Correct-by-Construction Network Programming for Stateful Data-Planes

Jedidiah McClurg
Colorado School of Mines
Golden, CO, USA
mcclurg@mines.edu

ABSTRACT

As switch hardware becomes faster, more stateful, and more programmable, functionality that was once confined to end hosts or the control plane is being pushed into the data plane. For example, recent work on adaptive congestion control and heavy hitter detection uses stateful switches to implement sophisticated functionality with only minor controller involvement. In applications where correctness depends on individual switches making coherent decisions, it is important that the switches have a consistent view of global state. However, such a consistency requirement makes it difficult to maintain efficiency (high throughput), due to the CAP theorem. Moreover, previous work on data-plane programming provides little to no built-in support for addressing this difficulty.

We propose *Callback State Machines* (CSMs), a new high-level declarative network programming abstraction which allows operators to write correct data-plane programs against global state. CSMs offer programmers useful consistency guarantees without the need to manage how global state is replicated/updated at the individual switch level. To aid in the implementation of this high-level programming framework, we present a flexible new intermediate representation (IR) called TAPIR that natively supports stateful data plane functionality, as well as a compiler to generate device-specific code such as P4 from TAPIR code. Additionally, we demonstrate the power of TAPIR itself by using it to build a working implementation of the CONGA congestion control system.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; **Programmable networks**; • **Computer systems organization** → *Reliability*; • **Software and its engineering** → *Domain specific languages.*

KEYWORDS

SDN, Petri nets, causal consistency, P4

ACM Reference Format:
Jedidiah McClurg. 2021. Correct-by-Construction Network Programming for Stateful Data-Planes. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), September 20–21, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3482898.3483362

SOSR '21, September 20–21, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9084-2/21/09...$15.00
https://doi.org/10.1145/3482898.3483362

1 INTRODUCTION

Software-defined networking (SDN) seeks to make networks more programmable. Early realizations of SDN (e.g., OpenFlow) require all state to reside on the *controller*—the switches effectively serve as caches for static forwarding tables, which can be (re)populated by the controller in response to network events. However, this model is beginning to change. SDN data planes are becoming more capable, with powerful devices emerging which are able to perform computations and update local state based on packet contents, all at line rate [6, 8, 67]. This has fueled an increased interest in pushing functionality which has traditionally been located on end hosts or in the control plane into the data plane. Rather than viewing a *network program* as simply a process that runs on the controller and interacts with switches, we can now view it as a distributed system or *data-plane program*, running atop the networking hardware.

Numerous types of data-plane programs can be found, both in the networking literature, and also deployed in industrial networked systems, such as:

- *Congestion control* — automatically adjusting forwarding paths, based on measured congestion;
- *Traffic management* — optimizing performance of the network along various quality metrics;
- *Monitoring* — performing accurate measurements of properties within the network;
- *Active networking* — allowing packets to carry small "programs" which are executed by forwarding devices as the packet moves through the network; and
- *Network OS/Runtimes* — virtualization functionality built on top of networking devices.

These applications are typically built in an ad-hoc way, using a combination of switch- and host-level functionality tuned to a specific SDN installation. This opens the door for bugs in the implementation, and also makes it difficult to build and prototype new systems based on current designs. We believe that a general and highly intuitive approach for building correct and efficient data-plane programs is needed to address these issues.

Unfortunately, prior work has not fully considered how to properly deal with *global state* in data-plane programs. There are two main concerns when adding state into the mix: (1) consistency—making sure that the program consistently maintains distributed views of the state, and (2) efficiency/availability—making sure that network performance is not penalized by maintenance of consistent views of the state. The CAP theorem applies in this context [58], meaning that there is an inherent tension between consistency and efficiency (since partition tolerance is non-negotiable).

This paper focuses on building a network programming language and runtime that abstracts away these concerns, and gives programmers the right balance between consistency and availability. A salient point here is that we cannot abstract away too much: programmers still need to write distributed, asynchronous programs. This ability is needed to write programs that detect and react to events such as attempted intrusion or congestion.

Our first key contribution is *Callback State Machines (CSMs)*, a declarative abstraction that can be used for describing network behavior. This abstraction is based on solid existing formalisms (*distributable nets* and *event structures* [5, 26, 73]) and combines the intuitiveness of automata-based network programming languages such as Kinetic [40] with the expressive programming constructs in high-level languages such as SNAP [4]. It also inherits support for formally specifying and verifying key program properties from *event nets* [49]. It has the desired features we described above: callbacks that are programmed against global state and run asynchronously, without the need to specify the low-level details of how the switches maintain consistent views of the global state. Our second key contribution is *Stateful Data-Plane Intermediate Representation (TAPIR)*, a new intermediate representation (IR) for stateful data planes. TAPIR allows operators to program in terms of imperative stateful packet-processing functions, instead of thinking at the level of flow tables. Using IR-to-IR translation stages, we show how to leverage TAPIR to build a compiler that produces executable code from CSMs. In this work, we target P4 switches [8], but our compiler could be easily extended with back-ends for different architectures. The compiler performs the following transformations:

- CSM (callbacks that read/write global state)
  ↓
- (per-switch) TAPIR programs with global variables
  ↓
- TAPIR programs that read/write only local state
  ↓
- P4 programs that read/write only local state

Any of these levels of abstraction are available to the programmer, and in Section 5 , we show that programming with the lower-level IR itself is straightforward, by using it to implement the CONGA congestion control system. Our intent is for programs to be written at the CSM level—this has the advantage of providing the programmer with guarantees regarding the efficiency and consistency of the program's resulting data-plane implementation.

The consistency guarantee provided by CSMs is a form of causal consistency, similar to one we described in previous work [51]—causally-related events (ones that occur at the same switch, or as a result of the same packet) are observed in the same order by all switches. Although this is a more relaxed consistency model than, e.g., sequential (atomic) consistency, it allows us to ensure a key property: the programmer can rely on the *control-flow* of the CSM to be observed consistently throughout the network—switches may be out-of-sync in their views of the CSM's "program counter", but not in conflicting ways.

This consistency model allows for an efficient implementation—the CSM executes entirely in the data plane, without involvement of an SDN controller (except possibly to initialize registers on switches upon network startup). The key practical advantage of our new CSM approach versus our previous relaxed-consistency approach [51] is that CSMs allow programming with *loops*, enabling applications such as load balancers.

2 MOTIVATION: COMPOSING LOAD BALANCER WITH STATEFUL FIREWALL

Before detailing our approach, we will first give a high-level overview of data-plane programming, and some of the challenges that it brings. Let us consider the example network shown in Figure 1, where H_2 is a server in a data center, and H_1 is a client requesting data from the server. Initially, all client traffic is directed through S_1, which functions as a permanently-enabled firewall, but as the load on the data-center increases, load-balancer S_{lb} may redirect some flows through S_{fw}. In this case, firewall functionality can also be enabled at S_{fw}. The key things to notice about this example are that (1) action must be taken in response to packet-related events occurring in the data plane, and (2) there are two different "processes" making changes to the state of the network (firewall and load balancer). The former means that it can become highly inefficient to involve the controller in handling these events. For example, S_{lb} would not be able to feasibly ask the controller to make a load-balancing decision on each incoming packet. The latter means that we need to think carefully about how these processes interact, since the processes are simultaneously changing global state in the network—in particular, the end-to-end forwarding behavior is being modified by both processes. These processes are shown graphically in Figure 2.

When compiling high-level stateful functionality into executable code, subtle problems can occur. For example, in Figure 2, if the load balancer goes up before the firewall is enabled, an important security policy will be violated. To avoid this fault, the program will need some type of *synchronization* between the two processes

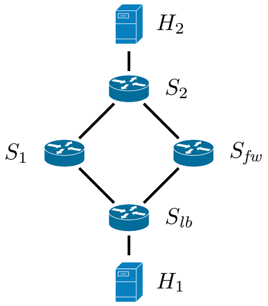

Figure 1: Example topology.

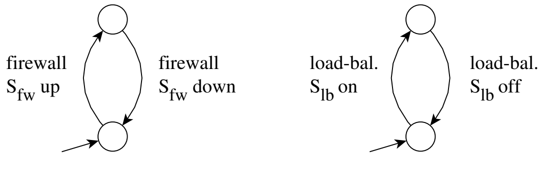

Figure 2: Example processes.

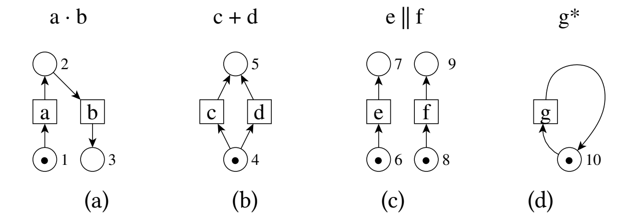

Figure 3: Composition operators and Petri nets: (a) sequencing, (b) choice, (c) concurrent composition, (d) iteration.

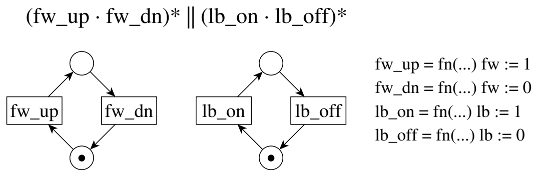

Figure 4: Example processes: Callback State Machine.

in order to prevent the policy violation—i.e., we need language support for writing such programs. More specifically, we need a language (and compiler) that offers the following guarantees to network programmers:

- Ability to read/write global network state such that basic consistency is maintained at all times, i.e., devices should not have incompatible views of state;
- Avoidance of packet buffering and other expensive synchronization operations that reduce throughput;
- *Composability* of programs, i.e., we should be able to run the firewall and load balancer at the same time; and
- Ways of ensuring that programs do not interact incorrectly (e.g., enforcing the security policy in Fig. 1-2).

Our new programming abstraction CSMs provides these guarantees while still allowing intuitive specification of network behavior, similar to the state machine shown in Figure 2.

Petri Nets. The semantics of CSMs can be defined in a straightforward and compositional way using *Petri nets*, a well-known state-machine-like concurrency model. While the programmer can easily use CSMs without understanding Petri nets in detail, some basic background in this area will assist in understanding the precise behavior of the language constructs.

A Petri net is a transition system where one or more *tokens* (denoted by black dots) can move between *places* (denoted by circles), as dictated by *transitions* (denoted by squares). An assignment of tokens to places (representing the "state" of the Petri net) is called a *marking*. Directed edges indicate where tokens can move—an edge can either connect a place to a transition (an *input place* of the transition), or a transition to a place (an *output place* of the transition). A transition is *enabled* when a token is present at all of its input places, and an enabled transition can *fire* by removing a token from each input place, and adding a token to each output place. A *trace* is a sequence of markings that results from firing enabled transitions. Figure 3 shows four Petri nets—the places are 1-10, the transitions are *a*-*g*, and a token is initially present at each of places 1, 4, 6, 8, and 10.

Petri nets provide a flexible framework for concurrency. For example, the Petri net in Figure 3(a) shows how *sequencing* can be modeled—transition *a* is the only one that is enabled, so it must fire first (moving the token to place 2), before transition *b* can fire. Figure 3(b) shows how *choice* can be modeled—either *c* can fire (moving the token to place 5), or *d* can fire, but not both. Figure 3(c) shows how *concurrency* can be modeled—transition *e* can fire (moving the token from place 6 to place 7), and *f* can fire independently. Figure 3(d) shows how *iteration* (cyclic behavior) can be modeled—transition *g* can fire an indefinitely many times.

Callback State Machines. In previous work, we introduced event nets [49, 51], a Petri-net-based language for event-driven network programming. In that language, each transition is labeled with an *event*. An event can be any phenomenon occurring at a specific *location* (specific port on a switch), but for simplicity, events are restricted to packet arrivals. Event nets must be *1-safe*, that is, no place should contain more than one token at any point during the program's execution. This allows each place to be labeled with a set of *forwarding rules* dictating how packets move through the network, and the current *configuration* is taken as the union of all rules on places containing a token. The configuration can change only in ways allowed by the event net—when a transition fires, the forwarding rules corresponding to its input places are "overwritten" by the forwarding rules corresponding to its output places.

Our work extends event nets—a CSM's behavior is defined in terms of a 1-safe Petri net where each transition is labeled with an event and a *callback*, and places are unlabeled (instead of signifying a set of static forwarding rules). When an event corresponding to an enabled transition occurs, the state of the Petri net is updated, and the callback is executed at the location of the event occurrence. A *callback* is a function that receives information about a specific occurrence of an event, and executes a piece of imperative code that can read/write to arbitrary *global variables*, which are readable network-wide, and can be used by switches to make forwarding decisions, etc. This adds significant flexibility to event nets, in which global state is limited to static network configurations.

Composition of CSMs. CSMs can be built from other CSMs using the *composition operators* shown in Figure 3. Figure 4 shows the CSM (and underlying Petri net) for the previously-described firewall/load-balancer example. For the purposes of this discussion, the specifics of the events are not important, so we will focus on the callbacks. This example features two single-bit global variables that determine end-to-end forwarding paths—switches read from these global variables to determine forwarding behavior. The *fw_up* and *fw_down* callbacks modify the *fw* global variable, and the *lb_on* and *lb_off* callbacks modify the *lb* global variable, as shown in Figure 4. If switch S_{lb} finds that $lb = 0$, it forwards all inbound (client) packets left (to S_1), and otherwise, it forwards some inbound packets right (to S_{fw}). If switch S_{fw} finds that $fw = 0$, it forwards all inbound packets to S_2, and otherwise, it drops disallowed inbound packets. A policy violation occurs if disallowed packets are allowed

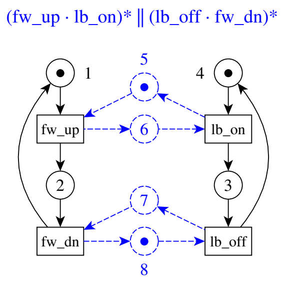

Figure 5: Condition var.

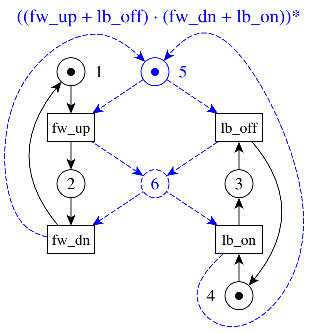

Figure 6: Mutex.

to reach the server H_2. On the surface this looks similar to the simple state machine in Figure 2, but the CSMs actually add a surprising amount of power, especially in terms of compositionality.

Unlike prior work, we can perform *correct* composition of callback nets. Correct means that we can easily add synchronization constructs, in order to ensure that processes do not interact in unwanted ways. For example, in Figure 5, we have added *condition variable* synchronization constructs to the Figure 4 CSM, which prevent it from violating the security policy. In particular, *lb_on* cannot fire until *fw_up* has fired, and subsequently, *fw_down* cannot fire until *lb_off* has fired. The programmer needs only to combine the original Figure 4 CSM with the condition variable CSM shown in Figure 5 using the concurrent composition operator ||.

An important property of this type of correct composition is that the resulting CSM is still *efficiently implementable.* The synchronization constructs in this context work via message-passing, and do not require blocking.

Consistency of the Program Control Flow. The most basic type of correctness that we expect from a data-plane network program is consistency of the program's control flow. That is, devices should not have incompatible views of where the "program counter" is—in terms of our Petri-net semantics for CSMs, this means that switches must not have incompatible views of the Petri net's trace. In our model, this is ensured by a condition called *locality.*

This condition can be explained by way of example—in Figure 5, we saw one way of ensuring that the Figure 4 CSM does not violate the security policy, but let us look at *different* way we can add synchronization to ensure correctness with respect to the policy. In Figure 6, we have added a *mutex*-like construct between the two processes. This ensures that there is a mutually-exclusive choice between firing *fw_dn* and *lb_on*, and also between *fw_up* and *lb_off.* The key detail to notice about this example is that place 6 is an input for *two* different transitions—we say that these transitions *fw_dn* and *lb_on* are *conflicting*.

The locality condition requires events associated with conflicting transitions such as *fw_dn* and *lb_on* to be detected at the same device. If this were not the case, and two such events were detected simultaneously on two different devices, there would need to be either consensus on the ordering (expensive), or acceptance of conflicting program state (incorrect program execution). Returning to the Figure 6 example, let us assume that the events associated with *fw_dn* and *lb_on* are detected at S_{fw} and S_{lb} respectively. If these two events occur (nearly) simultaneously, and both S_{lb} and S_{fw} see a token at place 6, they can fire *lb_on* and *fw_dn* respectively, resulting in an inconsistent program state—S_{lb} believes there to be a token at place 3, and S_{fw} believes there to be a token at place 1, but there is no execution of the Petri net where these two traces could be reconciled.

If the programmer has written a CSM that fails to satisfy the locality condition, we can report an error message pointing to the problematic conflicting transitions. Otherwise, the state of the implemented program will be consistent—switches may have differing views of the Petri net trace, but these traces will be compatible.

Comparison to SNAP. It is now instructive to see how consistency/conflict resolution is handled in prior work. In the SNAP [4] approach, each global variable (such as *fw* and *lb*) is stored on a *single* device. The programmer does not need to be concerned *which* device—the compiler picks the appropriate one. Consistency is maintained in the following way: if certain packets must be processed differently depending on the value of a specific state variable, those packets are (automatically) directed through the appropriate device which contains that variable. This has several drawbacks: (1) It is not tolerant of failures—if a switch containing a certain variable goes down, the program will not be able to execute correctly; and (2) it can cause poor performance, e.g., congestion when packets need to access the same variable. Thus conflict-resolution is done at the expense of concurrency.

Our approach, on the other hand, does not need to "place" state variables. Reads/writes to state happen locally, and state is lazily distributed to other nodes in a way that maintains consistency. Instead of requiring each state variable to be contained at a single node, state is replicated across all devices, making the system more resistant to node failures. The only constraint we need is to require *conflicting* events (as in the Figure 6 load balancer example) to be detected at the same switch.

Consistency of Global Variables. As mentioned in Section 1, our implementation of global variables ensures a form of causal consistency. Global variables are built from conflict-free replicated datatype (CRDT) [66] registers, and updates to these are propagated lazily by piggybacking on data packets. In this way, causality is maintained, i.e., devices that have received a data packet which passed through a switch with a newer view of the global state will incorporate this new view of the state into their own. Returning to the firewall/load-balancer example in Figure 4, firing the *fw_up* transition at S_{fw} updates the *fw* global variable, and causes that switch to immediately begin dropping disallowed packets. However, the update to global variable *fw* is propagated lazily, meaning S_2 will not learn about this new value until it receives a packet from S_{fw}. Similarly, S_{lb} will learn about the new value only when the server sends a response that passes through S_{fw}. Once S_{lb} learns that $fw = 1$, it may make sense to proactively drop disallowed packets destined to S_{fw}—although there is a delay between the time that the global variable changes, and the time that S_{lb} learns about this change, it does not affect correctness with respect to the security policy. This relaxed form of consistency is efficiently implementable, and the CRDTs ensure that switches have compatible views of the values of global variables.

Automatic Synthesis of Synchronization. Additionally, our model enables tools [49] to automatically compose programs in ways that respect certain high-level correctness properties. That is, our model makes it possible to automatically insert code which prevents various types of unwanted races when multiple programs are deployed simultaneously.

3 CALLBACK STATE MACHINES

In this section, we will formalize our new high-level network programming language. Since the language is implemented using a new intermediate representation, we will begin with a discussion of the IR.

3.1 TAPIR: Stateful Dataplane Intermediate Representation

TAPIR is a new IR designed to simplify programming of per-switch data-plane behavior. While languages like P4 can be used for this purpose, we show how they are tied to networking hardware architectures in ways that make them sub-optimal for the types of source-to-source translation stages we need.

Pipelined Stateful Data-Planes. The P4 switch programming language supports *registers*, which can be read/written using values computed from the headers of incoming packets. P4 is a "schema"-like language, which allows a sequence of *tables* to be (conditionally) applied to the packet, and the tables must be separately populated with *forwarding rules* to achieve the desired behavior. Switches that target P4 are able to achieve line rate by using a *pipelined* architecture which executes the P4 program. At a high level, the switch is structured as an *ingress pipeline*, followed by a queueing mechanism, followed by an *egress pipeline*. Each arriving packet is processed by the ingress pipeline (with special packet *metadata* fields set to indicate which port the packet arrived on), and the P4 program can set metadata fields which tell the subsequent queueing mechanism which output port(s) to send the packet to. The queueing mechanism duplicates the packet if needed, and sends each copy through the egress pipeline, where the P4 program can make additional modifications to the packet (except to the output port, which is now fixed).

Need for a New IR. Describing code transformations in terms of this schema-like language is difficult. Our first goal is to make the dataplane programming process more accessible for compiler developers, by moving away from a flow-table-based model towards a more familiar imperative programming language. Our new intermediate representation TAPIR is a simple imperative programming language used in our compiler. Conceptually, the language is similar to P4, in that it provides a way of describing basic packet-processing functionality—specifically, functions which accept a packet, optionally perform some modifications and/or update local switch registers, and then send the packet to another port(s) on the switch. Our IR is different in that it provides a concise and straightforward way of encoding *both* the control flow and match-action table contents.

IR Syntax and Semantics. The full syntax of the IR is shown in Figure 7—the syntax is similar to (a subset of) Rust. In IR programs, the base data are `true` and `false` (of type `bool`), and fixed-width integers of a certain size (e.g., `int32`, `int64`, etc.). Data can be structured into *tuples* such as $(\texttt{false}, 1, 2, \ldots)$, fixed-length *arrays* such as $[1, 2, \ldots]$, and *records* such as $\{\texttt{field1}{:}100, \texttt{field2}{:}200\}$.

$$\begin{array}{rcll}
id & \in & \mathit{Ident} & \textit{(identifier)}\\
n & \in & \mathbb{Z} & \textit{(numeric constant)}\\
\tau & ::= & \texttt{bool} \mid \texttt{int}_n \mid (\tau, \ldots) \mid [\tau; n] \mid \{id : \tau, \ldots\} & \textit{(type)}\\
 & \mid & \tau \to \tau & \\
tid & ::= & id \mid id : \tau & \textit{(typed ident.)}\\
b & ::= & \texttt{true} \mid \texttt{false} & \textit{(boolean)}\\
e & ::= & b \mid n \mid id \mid id(e, \ldots) \mid [e, \ldots] \mid e[e] \mid (e, \ldots) & \textit{(expression)}\\
 & \mid & \{id : e, \ldots\} \mid e.e \mid e + e \mid e - e \mid e * e \mid -e \mid \{s;\ \ldots\} & \\
 & \mid & \texttt{if}(c)\ e\ \texttt{else}\ e \mid e\ \texttt{as}\ \tau \mid id{::}id(e, \ldots) & \\
c & ::= & b \mid e \neq e \mid e = e \mid e > e \mid e < e \mid \neg c \mid c \wedge c & \textit{(condition)}\\
 & \mid & c \vee c & \\
s & ::= & e \mid \texttt{skip} \mid \texttt{let}\ tid = e \mid \texttt{let}\ \texttt{mut}\ tid = e & \textit{(statement)}\\
 & \mid & e = e \mid \texttt{push\_output}(id, e, n, id) \mid \texttt{for}(id\ \texttt{in}\ n\ ..\ n)\ s & \\
f & ::= & \texttt{fn}\ id(tid, \ldots)\ s & \textit{(function)}
\end{array}$$

Figure 7: TAPIR intermediate representation syntax.

Variables can be created using `let` bindings, and later destructively modified using assignment (when created as mutable using `let mut`). There is a `for` loop, where we require that the loop bounds evaluate to constants at compile time (for compatibility with the bounded programming model provided by P4 and other hardware switches). There is also an `if` statement (standard `else if` syntax is also supported, but is omitted for conciseness). Note that the IR is an expression-based language, and "return values" are simply the last expression in a block. For example, the code

```
let a = {
  let x = 1; let y = 2; x + y
}
```

sets the value of `a` to $1 + 2 = 3$.

A packet is modeled as a record, e.g.,

$$\{\texttt{ip\_proto}{:}\texttt{int8}, \texttt{ip\_dst}{:}\texttt{int32}, \ldots, \texttt{data}{:}\ldots\}$$

where `ip_proto` etc. are the standard header fields (TCP/IP, etc.), and the `data` field(s) can hold a custom payload of custom type. These fields are stored in the packet, and are readable/writable at switches. Similarly, a switch is modeled as having a record type, and in this case, the fields represent stateful registers on the switch, which can be read/written when packets arrive. Custom packet headers can also contain bounded *stack* data structures, which provide `push` and `pop` operations, but we elide discussion of this, and instead use only arrays for clarity of the presentation.

Typechecking IR Programs. Before attempting to translate an IR program to P4, we perform type checking—this is useful for preventing tricky bugs due to unexpected bit-width conversions, etc. The IR is strongly-typed, and has boolean, fixed-width integer, tuple, fixed-length array, record, and function types. Although not shown in the syntax, the programmer can specify a custom type for the `data` field of packets, which we refer to as τ_{pk}, as well as a custom type for switches, τ_{sw}. Values can be affixed with a type annotation, such as `let x = 123 as int48`. The integer types can be signed or unsigned, and the width can be an *expression*, as long as it evaluates to a constant integer at compile time, e.g., `let n = 63; let y = 124 as int(n+1)`, which gives `y` type `int64`.

The typechecking functionality first performs *constant propagation*, eliminating constant `let` bindings, and replacing the name with the corresponding value. A simple bottom-up typechecking algorithm is then employed to confirm that all type annotations are correct. By default, integer constants are taken to be signed 64-bit integers. Explicit type conversions can be performed between integer types, e.g., `let x = 123 as int32; let b = x as int64`. The compiler makes sure that the proper code is emitted to handle this conversion cleanly (sign extension, etc.). Type annotations are required on function parameters, but the function's return type can be *inferred*.

Global Variables. We define a *global variable* as a Conflict-free Replicated Data Type (CRDT) register. We allow global variables to be read/written via the $id{::}id(e, \ldots)$ form of expression in the Figure 7 IR grammar. This can be used for CRDT operations such as `reg::write(...)`, `reg::read(...)`, etc., where `reg` has been declared as a CRDT register.

Callbacks. We define a *callback* to be an IR function which takes the event-triggering packet and location as parameters, and performs an action(s) such as reading/writing to global variables. Specifically, a callback is an IR function with the following type:

$$(\tau_{pk}, \tau_{sw}, \texttt{int}_{32}, \texttt{int}_{9}, \texttt{int}_{8}, \texttt{bool}) \to \tau_{pk},$$

where the parameters are (1) a record representing the input packet, (2) a record representing the switch where the packet arrived, (3) the ID of the switch, (4) the ID of the input port, (5) the clone ID (which will be discussed later), and (6) a boolean flag indicating whether the packet has arrived directly from (or should be forwarded directly to) an end host. This function can read/write to the current switch, and return a modified version of the input packet.

Callbacks will be used in CSMs to perform updates to global state, but they are also used to specify the forwarding behavior of switches. Each switch has an ingress callback and an egress callback that process each packet after it arrives at an input port, and before it is emitted from an output port respectively. Ingress/egress callbacks may read from local/global state, but may only write to *local* state—global writes must be handled via a CSM, to ensure correct and efficient management of global state.

Mechanisms for Multicast. The function type for callbacks allows a *single* output packet, but there are some network applications that require *multicast*, i.e., multiple output packets being produced. This is enabled in our IR by multiple calls to

$$\texttt{push\_output}(\texttt{pk}, \texttt{port\_id}, \texttt{unique\_flag}, \texttt{ca})$$

in the ingress callback, each of which indicates that a copy of the input packet should be emitted to the output port `port_id`. The `unique_flag` parameter is passed through as the `clone_id` parameter to the egress callback `ca` when it is called on the packet, allowing multiple copies of the same packet to be distinguished, if needed. We configure P4's queueing mechanism to send each copy of the packet to the indicated output port, where it is processed by the indicated egress callback and transmitted from the switch.

$$\begin{array}{lcll}
l & ::= & (n, n) & \textit{(location)} \\
e & ::= & (c, l) & \textit{(event)} \\
m & ::= & (e, ca) \mid m \cdot m \mid m + m \mid m \| m \mid m^{*} & \textit{(CSM)}
\end{array}$$

Figure 8: CSM language syntax.

Initialization and Topology Specification. The IR programmer can define a packet initialization function

$$\texttt{init\_packet}(\texttt{pk}, \texttt{sw}, \texttt{sw\_id}, \texttt{input\_port}),$$

which is called on each packet entering the network from a host—this allows header fields to be set to default values. There is also an initialization function for switches $\texttt{init\_switch}(\texttt{sw}, \texttt{sw\_id})$, which is called once per switch, and is used to initialize switch registers to desired values. Unless initialized, all custom header fields and all registers are zeroed.

Finally, for the purposes of experimentation and rapid prototyping, the programmer can also specify the desired network topology (not shown in the IR). Our prototype implementation generates a custom Mininet harness which implements it (details in Section 5).

3.2 Callback State Machines

Leveraging our IR, we develop a high-level declarative language for stateful data-plane programming, and show how its behavior can be described in terms of Petri nets. In Section 2, we described CSMs at a high level, and now we formalize the definitions.

Callback State Machines. The syntax of the language describing CSMs is shown in Figure 8. We define a *location* to be a switch-port pair (sw, pt), and an *event* to be a pair (c, l), where l is a location and c is a condition (see Figure 7). In practice, the condition can refer to packet headers, registers on the switch, etc., but for conciseness, we will require $c = \texttt{true}$, so that we can identify an event simply by its location (signifying arrival of some packet at that location). Finally, we define a *CSM* to be either an event paired with a callback, or composite CSMs built with the shown composition operators. We next formalize the definition of Petri nets, and define the semantics of CSMs using 1-safe Petri nets, where each transition is associated with an event and a callback.

Petri Nets. We define a Petri net $N = (T, P, D, M_0)$ to be a tuple where T is a set of transitions, P is a set of places, and $D \subseteq (T \times P) \cup (P \times T)$ is a set of directed edges connecting transitions to places or vice-versa. Input places of a transition are defined as $ins(t) = \{p \mid (p, t) \in D\}$, and output places are $outs(t) = \{p \mid (t, p) \in D\}$. A marking $M : T \to \mathbb{N}$ is a map that assigns a number of tokens to each place (M_0 is the initial marking). A transition $t \in T$ is enabled by a marking if $M(p) > 0$ for all $p \in ins(t)$. A marking M can update to a new marking M' by firing an enabled transition, denoted $M \xrightarrow{t} M'$, and the updated marking is

$$M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in ins(t) - outs(t) \\ M(p) + 1 & \text{if } p \in outs(t) - ins(t) \\ M(p) & \text{otherwise.} \end{cases}$$

A trace is a sequence of markings $M_0 M_1 \cdots M_n$ such that for all $0 \le j < n$, there exists some $t_j \in T$ such that $M_j \xrightarrow{t_j} M_{j+1}$. A Petri

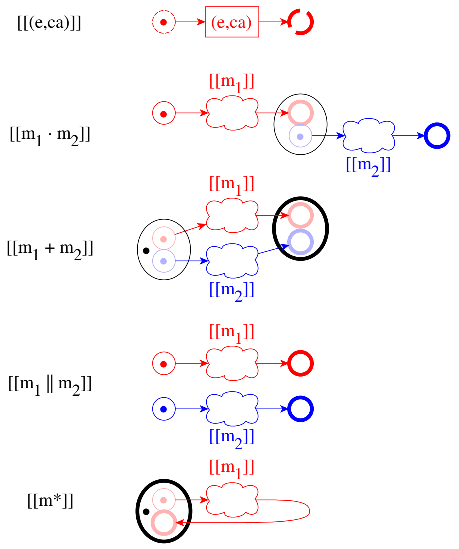

Figure 9: CSM semantics, using 1-safe Petri nets.

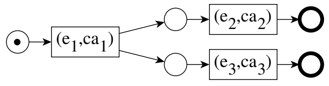

Figure 10: Semantics for $[\![(e_1, ca_1)\cdot((e_2, ca_2)||(e_3, ca_3))]\!]$.

net is 1-safe if for all traces, every marking M within the trace has $M(p) \leq 1$ for all p.

CSM Semantics. CSM semantics is defined graphically in Figure 8. The output of the function $[\![\cdot]\!]$ is a 1-safe Petri net, where each transition is labeled with an event and a callback, and one or more places is marked as "final" (denoted by a thick border). Note that the composition operators ensure 1-safety of the resulting Petri nets. The function $[\![\cdot]\!]$ is defined recursively, using an operation that merges pairs of places. This merge operation is shown visually using the large black places–red/blue places inside these are deleted by the merge operation. The input/output places for the (e, ca) form of transition are dashed to indicate that *one or more* input/output places will be generated as needed, with respect to the sequencing operator. For example, in Figure 10, *two* output places have been generated for the (e_1, ca_1) transition, to match up with the two initial places generated for the parallel composition $(e_2, ca_2)||(e_3, ca_3)$.

When producing a Petri net from a CSM, we perform two checks: (1) that the locality condition (Section 2) has been satisfied, and (2) that each merge operation not involving dashed places involves *exactly two* places. The latter disallows the programmer from writing $(A||B)\cdot(C||D)$, requiring instead $(A||B)\cdot(e, ca)\cdot(C||D)$ where (e, ca) serves to generate a *barrier* ensuring that *both* A and B have fully executed before continuing. While we could automatically insert "dummy" transitions in the Petri net to produce the same effect, we would then need to choose proper locations for each of these, necessitating a constraint solver.

4 CSM COMPILER

In this section, we detail the steps involved in transforming a CSM into executable code that can be run on modern switches. Note that although we target P4 switches, our techniques also apply to other platforms, and other compiler backends could be added easily.

Compilation Stages. The compiler performs the following transformations, which are described in the subsequent sections:

- *L3:* CSM (callbacks that read/write global state)
  ↓
- *L2:* (per-switch) IR programs with global variables
  ↓
- *L1:* IR programs that read/write only local state
  ↓
- P4 programs that read/write only local state

4.1 Compilation: L3 → L2.

The first stage of the compilation translates CSMs into IR programs with global registers. The Petri net obtained from the CSM can be encoded along with the topology declaration:

```
let topology = [
  // ... declare topology ...
  // declare CSM:
  event([1,2,3], S1:2, [4,5], callback, 123),
  marking([1,2,3])
]
```

In this example, the places $1, 2, 3$ are initially marked, and a packet arrival at location $(S_1, 2)$ allows the transition to fire, moving the tokens to places $4, 5$, and calling `callback` with the `clone_id` parameter set to 123. This unique ID can be used to distinguish multiple events using the same callback.

We translate the Petri net into global registers as follows. For each place, we generate a single-bit global register. We generate a custom `init_switch` function to initialize these globals to match the specified initial marking. At the beginning of each ingress callback, we read the globals and check the current marking. We then insert a sequence of `if` statements to check whether the current marking and current switch and port match a transition in the Petri net. Within the body of each of these, we first insert the statements of the corresponding `callback`, and then insert code to update the globals to match the new marking.

4.2 Compilation: L2 → L1.

The second compilation stage translates IR programs with global registers into IR programs with only local registers accesses. Global registers are declared along with the topology declaration. Currently, we support two types of CRDT global registers: *Increment* (unsigned) and *Last-writer wins (LWW)* (signed). Bit-width of the registers can be specified.

```
let topology = [
  // ... declare topology ...
  // declare globals:
  global("counter", 32, "inc"),
  global("test", 64, "lww")
]
```

These global registers are accessed in the following way:

```
// read the counter
let x = counter::read(swt, swt_id);
// increment the counter by 2
```

```
counter::inc(swt, swt_id, 2 uint32);

// read the LWW register
let y = test::read(swt, swt_id);
// write 123 to the LWW register
test::write(swt, swt_id, 123 int64)
```

In general, CRDT registers rely on causal ordering, so for this, we use Lamport timestamps [41]. We add a new custom header field `time` to packets, and a new register `time` to each switch. For each global, we store a data structure in the packet header fields, and in registers on each switch. The type of this data structure differs for each type of CRDT register. For example, an increment register is stored as an array of (per-switch) counters, and an LWW register is stored as a register value along with a timestamp [66].

At the beginning of each ingress callback, we insert the following code, where *merge* is IR code for the state-based merge for that CRDT type, and *max* computes the maximum:

```
// update the local timestamp
swt.time = max(swt.time, pkt.time) + 1;
// update local copy of globals
swt.count = merge(swt.count, pkt.count);
swt.test = merge(swt.test, pkt.test)
```

At the end of each egress callback, we insert the following:

```
// send out local timestamp
swt.time = swt.time + 1;
pkt.time = swt.time;
// send out local copy of globals
pkt.count = merge(swt.count, pkt.count);
pkt.test = merge(swt.test, pkt.test)
```

At this point, we have IR code with only local reads/writes, so we can now proceed to emitting P4 code.

4.3 Compilation: L1 → P4.

The final stage of the compiler produces P4 code from an IR program. We support P4_14, to provide backward-compatibility with older P4 installations. P4_14 does not have the expression-level `if` construct, `let` bindings, or data structures like arrays and tuples. Thus, we first perform several transformations on the code to simplify it before P4 code generation.

First, we flatten all expression-level blocks into statement-level blocks. This is shown in Figure 11. In particular, we first eliminate blocks appearing in a `let` binding, by pulling out all statements, and then `let`-binding the final expression (Figure 11(a)). We then eliminate `if` expressions appearing in a `let` binding, by introducing a temporary mutable variable, and assigning the final expression in each branch to this variable (Figure 11(b)). Finally, we pull reads/writes of *switch* fields out of expressions, so that they appear at the statement level (Figure 11(c)). This is because P4 only has statement-level read/write functionality for registers. After applying these transformations, the control-flow in the resulting IR code is implementable using P4's `if` blocks, and statements like `register_write` etc.

In order to translate IR data structures into flat integer types which can be handled by P4, we need to perform the transformations shown in Figure 12. For example, we recursively flatten arrays into lists of flat integers (Figure 12(a)). Similarly, we (recursively) flatten records in a similar way (Figure 12(b)), and tuples follow a similar pattern.

```
      let a = {              let x = 1;
        let x = 1;           let y = 2;
(a)     let y = 2;           let a = x + y
        x + y
      }
-------------------------------------------------
      let b = if(a > 1) {    let umut t = 0;
          123                if(a > 1) {
(b)     } else {               t = 123
          124                } else {
        }                      t = 124
                             }
-------------------------------------------------
      let c =                let t1 = swt.one;
        swt.one + swt.two;   let t2 = swt.two;
(c)   let d =                let c =
        pkt.one + pkt.two      t1 + t2;
                             let d =
                               pkt.one + pkt.two
```

Figure 11: Step 1: Flattening IR statements

```
      let a =               let a_0_0 = 1;
        [[1,2],             let a_0_1 = 2;
(a)      [3,4],             let a_1_0 = 3;
         [5,6]]             let a_1_1 = 4;
                            let a_2_0 = 5;
                            let a_2_1 = 6
--------------------------------------------
      let b =               let b_foo = 123;
(b)     {foo:123,           let b_bar = true;
         bar:true,          let b_baz_0 = 1;
         baz:[1,2]}         let b_baz_1 = 2
```

Figure 12: Step 2: Flattening IR assignments/datatypes

```
pkt.one = 1;          pkt.one = 1;
let mut two = 2;      pkt.meta.two = 2;
pkt.one =             pkt.one =
  pkt.one + two;        pkt.one +
                        pkt.meta.two
```

Figure 13: Step 3: Flattening IR variables

The last step of the translation before emitting P4 code involves eliminating `let` bindings, as shown in Figure 13. Since P4_14 does not have support for such a construct, we translate all `let` bindings into packet metadata-field writes. These metadata fields essentially function as "temporary variables" stored in the packet during processing on the switch. They are separate from the packet's custom header fields, and are not transmitted with the packet.

After the Figure 11-13 transformations are performed, the body of each callback function only contains `if` statements, and straight-line code containing only reads/writes to packet or switch fields. Each field is of a flat integer type. This maps readily into P4 code: the `if` blocks can be emitted directly, and each executable statement (field read/write) can be emitted as a call to `apply(table)`, where `table` is an empty P4 table whose default action is the executable statement. For example, writes to switch fields such as `swt.field=e` become `register_write(field,0,e)`, reads from switch fields such as `pkt.meta.field=swt.field` become `register_read(routing_metadata.field,field,0)`, and accesses to custom packet fields such as `pkt.one` become `data.one`,

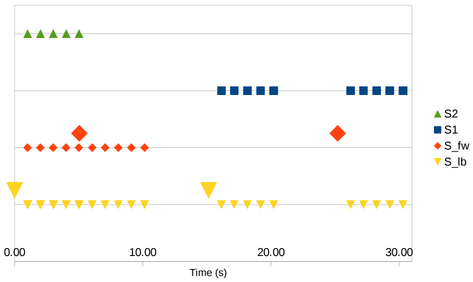

Figure 14: Property violation: malicious packets reach server (Figure 4 example).

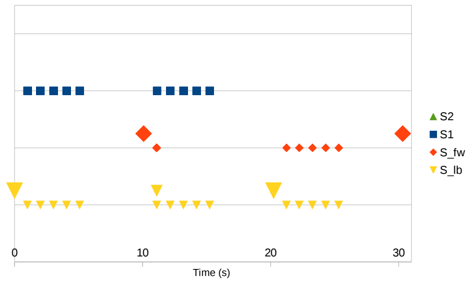

Figure 15: Correct operation: malicious packets blocked (Figure 5-6 examples).

where `data` is a custom P4 header holding all the (now flat integer) custom fields. The IR `push_output` function is implemented by conceptually "pushing" the desired output port, egress callback, and unique ID to a bounded "stack" in packet metadata fields.

A P4 parser is built for the `data` header. The `ingress` block of the P4 program (entrypoint which processes packets from the ingress queue) first checks whether an incoming packet contains this custom header—for simplicity, we indicate this with a special flag in the PCP bits of a VLAN header. If the packet is not flagged, we add the custom header (and the VLAN header, if necessary). We apply tables which set the `sw_id`, `input_port`, and `is_edge` callback parameters, and then emit the P4 code for the callback's body as described above.

Handling `push_output`. We use the *multicast groups* supported by many P4 switches to ensure that the packet is sent to each of the ports contained in the output-port stack. At the end of the ingress block, we apply tables which match on the contents of the output-port stack, and set the `intrinsic_metadata.mcast_grp` field accordingly. We map a unique multicast group ID to each potential combination of port IDs in the stack. The number of multicast groups is kept low by limiting the size of the stack (multicasting many packets is not common in our applications).

The `egress` block of the P4 program processes each packet after the ingress pipeline has moved it to a specific output port queue (as dictated by the multicast group). We have set up our multicast group assignments such that the multicast mechanism sets `intrinsic_metadata.egress_rid` to correspond to the index of this packet in the output-port stack. Thus, we apply tables which set metadata fields `clone_id` and `callback` to the unique ID and specified egress callback used in the `push_output` call. We then emit an `if` block for each possible egress callback in the IR program, and match on the `callback` ID to determine which one should handle the packet. Finally, the egress queue ends with tables which strip the VLAN and custom `data` header when the packet is transmitted directly to a host.

5 IMPLEMENTATION & EVALUATION

We have fully implemented the system described in Sections 3-4. The compiler consists of 3500+ lines of OCaml code, and the utilities for setting up and running experiments consist of about 1200+ lines of Java code. We ran all experiments on an Ubuntu Linux machine with 20GB RAM and a 3.2 GHz 4-core Intel i5-4570 CPU.

The main research questions we ask in this section are

(1) Do applications built using our approach function correctly and efficiently?
(2) Is it straightforward to use our approach to write real-world dataplane applications?

To answer Question #1, we implemented the CSM examples shown in Figures 4-6, on the topology shown in Figure 1. Our results are shown in Figures 15-14. In these graphs, each point represents a packet received at a given switch, at the time indicated by the point's position relative to the horizontal axis. Larger points correspond to state-change events. After each state change event, we attempted to send 5 malicious packets from client H_1 to server H_2. In Figure 14, the S_{lb} state change event at time 0 causes the load balancer to send packets to the firewall, allowing malicious packets to be received at S_2 until the firewall state change event occurs, at time 5 (policy violation). In Figure 15, the S_{lb} state change event at time 0 is ignored, since the corresponding CSM requires that the *firewall* state must change first. The firewall is enabled at time 10, and at time 20, a second S_{lb} state change event is received, and now the CSM allows the load balancer to be enabled. Note that at no time in Figure 15 does a malicious packet reach S_2.

We answered Question #2 by using our IR to build a working implementation of the CONGA adaptive congestion control system [1]. While that paper's approach required developing a custom ASIC, our approach only required the developer to write a small amount of code in a high-level imperative language.

Network Setup. The topology used throughout this experiments is shown in Figure 16. It corresponds to a simplified version of a *leaf-spine architecture*, where switches $S1, S2, S3$ are the spine, and $S4, S5, S6$ are leaves. The key thing to notice about this type of topology is that there are several paths between each group of hosts, allowing for traffic to circumvent congested paths.

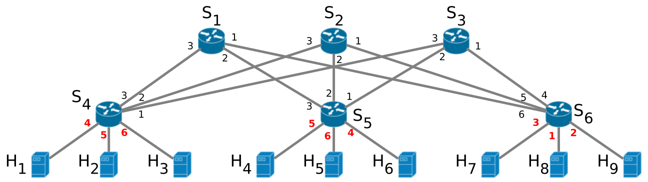

Figure 16: CONGA topology.

We used the Mininet network emulator [15] to set up the network and run experiments using a software switch that implements P4's *behavioral model*. These are at a severe performance disadvantage to hardware switches such as Barefoot Tofino, which can execute code at line rate, but this setup allows us to easily experiment with different topologies etc. We modified the simple-switch Mininet interface to enable the simulation to automatically install forwarding tables and initialize registers on the switches. Our compiler emits a special Mininet startup script which builds a topology to match the one specified in the IR program, and then brings up the simple-switches, and sets their initial forwarding tables and register contents, as required by the program.

CONGA IR Implementation. We implemented CONGA using our IR language. In this section we will walk step by step through the CONGA approach, and simultaneously see how this can be written using our language.

First, we start out with the topology shown in Figure 16. CONGA requires this type of leaf-spine topology—the basic idea is that the leaf switches choose which outgoing port to send each flow, based on congestion experienced previously by packets along those uplinks. Appendix A.1 contains the IR code encoding the topology.

We now specify the data structures used in our approach. CONGA requires packets to be tagged with several special fields. Specifically, `lbtag` tracks which port a leaf switch used to forward the packet, and `ce` keeps track of the maximum congestion the packet experiences in the network. This congestion information is *lazily* propagated back to the sender, by piggybacking it on data packets, using the corresponding `fb_lbtag` and `fb_metric` tags.

```
struct Packet {
    lbtag:uint9,
    ce:uint32,
    fb_lbtag:uint9,
    fb_metric:uint32
}
```

Leaf switches need to maintain several registers. The `flowlets` register contains one entry for each host (in this case, there are 9). This is used to keep track of flows being sent to the various destination hosts. Periodically, the *age* bit is checked, and if set, the flow expires (arriving packets reset the age bit). If the *valid* bit is set, then the *port* records where packets in the flow are currently being forwarded. The `to_table` register records a per-port congestion metric for each switch, representing the extent of congestion on the various uplinks. The `from_table` register records a per-port congestion metric to send *back* to each switch—the "pointer" bit records which of the metrics should be send next, when piggybacking on a data packet. All of these data structures need to be initialized properly on every switch, which is specified by defining an `init_switch` function (IR code shown in Appendix A.2).

```
struct Switch {
    // (port,valid,age)
    flowlets:[(uint9,bool,bool);9],
    to_table:[[uint32;7];9],
    // ("pointer",metric)
    from_table:[(uint9,[uint32;7]);9]
}
```

As discussed in Section 3, the ingress callback is used to process packets arriving at the switch, and the `is_edge` parameter is used to determine whether the packet is arriving directly from a host. In this case, this is useful for determining whether the switch should behave as a leaf or spine. The IR code for the ingress callback is shown in Appendix A.3.

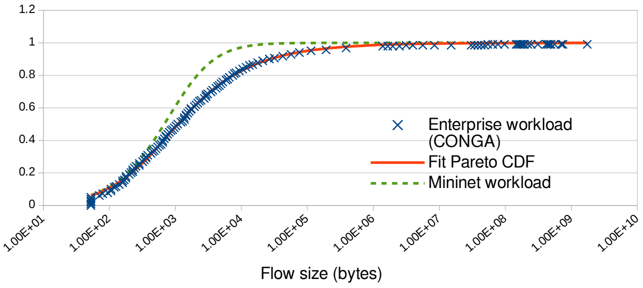

Figure 17: CDF of flow sizes used in the experiments.

As discussed in Section 3, the egress callback is used to process each packet after it is processed by the ingress callback. The ingress callback determines which port the packet should be forwarded to (using the `push_output` function), and the egress callback does any final processing before the packet is emitted from that port. In this case, we check to see if the packet is exiting to a host (i.e., at a leaf switch), in which case we store the collected congestion information. The egress callback IR code is shown in Appendix A.4.

Example Workload. To simulate a realistic workload, we first mapped the "enterprise workload" cumulative distribution function (CDF) from the CONGA paper. We were able to fit this to a Pareto distribution (Figure 17), but we found that the large flows (> 10MB) take a very long time to complete in Mininet with the simple-switches. Thus, we scaled the distribution as shown by the dashed line in the figure.

We built a simple HTTP server which accepts and serves requests for files of a certain size. We also build an HTTP client which connects to the server and sends requests at a fixed rate. The file sizes requested by the client are sampled from the CDF shown in Figure 17. For the experiment shown in Figure 18, we started a server on every host in the Figure 16 network simulation. We also started a client on each host, each of which made a connection to all servers except the ones which share the same leaf switch.

In Figure 18, we increased the rate at which the clients are making requests, and compared the average flow-completion time (FCT) of our CONGA implementation versus our ECMP implementation. As seen in the CONGA paper, CONGA exhibits similar FCT to that of ECMP on the enterprise workload—our graph roughly matches Figure 9(a) in the CONGA paper.

6 DISCUSSION AND FUTURE WORK

While we believe that the CSM approach is an important step toward simplifying correct and efficient data-plane programming, there are several limitations we hope to overcome in future work.

- Our backend uses only default entries in P4 tables. This may cause inefficiency, by producing many small tables instead of a single table with multiple match-action rules. Additionally, it makes functionality like least-prefix matching more difficult to encode using the IR. We hope to address these

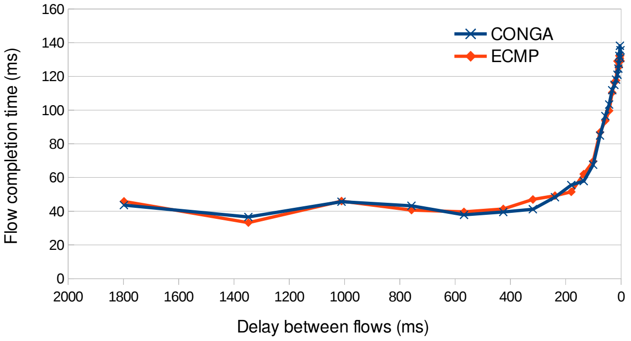

Figure 18: Performance of IR CONGA implementation.

issues via static analysis to automatically combine tables when possible, and extending the IR with more powerful pattern matching constructs.

- Our implementation of multicast must encode each possible set of output ports, which can cause inefficiency when using a large number of ports. Although we have not encountered applications which necessitate extensive multicasting, we hope to develop a more efficient multicasting mechanism.
- We are currently working to develop an *algebra* of CSMs, to enable equational reasoning, e.g., $A{\cdot}(B{+}C) = (A{\cdot}B){+}(A{\cdot}C)$. This is closely related to concurrent Kleene algebra [9].

7 RELATED WORK

Correctness of Network Programs. Different notions of consistency at the packet-level have been shown to be useful for reasoning about correctness of network behavior, and there are many approaches for achieving/ensuring this [25, 30, 36–38, 43, 48, 50, 62, 80]. However, packet-level consistency is not always sufficient [21]. Furthermore, in the stateful context, things become even more complicated. We need to be able to describe and ensure correctness in an environment where *events* initiate changes in the network [4, 17, 18, 24, 40, 49, 51, 56, 65, 74, 76, 77]. We also need to be able to *compose* event-driven programs correctly [10, 61].

Consensus Routing [32] was designed to address the tension between consistency and availability in networks. This is also a challenge in datacenters, since it is important that the network have both high throughput and high availability. One way to achieve this is via *stateless* functionality [33]. Another approach is to allow distributed state, but use relaxed consistency models [59], which is the direction we take.

There has also been work on correctness at the hardware/switch level, e.g., Packet Transactions in P4 [67]. P4 has been used to provide strong consistency [14]. Hyper4 enables correct composition of multiple P4 programs [23].

Data-Plane Programming. Over the last few years, there has been a trend towards pushing functionality which normally occurred in the control-plane into the data-plane [64]. As we mentioned briefly in the Introduction, numerous examples of these *data-plane programs* can be found across several application areas:

- **Adaptive Congestion control**—the network automatically adjusts forwarding decisions, based on levels of congestion, in order to adapt readily to changing traffic patterns. These include CONGA [1], HULA [34], DCTCP [2], Fastpass [60], ExpressPass [13], Hermes [79], DRILL [22], and others.
- **Traffic management**—these optimize the performance of the network along various quality metrics, such as consistency [11], connectivity [44], deadlock prevention [27], robustness [72], I/O performance [71], etc.
- **Monitoring**—these perform measurements of properties in the network, beyond what can be done with a simple host-based monitor, and include Felix [12], Pivot Tracing [47], OpenSketch [75], In-band Network Telemetry [39], Hardware-software co-design [54], Heavy hitter detection [68], SwitchPointer [70], Path Queries [55], and NetQRE [78].
- **Active networking**—packets can carry small "programs" which are executed by forwarding devices as the packet moves through the network. Millions of Little Minions [28] is a recent example, which is able to implement various data-plane applications, such as load balancing, etc.
- **Network OS/Runtimes**—these provide virtualization functionality built on top of networking devices, such as Participatory Networking [19], E2 [57], etc.

In contrast to the huge variety of ways these individual data-plane applications were developed, our work provides a general approach for building such applications, and ensuring that they operate correctly.

Concurrent Programming for Networks. Dudycz et al. [16] present an algorithm to *compose* network updates correctly with respect to loop freedom, and show that the problem of *optimally* doing so is NP-hard. Li et al. [42] present an algebra-based approach for reasoning about composition of updates. Beyond network updates, there has been work on composing network programs. For example, Pyretic has a programming language which allows sequential/parallel composition of static policies—dynamic behavior can be obtained via a sequence of policies [52]. NetKAT is a mathematical formalism and compiler which also allows composition of static policies [3, 69]. CoVisor is a hypervisor that allows multiple controllers to run concurrently (sequential or parallel composition). It can incrementally update the configuration based on intercepted messages from controllers, and does not need to recompile the full composed policy [29]. The PGA system addresses the issue of how to handle distributed conflicts, via customizable constraints between different portions of the policies, allowing them to be composed correctly [61]. Bonatti et al. [7] present an algebra for properly composing access-control policies. Canini et al. [10] use an approach based on software transactional networking to handle conflicts. We deal with conflicts through the *locality* condition of CSMs.

Handling persistent *state* properly in network programming is a difficult problem. Although basic support is provided by switch-level mechanisms for stateful behavior [6, 8, 67], global coordination still needs to be handled carefully at the language/compiler level. FAST [53], OpenState [6], and Kinetic [40] provide a finite-state-machine-based approach to stateful network programming. Gao et al. [20] present Trident, an approach for integrating network functions (NF) and SDN. Arashloo et al. [4] present SNAP, a high-level language for writing network programs. SNAP has a language with support for sequential/parallel composition of stateful policies, as well as built-in features beyond what we provide (such as atomic blocks). However, none of these approaches examine how to avoid/handle (or analyze) distributed conflicts. McClurg et al. [51] present an approach which formalizes event-driven network programs using event structures, and show how to deal with distributed conflicts. Our approach is conceptually similar, but provides a more flexible model, while retaining the ability to utilize consistency properties presented there.

SNAP [4] is the language/compiler most closely-related to our work. It combines NetKAT's ability to describe static configurations with the ability to read/write persistent state that can influence the processing of future packets. There are several key differences between our approach and SNAP:

(1) *Network model* — SNAP uses a "one big switch" model, meaning the program simply moves packets between end hosts. This model is essential for their approach, and it is not possible to access the "internals" of the network. Our model, on the other hand, allows a high-level program to be written such that individual devices within the network can be used (e.g., packets can be made to travel specific paths, which is important in applications like congestion control).
(2) *Execution model* — A SNAP program essentially runs in a "loop", i.e., the packet-processing function is applied each time a packet is sent into the network from an end host. Our approach is callback-based, i.e., certain *events* (e.g., arrival of certain packets at certain ports) cause corresponding blocks of code to be executed.
(3) *State placement / Conflict resolution* — Since this is the key difference, we discuss it in Section 2.
(4) *Composability* — Our tool makes it easy to compose programs. At the level of SNAP packet-processing functions, it is not clear what the composition of two programs looks like (this only becomes clear at the level of the compiled program representation, *xFDDs*).

Network Repair and Network Update Synthesis. While this paper focuses on correct-by-construction programming, there are other approaches for ensuring correctness of network programs. Saha et al. [63] and Hojjat et al. [24] present approaches for repairing a buggy network configuration using SMT and a Horn-clause-based synthesis algorithm respectively. Instead of repairing a static configuration, McClurg et al. [49] repair a network *program.* That work presents a new language called *event nets*, and a synchronization synthesis framework that helps users properly compose several processes into a single correct network program.

A *network update* is a simple network program—a situation where the global forwarding state must change. In the networking community, there are several proposals for packet- and flow-level consistency properties that should be preserved during an update. For example, per-packet and per-flow consistency [48, 62], and inter-flow consistency [45]. Many approaches solve the problem with respect to different variants of these consistency properties [25, 31, 35, 46, 50, 80]. In contrast, we provide a new language for succinctly describing how multiple updates can be composed, and this allows us to leverage approaches for synthesizing a composition which respects customizable properties over packet traces.

8 CONCLUSION

We proposed a network programming language CSM that allows operators to write data-plane programs against global state. This is enabled by a new intermediate representation TAPIR, and a compiler that produces efficient code for stateful switches. Our key insight is that we can allow programming against global state and allow programs with asynchronous control flow: CSM has a callback mechanism for specifying how networks react to events such as a congestion or an attempted intrusion.

In the future, we plan to explore both the theoretical and the practical directions enabled by this work, such as studying the formal properties of CSM, and extending it to help with non-SDN tasks such as bringing up servers on-demand.

ACKNOWLEDGMENTS

We are deeply grateful to Pavol Černý, Nate Foster, and Friedrich Weber for their advice and assistance on an early version of this work. We thank the anonymous reviewers for their constructive comments, and our shepherd David Walker, for providing further insight and direction. This work was funded by the National Science Foundation under awards NSF CCF 1849622 and 1952816.

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: distributed congestion-aware load balancing for datacenters. In *SIGCOMM.* ACM, 503–514.
[2] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM.* 63–74.
[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptist Jeannin, Jean-Baptiste, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. *POPL* (2014).
[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. *SIGCOMM* (2016).
[5] Eric Badouel, Benoıt Caillaud, and Philippe Darondeau. 2002. Distributing Finite Automata Through Petri Net Synthesis. *Formal Asp. Comput.* 13, 6 (2002), 447–470.
[6] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. 2014. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *ACM SIGCOMM CCR* (2014).
[7] Piero A. Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. 2000. A modular approach to composing access control policies. In *ACM Conference on Computer and Communications Security.* ACM, 164–173.
[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM CCR* (2014).
[9] Paul Brunet, Damien Pous, and Georg Struth. 2017. On Decidability of Concurrent Kleene Algebra. In *CONCUR (LIPIcs, Vol. 85).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:15.
[10] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. 2013. Software transactional networking: concurrent and consistent policy composition. In *HotSDN.* ACM, 1–6.
[11] Carmelo Cascone, Luca Pollini, Davide Sanvito, and Antonio Capone. 2015. Traffic Management Applications for Stateful SDN Data Plane. In *EWSDN.* IEEE Computer Society, 85–90.
[12] Haoxian Chen, Nate Foster, Jake Silverman, Michael Whittaker, Brandon Zhang, and Rene Zhang. 2016. Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis. In *SOSR.* ACM, 14.
[13] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *SIGCOMM.* ACM, 239–252.
[14] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos Made Switch-y. *Computer Communication Review* 46, 2 (2016), 18–24.

[15] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and Ligia Rodrigues Prete. 2014. Using Mininet for emulation and prototyping Software-Defined Networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*. 1–6. https://doi.org/10.1109/ColComCon.2014.6860404

[16] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. 2016. Can't Touch This: Consistent Network Updates for Multiple Policies. In *DSN*. IEEE Computer Society, 133–143.

[17] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. 2016. SDNRacer: concurrency analysis for software-defined networks. In *PLDI*. ACM, 402–415.

[18] 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

[19] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory networking: an API for application control of SDNs. In *SIGCOMM*. ACM, 327–338.

[20] Kai Gao, Taishi Nojima, and Yang Richard Yang. 2018. Trident: toward a unified SDN programming framework with automatic updates. In *SIGCOMM*. ACM, 386–401.

[21] Soudeh Ghorbani and Brighten Godfrey. 2014. Towards correct network virtualization. In *HotSDN*. ACM, 109–114.

[22] Soudeh Ghorbani, Zibin Yang, Philip Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *SIGCOMM*. ACM, 225–238.

[23] David Hancock and Jacobus E. van der Merwe. 2016. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *CoNEXT*. ACM, 35–49.

[24] Hossein Hojjat, Philipp Ruemmer, Jedidiah McClurg, Pavol Cerny, and Nate Foster. 2016. Optimizing Horn Solvers for Network Repair. *FMCAD* (2016).

[25] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-driven WAN. *HotNets* (2013).

[26] Richard P. Hopkins. 1990. Distributable nets. In *Applications and Theory of Petri Nets (Lecture Notes in Computer Science, Vol. 524)*. Springer, 161–187.

[27] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2016. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *HotNets*. ACM, 92–98.

[28] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of little minions: using packets for low latency network programming and visibility. In *SIGCOMM*. ACM, 3–14.

[29] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. *NSDI* (2015).

[30] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. 2014. Dynamic Scheduling of Network Updates. In *SIGCOMM*. 539–550.

[31] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic Scheduling of Network Updates. *SIGCOMM* (2014).

[32] J. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. 2008. Consensus Routing: The Internet as a Distributed System. *NSDI* (2008).

[33] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *NSDI*. USENIX Association, 97–112.

[34] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *SOSR*. ACM, 10.

[35] Naga Praveen Katta, Jennifer Rexford, and David Walker. 2013. Incremental Consistent Updates. In *HotSDN*. ACM, 49–54.

[36] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*. 99–112.

[37] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *NSDI*.

[38] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. 2012. VeriFlow: Verifying Network-wide Invariants in Real Time. *ACM SIGCOMM CCR* (2012).

[39] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band Network Telemetry via Programmable Dataplanes. In *SOSR (Demo paper)*.

[40] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable Dynamic Network Control. *NSDI* (2015).

[41] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[42] Geng Li, Yang Richard Yang, Franck Le, Yeon-Sup Lim, and Junqi Wang. 2019. Update Algebra: Toward Continuous, Non-Blocking Composition of Network Updates in SDN. In *INFOCOM*. IEEE, 1081–1089.

[43] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*. ACM, 411–422.

[44] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring Connectivity via Data Plane Mechanisms. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 113–126. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/liu_junda

[45] Weijie Liu, Rakesh B Bobba, Sibin Mohan, and Roy H Campbell. 2015. Inter-Flow Consistency: Novel SDN Update Abstraction for Supporting Inter-Flow Constraints. *NDSS* (2015).

[46] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. 2014. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies. In *HotNets*.

[47] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2016. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *USENIX Annual Technical Conference*. USENIX Association.

[48] Ratul Mahajan and Roger Wattenhofer. 2013. On Consistent Updates in Software Defined Networks. In *SIGCOMM*.

[49] Jedidiah McClurg, Hossein Hojjat, and Pavol Cerny. 2017. Synchronization Synthesis for Network Programs. *CAV* (2017).

[50] Jedidiah McClurg, Hossein Hojjat, Pavol Cerny, and Nate Foster. 2015. Efficient Synthesis of Network Updates. *PLDI* (2015).

[51] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerny. 2016. Event-driven Network Programming. *PLDI* (2016).

[52] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software Defined Networks. *NSDI* (2013).

[53] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. 2014. Flow-level State Transition as a New Switch Primitive for SDN. In *HotSDN*.

[54] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Mohammad Alizadeh, David Walker, Jennifer Rexford, Vimalkumar Jeyakumar, and Changhoon Kim. 2016. Hardware-Software Co-Design for Network Performance Measurement. In *HotNets*. ACM, 190–196.

[55] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In *NSDI*. USENIX Association, 207–222.

[56] Tim Nelson, Andrew D Ferguson, MJ Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks. *NSDI* (2014).

[57] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: a framework for NFV applications. In *SOSP*. ACM, 121–136.

[58] Aurojit Panda, Colin Scott, Ali Ghodsi, Teemu Koponen, and Scott Shenker. 2013. CAP for networks. In *HotSDN*. ACM, 91–96.

[59] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. 2017. SCL: Simplifying Distributed SDN Control Planes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 329–345. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-aurojit-scl

[60] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized Zero-queue Datacenter Network. In *SIGCOMM*.

[61] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *SIGCOMM*. ACM, 29–42.

[62] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. *SIGCOMM* (2012).

[63] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. 2015. NetGen: Synthesizing Data-plane Configurations for Network Policies. In *SOSR*.

[64] Liron Schiff, Michael Borokhovich, and Stefan Schmid. 2014. Reclaiming the Brain: Useful OpenFlow Functions in the Data Plane. In *HotNets*. ACM, 7:1–7:7.

[65] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, Hrishikesh B. Acharya, Kyriakos Zarifis, and Scott Shenker. 2014. Troubleshooting blackbox SDN control software with minimal causal sequences. In *SIGCOMM*. ACM, 395–406.

[66] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[67] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. *SIGCOMM* (2016).

[68] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data

Plane. In *SOSR*. ACM, 164–176.

[69] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. *ICFP* (2015).

[70] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed Network Monitoring and Debugging with SwitchPointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 453–456. https://www.usenix.org/conference/nsdi18/presentation/tammana

[71] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony I. T. Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: a software-defined storage architecture. In *SOSP*. ACM, 182–196.

[72] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. 2015. Central Control Over Distributed Routing. In *ACM SIGCOMM*.

[73] Glynn Winskel. 1987. *Event Structures*. Springer.

[74] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. Automated Bug Removal for Software-Defined Networks. In *NSDI*. USENIX Association, 719–733.

[75] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *NSDI*. USENIX Association, 29–42.

[76] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. 2014. NetEgg: Programming Network Policies by Examples. *HotNets* (2014).

[77] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. 2015. Scenario-based Programming for SDN Policies. *CoNEXT* (2015).

[78] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. In *SIGCOMM*. ACM, 99–112.

[79] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. 2017. Resilient Datacenter Load Balancing in the Wild. In *SIGCOMM*. ACM, 253–266.

[80] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. 2015. Enforcing Generalized Consistency Properties in Software-Defined Networks. *NSDI* (2015).

A CONGA–IR CODE

A.1 Network Topology

```
let topology = [
  switch(S3, ingress),
  switch(S2, ingress),
  switch(S1, ingress),
  switch(S6, ingress),
  switch(S5, ingress),
  switch(S4, ingress),
  host(H8, 00:00:00:00:00:08, 10.0.0.8, S6:1),
  host(H9, 00:00:00:00:00:09, 10.0.0.9, S6:2),
  host(H7, 00:00:00:00:00:07, 10.0.0.7, S6:3),
  host(H1, 00:00:00:00:00:01, 10.0.0.1, S4:4),
  host(H6, 00:00:00:00:00:06, 10.0.0.6, S5:4),
  host(H2, 00:00:00:00:00:02, 10.0.0.2, S4:5),
  host(H4, 00:00:00:00:00:04, 10.0.0.4, S5:5),
  host(H3, 00:00:00:00:00:03, 10.0.0.3, S4:6),
  host(H5, 00:00:00:00:00:05, 10.0.0.5, S5:6),
  link(S3:1, S6:4),
  link(S2:1, S6:5),
  link(S3:2, S5:1),
  link(S1:1, S6:6),
  link(S2:2, S5:2),
  link(S3:3, S4:1),
  link(S1:2, S5:3),
  link(S2:3, S4:2),
  link(S1:3, S4:3)
]
```

A.2 Switch Initialization

```
fn init_switch(mut swt:Switch, swt_id:uint(32)) {
  for(i in 0 .. 8) {
    // init each flowlet to (port=0, valid=false, age=true)
    swt.flowlets[i] = (0 uint9, false, true)
  };
  for(i in 0 .. 8) {
    // set each "pointer" to 1
    swt.from_table[i].0 = 1 uint9
  };
  // initialize all allowable shortest-paths to have cost 1
        in the to_table (0 represents infinity)
  if swt_id == 1 uint32 { // from S1...
    // to H1, via port 3
    swt.to_table[0][3] = 1 uint32;
    // to H2, via port 3
    swt.to_table[1][3] = 1 uint32;
    // to H3, via port 3
    swt.to_table[2][3] = 1 uint32;

    // to H4, via port 2
    swt.to_table[3][2] = 1 uint32;
    // to H5, via port 2
    swt.to_table[4][2] = 1 uint32;
    // to H6, via port 2
    swt.to_table[5][2] = 1 uint32;

    // to H8, via port 1
    swt.to_table[7][1] = 1 uint32;
    // to H9, via port 1
    swt.to_table[8][1] = 1 uint32;
    // to H7, via port 1
    swt.to_table[6][1] = 1 uint32
  }
  // ...
  else {}
}
```

A.3 Ingress Callback

```
fn ingress(mut pkt:Packet, mut swt:Switch,
      swt_id:uint(32), input_port:uint(9),
      clone_id:uint(8), is_edge:bool, queue_time:uint(32)) {
  // if this packet has entered the network from a host
  if is_edge {
    // grab the corresponding row of flowlet table,
          to_table, and from_table
    let mut temp =
    if pkt.ip_dst == 10.0.0.1 uint32 {
      (swt.flowlets[0], swt.to_table[0], swt.from_table[0])
    } else if pkt.ip_dst == 10.0.0.2 uint32 {
      (swt.flowlets[1], swt.to_table[1], swt.from_table[1])
    }
    // ...
    else {
      (swt.flowlets[8], swt.to_table[8], swt.from_table[8])
    };
    let mut flowlet = temp.0;
    let mut table = temp.1;
    let mut fr_table = temp.2;

    let mut port = flowlet.0;
    if(flowlet.1) {
      // if the flowlet is still active, use the flowlet's
            current port
    } else {
      // if the flowlet has expired, make a load balancing
            decision (find the minimum entry in this row of
            the to_table)
      let mut min = 4294967295 uint32;
      for(i in 0 uint9 .. 6 uint9) {
        if table[i] > 0 uint32 && table[i] < min {
          min = table[i];
          port = i
        } else {}
      };

      flowlet.0 = port;
      flowlet.1 = true
    };
    // mark the packet's tag, and assign it to
          corresponding port
    pkt.lbtag = port;
    pkt.ce = 1 uint32;
    push_output(pkt, port, 123 uint(8), egress);
    // reset the age bit
    flowlet.2 = false;
    // load a response metric
    for(i in 1 uint9 .. 6 uint9) {
      if fr_table.0 == i && fr_table.1[i] > 0 uint32 {
        pkt.fb_metric = fr_table.1[i];
        pkt.fb_lbtag = i
      } else {}
    };
    // make sure we increment the "pointer" so that the
          next metric will be considered later
    if fr_table.0 == 6 uint9 { fr_table.0 = 1 uint9 }
    else { fr_table.0 = fr_table.0 + 1 uint9 };

    // save any modifications to flowlet table and
          from_table
    if pkt.ip_dst == 10.0.0.1 uint32 {
      swt.flowlets[0] = flowlet; swt.from_table[0] =
            fr_table
    } else if pkt.ip_dst == 10.0.0.2 uint32 {
      swt.flowlets[1] = flowlet; swt.from_table[1] =
            fr_table
    }
    // ...
  else {
   swt.flowlets[8] = flowlet; swt.from_table[8] =
      fr_table
  }
 } else {
  // if the packet has not just entered from a host...
  // first, load any incoming fb_lbtag and fb_metric into
     to_table
  if pkt.fb_lbtag > 0 uint9 {
   let mut temp =
   if pkt.ip_src == 10.0.0.1 uint32 { swt.to_table[0] }
   else if pkt.ip_src == 10.0.0.2 uint32 {
      swt.to_table[1] }
   // ...
   else { swt.to_table[8] };
   for(j in 0 uint9 .. 6 uint9) {
    if(pkt.fb_lbtag == j) {
     temp[j] = pkt.fb_metric
    } else {}
   };
   if pkt.ip_src == 10.0.0.1 uint32 {
    swt.to_table[0] = temp
   } else if pkt.ip_src == 10.0.0.2 uint32 {
    swt.to_table[1] = temp
   }
   // ...
   else {
    swt.to_table[8] = temp
   }
  } else {};

  // do basic ECMP forwarding
  if swt_id == 1 uint32 && pkt.ip_dst == 10.0.0.4 uint32 {
   push_output(pkt, 2 uint(9), 123 uint(8), egress)
  } else if swt_id == 1 uint32 && pkt.ip_dst == 10.0.0.7
     uint32 {
   push_output(pkt, 1 uint(9), 123 uint(8), egress)
  }
  // ...
  else if swt_id == 2 uint32 && pkt.ip_dst == 10.0.0.4
     uint32 {
   push_output(pkt, 2 uint(9), 123 uint(8), egress)
  } else if swt_id == 2 uint32 && pkt.ip_dst == 10.0.0.7
     uint32 {
   push_output(pkt, 1 uint(9), 123 uint(8), egress)
  }
  // ...
  else if swt_id == 3 uint32 && pkt.ip_dst == 10.0.0.4
     uint32 {
   push_output(pkt, 2 uint(9), 123 uint(8), egress)
  } else if swt_id == 3 uint32 && pkt.ip_dst == 10.0.0.7
     uint32 {
   push_output(pkt, 1 uint(9), 123 uint(8), egress)
  }
  // ...
  else {}
 }
 pkt
}
```

A.4 Egress Callback

```
fn egress(mut pkt:Packet, mut swt:Switch, swt_id:uint(32),
   input_port:uint(9), clone_id:uint(8), is_edge:bool,
   queue_time:uint(32)) {
 // if the packet has reached its destination host
 if is_edge {
  // grab the corresponding entry in the from_table
  let mut temp =
  if pkt.ip_src == 10.0.0.1 uint32 {
   swt.from_table[0]
  } else if pkt.ip_src == 10.0.0.2 uint32 {
   swt.from_table[1]
  }
  // ...
  else {
   swt.from_table[8]
  };
  // save this packet's congestion metrics
  for(j in 0 uint9 .. 6 uint9) {
    if(pkt.lbtag == j) {
      temp.1[j] = pkt.ce;
      temp.0 = j
    } else {}
  };
  // save the edited entry to the from_table
  if pkt.ip_src == 10.0.0.1 uint32 {
   swt.from_table[0] = temp
  } else if pkt.ip_src == 10.0.0.2 uint32 {
   swt.from_table[1] = temp
  }
  // ...
  else {
   swt.from_table[8] = temp
  }
 } else {
  // if the packet hasn't reached its destination, update
     packet's congestion metric
  pkt.ce = pkt.ce + queue_time
 };
 pkt
}
```