Network Programming via Computable Products

Dennis Volpano

The Johns Hopkins University Applied Physics Lab

Laurel, MD, U.S.A.

Dennis.Volpano@jhuapl.edu

ABSTRACT

The User Plane Function (UPF) aims to provide network services in the 3GPP 5G core network. These services need to be implemented on demand inexpensively with provable properties. Existing network dataplane programming languages are not up to the task. A new software paradigm is presented for the UPF. It is inspired by model checking a concurrent reactive system where conceptually each component of the system is modeled as an extended finite-state machine and their product is verified. We show how such a product can be computed for one example of a UPF and how its state invariants can be inferred, thereby eliminating the need to formally verify the product separately. Code can be generated from the product and regenerated on the fly to remain optimal for the probability distribution of network traffic the UPF must process.

1 INTRODUCTION

The User Plane Function (UPF) in the 3GPP 5G core network [36] needs new techniques for building software that implements network functions at the edge quickly, reliably with provable guarantees, and inexpensively. Provisioning devices will likely be fully automatic and the software can be complex. Practical network functions are reactive systems responding to inputs based on history and time. They're not just packet-processing pipelines. They have control logic that manages timers, caches and mutable state.

Much work has been done in the design of high-level network programming languages [2, 4, 5, 7, 8, 11, 14, 20, 24–26, 29, 33, 39]. In general, they are either too narrow in scope or lack support for reuse and scalable proofs about mutable state and timers. Godefroid observed that model checking a concurrent reactive system conceptually amounts to modeling each component of the system as an extended finite-state machine and then verifying the product of all such machines [15]. This idea can be applied to the UPF, instances of which can be defined as the product of independent concurrent components represented by finite-state recognizers. A product can be transformed into branching logic and then implemented on a specific target platform. The approach is illustrated for a basic switch function implemented on an open target platform using Intel's Data Plane Development Kit (DPDK) [12].

2 A BASIC SWITCH FUNCTION

We give four independent concurrent components for a 4-port switch UPF. It has one uplink port, namely port 1, which is in a different broadcast domain than ports 2-4. The components are

(1) H – (hub) floods a frame to every port except the port at which it arrived and the uplink port.
(2) B – (bridge) forwards a frame to the port behind which the frame's destination MAC address was learned.
(3) M – learns the ports of MAC addresses.

(4) I – interleaves ingress and egress activity guaranteeing that every received frame is transmitted.

No component depends on another so all are independent and form reusable building blocks of a switch. Components are recognizers that run concurrently on a trace. For example, Table 1 shows a trace of our 4-port switch in the presence of the ARP protocol [13]. Each

Table 1: A trace of 4-port switch with uplink port 1

time	dest address (da)	src address (sa)	proto	location
t	ff:ff:ff:ff:ff:ff	04:0c:ce:d2:08:6c	arpreq	{2i}
t + 1	ff:ff:ff:ff:ff:ff	04:0c:ce:d2:08:6c	arpreq	{3e, 4e}
t + 2	04:0c:ce:d2:08:6c	7c:d1:c3:e8:a4:67	arpreply	{3i}
t + 3	04:0c:ce:d2:08:6c	7c:d1:c3:e8:a4:67	arpreply	{2e}

port is divided into an ingress and egress interface, denoted by i and e. At time t, an ARP request arrives at the ingress interface of port 2. Then at time $t + 1$, the request is at the egress interfaces of ports 3 and 4 as we would expect since port 1 is uplink and the frame is flooded to all ports except its ingress port. An ARP reply is received at time $t + 2$ at port 3 and fowarded to port 2 at time $t + 3$ because its destination address was learned there at time t. Elements of a trace are referenced within a recognizer by free variables t (current time), f (frame in the trace at time t), loc (location of f) and $port$ (the ingress port of f when f is located at an ingress interface).

2.1 Hub component $H(self)$

Hub component $H(self)$ is defined in Table 2 using a special type of recognizer called a λ-SFA. It is a type of deterministic symbolic finite automaton (SFA) [37, 38] with lambda bindings that allow it to more succinctly remember history.[1] $H(self)$ has three transitions

Table 2: $H(self)$ relays between non-uplink ports

H1 → H1
$loc = \{port\ i\} \Rightarrow (port = uplink\text{-}port \lor f.da = haddr(port))$
H1 → H2
$\lambda x.\ loc = \{port\ i\} \land port \neq uplink\text{-}port \land f.da \neq haddr(port)$
H2 → H1
$(self\ e \in loc \land ((bcast(x.f.da) \land \neg arp\text{-}reqrx(x.f, x.port)) \lor$
$ucast(x.f.da))) \Rightarrow (f = x.f \land self \neq x.port \land self \neq uplink\text{-}port)$

and two states H1 and H2 where H1 is the start state (the first transition listed is always from the start state). The proposition that labels a transition is shown below it. A transition from H1 to H2 occurs when a frame arrives at an ingress port other than the uplink

---

[1] λ is an input binding operator as in λ calculus, not a name for the null string as in finite automata.

port and its destination hardware address $f.da$ doesn't match the hardware address of the port, which indicates link-layer forwarding rather than handling traffic destined for the switch. Otherwise it stays in H1. Notice that if $H(self)$ were started in state H1 at time $t + 1$, then it stays in H1 because $loc = \{3e, 4e\}$ at that time, and thus $loc = \{port\ i\}$ is false then. This is a stutter step that allows the SFA to ignore actions in a trace that are not of interest to it in state H1, namely egress activity [22]. For the trace in Table 1, the bindings of the free variables of $H(self)$ are given in Table 3.

Table 3: Free variables of $H(self)$ bound by trace in Table 1

time	f.da	x.f.da	port	x.port
t	ff:ff:ff:ff:ff:ff	–	2	–
t + 1	ff:ff:ff:ff:ff:ff	ff:ff:ff:ff:ff:ff	–	2
t + 2	04:0c:ce:d2:08:6c	ff:ff:ff:ff:ff:ff	3	–
t + 3	04:0c:ce:d2:08:6c	04:0c:ce:d2:08:6c	–	3

With respect to our 4-port switch, $H(self)$ has four recognizer instances $H(1)-H(4)$, one for each port. Assuming that the ARP request in the trace is not a request for the hardware address of port 2 ($\neg arp\text{-}reqrx(x.f, 2)$ is true), each instance can make a transition on every entry in the trace, albeit for different reasons in some states. At time $t + 3$, for instance, all but $H(2)$ move from H2 to H1 by vacuously satisfying its condition since only 2e is a member of loc. But $H(2)$ must satisfy its consequent ($f = x.f \wedge 2 \neq 3 \wedge 2 \neq 1$). If loc were $\{2e, 3e\}$ then while $H(2)$ can transition out of state H2, $H(3)$ cannot. We say $H(3)$ is "stuck" in this case. If loc were $\{2e, 4e\}$ then $H(2)$ and $H(4)$ can both transition out of H2 as $H(4)$ would also satisfy its consequent ($f = x.f \wedge 4 \neq 3 \wedge 4 \neq 1$). The fact that loc doesn't include 4e in the trace suggests the switch learned the port for MAC address 04:0c:ce:d2:08:6c. That brings us to our second component, namely bridging.

2.2 Bridging component $B(self)$

The bridging component is given in Table 4. It forwards a unicast frame only to the port behind which the unicast destination address was learned. Like H, it is parameterized on $self$. The port behind which a MAC address is learned is stored in MAC learning table mlt and mto is the MAC learning table timeout governing when table entries expire. For all $i \in dom(mlt)$, $mlt(i).mac$ is a MAC address that was last seen as an ingress source address at time $mlt(i).t$ at port $mlt(i).port$. From state B2, a unicast frame can

Table 4: $B(self)$ bridges between non-uplink ports

B1 → B1
$loc = \{port\ i\} \Rightarrow (port = uplink\text{-}port \vee f.da = haddr(port))$
B1 → B2
$loc = \{port\ i\} \wedge port \neq uplink\text{-}port \wedge f.da \neq haddr(port)$
B2 → B1
$(self\ e \in loc \wedge ucast(f.da)) \Rightarrow ($
$\exists i.\ mlt(i).mac = f.da \wedge t - mlt(i).t \leq mto \wedge mlt(i).port = self \vee$
$\forall i.\ mlt(i).mac \neq f.da \vee t - mlt(i).t > mto)$

exit port $self$ only if the frame's destination address has an entry

in mlt, the entry is unexpired and the port at which the destination address was learned matches the egress port (dash underlined condition), or the port for the destination address is unknown or expired (underlined condition). The latter condition allows a unicast frame to be flooded.

2.3 Learning component M

The MAC learning table is managed by the learning component defined in Table 5. It has only one state and merely constrains the

Table 5: M learns MAC addresses at non uplink ports

ML → ML
$\lambda x.\ [\ (loc = \{port\ i\} \wedge port \neq uplink\text{-}port \wedge ucast(f.sa) \wedge$
$(\exists k.\ x.mlt(k).mac = f.sa \vee \exists k.\ t - x.mlt(k).t > mto)) \Rightarrow$
$\exists k.\ mlt = x.mlt(k)\ \{mac = f.sa, t = t, port = port\}]\ \wedge$
$[\ (loc \neq \{port\ i\} \vee port = uplink\text{-}port \vee \neg ucast(f.sa) \vee$
$(\forall k.\ x.mlt(k).mac \neq f.sa \wedge \forall k.\ t - x.mlt(k).t \leq mto)$
$) \Rightarrow mlt = x.mlt\ ]$

MAC learning table in that either the table is updated (dash underlined condition) or remains unchanged (underlined condition). An update occurs if a frame arrives at a non-uplink port with a unicast source MAC address and either that address is already in the table or it's not but there's room in the table for it because there's an expired entry. Otherwise the table remains unchanged ($mlt = x.mlt$). It also remains unchanged on egress activity in a trace ($loc \neq \{port\ i\}$).

We expect a frame to be output in response to every frame input. The response can be the input frame, a rewrite of it or some other response frame. This much will be determined by other components, however, we still need a component to enforce an egress action after every ingress action. The interleaving component I accomplishes this. It has an ingress transition **I1** → **I2** labeled with $loc = \{port\ i\}$ and an egress transition **I2** → **I1** labeled with $loc \subseteq egress$.

3 TENSOR PRODUCT

Tensor product $H(self) \times B(self) \times I \times M$ gives the semantics of our 4-port switch function and is shown in Table 6. The product is computed with the help of the Yices SMT solver [40], which eliminates transitions with unsatisfiable propositions. Notice how the product automatically creates the desired control logic, splitting frame processing into handling frames destined for the switch (e.g. management frames or frames to be routed), conveyed by the underlined condition, and those that are not (switched), conveyed by the dash underlined condition. This happens because interleaving component I doesn't allow the hub component to spin on successive ingress frames arriving at the uplink port and remain in state H1. In state H1I2, $loc \subseteq egress$ is true which makes constraint $loc = \{port\ i\} \Rightarrow (port = uplink\text{-}port \vee f.da = haddr(port))$ on H1 → H1 vacuously true. Handling frames destined for the switch function occurs in state H1B1I2ML, which is incomplete with respect to the components presented because none of them is concerned with handling such frames. Thus this state merely requires $loc \subseteq egress$ to transition out.

Table 6: λ-SFA for $H(self) \times B(self) \times I \times M$

H1B1I1ML → H1B1I2ML

$\lambda x.\ loc = \{port\ i\} \wedge \underline{(port = uplink\text{-}port \vee f.da = haddr(port))} \wedge$
$((port \neq uplink\text{-}port \wedge ucast(f.sa) \wedge$
$(\exists k.\ x.mlt(k).mac = f.sa \vee \exists k.\ t - x.mlt(k).t > mto)) \Rightarrow$
$\quad \exists k.\ mlt = x.mlt(k)\{mac = f.sa, t = t, port = port\}) \qquad \wedge$
$((port = uplink\text{-}port \vee \neg ucast(f.sa) \vee (\forall k.\ x.mlt(k).mac \neq f.sa$
$\wedge\ \forall k.\ t - x.mlt(k).t \leq mto)) \Rightarrow mlt = x.mlt)$

H1B1I1ML → H2B2I2ML

$\lambda x.\ loc = \{port\ i\} \wedge port \neq uplink\text{-}port \wedge f.da \neq haddr(port) \wedge$
$\overline{}$
$((ucast(f.sa) \wedge$
$(\exists k.\ x.mlt(k).mac = f.sa \vee \exists k.\ t - x.mlt(k).t > mto)) \Rightarrow$
$\quad \exists k.\ mlt = x.mlt(k)\{mac = f.sa, t = t, port = port\}) \qquad \wedge$
$((\neg ucast(f.sa) \vee (\forall k.,\ x.mlt(k).mac \neq f.sa \wedge$
$\quad \forall k.\ t - x.mlt(k).t \leq mto)) \Rightarrow mlt = x.mlt)$

H1B1I2ML → H1B1I1ML

$loc \subseteq egress \qquad$ No action taken for frames destined for switch.

H2B2I2ML → H1B1I1ML

$self\ e \in loc \Rightarrow [\ ($
$\quad ((\neg bcast(x.f.da) \vee arp\text{-}reqrx(x.f, x.port)) \wedge \neg ucast(x.f.da)) \vee$
$\quad (f = x.f \wedge self \neq x.port \wedge self \neq uplink\text{-}port)\ )\ \wedge$
$\qquad (\neg ucast(f.da) \vee$
$\quad (\exists i.\ mlt(i).mac = f.da \wedge t - mlt(i).t \leq mto \wedge mlt(i).port = self$
$\quad \vee\ \forall j.\ mlt(j).mac \neq f.da \vee t - mlt(j).t > mto))\ ]$
$\wedge\ loc \subseteq egress \wedge mlt = x.mlt$

The switch function between non-uplink ports, on the other hand, is complete. On the ingress side (state H1B1I1ML), a frame arriving at a non uplink port that is not destined for the hardware address of the port causes an update to the MAC learning table if its source hardware address is unicast. If the source address is already in the table or there's an expired entry allowing it to be inserted then the learned port and timestamp fields are reset. If for some reason the source address is not unicast or it is but it's not already in the table and no entries in the table are expired then the MAC learning table remains unchanged. On the egress side (state H2B2I2ML), we have *self* among the egress ports for the output frame if the input frame is a broadcast but not an ARP request for the ingress port's hardware address or it's a unicast. In this case, the current frame f to be output is constrained to be $x.f$ and *self* cannot be the uplink port or the ingress port ($x.port$). In addition, if the destination hardware address of f, which is the destination address of $x.f$ since $f = x.f$, is unicast then *self* is governed by the learning component.

4 CODE GENERATION FOR DPDK PLATFORM

Every formula governing a transition in a product is converted into a minimum disjunctive normal form (DNF). Branching logic is then computed for each DNF formula. Finally, the disjuncts of these formulas are discharged into C code using the DPDK API.

Ideally, both branch size and expected running time should be optimized but this isn't always possible. Minimizing expected running time requires minimizing expected residuals:

Definition 4.1. Given a predicate p and a set of disjuncts D, let $res(p, D) = \emptyset$ if $p \in D$. Otherwise, a predicate q is in $res(p, D)$ if $\neg q \notin res(p, D)$, $q \neq p$ and there's a disjunct $d \in D$ such that q occurs in d and $d \wedge p$ is satisfiable. If A is a truth assignment for members of $res(p, D)$ then the expected residual of p relative to D and A is

$$\Pr[p \mid A] \times |res(p, D)| + (1 - \Pr[p \mid A]) \times |res(\neg p, D)|$$

For instance, consider DNF formula $(C \wedge B) \vee (F \wedge B) \vee E$, so $D = \{C \wedge B, F \wedge B, E\}$. Suppose predicate B is more likely to be true than E, C and F, reflected say by the distribution $\Pr[B] = 12/16$, $\Pr[C] = 2/16$ and $\Pr[F] = \Pr[E] = 1/16$. If B is true then C, F and E remain to be evaluated, thus $|res(B, D)| = 3$. And if it's false then only E remains, so $|res(\neg B, D)| = 1$. Residuals can likewise be computed for the other predicates. The expected residuals of the predicates then with respect to D and $A = \emptyset$ become:

$$\Pr[B] \times 3 + (1 - \Pr[B]) \times 1 = 36/16 + 4/16 = 40/16$$
$$\Pr[C] \times 3 + (1 - \Pr[C]) \times 3 = 6/16 + 42/16 = 48/16$$
$$\Pr[F] \times 3 + (1 - \Pr[F]) \times 3 = 3/16 + 45/16 = 48/16$$
$$\Pr[E] \times 0 + (1 - \Pr[E]) \times 3 = 0 + 45/16 = 45/16$$

Since B has the least expected residual, branching would begin by evaluating B to minimize expected running time. Note that by starting this way, the final branch size will not be minimal since the minimum size is achieved by evaluating E first. So it is not always possible to minimize both size and expected running time.

Residual calculations are then made for $D = \{C, F, E\}$ for the "then" branch and for $D = \{E\}$ for the "else" branch, each with respect to $A = \{B\}$. If B is a predicate asserting a frame is a broadcast, for instance, and C is a predicate asserting the frame is an ARP request then $\Pr[C|A]$ is the probability the frame is an ARP request given it's a broadcast. This can vary depending on the network environment of the UPF. An advantage of our approach is that branching logic can be regenerated continuously in response to observed traffic that causes the distribution to change. So the UPF can adapt in real time and remain optimal for the given environment.

After branching is computed for each DNF formula, the formula's disjuncts are discharged in the context of declarations provided by a service-discipline wrapper. This requires distinguishing checkable predicates from enforceable ones. The former translates into guards and the latter into statements of the generated C code. An *enforceable* predicate is one whose truth can always be guaranteed at run time, otherwise, it is *checkable*. For example, the formula on the transition from H2B2I2ML in Table 6 has disjunct:

$self\ e \in loc \wedge \underline{ucast(x.f.da)} \wedge f = x.f \wedge$
$self \neq x.port \wedge \underline{self \neq uplink\text{-}port} \wedge$
$\underline{\exists i.\ mlt(i).mac = f.da \wedge t - mlt(i).t \leq mto \wedge mlt(i).port = self} \wedge$
$loc \subseteq egress \wedge mlt = x.mlt$

Underlined predicates are checkable and all others are enforceable. Our wrapper code within which generated code runs always guarantees $loc \subseteq egress$, so this predicate can be eliminated at compile time. Further, the wrapper code runs on a single Intel core so there's no way for a concurrent thread to change the MAC learning table before entering state H2B2I2ML. Thus $mlt = x.mlt$ can be eliminated (no locking required at run time). Both predicates

are enforceable. In contrast, the existential constraint on *mlt* is checkable. On the surface, there's nothing to suggest it cannot be enforced by an implementation that sets the fields of *mlt* as prescribed. But this cannot be done as it implies control over network function inputs! M has a single state invariant Φ_{ML} given in Table 7. It relates the contents of the MAC learning table to an input sequence, specifically that $f.da$ was learned at port *self* in the past. This prevents enforcement of the constraint since no implementation can control what is learned at a port.

A discharge table maps predicates to be discharged into C, leveraging Intel's Data Plane Development Kit (DPDK) [12]. The DPDK provides a rich API. For instance, checkable predicate $ucast(f.da)$ can be discharged directly into C using the DPDK Ethernet API:

```
is_unicast_ether_addr(dst_haddr(bufs[buf]))
```

It will be much easier to prove discharge tables correct than to prove entire C programs correct. Furthermore, it need only be done once. Thereafter, proving any property of C code generated for a network function will reduce to proving properties of finite-state machines (λ-SFA), which are easier to reason about than C code.

Our service-discipline wrapper is a simple round-robin service wrapper written in C (580 lines of code) using the DPDK API (v17.05) [12] and running on an 8-core Intel Xeon 2.1Ghz server with 4 X540-AT2 10Gb Ethernet NICs, one for each port of our switch function. It repeatedly gets for each port a burst of frames using the DPDK API. For each ingress frame, it resets the port mask and current time by reading the timestamp counter register. It then executes our generated code, producing an output frame and a port mask defining the egress ports of the frame. It is a simple service discipline. Other disciplines like deficit round robin could be used instead.

5 PROVING COMPONENT PROPERTIES

The correctness of a given component is established relative to a requirement formulated as a property of a timed state sequence [1]. One formulates invariants for the states of the component and proves them by mutual induction. As examples, we have formulated invariants for state B1 of learned forwarding component $B(self)$ and state ML of MAC learning component M. They are shown in Table 7. Φ_{B1} relates the current frame to the MAC learning table, and Φ_{ML} relates the MAC learning table to timed state sequences. More precisely, Φ_{B1} says if a unicast frame, arriving at a non-uplink port, is not destined for the switch and at the next time step τ_{i+1} it exits at port *self* then the MAC learning table at time τ_{i+1} either has an unexpired entry for it, consisting of its destination MAC address and the port *self*, or does not. Φ_{ML} on the other hand states what is true of all destination MAC address/port pairs (m, p) stored in the MAC learning table relative to timed state sequences. Specifically, destination address m is the source MAC address of a frame that arrived at port p at some time τ_j prior to τ_k where $\tau_k - \tau_j \leq mto$.

Putting the two invariants together then gives us that p is the port at which destination address m was seen as a source MAC address within the last *mto* seconds. Note the invariants alone are insufficient for relating the current frame to a timed state sequence but together they accomplish it in the product state H1B1I1ML, which has partial invariant $\Phi_{\text{B1}} \wedge \Phi_{\text{ML}}$.

The invariant of a product state in general is the conjunction of invariants of its component states. The proof is a straight-forward extension of the standard correctness proof for product automata [21]. This homomorphic property is what allows proofs about properties of individual components to scale up to proofs about properties of products at no extra cost. This is key to making verification practical for 5G providers.

State invariants are proven by mutual induction on the length of a timed state sequence. Suppose $\hat{\delta}$ is the multistep transition function for a transition function δ [21], defined as $\hat{\delta}(q, (w_0, \tau_0), \sigma) = (q, \sigma)$ and for $n > 0$, $\hat{\delta}(q, (w_0 \cdots w_n, \tau_0 \cdots \tau_n), \sigma) = (p, \sigma'')$ if

$$\hat{\delta}(q, (w_0 \cdots w_{n-1}, \tau_0 \cdots \tau_{n-1}), \sigma) = (q', \sigma')$$

and $\delta(q', (w_n, \tau_n), \sigma') = (p, \sigma'')$. Note there is no empty timed state sequence; (w_0, τ_0) reflects the initial state and τ_0 the time at which initialization of that state is complete. It forms the base case for induction over sequences. Then we can show for all MAC addresses m and sequences $\mu = (w_0 \; w_1 \; \cdots \; w_n, \tau_0 \; \tau_1 \; \cdots \; \tau_n)$ satisfying

$$(w_0, \tau_0) \models \forall d. \, \tau_0 - mlt(d).t > mto \wedge mlt(d).mac \neq m$$

if σ and σ_0 are mappings where $\sigma_0(x) = (w_0, \tau_0)$ and $\hat{\delta}(\text{ML}, \mu, \sigma_0) = (\text{ML}, \sigma)$ then $\Phi_{\text{ML}}(\mu)$ holds. Proof is by induction on n.

6 RELATED WORK

Much work has been done in the design of high-level network programming languages to configure multiple packet-forwarding devices into a particular network topology [29]. Frenetic [14], NDlog [26], OpenBox [8], Nettle [39] and P4 [7]. All lack an explicit treatement of time and the ability to reason about timeouts. In [11], the aim is to verify bounded execution and crash freedom for dataplanes constructed as a packet processing pipeline of Click elements that do not share mutable state beyond the packet and its metadata. The efforts of [25, 33] involve annotating P4 dataplane code with assertions and looking for an initial state that leads to their violation. None of this work can reason about time, history or mutable state. OpenBox is unique in that it attempts to define the intersection of packet-processing pipelines via a merge algorithm on packet processing graphs. However the algorithm is described informally so its soundness is difficult to assess, especially with potential packet modification conflicts.

An intermediate network program representation, called a network transaction automaton, is described in [23, 24]. However the product of such automata is not well defined. A transition can assign to a variable and the product construction requires taking the union of two assignments. But what is the union of $x := 0$ and $x := 1$? NetKat [2] allows one to specify forwarding policies via a small set of primitive commands and combinators. NetKat expressions can be represented as deterministic finite automata (DFA). So the intersection of policies is defined by the standard product of DFA, which is an instance of a tensor product. Temporal NetKat [5], NetKat extended with linear temporal operators, also lacks an explicit treatment of time.

Emphasis on reusability can be found in the early work around kernel network stack development: x-kernel [17], Scout [31, 34], and later in extensible routers [10, 18, 20] and decomposition of security services in SDN networks [35]. Click [20], is a Linux-based

Table 7: State invariants Φ_{B1} and Φ_{ML}.

$$\Phi_{B1}(w_0 \cdots w_n, \tau_0 \cdots \tau_n):$$
$$\forall i. 0 \leq i < n.$$
$$(\ (((w_i, \tau_i) \models loc = \{port\ i\} \wedge port \neq uplink\text{-}port \wedge f.da \neq haddr(port)) \ \wedge\ ((w_{i+1}, \tau_{i+1}) \models self\ e \in loc \wedge ucast(f.da))\ ) \Rightarrow$$
$$(w_{i+1}, \tau_{i+1}) \models (\exists i.\ mlt(i).mac = f.da \wedge \tau_{i+1} - mlt(i).t \leq mto \wedge mlt(i).port = self\ \vee$$
$$\forall i.\ mlt(i).mac \neq f.da \vee t - mlt(i).t > mto$$
$$)$$

$$\Phi_{ML}(w_0 \cdots w_n, \tau_0 \cdots \tau_n):$$
$$\forall d \in dom(mlt).\ \forall m, p.\ \forall k. 0 \leq k \leq n.$$
$$((w_k, \tau_k) \models mlt(d).mac = m \wedge \tau_k - mlt(d).t \leq mto\ \wedge\ mlt(d).port = p\ ) \Leftrightarrow$$
$$(w_0 \cdots w_k, \tau_0 \cdots \tau_k) \models \exists j. 0 \leq j < k.\ ($$
$$(w_j, \tau_j) \models (loc = \{p\ i\} \wedge f.sa = m \wedge ucast(f.sa) \wedge \exists i.\ \tau_j - mlt(i).t > mto \vee mlt(i).mac = f.sa) \wedge$$
$$(w_k, \tau_k) \models \tau_k - \tau_j \leq mto \wedge mlt(d).t = \tau_j$$
$$)$$

platform for building a single network stack from reusable C++ classes or "elements" linked together to form a packet-processing chain. An element can be an arbitrarily-complex computation though in practice it usually implements some basic step in a network stack like fetching a route or decrementing a TTL. The work does not facilitate rigorous construction of network functions from reusable parts. Although packet-processing functions may be reusable they are not expressed in a way that is well suited for combining them algorithmically. In Click, they are C++ programs.

On the formal verification front, work has been done verifying controllers of software-defined networks (SDN) and dataplanes. A compiler and run-time system for NetCore [30] is verified with mechanical support in [16]. NICE [9], FlowLog [32], Kuai [27] and Kinetic [19] use model checking to verify temporal and nontemporal properties of applications like MAC address learning. Vericon [3] takes a different approach, formulating invariants of networks and properties of SDN programs in first-order logic and then checking satisfiability using Z3.

In [11], the aim is to verify bounded execution and crash freedom for dataplanes constructed as a packet processing pipeline of Click elements that do not share mutable state beyond the packet and its metadata. The efforts of [25, 33] involve annotating P4 dataplane code with assertions and looking for an initial state that leads to their violation. None of this work can reason about time, history or mutable state.

Zen is a modeling language that allows one to express and analyze a wide variety of network functions written in C# [6]. Composing two Zen models is purely operational in that a function of one model can call a function of the other. No attempt is made to define it denotationally, for instance, in terms of a new a property exhibited by the composition that a programmer can inspect. A declarative language limited to application-layer gateway processing is given in [4]. Using Z3 one can verify the correctness of packet filtering and rewrite rules.

7 CONCLUSIONS

Rather than writing networking software and then reasoning about it, the approach presented here involves generating code from products of primitive reusable components that capture various network behaviors. An example product was given with four components. These can be mechanically combined to produce a new functional specification from which code is ultimately generated. It is easy to add other components that introduce new features like per-port stateful firewalling, network address translation and so on.

No ex post facto reasoning about generated code is needed once discharge tables are proved correct. Generated code is not modified directly since changes are made at the reusable component level. Consequently, opportunities for introducing low-level bugs in C are eliminated. Contrast this with the state of the art where bugs can be introduced and then code must be analyzed to detect them. If such analysis requires one to annotate dataplane code with assertions and then check whether the code is a model of them then why bother write the code at all? Instead one should focus on the assertion logic and derive code from it, making model checking unnecessary. Others are reaching the same conclusion for SDN controller software [28]. The challenge then shifts from verifying code to compiling logical assertions into code that rivals the performance of handwritten dataplane code, a challenging but more tractable problem.

REFERENCES

[1] Rajeev Alur and Thomas Henzinger. 1994. A Really Temporal Logic. *Journal of the Association for Computing Machinery* 41, 1 (1994), 181–204.

[2] 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 41st ACM Symposium on Principles of Programming Languages*. 113–126.

[3] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Shapira, and Asaf Valadarsky. 2014. Vericon: Towards Verifying Controller Programs in Software-defined Networks. In *Proceedings of PLDI'14*. 282–293.

[4] Hampus Balldin and Christoph Reichenbach. 2020. A Domain-Specific Language for Filtering in Application-Level Gateways. In *Proceedings 19th ACM SIGPLAN Int'l Conference on Generative Programming: Concepts and Experiences*. 111–123.

[5] Ryan Beckett, Michael Greenberg, and David Walker. 2016. Temporal NetKAT. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 386–401.

[6] Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *Proceedings of HotNets '20*.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014).

[8] Anat Bremier-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: A Software-Defined Framework for Developing, Deploying and Managing Network Functions. In *Proceedings of ACM SIGCOMM'16*. 511–524.

[9] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Proc. NSDI'12*. 127–140.

[10] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. 1998. Router Plugins: A Software Architecture for Next Generation Routers. In *Proc. SIGCOMM'98*. 229–240.

[11] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *Proceedings of USENIX NSDI'14*. 101–114.

[12] DPDK 2018. *Data Plane Development Kit, Programmer's Guide, Release 18.08.0.* dpdk.org.

[13] Kevin R. Fall and W. Richard Stevens. 2012. *TCP/IP Illustrated, Volume 1: The Protocols* (second ed.). Pearson Education.

[14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ACM.

[15] Patrice Godefroid. 2016. Between Testing and Verification: Dynamic Software Model Checking. In *Dependable Software Systems Engineering*, J. Esparza et al. (Ed.). 99–116.

[16] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-verified Network Controllers. In *Proc. 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 483–494.

[17] N.C. Hutchinson and L.L. Peterson. 1991. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Trans. Software Engineering* 17, 1 (1991), 64–76.

[18] Ralph Keller, Lukas Ruf, Amir Guindehi, and Bernhard Plattner. 2002. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *Proc. IFIP-TC6 4th Int'l Working Conference on Active Networks*. 20–31.

[19] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable Dynamic Network Control. In *Proceedings NSDI'15*. 59–72.

[20] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. on Computer Systems* 18, 3 (2000), 263–297.

[21] Dexter C. Kozen. 1997. *Automata and Computability.* Springer.

[22] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans on Programming Languages and Systems* 16, 3 (1994), 872–923.

[23] Hao Li, Peng Zhang, Guangda Sun, Chengchen Hu, Danfeng Shan, and Tian Pan. 2020. An Intermediate Representation for Network Programming Languages. In *Proceedings APNet'20*. 1–7.

[24] Hao Li, Peng Zhang, Guangda Sun, Chengchen Hu, Danfeng Shan, Tian Pan, and Qiang Fu. 2020. A Modular Compiler for Network Programming Languages. In *Proceedings CoNEXT'20*. 198–210.

[25] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Cașcaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical Verification for Programmable Data Planes. In *Proceedings of ACM SIGCOMM'18*. 490–503.

[26] Boon Thau Loo and Wenchao Zhou. 2012. *Declarative Networking.* Morgan & Claypool Publishers.

[27] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. 2014. Kuai: A Model Checker for Software-defined Networks. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. 163–170.

[28] J. McClurg. 2018. *Program Synthesis for Software-Defined Networking.* Ph.D. Dissertation. University of Colorado Boulder.

[29] Oliver Michel, Roberto Bifulco, Gabor Retvari, and Stefan Schmid. 2021. The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications. *Comput. Surveys* 54, 4 (2021).

[30] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A Compiler and Run-time System for Network Programming Languages. In *Proceedings of 39th ACM Symposium on Principles of Programming Languages*.

[31] D. Mosberger and L.L. Peterson. 1996. Making Paths Explicit in the Scout Operating System. In *Proc. OSDI'96*. 153–167.

[32] Tim Nelson, Arjun Guha, Daniel Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2013. A Balance of Power: Expressive, Analyzable Controller Programming. In *Proc. Hot Topics in Software Defined Networking (HotSDN13)*.

[33] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos. 2018. Verification of P4 Programs in Feasible Time using Assertions. In *Proc. of ACM CoNEXT'18*. 73–85.

[34] L.L. Peterson, S.C. Karlin, and K. Li. 1999. OS Support for General-Purpose Routers. In *Proc. HotOS-VII*. 38–43.

[35] Seugwon Shin, Phillip Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. 2013. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proc. Network and Distributed System Security Symposium*.

[36] UPF 2017. *System Architecture for the 5G System, TS23.501.* portal.3gpp.org.

[37] Gertjan van Noord and Dale Gerdemann. 2001. Finite State Transducers with Predicates and Identities. *Grammars* 4 (2001).

[38] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic Regular Expression Explorer. In *Proc. 3rd Int'l Conference on Software Testing, Verification and Validation*. 498–507.

[39] Andreas Voellmy and Paul Hudak. 2011. Nettle: Taking the Sting Out of Programming Network Routers. In *Proceedings of Practical Aspects of Declarative Languages (PADL 2011)*. Springer Verlag, 235–249. LNCS 6539.

[40] Yices 2020. *Yices 2 SMT Solver.* yices.csl.sri.com.