), shifting ($\langle\langle, \rangle\rangle_a, \rangle\rangle_l$), concatenation (++), slicing ($\cdot[lo : hi]$), and bitwise operators (&, |, $\oplus$, …). We also permit all signed and unsigned comparison operators (=, $<_s$, $<_u$, $>_u$, $>_s$, …). The quantifier-free theory of bitvectors with these defined function symbols is called QFBV and the quantified variant is called QBV. When we allow expressions (which are sometimes called terms) to contain uninterpreted functions $t, F \in \mathsf{Func}$, our theory is UFBV or QFUFBV (with and without quantifiers respectively). Our function symbols also have types $2^w \to 2^\ell$ for $w, \ell \in \mathbb{N}$, in the grammar, we write this as $F^{w,\ell}$, but in practice, as with bitwidths elsewhere, we omit these annotations. Lower case greek symbols $\varphi$, $\psi$, $\chi$ range over bitvector formulae in UFBV.

Notice that our theories differs along two dimensions, the language of formulae ($\varphi$), and the language of expressions ($e$). To indicate that a formula $\varphi$ is syntactically valid in theory $\mathcal{T}$, we write $\varphi \in \mathsf{Form}(\mathcal{T})$. We also write $e \in \mathsf{Expr}(\mathcal{T})$, when $e$ is in $\mathcal{T}$'s language of expressions.

The semantics is largely standard, except for its use of configs $\sigma \in \mathsf{Config}$. The set of configs (Config) is the set of functions with type $\mathsf{Func} \to 2^* \to 2^*$. For convenience, we restrict (wlog[1]) the co-domain of $\sigma$ to be functions from $2^*$ to $2^*$. However, because each $F^{w,\ell}$ has type $2^w \to 2^\ell$, it must be that $\sigma(F) : 2^w \to 2^\ell$. Intuitively, for a function symbol $F \in \mathsf{Func}$, we have that $\sigma(F)$ is a function definition for $F$. In this sense, configs $\sigma$ can be viewed as finite sets modeling first-order logic formulae. In addition to configs, we need to define the runtime packet $pkt \in \mathsf{Packet}$. A packet is a valuation function $pkt : \mathsf{Var} \to 2^*$ on variables. We stipulate a standard evaluation function for expressions $\mathcal{E}\,[\![e]\!]^{\sigma}\ pkt = v$ and a satisfaction relation for formulae $\sigma \models_{pkt} \varphi$.

### 3.2 Syntax and Semantics of the Guarded Pipeline Language (GPL)

This section presents our modeling language for tables, GPL($\mathcal{T}$). The language is parametric over the bitvector theory used in expressions and assumptions. By default, we will assume $\mathcal{T}$ is QFBV (i.e., no quantifiers or uninterpreted functions), and will write GPL to denote GPL(QFBV).

The syntax and semantics of GPL($\mathcal{T}$) are presented in Figure 3. A GPL($\mathcal{T}$) program is mostly standard comprising: assignment $[x]_w := e$ which assigns $e \in \mathsf{Expr}(\mathcal{T})$ to the $w$-bit variable $x$;

[1] $F(x, y) = \langle [p]_m, [q]_n \rangle$ can be seen as syntactic sugar for $F(x{+}{+}y)[0 : m] = [p]_m \wedge F(x{+}{+}y)[m{+}1 : m{+}n{+}1] = [q]_w$

assumption ($\mathsf{asm}\ \varphi$) which assumes the truth of $\varphi \in \mathsf{Form}(\mathcal{T})$; sequential composition ($c; c$); and finally nondeterministic choice ($c \ [] \ c$).

The main non-standard constructs found in GPL($\mathcal{T}$) are table declarations and table applications. A table declaration $T = \langle t, n, \mathbf{a} \rangle$ is a tuple comprising a table name variable $t \in \mathsf{Table} \subseteq \mathsf{Func}$, a natural bitwidth indicating the size of its key domain, $n \in \mathbb{N}$, and a set of possible actions $\mathbf{a} \subseteq \mathsf{Action}$. An action $a = \lambda d : w.\ p$ is a function parameterized on a variable $d$ of bitwidth $w$ and runs a straight-line program $p$ that may read $d$ (a *straight-line program* never uses nondeterministic choice). For an argument $[v]_w$, we write $a(v)$ to mean the substitution $p[d \mapsto v]$. When $w = 0$ we use the syntactic sugar $\lambda().\ p$. We use $2^n$ to refer to the set of bitvectors of width $n$. To evoke tables' functionality, we stylize their declarations as follows: $t : 2^n \to \mathbf{a}$.

For instance, below we declare the *Agg* and *Group* tables from Figure 1, first defining the actions, and then declaring the tables. The key width for the *Group* table is 32 as it reads the IPv4 destination address, a 32-bit field. The *Agg* table reads the grp field, which is also 32 bits.

$$\mathsf{set\_group}\ g \triangleq \mathsf{grp} := g \qquad \mathsf{set\_port}\ p \triangleq \mathsf{port} := p \qquad \mathsf{nop}() \triangleq \mathsf{asm\ true}$$

$$\mathit{Group} : 2^{32} \to \{\mathsf{set\_group}\} \qquad \mathit{Agg} : 2^{32} \to \{\mathsf{set\_port}, \mathsf{nop}\}$$

A table application is written $t(e)$ for some declared table $t : 2^n \to \mathbf{a}$ and expression $e \in$ QFUFBV of width $n$. This variant is highlighted in Figure 3. To indicate that a program $p$ may reference a set of declarations $\mathbf{T}$, we write the stylized pair $p[\mathbf{T}]$. With the above definitions, the link aggregation example from Figure 1 is written as follows:

$$\mathit{Group}(\mathsf{ipv4.dst}); \mathit{Agg}(\mathsf{grp})$$

Semantically, a GPL($\mathcal{T}$) program $p$ takes in a config $\sigma \in \mathsf{Config}$ and returns a function from packets (Packet) to sets of packets ($\mathcal{P}(\mathsf{Packet})$). Formally, we have a function $[\![p]\!]^\sigma : \mathsf{Packet} \to \mathcal{P}(\mathsf{Packet})$, whose semantics are provided in Figure 3. Assignment $x := e$ uses the $pkt$ and $\sigma$ to evaluate $e$ to a bitvector $v$, returning a singleton set containing the packet $pkt[x \mapsto v]$. The notation $pkt[x \mapsto v]$ indicates the packet that is identical to $pkt$ except on variable $x$, which is mapped to $v$. Next, assumptions ($\mathsf{asm}\ \varphi$) evaluate whether $pkt$ satisfies $\varphi$ in $\sigma$: if so, it returns the singleton packet set $\{pkt\}$, otherwise it returns the empty set $\emptyset$. Sequential composition ($p_1; p_2$) is the composition of the denotation of $c_1$ composed with the denotation of $p_2$ lifted to sets in the natural way. Similarly, the semantics of nondeterministic choice ($p_1 \ [] \ p_2$) is the union of the denotations of the disjuncts. Again, GPL($\mathcal{T}$)'s most novel construct is table application, $t(x)$, which, semantically, looks up $t$ in the config $\sigma$. Then, $\sigma(t)$ returns a pair $\langle i, d \rangle$ of an action identifier $i$ and action data $d$. The semantics then select the $i$th action $a_i$, and run it with its argument.

From this model, we can also define syntactic sugar for trivial and conditional statements. The trivial statement skip does nothing. Conditionals are encoded in the standard way using a combination of assumes and nondeterministic choice:

$$\mathsf{skip} \triangleq \mathsf{asm\ true} \qquad \mathsf{if}(b)\{c_t\}\{c_f\} \triangleq \mathsf{asm}\ b; c_t \ [] \ \mathsf{asm}\ \neg b; c_f$$

For a formula $\varphi \in \mathsf{Form}(\mathcal{T})$, construing $[\![p]\!]^\sigma$ to be a relation lets us write $[\![p]\!]^\sigma \models \varphi$ to indicate that $p$ satisfies $\varphi$ under config $\sigma$. We find it more evocative to write this as $p[\sigma] \models \varphi$. We also define $p \models \varphi$ to be $\forall \sigma.\ p[\sigma] \models \varphi$.

*An aside on types.* GPL requires a type system to keep track of bitwidths and ensure they are used consistently throughout a program. However, we will elide this detail as it is standard and unsurprising. We will also omit bitwidths in examples when they are obvious or irrelevant.

### 3.3 Modeling Tables as Uninterpreted Functions

In this section, we show how we can model GPL($\mathcal{T}$)'s tables using uninterpreted functions. We do so by defining a restriction of GPL($\mathcal{T}$) that corresponds to Dijkstra's guarded command language (GCL($\mathcal{T}$)), and then defining a mapping from GPL($\mathcal{T}$) to GCL($\mathcal{T}$), and proving equivalence.

Formally, GCL($\mathcal{T}$) is the subset of GPL($\mathcal{T}$) that excludes table application. Just as we've been letting $p$ range over GPL($\mathcal{T}$) programs, let $c$ range over GCL($\mathcal{T}$) programs. For GCL($\mathcal{T}$), the default theory is QFUFBV, so we take the convention that GCL indicates GCL(QFUFBV).

We can define $\mathsf{model} : \text{GPL} \to \text{GCL}$ for an application of table $t : 2^w \to \mathbf{a}$. Intuitively, $\mathsf{model}(t(x))$ treats $t$ as an uninterpreted function, and applies it to the key $x$ to produce an action and argument and then runs that action with its argument.

$$\mathsf{model}(t(x)) \triangleq \langle i, d\rangle := t(x); \mathsf{run}_{\mathbf{a}}(i, d)$$

where $\mathsf{run}_{\mathbf{a}}(i, d)$ selects the $i$th action from the set $\mathbf{a} = \{a_0, \ldots, a_n\}$ and runs it with argument $d$:

$$\mathsf{run}_{a_0,\ldots,a_n}(i, d) \triangleq \mathsf{asm}\ i = 0; a_0(d)\ [] \cdots [] \ \mathsf{asm}\ i = n; a_n(d)$$

As an example, consider the pipeline from Figure 1. We recapitulate its definition in GPL below and show its translation into GCL:

*Action Definitions*
set_group $\triangleq \lambda g.$ grp $:= g$
set_port $\triangleq \lambda p.$ port $:= p$
nop $\triangleq \lambda().$ asm true

*Table Definitions*
*Group* : $2^{32} \to$ {set_group}
*Agg* : $2^{32} \to$ {set_port, nop}

*GPL pipeline*
*Group*(ipv4.dst);
*Agg*(grp)

$\overset{\text{model}}{\mapsto}$

*GCL Model*
$\langle a, g\rangle :=$ *Group*(ipv4.dst);
grp $:= g$;
$\langle b, p\rangle :=$ *Agg*(grp);
if($b$ = set_port){
  port $:= p$
}{// *else* $b$ = nop
  skip
}

Observe that both tables *Group* and *Agg* have been replaced by function calls that compute output variables $a$ and $d$. After *Group* is called, we can ignore $a$ since *Group* only has one action, and simply assign $d$ to grp. Then we run the *Agg* function to compute $b$ and $p$. We then inspect $b$ to determine which action should be run. If $b$ indicates the set_port action, then port is assigned the action data value $p$, otherwise, $b$ is nop and nothing happens.

We prove that this translation is semantics-preserving.

Theorem 3.1 (Adequacy). $[\![p]\!]^\sigma = [\![\mathsf{model}(p)]\!]^\sigma$

Proof. By induction on $p$. Let $p = t(x)$, as the remaining cases are immediate or by IHs. Let $\mathbf{a} = \{a_0, \ldots, a_n\}$, and $\langle j, d\rangle = \sigma(t)$.

$$\begin{aligned}
[\![\mathsf{model}(t(x))]\!]^\sigma &= [\![\langle i, d\rangle := t(x); \mathsf{run}_{\mathbf{a}}(i, d)]\!]^\sigma \\
&= [\![\mathsf{run}_{\mathbf{a}}(j, d)]\!]^\sigma \\
&= [\![\mathsf{asm}\ i = 0; a_0(d)\ [] \cdots [] \ \mathsf{asm}\ i = n; a_n(d)]\!]^\sigma \\
&= [\![a_j(d)]\!]^\sigma \\
&= [\![t(x)]\!]^\sigma
\end{aligned}$$

□

Our model is the first to precisely characterize the semantics of tables in a logical formalism [32, 40, 43]. We will use it to generate precise symbolic representations of GPL programs.

### 3.4 Symbolic Compilation

By Theorem 3.1, to generate a symbolic model of $p \in \text{GPL}$, we need only compile its GCL model $c = \text{model}(p)$. We rely heavily on previous work [16, 25] to produce our symbolic compiler. Our first step is to normalize programs into the passive form [25]. A program is *passive* if it does not have any assignments. We can *passify* a program $c$ by replacing assignments with assumes. Doing so requires minting a new variable index each time a variable is written, and doing some careful bookkeeping to ensure that indices are synchronized across join points. The function $\text{passify} : \text{GCL}(\mathcal{T}) \times \mathbb{N}^{\text{Var}} \to \text{GCL}(\mathcal{T}) \times \mathbb{N}^{\text{Var}}$, takes in two arguments, a GCL program $c$ and a map $\mathcal{I}$ from variables to indices. It returns a passive $c'$ and a map $\mathcal{I}'$ holding the maximum index for each variable. We define passify below:

$$
\begin{array}{lcl}
\text{passify}(x := e, \mathcal{I}) & \triangleq & \textbf{let } \mathcal{J} = \mathcal{I}[x \mapsto \mathcal{I}(x) + 1] \textbf{ in} \\
& & (\text{asm } x_{\mathcal{J}(x)} = \text{subst}(\mathcal{I}, e), \mathcal{J}) \\
\text{passify}(\text{asm } \varphi, \mathcal{I}) & \triangleq & (\text{asm } \mathcal{I}(\varphi), \mathcal{I}) \\
\text{passify}(c_1; c_2, \mathcal{I}) & \triangleq & \textbf{let } c_1', \mathcal{I}_1 = \text{passify}(c_1, \mathcal{I}) \textbf{ in} \\
& & \textbf{let } c_2', \mathcal{I}_2 = \text{passify}(c_2, \mathcal{I}_1) \textbf{ in} \\
& & (c_1'; c_2', \mathcal{I}_2) \\
\text{passify}(c_1 \,[]\, c_2, \mathcal{I}) & \triangleq & \textbf{let } c_1', \mathcal{I}_1 = \text{passify}(c_1, \mathcal{I}) \textbf{ in} \\
& & \textbf{let } c_2', \mathcal{I}_2 = \text{passify}(c_2, \mathcal{I}) \textbf{ in} \\
& & \textbf{let } r_1, r_2, \mathcal{J} = \text{merge}(\mathcal{I}_1, \mathcal{I}_2) \textbf{ in} \\
& & (c_1'; r_1 \,[]\, c_2'; r_2, \mathcal{J})
\end{array}
$$

where $\mathcal{I} : \text{Var} \to \mathbb{N}$ is a map from variables to natural indices. We define $\mathcal{Z}$ to be the map that indexes each variable with 0. We always initialize passify with $\mathcal{Z}$. In the above function, each time we see an assignment $x := e$, we rename $e$ according to the current set of indices using a substitution function $\text{subst}(e, \mathcal{I})$, which returns an expression $e$ whose variables have been indexed according to $\mathcal{I}$. We then increment the index for $x$. Translating assumptions ($\text{asm } \varphi$) is similar, we annotate all the variables in $\varphi$ with their current indices, written $\text{subst}(\varphi, \mathcal{I})$. The sequence case is homomorphic: after passifying $c_1$ we passify $c_2$ with the updated indices from $c_1$.

The hard case is passifying choice ($c_1 \,[]\, c_2$), where we add so-called *residuals* $r_1$ and $r_2$ to each passified program disjunct ($c_1'$ and $c_2'$ above). These residuals are computed by $\text{merge} : \mathbb{N}^{\text{Var}} \times \mathbb{N}^{\text{Var}} \to \text{GCL} \times \text{GCL} \times \mathbb{N}^{\text{Var}}$ which takes in the indexing functions $\mathcal{I}_1$ and $\mathcal{I}_2$ that result from passifying $c_1$ and $c_2$ and returns so-called *residuals* $r_1$ and $r_2$. The residuals synchronize the indices between $c_1'$ and $c_2'$. The residual $r_1$ finds the variables that have a lower maximum index in $c_1'$ than they do in $c_2'$ and assumes a chain of equalities $x_i = x_{i+1}$ that "catch up" to the max indices of $c_2'$. The residual $r_2$ is symmetric. We define merge formally below

$$
\begin{array}{lcl}
\text{merge}(\mathcal{I}_1, \mathcal{I}_2) & \triangleq & \textbf{let } r_1 = \text{asm } (\bigwedge \{x_i = x_{i+1} \mid \mathcal{I}_1(x) \le i < \mathcal{I}_2(x), x \in \text{Var}\}) \textbf{ in} \\
& & \textbf{let } r_2 = \text{asm } (\bigwedge \{x_i = x_{i+1} \mid \mathcal{I}_2(x) \le i < \mathcal{I}_1(x), x \in \text{Var}\}) \textbf{ in} \\
& & \textbf{let } \mathcal{J} = \{x \mapsto \max\{I_1(x), I_2(x)\} \mid x \in \text{Var}\} \textbf{ in} \\
& & (r_1, r_2, \mathcal{J})
\end{array}
$$

Note that the size of the added residuals is quadratic in the size of the input program [25]. Of course the translation is semantics-preserving, after some bookkeeping to relate the lowest and highest indices with the inputs and outputs of the original program [25].

To understand passify by example, let's return to Figure 1, for which we compute the following:

*GCL Model*

$\langle a, g\rangle := \mathit{Group}(\mathsf{ipv4.dst})$;
$\mathsf{grp} := g$;
$\langle b, p\rangle := \mathit{Agg}(\mathsf{grp})$;
$\mathsf{if}(b = \mathsf{set\_port})\{$
  $\mathsf{port} := p$
$\}\{$// *else* $b$ *is* nop
  skip
$\}$

$\xmapsto{\pi_1 \circ \mathsf{passify}(-, \mathcal{Z})}$

*Passive Form GCL Model*

$\mathsf{asm}\ \langle a_1, g_1\rangle = \mathit{Group}(\mathsf{ipv4.dst}_0)$
$\mathsf{asm}\ \mathsf{grp}_1 = g_1$;
$\mathsf{asm}\ \langle b_1, p_1\rangle = \mathit{Agg}(\mathsf{grp}_1)$
$\mathsf{if}(b_1 = \mathsf{set\_port})\{$
  $\mathsf{asm}\ \mathsf{port}_1 = p_1$
$\}\{$// *else* $b_1$ *is* nop
  $\mathsf{asm}\ \mathsf{port}_1 = \mathsf{port}_0$
$\}$

Notice the residual that was added to the nop branch of the choice operator. Because the set_port branch in the original program (above left) updated port to $d$, the passive equivalent incremented port's index to 1. Now, to synchronize the indices across branches, and capture that the value remained unchanged, passify adds the residual $\mathsf{asm}\ \mathsf{port}_1 = \mathsf{port}_0$ to the nop branch.

Assuming that a program is in passive form, we can generate a linear-size symbolic representation[2]. The following symbolic compilation function $N : \mathrm{GCL}(\mathcal{T}) \to \mathcal{T}$, precisely captures the executions of a passive program $c$:

$$\begin{array}{rcl} N(\mathsf{asm}\ \varphi) & \triangleq & \varphi \\ N(p_1; p_2) & \triangleq & N(p_1) \wedge N(p_2) \\ N(p_1 \,[]\, p_2) & \triangleq & N(p_1) \vee N(p_2) \end{array}$$

The following shows the result of running $N$ on the passified example program:

*Passive Form GCL Model*

$\mathsf{asm}\ \langle a_1, g_1\rangle = \mathit{Group}(\mathsf{ipv4.dst}_0)$
$\mathsf{asm}\ \mathsf{grp}_1 = g_1$;
$\mathsf{asm}\ \langle b_1, p_1\rangle = \mathit{Agg}(\mathsf{grp}_1)$
$\mathsf{if}(b_1 = \mathsf{set\_port})\{$
  $\mathsf{asm}\ \mathsf{port}_1 = p_1$
$\}\{$// *else* $b_1$ *is* nop
  $\mathsf{asm}\ \mathsf{port}_1 = \mathsf{port}_0$
$\}$

$\xmapsto{N}$

*Symbolic Model*

$\langle a_1, g_1\rangle = \mathit{Group}(\mathsf{ipv4.dst}_0) \wedge$
$\mathsf{grp}_1 = g_1 \wedge$
$\langle b_1, p_1\rangle = \mathit{Agg}(\mathsf{grp}_1) \wedge$
$(b_1 = \mathsf{set\_port} \wedge \mathsf{port}_1 = p_1$
$\vee\ b_1 = \mathsf{nop} \wedge \mathsf{port}_1 = \mathsf{port}_0)$

With a symbolic pipeline in hand, we can check whether it satisfies a spec $\varphi$ via implication. However, we must be sure to update $\varphi$ with respect to passify's output index mapping $\mathcal{I}$, that is $\mathsf{subst}(\varphi, \mathcal{I})$. In our example, since $\mathcal{I}(\mathsf{port}) = 1$, we check the following:

$$\left(\begin{array}{l} \langle a_1, g_1\rangle = \mathit{Group}(\mathsf{ipv4.dst}_0) \wedge \\ \mathsf{grp}_1 = g_1 \wedge \\ \langle b_1, p_1\rangle = \mathit{Agg}(\mathsf{grp}_1) \wedge \\ (b_1 = \mathsf{set\_port} \wedge \mathsf{port}_1 = p_1 \\ \ \vee\ b_1 = \mathsf{nop} \wedge \mathsf{port}_1 = \mathsf{port}_0) \end{array}\right) \Rightarrow\ \mathsf{port}_1 \neq \mathsf{NONE}$$

We define symbolic compilation using $\textsc{VCGen} : \mathrm{GPL}(\mathcal{T}) \times \mathcal{T} \to \mathcal{T}$, as shown below:

$$\begin{array}{rl} \textsc{VCGen}(p, \varphi) \triangleq & \textbf{let } c = \mathsf{model}(p) \textbf{ in} \\ & \textbf{let } c', \mathcal{I} = \mathsf{passify}(c, \mathcal{Z}) \textbf{ in} \\ & N(c') \Rightarrow \mathsf{subst}(\varphi, \mathcal{I}) \end{array}$$

[2]The standard presentation of compact symbolic compilation [25] also uses an additional *wrong execution function* $W$ which captures when programs violate assert statements. But $\mathrm{GCL}(\mathcal{T})$ has no assertions, so it can never "go wrong."

We prove that VCGen is a precise ci-spec:

Theorem 3.2 (Symbolic Compilation). *VCGen$(p, \varphi)$ is a precise ci-spec for $p$ and $\varphi$.*

Proof. By Theorem 3.1 and [25]. □

Hence, the sentence VCGen$(p, \varphi)$ is a valid formula for validating configs. However, this formula, being almost a line-for-line translation of the initial problem $p \models \varphi$, with the added complexity of indexed variables, is not a significant improvement on the original program. Further, finding counterexamples for a concrete $\sigma$ (i.e., satisfying $N(c) \wedge \neg\text{subst}(\varphi, \mathcal{I})$) is NExpTime-complete [30]. Checking such a formula on every control-plane update could incur significant latency.

## 4 Efficiently Control-Monitorable Sentences and Their Inference

Rather than repeatedly running NExpTime-complete checks, we propose a class (ECMS) of first-order sentences that can be checked efficiently—i.e., with polynomial complexity for a fixed set of typed functions $\mathbf{F}$. Specifically, we will characterize the complexity of monitoring an ECMS in terms of its *expression complexity*, a concept from database theory [1]. We then define an algorithm that computes a precise ci-spec, by leveraging quantifier elimination (Qe), making sure to show that this precise ci-spec is an ECMS. Finally, we show that any algorithm that computes a ci-spec in ECMS can solve the Qe for UFBV. This equivalence means that computing an ECMS may still incur a combinatorial blowup—i.e., the formula we generate will have exponential size in cases where bit-blasting is required. However, as shown in our experiments, we avoid bit-blasting in the common case. So working with formulae in ECMS is useful in practice.

First, we define a syntactic set of sentences that are efficiently monitorable by the control plane:

*Definition 4.1 (Efficiently Control-Monitorable).* A sentence $\psi$ of UFBV over a fixed set of functions $\mathbf{F} = \{F_1, \ldots, F_n\}$ is said to be *efficiently control-monitorable* ($\psi \in$ ECMS) if, for variable sets $\mathbf{z} = \{z_1, \ldots, z_n\}$, $\mathbf{y} = \{y_1, \ldots, y_n\}$ and $\mathbf{x} \subseteq \mathbf{z} \cup \mathbf{y}$, $\psi$ can be written $z_1 = F_1(y_1) \wedge \cdots \wedge z_n = F_n(y_n) \implies \varphi(\mathbf{x})$ where $\varphi \in$ QFBV. For brevity, we write $\psi$ as $\mathbf{z} = \mathbf{F}(\mathbf{y}) \implies \varphi(\mathbf{x})$.

To calculate the *expression complexity*,[3] one fixes the database, and expresses complexity in terms of the size of the query. In contrast, to calculate the *data complexity*, one fixes the query, and expresses complexity in terms of the size of the database. The *combined complexity* expresses complexity in terms of the sizes of both the query and the database [1]. In our setting, we focus on expression complexity. First, we fix the control plane interface to be a set $\mathbf{F} = \{F_1, \ldots, F_n\}$ and their associated types, e.g. $F_i : 2^{w_i} \to 2^{\ell_i}$. With a fixed $\mathbf{F}$, the number of functions $n$, and every $w_i$ and $\ell_i$ are also fixed, which means a config $\sigma$ comprises finite functions between fixed-size domains. We show the expression complexity is polynomial.

Theorem 4.2. *For a fixed config $\sigma$, and $\psi \in$ ECMS, checking $\sigma \models \psi$ is polynomial.*

Proof. Let $\psi \in$ ECMS. This means there is $\varphi \in$ QFBV, and variable sets $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ such that $\psi = \mathbf{z} = \mathbf{F}(\mathbf{y}) \implies \varphi(\mathbf{x})$ and $\mathbf{x} \subseteq \mathbf{y} \cup \mathbf{z}$. First, observe that given a valuation $\mu : \mathbf{x} \to 2^*$, checking $\models_\mu \varphi(\mathbf{x})$ is polynomial in the size of $\varphi$—simply evaluate the formula. Since $\sigma$ is fixed, $M = \sigma(F_1) \times \cdots \times \sigma(F_n)$ has a fixed size. Each element $\mu_i \in M$ corresponds to a valuation $\mu_i : \mathbf{z} \cup \mathbf{y} \to 2^*$, and since $\mathbf{x} \subseteq \mathbf{z} \cup \mathbf{y}$, we can write $\mu_i : \mathbf{x} \to 2^*$. It suffices to perform the fixed number of polynomial checks $\models_{\mu_i} \varphi(\mathbf{x})$ for $i = 1, \ldots, |M|$. □

The analysis above also shows that the data complexity, and hence the combined complexity, is exponential in the size of $\sigma$. However, this only captures the uninteresting observation that in

[3] Also called query complexity

the worst case, a monitor must inspect every combination of elements in a pipeline's tables. The expression complexity captures the complexity of each validation.

We show, in the remainder of this section, that rather than computing general first-order-logic formulae, it suffices to compute formulae in ECMS. We do this by showing that inferring a precise ECMS is formally equivalent to quantifier elimination (QE). While quantifier elimination algorithms normally define QE using an existential quantifier variable and use structural recursion to define it over the full grammar, it's more convenient to use the universal variant of QE as below:

*Definition 4.3 (Quantifier Elimination).* Given a formula $\varphi(x_0, \mathbf{x}) \in \text{QFBV}$, with $x_0 \in \text{Var}$, $\mathbf{x} \subseteq \text{Var}$ and $x_0 \notin \mathbf{x}$, a solution to the quantifier elimination problem is a formula $\psi(\mathbf{x})$ on only the variables $\mathbf{x}$ such that $\psi(\mathbf{x}) \Leftrightarrow \forall x_0.\varphi(x_0, \mathbf{x})$. We write $\psi(\mathbf{x}) = \text{QE}(\forall x_0.\varphi(x_0, \mathbf{x}))$.

A corollary of our construction in the following sections will be that restricting ci-specs to ECMS does not affect the expressiveness. That is, computing the weakest ECMS is equivalent to computing the weakest first-order *sentence*.

### 4.1 QE Computes Precise ci-specs

To infer a precise ECMS constraining configs $\sigma$ such that $p[\sigma] \models \varphi$, we compile $p$ to a GCL program $c$, and then lift out the functions. We define a lift function that, loosely speaking, separates out the control plane (i.e. the tables) from the data plane (the forwarding behavior). To do this, we introduce ghost variables $\mathbf{z}$ and $\mathbf{y}$ that capture the inputs and outputs of the tables $\mathbf{t}$. Then we write $\mathbf{z} = \mathbf{t}(\mathbf{y})$ to nondeterministically capture all potential table rows—this space is collapsed to the runtime key $x$ in the data plane program. We use $\mathbf{x}$ to indicate the remaining variables that occur in $c$. Formally we write $\text{lift}(c) = \langle \mathbf{z} = \mathbf{t}(\mathbf{y}), d \rangle$ to indicate the following, lifted in the expected way:

$$\text{lift}\,(\langle a, d\rangle = t(x); \text{run}_{\mathbf{a}}(a, d)) \triangleq \langle \langle z_a, z_d\rangle = t(y_x), \text{asm}\, y_x = x; \text{run}_{\mathbf{a}}(y_a, y_d)\rangle$$

Notice that the output program $d$ has no uninterpreted functions, that is $c' \in \text{GCL(QFBV)}$. The relationship between $d$ and $c$ can be captured below:

$$(\text{asm}\,(\mathbf{z} = \mathbf{t}(\mathbf{y}))\,; d) \equiv_{\mathbf{x}} c \tag{1}$$

where $\equiv_{\mathbf{x}} \subseteq \text{GCL} \times \text{GCL}$ relates programs that are equivalent on the variables $\mathbf{x}$. We can see this relationship by running lift on our example from Figure 1 as below:

$$\begin{array}{l} \langle z_a, z_g\rangle = \mathit{Group}(y_1) \\ \qquad \wedge \\ \langle z_b, z_p\rangle = \mathit{Agg}(y_2) \end{array} \qquad \begin{array}{l} \text{asm}\, y_1 = \text{ipv4.dst}; \\ \text{grp} := z_g; \\ \text{asm}\, y_2 = \text{grp}; \\ \text{if}(z_b = \text{set\_port})\{\text{port} := z_p\}\{\text{skip}\} \end{array}$$

The formula on the left "queries" the pipeline's interface with the *Group* and *Agg* tables, using the ghost variables $\mathbf{y}$ and $\mathbf{z}$ to capture the results. Then, the program on the right uses these variables to capture the forwarding behavior. Below, we recombine these components according to Equation (1):

$$\begin{array}{l} \langle a, g\rangle := \mathit{Group}(\text{ipv4.dst}); \\ \text{grp} := g; \\ \langle b, p\rangle := \mathit{Agg}(\text{grp}); \\ \text{if}(b = \text{set\_port})\{ \\ \quad \text{port} := p; \\ \}\{\text{skip}\} \end{array} \quad \equiv_{\text{ipv4.dst,grp,port}} \quad \begin{array}{l} \text{asm}\left(\begin{array}{l} \langle z_a, z_g\rangle = \mathit{Group}(y_1) \wedge \\ \langle z_b, z_p\rangle = \mathit{Agg}(y_2) \end{array}\right); \\ \text{asm}\, y_1 = \text{ipv4.dst} \\ \text{grp} := z_g; \\ \text{asm}\, y_2 = \text{grp}; \\ \text{if}(z_b = \text{set\_port})\{ \\ \quad \text{port} := z_p \\ \}\{\text{skip}\} \end{array}$$

Notice that in the lifted program on the right we've lifted all function calls to the start of the program. We then use the assumptions like asm $y_1$ = ipv4.dst to collapse the space of lookups to precisely those where we looked up the value of ipv4.dst in the function *Group*.

Because the lifting stage preserves equivalence on the relevant variables (Equation (1)), it will intercede after the modeling stage. To evoke the fact that lift separates the control plane from the data plane, we will define a control plane symbolic compilation function $\mathcal{C} : \text{GPL} \to \text{UFBV}$ and a data plane symbolic compilation function $\mathcal{D} : \text{GPL} \times \text{QFBV} \to \text{QFBV}$. We define these below:

$$\begin{array}{l} \mathcal{C}(p) \triangleq \ \textbf{let } c = \mathsf{model}(p) \textbf{ in} \\ \qquad\qquad\ \textbf{let } \vartheta, d = \mathsf{lift}(c) \textbf{ in} \\ \qquad\qquad\ \vartheta \end{array} \qquad \begin{array}{l} \mathcal{D}(p, \varphi) \triangleq \ \textbf{let } c = \mathsf{model}(p) \textbf{ in} \\ \qquad\qquad\ \textbf{let } \vartheta, d = \mathsf{lift}(c) \textbf{ in} \\ \qquad\qquad\ \textbf{let } d', \mathcal{I} = \mathsf{passify}(d, \mathcal{Z}) \textbf{ in} \\ \qquad\qquad\ \textbf{let } \varphi' = \mathsf{subst}(\varphi, \mathcal{I}) \textbf{ in} \\ \qquad\qquad\ N(d') \Rightarrow \varphi' \end{array}$$

Both of these functions start the same, by modeling $p \in \text{GPL}$ as a GCL program $c$ and then lifting the control plane $\vartheta$ out of the data plane $d$. The control plane function $\mathcal{C}$, stops here and returns $\vartheta$. The data plane function continues its symbolic compilation, by computing a passive version $d'$ of $d$ by calling $\mathsf{passify}(d, \mathcal{Z})$ (recall that $\mathcal{Z}$ zero-initializes all passivization indices). Then, the data plane function normalizes the spec $\varphi$ corresponding to the output indices $\mathcal{I}$, which produces $\varphi'$. Finally, $\mathcal{D}$ returns the formula $N(d') \Rightarrow \varphi'$.

The following lemma shows that $\mathcal{C}$ and $\mathcal{D}$ precisely characterize pipelines:

Lemma 4.4 (Lifting). $\text{VCGen}(p, \varphi) \iff \mathcal{C}(p) \Rightarrow \mathcal{D}(p, \varphi)$

Proof. By Equation (1) □

The final step is to use Qe to eliminate the packet variables $\mathbf{x}$ from $\mathcal{D}(p, \varphi)$. Since $\mathcal{D}(p, \varphi)$ is a formula over the original data plane variables $\mathbf{x}$ as well as on the ghost variables $\mathbf{y}$ and $\mathbf{z}$, the result of using Qe to eliminate $\mathbf{x}$, will be a formula $\psi(\mathbf{y}, \mathbf{z})$ over just the variables $\mathbf{y}$ and $\mathbf{z}$. In fact, a key result in the domain of logical abduction [17, 18] is that $\psi(\mathbf{y}, \mathbf{z})$ is the *weakest* formula on the variables $\mathbf{y}$ and $\mathbf{z}$ such that $\psi(\mathbf{y}, \mathbf{z}) \Rightarrow \mathcal{D}(p, \varphi)$. Combining this weakness with the fact that Qe is equivalence-preserving, we can see that Qe suffices to solve the ci-spec inference problem.

Formally, we define a procedure $\text{PrecSpec}(p, \varphi)$ as follows:

$$\text{PrecSpec}(p, \varphi) \triangleq \mathcal{C}(p) \Rightarrow \text{Qe}(\forall \mathbf{x}. \mathcal{D}(p, \varphi))$$

where $\mathbf{x} = \text{Var} \setminus \mathbf{y}$ where $\mathbf{y}$ is the set of all ghost variables that occur in $\mathcal{C}(p)$. Said another way, $\mathbf{x}$ is the set of indexed data plane variables.

Now, based on the observations we've made so far, we can prove that $\text{PrecSpec}(p, \varphi)$ precisely captures the control plane configs $\sigma$ that make $p$ satisfy its spec $\varphi$:

Theorem 4.5. $\text{PrecSpec}(p, \varphi)$ *is a precise ci-spec.*

Proof. By Lemma 4.4, [17, 18], and Definition 4.3. □

Finally, by examining its syntax, we'll see that $\text{PrecSpec}(p, \varphi) \in \text{ECMS}$! Here's how: since $\mathcal{C}(p)$ can be written as $\mathbf{z} = \mathbf{F}(\mathbf{y})$, and since $\varphi \in \text{QFBV}$, then $\text{Qe}(\forall x. \mathcal{D}(p, \varphi))$ is QFBV. Further, since $\text{PrecSpec}(p, \varphi)$ is indeed precise, the fact that it is also in ECMS means that we have not given up any precision in restricting our ci-specs to be efficiently monitorable.

At first blush, it seemed that ci-spec inference would require us to learn arbitrary first-order logic formulae. We've shown here that it suffices to learn formulae in ECMS, and specifically, that we need only eliminate quantifiers in the theory of bitvectors.

### 4.2 Precise ci-spec Inference in ECMS Solves $\textsc{Qe}$

Unfortunately, we can show that precise ci-spec inference in ECMS solves $\textsc{Qe}$ in the theory of bitvectors—whose best known algorithms [5, 15] require bit-blasting the finite domain of quantification. Hence, for ci-spec inference, we also resort to bit-blasting in the worst case.

Consider a formula $\varphi \in \text{QFBV}$. We want to compute $\textsc{Qe}(\forall x_0.\, \varphi(x_0, \mathbf{x}))$. To do this, we will use the GPL program $t(\mathbf{x})$ where $t : 2^{|x_1|+\cdots+|x_n|} \to \{\lambda().\text{skip}\}$, and compute its ci-spec w.r.t. $\varphi(x_0, \mathbf{x})$. Now, $\textsc{PrecSpec}(t(\mathbf{x}), \varphi(x_0, \mathbf{x}))$ gives us the following formula:[4]

$$\langle z_a, z_d \rangle = t(\mathbf{y}) \Rightarrow \textsc{Qe}(\forall x_0, \mathbf{x}.\, \mathbf{y} = \mathbf{x} \Rightarrow \varphi(x_0, \mathbf{x})) \tag{2}$$

Notice that the call to $\text{run}_\text{a}(z_a, z_d)$ has disappeared. This is because the choice to run one of the actions in the singleton set $\{\text{skip}\}$ will deterministically run that single action. As a result, $z_a$ and $z_d$ do not occur except for in the leftmost assumption. So, we can use the so-called "one-point rule" (also known as destructive equality resolution), to rewrite Equation (2) into the following:

$$\textsc{Qe}(\forall x_0, \mathbf{x}.\, \mathbf{y} = \mathbf{x} \Rightarrow \varphi(x_0, \mathbf{x})) \tag{3}$$

Next, we apply the one-point rule again and swap $\mathbf{y}$ for $\mathbf{x}$, which then lets us eliminate the innermost $\forall \mathbf{x}$, since the variables in $\mathbf{x}$ no longer occur. We that the following formula:

$$\textsc{Qe}(\forall x_0.\varphi(x_0, \mathbf{x})) \tag{4}$$

which is equivalent to our original sentence. Having just proved it, we state the theorem below.

Theorem 4.6. $\textsc{PrecSpec}(t(\mathbf{y}), \varphi(\mathbf{x}, \mathbf{y})) = \forall \mathbf{y}.\textsc{Qe}(\forall \mathbf{x}.\varphi(\mathbf{x}, \mathbf{y}))$ *where* $t : 2^{|y_1|+\cdots+|y_n|} \to \{\lambda().\text{skip}\}$.

Proof. As above. □

The downside of having shown the equivalence of $\textsc{Qe}$ and ci-spec inference in ECMS is that the best-known algorithms resort to bit-blasting in the worst case. However, in what follows, we exploit domain insights to develop an algorithm that can eliminate quantifiers effectively.

## 5 Programmatic $\textsc{Qe}$

Since inferring ci-specs is intractable in general, we pursue heuristic techniques that work well in practice. A standard maneuver when dealing with large, intractable problems is to decompose the problem into smaller, easier-to-solve, sub-problems. We exploit the fact that ci-spec inference commutes with choice (i.e., []). That is, given a GPL program $p_1 \,[]\, p_2$ and a spec $\varphi$, it is the case that $\textsc{PrecSpec}(p_1 \,[]\, p_2, \varphi) \Leftrightarrow \textsc{PrecSpec}(p_1, \varphi) \wedge \textsc{PrecSpec}(p_2, \varphi)$. By reasoning inductively, this relationship can be generalized over all paths: $\textsc{PrecSpec}(p, \varphi) \Leftrightarrow C(p) \Rightarrow \bigwedge_{\pi \in \text{paths}(p)} \textsc{PrecSpec}(\pi, \varphi)$. We define $\text{paths} : \text{GCL} \to \mathcal{P}(\text{GCL})$ below:

$$\begin{aligned}
&\text{paths} : \text{GCL} \to \mathcal{P}(\text{GCL}) \\
&\text{paths}(x := e) \triangleq \{x := e\} \\
&\text{paths}(\text{asm}\,\varphi) \triangleq \{\text{asm}\,\varphi\} \\
&\text{paths}(c_1; c_2) \triangleq \{\pi_1; \pi_2 \mid \pi_i \in \text{paths}(c_i), i = 1, 2\} \\
&\text{paths}(c_1 \,[]\, c_2) \triangleq \text{paths}(c_1) \cup \text{paths}(c_2)
\end{aligned}$$

Notice that we have defined paths on the GCL level. That is for a program $c \in \text{GCL}$, $\text{paths}(c) \subseteq \text{GCL}(\mathcal{T})$ is the set of straight-line programs (aka paths) through $c$. We then define $\text{paths}(p)$ for a program $p \in \text{GPL}$ by first compiling $p$ to its data plane-only representation using model and lift. That is $\text{paths}(p) = \text{paths} \circ \pi_2 \circ \text{lift} \circ \text{model}(p)$. Further, we have defined $\pi \in \text{GCL}(\text{UFBV})$. We define $\textsc{PrecSpec}(\pi, \varphi)$ to be $\textsc{Qe}(\forall \mathbf{x}.\mathcal{D}(\pi, \varphi))$, where $\mathbf{x}$ is the set of non-ghost variables in $\mathcal{D}(\pi, \varphi)$.

[4] Technically, PrecSpec computes a formula where each variable has a passive index of 0, that is $x_{00}$, $\mathbf{x}_0$, $\mathbf{y}_0$, but by erasing the indices, we get the formula shown in Equation (2)

### 5.1 Paths Produce Smaller QE Problems

Computing the ci-spec for a single path is much more tractable than doing so for a whole program. Aside from being much smaller programs, aggressive compiler optimizations are much more powerful on paths. For instance, we use standard compiler transformations for dead code elimination ($\mathsf{dce} : \mathrm{GCL} \times \mathrm{Var} \to \mathrm{GCL} \times \mathrm{Var}$) and expression propagation ($\mathsf{prop} : \mathrm{GCL} \times \mathrm{GCL} \to \mathrm{GCL}$), as defined below:

$$
\begin{array}{lcl}
\mathsf{dce}(x := e(\mathbf{y}), R) & \triangleq & \begin{cases} \langle \mathsf{asm}\ \mathsf{true}, R \rangle & x \notin R \\ \langle x := e, \mathbf{y} \cup (R \setminus \{x\}) \rangle & x \in R \end{cases} \\
\mathsf{dce}(\mathsf{asm}\ \varphi(\mathbf{y}), R) & \triangleq & \langle \mathsf{asm}\ \varphi(\mathbf{y}), \mathbf{y} \cup R \rangle \\
\mathsf{dce}(c_1; c_2, R) & \triangleq & \textbf{let}\ c_2', R_2 = \mathsf{dce}(c_2, R)\ \textbf{in} \\
 & & \textbf{let}\ c_1', R_1 = \mathsf{dce}(c_1, R_2)\ \textbf{in} \\
 & & \langle c_1'; c_2, R_1 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{prop}(x := e, c) & \triangleq & c[x \mapsto e] \\
\mathsf{prop}(\mathsf{asm}\ \varphi, c) & \triangleq & \mathsf{asm}\ \varphi; c \\
\mathsf{prop}(c_1; c_2, c_3) & \triangleq & \mathsf{prop}(c_1, \mathsf{prop}(c_2, c_3))
\end{array}
$$

The function dce, at every step, removes assignments $x := e$ when $x$ is not in the set of read variables $R$. Similarly, prop propagates $x := e$ by substituting $e$ for $x$ in the rest of the path. This substitution must be done carefully to avoid "capture". For an imperative path like this one, substitution stops once $x$ appears on the left-hand side of an assignment. This definition differs from typical definitions of constant or expression propagation, which need to merge sets of facts at join points. Because we're reasoning about straight-line code, the set of facts never diverges.

These compiler optimizations are actually doing heuristic quantifier elimination *at the program level*. Notice that after the lifting stage, control plane variables will *never* occur in assignments, only the data plane variables will. So, using dce and prop to eliminate as many assignments as possible before running Qe is a clear advantage of path decomposition.

However, the ability to generate smaller and more-optimizable Qe instances doesn't mean much if there are exponentially many of them to solve. Since $\mathsf{paths}(p)$ is exponential[5] in the size of $p$, it remains intractable to compute $\textsc{PrecSpec}(\pi, \varphi)$ for every $\pi \in \mathsf{paths}(p)$.

Luckily, we don't always need to examine every program path. In fact, we only need to explore paths that violate $\varphi$. Since the ci-spec for a path that doesn't violate $\varphi$ is $\top$, then $\textsc{PrecSpec}(c, \varphi)$ is equivalent to the ci-spec for only the paths that violate $\varphi$. Let's call this set of buggy paths $B$. In our experience (Section 8.1), we've seen that the number of these "buggy" paths can be orders of magnitude smaller than the size of $\mathsf{paths}(p)$.

Furthermore, we don't even need to analyze every buggy path in $B$. In fact, our experience has shown (Section 8.4) that the ci-spec for a single path generalizes to solve many paths. For instance, if we added a parser to our example from Figure 1 that either validated one of the main Layer 4 protocols, TCP or UDP, then the ci-spec for either parser path would generalize to the other.

### 5.2 A Path-Based Iterative Strengthening Algorithm

Our algorithm, CegQe, iteratively strengthens a candidate ci-spec using counterexamples. The procedure Strengthen takes in a spec $\varphi$, a candidate ci-spec $\psi_i$, and a spec-violating path $\pi \models \psi_i \wedge \neg\varphi$, and computes a new candidate ci-spec $\psi_{i+1}$ such that $\psi_{i+1} \Rightarrow \psi_i$, and $\pi \models \psi_{i+1} \Rightarrow \varphi$. We define Strengthen as follows:

$$\textsc{Strengthen}_\pi(\psi, \varphi) \triangleq \psi \wedge \textsc{PrecSpec}(\pi, \varphi)$$

By definition, $\textsc{Strengthen}_\pi(\psi, \varphi) \Rightarrow \psi$. Similarly, since $\pi \models \textsc{PrecSpec}(p, \varphi) \Rightarrow \varphi$, then $\pi \models \textsc{Strengthen}_\pi(\psi, \varphi) \Rightarrow \varphi$, indicating that the strengthened ci-spec prohibits $\pi$ from violating $\varphi$.

[5]The well-known *path explosion problem.*

The algorithm iteratively strengthens $\psi$ until an SMT solver proves $\psi$ is stronger than $\mathcal{D}(p, \varphi)$. To maintain precision, $\textsc{Strengthen}_\pi$ will never "overshoot" $\mathcal{D}(p, \varphi)$. That is, as long as $\pi \in \mathsf{paths}(p)$, the invariant $\mathcal{D}(p, \varphi) \Rightarrow \textsc{Strengthen}_\pi(\psi, \varphi)$ holds. This formula is similar to `bf4`'s necessity constraint (which they write $OK \models \phi$, where $\phi$ is a new candidate ci-spec). While `bf4` checks this constraint using an SMT solver after each operation, Capisce maintains this as invariant, which holds because the conjunction of path-based ci-specs is equivalent to the full program ci-spec.

We define the following set $\textsc{BadPath}_p$ to capture all paths $\pi$ that witness the insufficiency of $\psi$ to prove $\mathcal{D}(p, \varphi)$. In practice, we use an SMT solver to produce one such path, when it exists.

$$\textsc{BadPaths}_p(\psi, \varphi) \triangleq \{\pi \in \mathsf{paths}(p) \mid \pi \models \psi \wedge \neg\mathcal{D}(p, \varphi)\}$$

Now, the algorithm can be stated formally. For a program $p$, a spec $\varphi$, and a candidate $\psi$, define:

$$\begin{array}{l}\textsc{CegQe}(p, \varphi) \triangleq \\ \quad \psi \leftarrow \top; \\ \quad \textbf{while } \pi \in \textsc{BadPaths}_p(\psi, \varphi) : \\ \quad\quad \psi \leftarrow \textsc{Strengthen}_\pi(\psi, \varphi); \\ \quad \textbf{return } \psi\end{array}$$

For any program $p$ and spec $\varphi$, the algorithm $\textsc{CegQe}(p, \varphi)$ terminates, because the set of paths through $p$ is finite. By initializing $\psi$ to be $\top$ we ensure that we will never overshoot the correct ci-spec. Similarly, the $\psi$ produced by CegQe is the most precise ci-spec. Correctness of CegQe comes from the fact that the final path is equivalent to $\bigwedge_{p \in B} \textsc{PrecSpec}(\pi, \varphi)$ for some set of bad paths $B \subseteq \mathsf{paths}(p)$. We sketch the proof of this algorithm below:

Theorem 5.1 (Correctness). $\textsc{PrecSpec}(p, \varphi) \iff C(p) \Rightarrow \textsc{CegQe}_{p,\varphi}(\top)$

Proof Sketch. CegQe terminates because it explores a finite set of paths—the continued strengthening of the candidate solution ensures that it never explores the same path twice. The forwards direction follows from the fact that $\textsc{CegQe}(p, \varphi)$ can be written as $\bigwedge_{\pi \in B}(\textsc{PrecSpec}(\pi, \varphi))$, for some set of bad paths $B \subseteq \mathsf{paths}(p)$. Since PrecSpec commutes with choice, the implication holds. The reverse direction follows by the emptiness of BadPaths which implies $\textsc{CegQe}(p, \varphi) \Rightarrow \mathcal{D}(p, \varphi)$. □

Finally, we have $C(p) \Rightarrow \textsc{CegQe}(p, \varphi) \in$ ECMS since $\textsc{CegQe}(p, \varphi) \in$ QFBV.

All told, we've been able to reduce the size of each expensive Qe sub-problem, by decomposing the program into its component paths, and using aggressive compiler optimizations to eliminate variables at the program level.

## 6 Specifications for Data Planes

The standard specification mechanism in data plane verification is assume-assert style specification [32]. Indeed, architectures [27, 45] for the P4 programming language have built-in functions called `assume` and `assert`. Even though they have no semantic effect on the program, programmers use these constructs with verification tools [2, 32] that reason about intermediate states of the system. Since GPL already has assumptions, we need only add assertions.

Unfortunately, adding assertions to GPL incurs a quadratic cost in the size of the formula [25], even along single paths. Assertions, written ast $\varphi$, characterize when programs "go wrong" by violating $\varphi$. The function $W : \mathrm{GCL}(\mathcal{T}) \to \mathcal{T}$, originally defined by Flanagan & Saxe [25], symbolically characterizes these executions for a passive program $c \in \mathrm{GCL}(\mathcal{T})$. It is defined below:

$$\begin{array}{rl} W(\mathsf{ast}\,\varphi) \triangleq & \neg\varphi \\ W(\mathsf{asm}\,\varphi) \triangleq & \bot \\ W(c_1; c_2) \triangleq & W(c_1) \vee N(c_1) \wedge W(c_2) \\ W(c_1 \,[]\, c_2) \triangleq & W(c_1) \wedge W(c_2) \end{array}$$

The quadratic size comes from the sequence rule. A program $c_1; c_2$ can either go wrong because $c_1$ goes wrong ($W(c_1)$), or because $c_1$ goes right ($N(c_1)$), and then $c_2$ goes wrong ($W(c_2)$).

Consequently, checking whether a candidate $\psi$ suffices for a passive program $c$, with asserts, means checking $\psi \Rightarrow \neg W(c)$, which asks if $\psi$ is sufficient to prove that the program never goes wrong. Because the spec $\varphi$ has become part of the program (e.g. via a terminal ast statement), we don't have an explicit spec $\varphi$ to reason about.

In the case that there is some $\pi \models \neg\psi \wedge W(c)$, we know that $\pi$ that contains *at least one* violated assertion. Now we generate the following quantifier elimination problem, which eliminates the data plane variables $\mathbf{x}$ from the wrong executions ($W$) of the path $\pi$:

$$\text{Q}_\text{E}\ (\forall \mathbf{x}.\ \neg W(\pi))$$

Unfortunately, $W(\pi)$ is quadratic in size [25]. In Section 5.2 we were checking the linear-size $N(\pi) \Rightarrow \varphi$. Now, we much larger Q$_\text{E}$ instances. To sidestep this growth, and maintain the compactness of the Q$_\text{E}$ sub-problems, we decompose the problem even further.

Let's proceed by example. For a path $c = c_1; \text{ast}\ \varphi_1; c_2; \text{ast}\ \varphi_2$, where $c_1$ and $c_2$ are ast-free, we would generate the following Q$_\text{E}$ problem:

$$\text{Q}_\text{E}\ (\forall x.\ \ (N(c_1) \Rightarrow \varphi_1) \wedge (N(c_1) \wedge \varphi \wedge N(c_2) \Rightarrow \varphi_2))$$

Observing DeMorgan's laws and the distributivity of Q$_\text{E}$ and $\forall$ over conjunction, this becomes:

$$\text{Q}_\text{E}(\forall \mathbf{x}. N(c_1) \Rightarrow \varphi_1) \quad \wedge \quad \text{Q}_\text{E}(\forall \mathbf{x}. N(c_1) \wedge \varphi_1 \wedge N(c_2) \Rightarrow \varphi_2)$$

Projecting this reasoning back up into the program, each of these subproblems corresponds the following conjunction of assert-free ci-spec inference problems:

$$\textsc{PrecSpec}(c_1, \varphi_1) \quad \wedge \quad \textsc{PrecSpec}(c_1; \text{asm}\ \varphi_1; c_2, \varphi_2)$$

In general, exploiting this distributivity lets us consider assert-final path *prefixes*. Given a counterexample packet *pkt*, our path generation scheme produces the path prefix that terminates in the *first* assertion that is violated. At the end, we find that even in the presence of assertions we will only ever produce *linear*-size Q$_\text{E}$ sub-problems (although there can be many such sub-problems).

## 7 Implementation

We have implemented a ci-spec inference library in OCaml called Capisce. Our library exposes a GPL AST, which makes heavy use of smart constructors. Our algorithm largely follows the structure outlined in the previous section. We discuss here the implementation of path selection and quantifier elimination. We also discuss how GPL can model real world pipeline programs.

*Path Selection.* The BadPaths function in CegQe$(p, \varphi)$ checks whether there exist any buggy paths. To compute this check in practice, we use both an SMT solver, and a tracing execution of the program. First, we use an SMT solver to check SAT$(\psi \wedge \neg\mathcal{D}(p))$, which returns a valuation of the input variables—that is, a packet *pkt*. We then define a tracing execution that runs *pkt* through the program $p$, accumulating its execution trace $\pi$ as it goes. We then use $\pi$ to strengthen $\psi$.

*Quantifier elimination.* To eliminate quantifiers, we rely on an ensemble of state-of-the-art solvers: Z3 [15] and Princess [5]. In our experience, it seems that Z3 is more efficient at bit-blasting, while Princess is better at eliminating formulae with arithmetic operations (+, −, etc). We find that combining the respective strengths of these two solvers is highly effective. Rather than racing the solvers, as is common, we rely on the fact that both solvers produce partial results when they time out. We can then pass these partially-eliminated formulae between the two solvers. We have found that Z3;Princess;Z3 generally suffices.

### 7.1 Modeling P4 in GPL

GPL provides a concise model of the core interface between the data plane and the control plane, namely tables. However, industry standard data plane programming languages, like P4, have a great deal more complexity, including headers, parsing, hash functions, and stateful operations. We discuss how we have modeled these features below:

*Headers:* Headers are similar to `structs` in the C language, with typed fields $f_1, \ldots, f_n$, which can be accessed using standard dot notation, e.g. $h.f_i$. Headers are also equipped with a validity bit $h$.`isValid()` that can be manually manipulated using the `setValid()` and `setInvalid()` methods. We explode headers a list of variables $\texttt{h\_f}_1, \ldots, \texttt{h\_f}_n$, one for each field. We also add an explicit validity bit to each header, e.g. `h_isValid`. Then `h.setValid()` and $h$.`setInvalid()` can be modeled as assignment of 0 or 1 to `h_isValid`. The validity bit for all headers is initialized to 0.

*Metadata*: Metadata is another `struct`-like data representation. They differ from headers only in that they have no validity bit.

*Parsers*: Parsers are often expressed using a finite state machine abstraction [8], however, because of limitations in programmable data plane hardware [9], these finite state machines are required to terminate within a given bound [13]. In practice, it is straightforward to unroll parser loops.

*Hash Functions*: Hash functions are often used for network functions like heavy-hitter detection [38], or load balancing via equal-cost multipath routing (ECMP). We could model hash functions using uninterpreted functions and concolic execution [37]. However, because they typically occur only once in a pipeline, we can usually get away with modeling them using nondeterminism.

*Stateful Operations*: Stateful operations are also used to support a variety of applications including in-network telemetry [38], and in-network caching [29], among others. The challenge in programming with state is that stateful `externs` in P4 programs are subject to data races,[6] except when surrounded by the `@atomic` annotation. For simplicity, we treat non-atomic register reads as producing nondeterministic values, while treating registers in `@atomic` blocks like fields in headers.

## 8 Experience

To assess the usefulness of Capisce, we investigate the following five research questions:

**RQ1**: Can CegQe to infer real safety properties?
**RQ2**: Are most program paths correct?
**RQ3**: Do ci-specs for individual paths generalize over many paths?
**RQ4**: Can ci-specs help programmers find bugs?
**RQ5**: How does Capisce compare to `bf4`?

In Section 8.1 we use Capisce to infer ci-specs for a suite of P4 programs collected from previous work [19], answering **RQ1** in the affirmative. Most ci-specs are inferrable in a few seconds, with several taking hours. Two programs, `fabric` and `linearroad`, reached our timeout of 24 hours, without having terminated. However, with simple fixes to each program, we can infer their ci-specs. We discuss these examples in Section 8.5.

To answer **RQ2** and **RQ3**, which refer to the guiding assumptions about the prevalence of bugs, we measure the proportion of paths that are covered after each strengthening step of CegQe. Our analysis in Section 8.4 shows that for the programs we analyzed, 40-96% of paths were initially correct. We also can see that the ci-spec for individual paths can generalize very well—in some cases over thousands of other paths.

To answer **RQ4**, we qualitatively analyzed the ci-specs Capisce computed. In Section 8.2, we discuss the programs that had absurd ($\bot$) ci-specs, that is, there was some buggy packet for every

[6] Section 18.4.1 of the P4 language specification[13]

| Program | Program Paths | Result | Time (s) | Explored Paths | Spec AST Size | Explored Ratio |
|---|---|---|---|---|---|---|
| | | | Absurd Programs | | | |
| `ts-switching` | 21 | $\bot$ | 0.160 | 2 | 1 | 0.095 |
| `mc-nat` | 39 | $\bot$ | 0.089 | 1 | 1 | 0.026 |
| | | | Fixes to Absurd Programs | | | |
| `ts-switching-fixed` | 21 | $\top$ | 0.030 | 0 | 1 | 0.0 |
| `mc-nat-fixed` | 39 | $\top$ | 0.027 | 0 | 1 | 0.0 |
| | | | Trivial Programs | | | |
| `resubmit` | 9 | $\top$ | 0.028 | 0 | 1 | 0.0 |
| `netpaxos-acceptor` | 0.116 | $\top$ | 30.0 | 0 | 1 | 0.0 |
| `ecmp` | 102 | $\top$ | 0.030 | 0 | 1 | 0.0 |
| `hula` | 3629 | $\top$ | 0.068 | 0 | 1 | 0.0 |
| `ndp-router` | 3843 | $\top$ | 2.9 | 0 | 1 | 0.0 |
| | | | Nontrivial Programs | | | |
| `arp` | 95 | $\varphi$ | 5.0 | 0.016 | 349 | 0.17 |
| `heavy-hitter-2` | 267 | $\varphi$ | 0.29 | 3 | 26 | 0.011 |
| `heavy-hitter-1` | 327 | $\varphi$ | 0.60 | 7 | 90 | 0.021 |
| `flowlet` | 649 | $\varphi$ | 1.8 | 9 | 127 | 0.014 |
| `simple_nat` | 66531 | $\varphi$ | 5.2 | 54 | 1421 | 0.00081 |
| `07-multiprotocol` | 54459 | $\varphi$ | 16 | 143 | 3138 | 0.0026 |
| `netchain` | 26726780 | $\varphi$ | $2.9 \times 10^3$ | 264 | 11658 | $9.9 \times 10^{-6}$ |
| `linearroad` | 54477696 | | timeout | | | |
| `fabric` | 133365047559893 | | timeout | | | |
| | | | Spec Smell Program Fixes | | | |
| `heavy-hitter-1-fixed` | 327 | $\varphi$ | 0.63 | 7 | 107 | 0.021 |
| `linearroad-fixed` | 54477696 | $\varphi$ | $5.9 \times 10^4$ | 3236 | 179885 | $5.9 \times 10^{-5}$ |
| `fabric-fixed` | 133365047559893 | $\varphi$ | $1.2 \times 10^3$ | 653 | 41140 | $4.9 \times 10^{-12}$ |

Fig. 4. Experience with using Capisce to check Header Validity on a broad range of P4 programs.

possible config. We were able to analyze these programs and fix their errors. We also analyzed the nontrivial specifications (Section 8.3), which revealed several hitherto-unknown bugs in the source programs. Further, a local analysis of `fabric`, on which Capisce timed out, directed us to fix bugs in its access control logic (Section 8.5.1).

To answer **RQ5**, we compared Capisce with `bf4` on our suite of programs (Section 8.6), and found that while Capisce is often much slower than `bf4`, its computed ci-specs are much safer.

We ran our experiments on a 64-core machine, with Intel Xeon Silver 4216 CPUs @ 2.10GHz, running Ubuntu 22.04. Each experiment was single-threaded.

### 8.1 Capisce in Practice

To understand the effectiveness of Capisce on real-world programs, we inferred ci-specs for the programs used as benchmarks in previous work [19]. These benchmark programs comprise both research and industrial programs that are publicly available on Github.

First, we ran Capisce's inference with respect to two well-studied [19, 20, 32] properties in data plane programming. The first, called *Header Validity*, asserts that every header $h$ is valid every time $h.f$ is read. The second, called *Determined Forwarding*, every packet is assigned forwarding behavior. In V1Model P4, which we use for our examples, we can check determined forwarding by ensuring that the variable `std_metadata.egress_spec` is assigned a value. As previous work

| Program | Program Paths | Result | Time (s) | Explored Paths | ci-spec Size | Explored Ratio |
|---|---|---|---|---|---|---|
| | | | Absurd Programs | | | |
| ecmp | 102 | $\bot$ | 0.320 | 4 | 1 | 0.039 |
| fabric | 133365047559893 | $\bot$ | 7.3 | 5 | 1 | $3.7 \times 10^{-14}$ |
| netchain | 26726780 | $\bot$ | 27 | 7 | 1 | $2.6 \times 10^{-7}$ |
| | | | Trivial Programs | | | |
| arp | 95 | $\top$ | 0.027 | 0 | 1 | 0.0 |
| linearroad | 54477696 | $\top$ | 0.054 | 0 | 1 | 0.0 |
| simple-nat | 5548 | $\top$ | 0.034 | 0 | 1 | 0.0 |
| | | | Nontrivial Programs | | | |
| resubmit | 9 | $\varphi$ | 0.016 | 2 | 17 | 0.22 |
| ts-switching | 21 | $\varphi$ | 0.10 | 1 | 4 | 0.048 |
| mc-nat | 39 | $\varphi$ | 0.27 | 3 | 21 | 0.077 |
| netpaxos-acceptor | 116 | $\varphi$ | 0.12 | 1 | 4 | 0.0086 |
| heavy-hitter-2 | 267 | $\varphi$ | 88 | 15 | 233 | 0.056 |
| heavy-hitter-1 | 327 | $\varphi$ | 0.10 | 11 | 187 | 0.034 |
| flowlet | 649 | $\varphi$ | 79 | 15 | 490 | 0.023 |
| hula | 3629 | $\varphi$ | 0.39 | 1 | 9 | 0.00028 |
| ndp-router | 3843 | $\varphi$ | 40 | 36 | 824 | 0.0094 |
| 07-multiprotocol | 54459 | $\varphi$ | 30 | 232 | 5034 | 0.0043 |
| | | | Spec Smells & Fixes | | | |
| ecmp-fixed | 102 | $\varphi$ | 0.28 | 3 | 34 | 0.029 |
| mc-nat-fixed | 27 | $\top$ | 0.029 | 0 | 1 | 0.0 |

Fig. 5. Experience with using Capisce to check Determined Forwarding on a broad range of P4 programs.

has shown [20, 32], satisfying these properties requires complicated invariants on the control plane configs that potentially involve multiple tables, which makes them excellent benchmarking properties. The results can be seen in Figures 4 and 5. The "Result" column categorizes the result of running CegQe on the program: $\top$ indicates that CegQe returned true; $\bot$ means that CegQe returned false, and $\varphi$ captures everything in between. The "Time" column presents the number of seconds to 2 significant figures. We also report the number of paths through the original program, as well as the number of concrete paths that CegQe explored.

Observe first, that most of our programs have non-trivial and non-absurd specifications. These are programs that would have been rejected by standard verifiers [21, 32, 40, 43]. Second, observe that most of these programs have reasonable solve times—a few seconds to a few minutes. Even for the larger programs that have hundreds of millions to quadrillions of paths, we are only exploring a minute fraction of those paths.

### 8.2 True Data-Plane Bugs

For programs that produced empty control plane properties ($\bot$), we inspected the programs to understand the errors. For *Header Validity*, many of these programs with true data plane bugs had made implicit assumptions about the packets that would successfully pass the parser, which is a well-documented pattern [20]. We describe how we incorporated these assumptions in the Section 8.2. Conversely, for programs with determined forwarding bugs, the fix is to specify that by default the packet should be dropped at the start of egress processing, which trivializes all ci-specs.

*8.2.1 Header Validity.* In this section, we discuss the true data-plane bugs that we found in the `mc-nat` program. The `mc-nat` program is an industrial R&D program. The parser for this program performs standard Ethernet and IPv4 parsing, which means that at the start of the pipeline, Ethernet is known to be valid, but IPv4 may or may not be. The error occurs in the first table `set_mcg` shown below. Its key is `ipv4.dstAddr`, and therefore to instrument it for *Header Validity*, we have asserted the validity of `ipv4` (top right), which is not guaranteed by the parser (below left).

```
// Table Definitions
set_mcg : bit<32> -> { ... }
// Parser
eth.isValid = 1;
if (eth.type == 0x0800){
  ipv4.isValid := 1; ...
} else {
  ipv4.isValid := 0; ...
}
...
```

```
// Buggy Pipeline
assert(ipv4.isValid = 1); // error!
set_mcg(ipv4.dstAddr); ...
```

```
// Fixed Pipeline
assume(ipv4.isValid = 1); // fix!
assert(ipv4.isValid = 1);
set_mcg(ipv4.dstAddr); ...
```

After inspecting the program, we concluded that the engineers only intended for this program to run on IPv4 packets. We realize this apparent assumption by adding an assume statement (shown above on the right). With this assumption, the assertion follows immediately, and Capisce returns the ci-spec true. The results for the fixed program are reported in Figure 4 under `mc-nat-fixed`. The bug in `ts-switching` has a similar character and similar fix.

*8.2.2 Determined Forwarding.* We also found true violations of *Determined Forwarding*. One such example can be found in the `ecmp` program. Improving on the previous example, the `ecmp` program does guard the accesses to the optionally-valid `ipv4` header with an if statement. The problem is that when the `ipv4` header is invalid, no code is run, which means that the forwarding behavior is not determined. The following presents an outline of the `ecmp` program.

```
// Table Definition
table ecmp_group : bit<32> -> { ... }
// Pipeline
determined := 0;
if (ipv4.isValid == 1 && ipv4.ttl > 8w0) {
  ecmp_group(ipv4.dst); // may or may not determine forwarding
  ...
} else {
  // does NOT determine forwarding!
}
assert (determined == 1); // violated when the else branch is run
```

This bug has several fixes. We could set all packets to be dropped at the start of the pipeline, or we could manually determine the forwarding behavior in the else branch. After applying the latter fix, to produce `ecmp-fixed`, Capisce computes a sensible ci-spec in 280ms.

### 8.3 Bugs Found by Inspecting ci-specs

For most programs, Capisce computes non-trivial and non-absurd ci-specs (indicated by $\varphi$ in Figures 4 and 5). We manually analyzed these specs, which gave us new insights about the programs. Concretely, we were able to discover real bugs in the programs. Borrowing the idea of *code smells*, we identify some simple "spec smells" that we have used to find bugs.

The first spec smell, called *prohibited action*, occurs when the inferred ci-spec prohibits one of the tables actions from ever occurring. It would be unusual for a programmer to implement an action that must never be used. The most likely explanation is that the program has a bug. The second smell, called *obligatory wildcard*, occurs when the inferred ci-spec requires a table to always wildcard one of its keys. Again, it would be unusual to declare a table with a useless key.

We return to mc-nat, which exemplifies the *prohibited action* code smell when analyzed w.r.t. the *Determined Forwarding* property. We then analyze heavy-hitter-1 which exemplifies the *obligatory wildcard* code smell.

*8.3.1 Prohibited Action.* Returning to the mc-nat program, we will set_mcg, more closely. It has the following three actions: set_output_mcg, drop, and nop shown below.

```
// Definitions
action set_output_mcg (mcast_group : 16) = meta.mcast_group := mcast_group
action drop () = std_meta.egress_spec := 511
action nop () = skip
table set_mcg : bit<32> -> { set_output_mcg, drop, nop }
// Pipeline
set_mcg(ipv4.dst); ...
```

Of set_mcg's actions, only one that sets the egress specification: drop. Further, set_output_mcg sets meta.mcast_group, which triggers an assignment to the egress_spec field later in the pipeline. Finally, nop does neither, and the egress_spec remains undefined. As a result, Capisce computes a spec that prohibits set_output_mcg from running the nop action. This is a *prohibited action* smell, and a true bug. To fix it bug, we can simply remove nop from the actions list. After doing this, Capisce computes the trivial ci-spec—that is, $\top$—in 29ms.

*8.3.2 Obligatory Wildcard.* In the heavy-hitter-1 program, we find an example of the *obligatory wildcard* spec smell when analyzing it w.r.t. *Header Validity*. The ci-spec computed by Capisce forces the ipv4.dst address to be wild-carded. We can examine the pipeline below to see why:

```
// Table Definitions
table count_table : bit<32> -> { ... }
table ipv4_lpm : bit<32> -> { ... }
table forward : bit<32> -> { ... }
// Pipeline --- ipv4 may or may not be valid
// To fix, assume ipv4.isValid = 1
count_table(ipv4.dst); ipv4_lpm(ipv4.dst); forward(nhop_ipv4);
```

After running a parser that optionally parses the IPv4 header (similar to the one shown in Section 8.2.1 for mc-nat), the heavy-hitter-1 program immediately reads the ipv4.dst address. Since the ipv4 header may be invalid, count_table *must not* read from it. Capisce recognizes this and forces count_table to wild-card its key. It seems strange that a single-key table should not be allowed to use any of its packet-classification power. This is likely not intended by the programmer, so we declare it a bug. We can fix it by assuming ipv4's validity. After doing so, Capisce computes a sensible spec in 630ms.

## 8.4 Analyzing Path Decomposition

The majority of our programs had non-trivial ci-specs. Even for programs with quadrillions of paths, Capisce explores only a small fraction of them. In the extreme, for fabric-fixed, while we do explore nearly 41k paths, this is 12 orders of magnitude smaller than the number of paths through the program itself. While this fraction is extreme for our dataset, the rightmost columns of Figures 4 and 5 show that Capisce explores a small fraction of paths.

While Figure 4 shows that it suffices to explore relatively few of a program's paths, we want a more fine-grained answer to our guiding assumptions (**RQ2** & **RQ3**). How many paths are actually buggy? How many paths are covered by each strengthening step?

To answer these questions, we measure how path coverage evolves as the candidate ci-spec gets stronger for a few of our small programs with nontrivial ci-specs. After the run finished, we measured the proportion of paths that satisfied the specification after assuming the new candidate

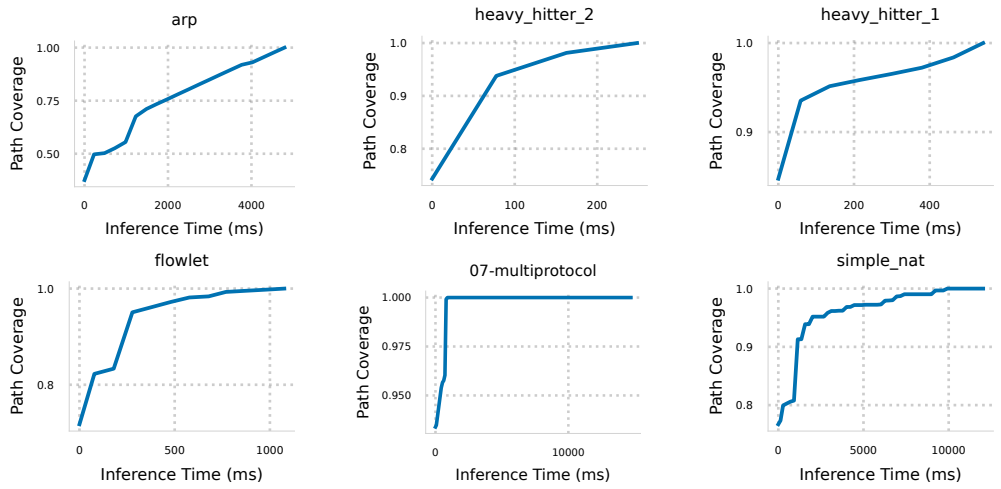



Fig. 6. Path coverage over time for Header Validity analysis of programs with fewer than 100k paths.

spec—we call this proportion *path coverage*. We restricted ourselves to programs with fewer than 100k paths to make this analysis tractable.

The results of this analysis are shown in Figure 6. Notice the high proportion of safe paths when the inference time is 0. At the start the candidate ci-spec is $\top$, so the path coverage metric at this point is measuring the proportion of safe paths. The proportion of safe paths is very high, the lowest being 40% and the highest being nearly 95%. This empirical evidence supports our first guiding assumption: programs are usually correct. Second, notice the steep inclines early towards the left of the figures. These indicate that strengthening is highly effective—many other paths were covered by PrecSpec$(\pi, \varphi)$. For instance, in the `simple-nat` run, the ci-spec $\psi_2$ that was computed by the second bad path, $\pi_2$, covered approximately 10% of the remaining buggy paths. This empirical evidence supports our second guiding assumption: the ci-spec for a single path suffices to cover many other paths.

### 8.5 Limitations

So far, we've seen that despite the theoretical difficulty of ci-spec, Capisce computes useful ci-spec for real-world programs. Unfortunately, because ci-spec inference is theoretically difficult, it is unsurprising that Capisce hits the 24h timeout on 2 programs: `linearroad` and `fabric`. However, these timeouts can be considered their own "spec smells." In diagnosing why these programs reached the timeout, we found issues in the code. After fixing them, Capisce produced ci-specs.

*8.5.1 Fabric.* ONOS's `fabric.p4` is a production-grade L2/L3 data plane program. Originally used as a target for an internal API, it has evolved to be a mid-level abstraction layer [12], as well as support higher-level user plane functionality [33].

We were unable to compute a ci-spec for `fabric` in 24 hours (in fact, it took 16 days). In analyzing the ci-spec for subprograms, we detected the *obligatory wildcard* code smell in the `acl` table. Concretely, the ci-spec forces the `acl` table to always wild card `icmp.type` and `icmp.code`. This is because there is no way to for the controller to ensure that `hdr.icmp` is always valid.

The issue arises in `fabric`'s treatment of tunneling. After running the metadata initialization below on the left, `lkp` metadata holds the innermost valid IPv4 protocol and ICMP header data, as shown in the type and code below:

```
// Metadata Initialization
if (inner_ipv4.isValid = 1){
  lkp.ip_proto := inner_ipv4.proto;
  if (inner_icmp.isValid = 1){
    lkp.icmp_type := inner_icmp.type;
    lkp.icmp_code := inner_icmp.code;
  } else {}
} elif (ipv4.isValid = 1) {
  ...
  lkp.ip_proto := ipv4.proto;
  if (icmp.isValid = 1) {
    lkp.icmp_type := icmp.type;
    lkp.icmp_code := icmp.code;
  } else {...}
} else {...}
```

```
// Buggy Table Keys
acl(eth_type,
    lkp.ip_proto, ...,
    icmp.type, // buggy!
    icmp.code, // buggy!
    ... );
```

```
// Fixed Table Keys
acl (eth_type,
     lkp.ip_proto, ...,
     lkp.icmp_type, // fix!
     lkp.icmp_code, // fix!
     ...);
```

Now, in the `acl` table on the right, even though both `eth_type.value` and `lkp.ip_proto` appear in the keys, they are not sufficient to determine the validity of `icmp`. Together, these two keys can only determine that either `icmp` or `inner_icmp` is valid, but not which. Consequently, reads to the `icmp` header reads must be wild-carded. In the fixed version of the program, `fabric-fixed`, we replaced `icmp` and `icmp_code` with their respective `lkp` counterparts. With these fixes, Capisce computes its ci-spec in about 20 minutes.

*8.5.2 Linearroad.* Despite our best efforts to minimize QE problem instances, `linearroad`'s use of complex machine arithmetic causes our ensemble of QE solvers to resort to bit-blasting. This kind of computation is not typical in data plane programs. In fact, `linearroad` is an experimental program that was written to evaluate use of P4 for implementing streaming database queries [28].

The complex machine arithmetic arises in the update_ewma_spd table, which computes an estimated weighted moving average (EWMA). The relevant pieces of it are shown below:

```
seg_meta.ewma_spd := seg_meta.ewma_spd * 96 + pos_report.spd * 16 >> 7
...
check_toll(..., seg_meta.ewma_spd, ...)
```

Since the value of the complicated machine arithmetic expression on the left flows into the key of the *CheckToll* table, we will need to reason about the possible values of seg_meta.ewma_spd. For instance, there are certain values, like `0xFE00`, that will never be assigned to seg_meta.ewma_spd, this causes solvers to bit-blast to compute these values. However, in our inspection of the source code and the test cases, it's clear that the programmers did not intend for there to be any correctness requirements on this key. Capisce provides an annotation mechanism for keys that allows us to avoid bit-blasting and generate possibly less precise ci-specs. Programmers can annotate specific table keys as *unconstrainable*, which means that the ci-spec cannot reject configs based on the value of these keys. After marking seg_meta.ewma_spd unconstrainable, Capisce produces a ci-spec in under 17 hours, after exploring nearly 180k paths.

### 8.6 Comparison to `bf4`

We compared Capisce to the most relevant tool in previous work, `bf4`. To do this, we serialized the programs in our benchmarks as P4 programs and passed them into `bf4`.

To showcase Capisce's improvement over `bf4`, we analyzed *Header Validity* and *Determined Forwarding* over the benchmark suite programs that both tools agreed had bugs.[7] We used `bf4` to report the number of "bugs" (a.k.a. violable assertion points) that were reachable both before assuming the inferred ci-spec. Then, each tool computed its ci-spec and reported the number of

[7]For instance, `fabric` is omitted because `bf4` incorrectly marks it bug-free. Similarly, `hula` is excluded because `bf4` incorrectly detects bugs. We manually verified these analyses by inspecting the P4 code.

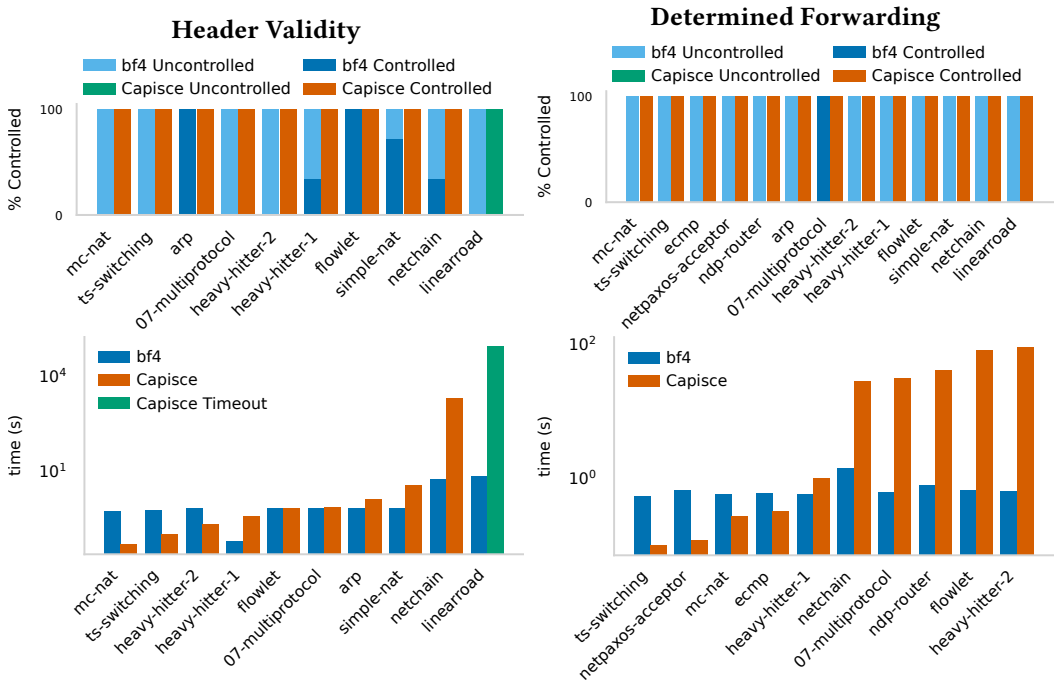



Fig. 7. Comparing analysis capabilities of bf4 and Capisce w.r.t time (bottom) and bugs controlled (top). Note the logarithmic $y$-axes on the time charts.

reachable bugs that remained. If a bug was not reachable after inferring the ci-spec, we say it was *controlled*. The results can be seen in Figure 7.

First, observe that for larger programs, bf4 is orders of magnitude faster than Capisce. Note that the bar graphs at the bottom of Figure 7 have logarithmic $y$-axes. However, bf4 can only control a fraction of the bugs that Capisce can. For *Header Validity*, while Capisce can control 96% of bugs, bf4 can only control 40% of bugs. The only bugs that Capisce cannot control are the bugs for which it times out. For *Determined Forwarding*, Capisce controls 100% of bugs, while bf4 controls only 1 out of 13 bugs for the programs in our benchmark suite.

In comparing Capisce with bf4, we have seen that with its extended runtimes, Capisce can control many more bugs than can bf4. This is unsurprising, as tools have different goals: bf4 quickly computes *necessary* ci-specs that maximize the number of controlled bugs, while Capisce produces safe (and indeed precise) ci-specs—that is, Capisce's ci-specs control all bugs.

## 9 Related Work

We survey some of the most relevant related work on data plane verification and spec inference.

*Logical Abduction.* QE-driven maximal spec inference has been well-studied in the formal methods literature [3, 17, 18]. In particular, the MaxSafeSpec algorithm can be used to produce "maximal" conjunctions of single-table specs.[8] Here, maximal does not mean "weakest," rather, it means that none of the single-table conjuncts can be safely weakened. This notion is indeed stronger than weakness: "maximal" specs are often non-trivially more restrictive than the weakest specs.

[8]The MaxSafeSpec algorithm uses general functions as its core model. In our domain we would specialize their general functions to table calls

*P4 Verification.* Our symbolic compilation is informed by previous work [19, 21, 32, 40, 43], though we are the first to prove our modeling approach correct. The p4v paper informally posed the problem of ci-spec inference [32]. The Π4 paper presents a dependent refinement type system for modular verification in the style of p4v [21]. The `p4-constraints` library offers a language for specifying ci-specs [39], but the language is semantically restricted and does not provide an inference mechanism. The `p4testgen` tool generates test cases for P4 programs, and can reason about ci-specs expressed in `p4-constraints` to reduce false alarms [37].

*Specification Inference.* The SPYRO tool [36] provides a general-purpose framework for spec synthesis which summarizes arbitrary queries from given DSL. In contrast, while our work is specialized to the domain of data planes, Capisce infers a precise ci-spec, without requiring a specific DSL. Further, while SPYRO's algorithm uses a syntactic CEGIS algorithm, our approach is more semantic—we compute ci-specs using deductive tools: symbolic analyses and QE.

The SafeP4 paper presents a type system for checking that a switch program has no invalid header reads, and is equipped with a limited inference procedure for single tables [20]. Our work improves upon SafeP4 in its expressiveness: Capisce protects against any assert-specifiable bug, while SafeP4 is limited to *Header Validity*.

Finally, *Config2Spec* [7], infers control properties of traditional networks using a refinement loop that uses both emulation and verification to generate high quality properties. *Config2Spec* focuses on network-wide control properties, while Capisce focuses on safe configs for individual switches.

*Control Plane Verification & Synthesis.* NETKAT takes a (co-)algebraic approach to specifying, verifying and compiling network-wide control planes [4]. Work on synthesizing consistent updates shows how to synthesize network updates so that each packet views a consistent snapshot of the network [34]. GENESIS [41], NetComplete [22], and Propane/AT [6] all synthesize legacy network configs from high-level specifications. Recent work on P4R-Type [31] develops a typed variant of P4Runtime, the generic control-plane API used by P4 programs. While P4Runtime enforces some type constraints dynamically, P4R-Type guarantees that type errors will not arise at runtime.

## Availability Statement

Capisce is available on Github [11] and Zenodo [10].

## Acknowledgments

We would like to thank Minseok Kwon, who contributed to early discussions on this project; Dragos Dumitrescu for his insights about `bf4`; Steffen Smolka for his insights about industrial ci-specs; Haobin Ni, Ryan Doenges, Oliver Richardson, and Mark Moeller for close-reading numerous introductions; the members of Cornell Netlab for countless insightful discussions; and the EPFL DCSL group for providing a welcoming and supportive environment. We would also like to thank the anonymous OOPSLA reviewers, whose feedback greatly improved Capisce. Our work has been supported in part by the National Science Foundation under grant FMiTF-1918396, the Defense Advanced Research Projects Agency (DARPA) under Contracts HR001120C0107 and HR001124C0429, and gifts from Fujitsu, Google, Keysight, and Juniper.

## References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.

[2] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. 2022. SwitchV: Automated SDN Switch Validation with P4 Models. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 365–379. https://doi.org/10.1145/3544216.3544220

[3] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 789–801. https://doi.org/10.1145/2837614.2837628

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

[5] Peter Backeman, Philipp Rummer, and Aleksandar Zeljic. 2018. Bit-Vector Interpolation and Quantifier Elimination by Lazy Reduction. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–10. https://doi.org/10.23919/FMCAD.2018.8603023

[6] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 437–451. https://doi.org/10.1145/3062341.3062367

[7] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. 2020. Config2Spec: Mining network specifications from network configurations. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 969–984.

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

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (aug 2013), 99–110. https://doi.org/10.1145/2534169.2486011

[10] Eric Hayden Campbell. 2024. cornell-netlab/capisce. https://doi.org/10.5281/zenodo.12785373

[11] Eric Hayden Campbell. 2024. cornell-netlab/capisce: Control interface specifications for Dataplane Pipelines. https://github.com/cornell-netlab/capisce

[12] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. 2021. Avenir: Managing Data Plane Diversity with Control Plane Synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 133–153. https://www.usenix.org/conference/nsdi21/presentation/campbell

[13] P4 Language Consortium. 2021. P4 16 Language Specification v.1.2.2. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

[14] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 128–148. https://doi.org/10.1007/978-3-642-35873-9_10

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[16] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (aug 1975), 453–457. https://doi.org/10.1145/360933.360975

[17] Isil Dillig and Thomas Dillig. 2013. Explain: a tool for performing abductive inference. In *International Conference on Computer Aided Verification*. Springer, 684–689. https://doi.org/10.1007/978-3-642-39799-8_46

[18] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. *SIGPLAN Not.* 48, 10 (oct 2013), 443–456. https://doi.org/10.1145/2544173.2509511

[19] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2020. Bf4: Towards Bug-Free P4 Programs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 571–585. https://doi.org/10.1145/3387514.3405888

[20] Matthias Eichholz, Eric Hayden Campbell, Nate Foster, Guido Salvaneschi, and Mira Mezini. 2019. How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 12:1–12:28. https://doi.org/10.4230/LIPIcs.ECOOP.2019.12

[21] Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. 2022. Dependently-Typed Data Plane Programming. *Proc. ACM Program. Lang.* 6, POPL, Article 40 (jan 2022), 28 pages. https://doi.org/10.1145/3498701

[22] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 579–594. https://www.usenix.org/conference/nsdi18/presentation/el-hassany

[23] fabric.p4 source code 2022. fabric.p4 source code. https://github.com/opennetworkinglab/onos/blob/2.2.2/pipelines/fabric/impl/src/main/resources/fabric.p4. Accessed 2022.

[24] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: an intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (apr 2014), 87–98. https://doi.org/10.1145/2602204.2602219

[25] Cormac Flanagan and James B. Saxe. 2001. Avoiding exponential explosion: generating compact verification conditions. *SIGPLAN Not.* 36, 3, 193–205. https://doi.org/10.1145/373243.360220

[26] Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. 2020. Using deep programmability to put network owners in control. *SIGCOMM Comput. Commun. Rev.* 50, 4 (oct 2020), 82–88. https://doi.org/10.1145/3431832.3431842

[27] P4.org Architecture Working Group. 2021. P4 16 Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.html.

[28] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the Fast Lane: A Line-Rate Linear Road. In *Proceedings of the Symposium on SDN Research* (Los Angeles, CA, USA) *(SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3185467.3185494

[29] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[30] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. 2016. Complexity of Fixed-Size Bit-Vector Logics. *Theor. Comp. Sys.* 59, 2 (aug 2016), 323–376. https://doi.org/10.1007/s00224-015-9653-1

[31] Jens Kanstrup Larsen, Roberto Guanciale, Philipp Haller, and Alceste Scalas. 2023. P4R-Type: A Verified API for P4 Control Plane Programs. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 290 (oct 2023), 29 pages. https://doi.org/10.1145/3622866

[32] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 490–503. https://doi.org/10.1145/3230543.3230582

[33] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay ThakuR, Larry Peterson, Jennifer Rexford, and Oguz Sunay. 2021. A P4-Based 5G User Plane Function. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)* (Virtual Event, USA) *(SOSR '21)*. Association for Computing Machinery, New York, NY, USA, 162–168. https://doi.org/10.1145/3482898.3483358

[34] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. 2015. Efficient synthesis of network updates. *SIGPLAN Not.* 50, 6, 196–207. https://doi.org/10.1145/2813885.2737980

[35] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (mar 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[36] Kanghee Park, Loris D'Antoni, and Thomas Reps. 2023. Synthesizing Specifications. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 285 (oct 2023), 30 pages. https://doi.org/10.1145/3622861

[37] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4-16. In *Proceedings of the ACM SIGCOMM 2023 Conference* (, New York, NY, USA,) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 136–151. https://doi.org/10.1145/3603269.3604834

[38] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 164–176. https://doi.org/10.1145/3050220.3063772

[39] Smolka Steffen, Ali Kheradmand, and Antonin Bas. [n. d.]. p4lang/p4-constraints: Constraints on P4 objects enforced at runtime. https://github.com/p4lang/p4-constraints

[40] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 518–532. https://doi.org/10.1145/3230543.3230548

[41] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 572–585. https://doi.org/10.1145/3009837.3009845

[42] switch.p4 source code 2020. switch.p4 source code. https://github.com/p4lang/switch. Accessed Feb, 2022.

[43] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. 2021. Aquila: A Practically Usable Verification System for Production-Scale Programmable Data Planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 17–32. https://doi.org/10.1145/3452296.3472937

[44] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. 2012. On Controller Performance in Software-Defined Networks. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/tootoonchian

[45] 2021. v1model.p4 source code. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4. (2021). Accessed Feb, 2022.

Received 2024-04-06; accepted 2024-08-18