2017-11-23

Modular network function virtualization

Volpano, Dennis

IEEE

2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): SWFAN 17: International Workshop on Software-Driven Flexible and Agile Networking

Modular Network Function Virtualization

Dennis Volpano
Naval Postgraduate School
Monterey, CA 93943

***Abstract*—Network functions like load balancers and stateful firewalls which traditionally have been packaged in a single proprietary device are now being virtualized in software across multiple physical devices networked together to achieve greater flexibility and scale. A virtualization can become very complex. Separating its definition from the software that implements it allows this complexity to be managed more easily. This paper describes some elementary behaviors that can be rigorously combined to produce modular definitions of new virtualizations. Behaviors are expressed using a new type of symbolic finite automaton called a λ-SFA. These automata can be formally analyzed and serve as a guide for synthesizing efficient code. As behaviors are combined, proofs of invariants for the result can leverage proofs of invariants for the elementary behaviors.**

I. Introduction

Network functions like switching, routing, load balancing and so on traditionally have been implemented in a single proprietary device with a fixed number of physical ports. Today there's interest in implementing such functions in software across commodity platforms networked together to scale a function to any number of physical ports. This task is called network function virtualization (NFV). We seek a methodology for this task that makes use of reusable components and facilitates verifying virtualizations are correct.

Industry efforts supporting NFV focus primarily on building scalable virtual machine (VM) architectures and interfaces to configure them [10], [15]. The software components comprising a virtualization each map to a VM image. As executables, VM images cannot be usefully combined beyond composing them. This is true of software components in general. Many network functions though are not compositions of behaviors but rather *intersections* of them. Further, with VM images there's no hope of formally verifying in any practical way that a function is correctly virtualized. Thus VM images and software components in general are unsuitable as reusable components in a virtualization methodology.

This paper proposes a methodology whereby a virtualization is formally defined through intersecting and composing reusable elementary behaviors. A formal definition can be more easily reasoned about than low-level code. Executable code is then synthesized from the formal definition. A set of elementary behaviors is given in the next section. It is shown how they can be reused to formally define virtualizations of three different network functions: a switch, a stateful firewall and a direct-return server load balancer. The goal in each case is to implement the function as a network of devices that scales up the number of physical ports the function can support. This is followed by an example of intersecting behaviors in Sec. IV. Proofs of invariants for complex behaviors leverage proofs of invariants for elementary ones. An example is given in Sec. IV-A. Code synthesis is an area of future work and is briefly discussed in Sec. V.

Approved for public release; distribution is unlimited.

II. Elementary Behaviors for NFV

Among the most fundamental behaviors is forwarding between ports. Consider 3 ports, namely 2, 4 and 6, and the behavior of forwarding every frame arriving at port 2 to either port 4 or 6. Suppose each port is divided into ingress and egress interfaces (e.g. $2i$ and $2e$ for port 2). Then the forwarding behavior is described in Fig. 1 by a new type of symbolic finite automaton called a λ-SFA. Its input is a finite

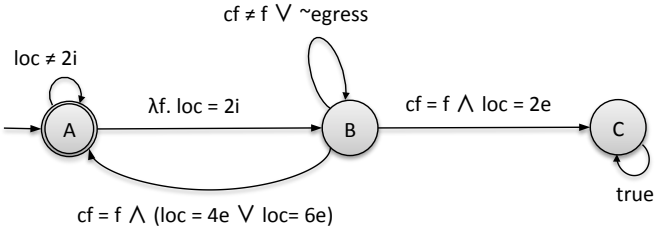

Fig. 1. λ-SFA for forwarding one frame from port 2 to port 4 or 6

sequence of triples (g, l, t) where g is an Ethernet frame and l its location at time t. These elements are referenced by *cf* (current frame), *loc* (location) and *ct* (current time). In this example, *egress* is shorthand for $loc = 2e \vee loc = 4e \vee loc = 6e$. In general, a transition is labeled with a proposition that can be any Boolean combination of linear arithmetic constraints on integers. It may also include a λ binding. For instance, any frame that causes transition from A to B can be referenced in the future since it's bound to λ variable f. The sequences this λ-SFA accepts are those in which every frame arriving at port $2i$ is eventually forwarded to port $4e$ or $6e$. No sequence in which the frame is forwarded to the originating port $2e$ is ever accepted. Which port the frame exits depends on other behaviors which are beyond the scope of forwarding. The port may be determined by learning as in a switch or a route table. Neither should be part of forwarding though as this would over constrain it. Forwarding specifies the most basic behavior which can then be constrained in different ways by intersecting it with other behaviors such as learning to get a switch or routing to get a router.

The forwarding λ-SFA buffers only one frame as there's only one λ variable. It is possible to queue up to k frames

978-1-5386-2784-6/17/$31.00 ©2017 IEEE

using only $2k + 2$ states, k variables and $O(k^2)$ transitions. These symbolic automata are exponentially more succinct than classical deterministic finite automata by virtue of λ variables.

A. MAC Address Learning

Consider learning MAC addresses behind port 4 of ports 2, 4 and 6. We wish to learn the source MAC address of a frame at ingress interface $4i$ and then for 15 seconds thereafter ensure all frames destined for that address are forwarded to egress interface $4e$. The timeout is soft in that a 15 second timer is reset if another frame with the same source address is received at $4i$ before the timeout occurs. A λ-SFA specifying this behavior is shown in Fig. 2. Here $g.ts$ is a time stamp

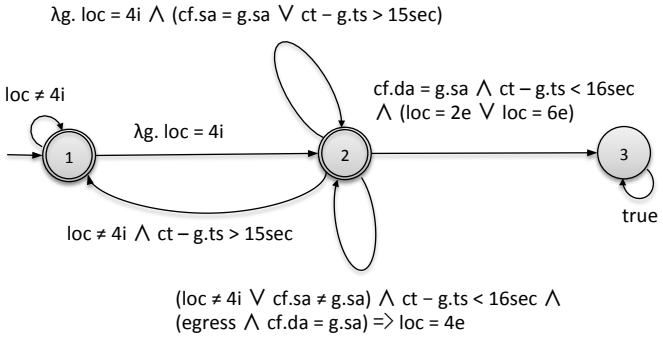

Fig. 2. Learning MAC addresses at port 4 of ports 2, 4 and 6

referring to the time g arrived at $4i$ while $g.sa$ refers to g's source address. The transition labeled

$$\lambda g.\, loc = 4i \wedge (cf.sa = g.sa \vee ct - g.ts > 15sec)$$

effectively resets the timer if another frame having source address $g.sa$ arrives at $4i$ before the timer expires. After the transition from state 1 to 2 is made, no frame destined for the source address of g is allowed at any egress port other than $4e$ for up to 15 seconds thereafter. After that, forwarding to any port is allowed. Note the machine does not constrain forwarding of broadcasts or multicast frames assuming that no frame is ever sent with such an address as its source hardware address. Multiple MAC addresses can be learned but not more than one simultaneously as there's only one λ variable. Learning up to k addresses can be done with $k + 2$ states, k variables and $O(k^2)$ transitions.

A λ variable's time stamp attribute provides a form of freeze quantifier like that of TPTL [2]. This makes a λ-SFA a type of timed automaton [1]. The difference is that a timed automaton is an ω-automaton with clock variables that can be reset. A clock variable is a way to store elapsed time between events for future reference. A λ-SFA is not an ω-automaton and has no clock variables. Computing elapsed time between events is done by storing the events in two different λ variables and computing the difference in their time stamps.

B. Source Socket Learning

Like switches, stateful firewalls also learn but they learn source sockets. A λ-SFA for learning source sockets at port 4 is identical to MAC address learning at port 4 except that

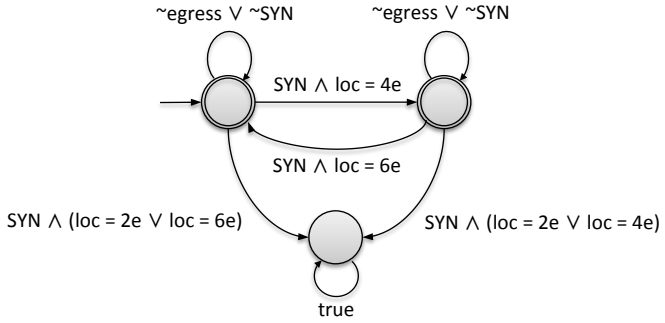

Fig. 3. Round-robin load balancing from port 2 to ports 4 and 6

it learns the port of a TCP source socket rather than a source hardware address. Specifically, $cf.da = g.sa$ is replaced in MAC learning by

$$cf.ethertype = IPv4 \ \wedge \ cf.ipproto = TCP \ \wedge \\ cf.destip = g.srcip \ \wedge \ cf.destport = g.srcport$$

and $cf.sa = g.sa$ is replaced by

$$cf.ethertype = IPv4 \ \wedge \ cf.ipproto = TCP \ \wedge \\ cf.srcip = g.srcip \ \wedge \ cf.srcport = g.srcport$$

C. Stateful Firewalling

A stateful firewall blocks packets destined for a socket behind a port unless traffic from that socket has been seen at the port in the recent past. Suppose external traffic arrives at port 2 and internal traffic at ports 4 and 6. A λ-SFA for stateful firewalling of ports 4 and 6 is obtained with only minor changes to the λ-SFA for source-socket learning. First, learn source sockets of packets arriving at ports 4 and 6. Next, add a transition from state 1 to 3 labeled $loc = 4e \vee loc = 6e$ so that no packet arriving at port 2 and destined for a socket behind ports 4 or 6 will be forwarded unless the socket has been learned at 4 or 6 in the last 15 seconds. Finally replace the transition from state 1 to itself with $loc \neq 4i \wedge loc \neq 4e \wedge loc \neq 6e$.

D. Load Balancing

Suppose we wish to distribute TCP connection requests from one ingress port to other ports in a round-robin fashion. The λ-SFA in Fig. 3 gives this semantics among 3 ports where requests arriving at port 2 are balanced across ports 4 and 6. Here SYN stands for $cf.ethertype = IPv4 \wedge cf.ipproto = TCP \wedge cf.SYN = 1 \wedge cf.ACK = 0$. Notice that round-robin balancing is not concerned with making sure all packets belonging to a TCP handshake are steered to the same server. Steering is beyond its scope and is introduced in a virtualization of load balancing using source-socket learning.

A slight modification of the λ-SFA for load balancing produces a λ-SFA for server load balancing. Suppose there are two servers, one behind port 4 with MAC address $M1$ and another behind port 6 with MAC address $M2$. Then a λ-SFA for server load balancing is shown in Fig. 4.

III. Modular Definitions of Virtualized Functions

A virtualized network function is defined using a repository of reusable λ-SFA. Ordinarily one of these functions would

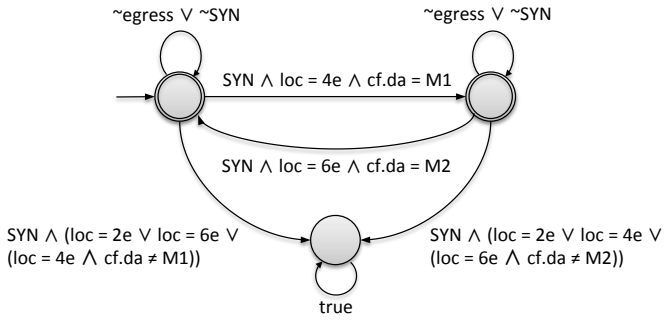

Fig. 4. Round-robin balancing of TCP connection requests arriving at port 2 to servers behind ports 4 and 6 with MAC addresses $M1$ and $M2$ respectively

be implemented by a single physical device having a fixed number of access ports. Virtualizing it means implementing it across many devices networked together to provide a much larger number of access ports. Three examples are given: a complete modular definition of a virtual switch, a virtual stateful firewall and a virtual direct-return server load balancer. Each is defined using the elementary behaviors of the preceding section. The general forms of these behaviors are summarized in Table I.

TABLE I
λ-SFA REPOSITORY FOR NFV

$\mathsf{ML}_Q(S,t)$	learn MAC addresses at all ports in S where t is a soft timeout on how long a source address is remembered at a port and $S \subseteq Q$
$\mathsf{SL}_Q(S,t)$	learn source sockets at all ports in S where t is a soft timeout on how long a source socket is remembered at a port and $S \subseteq Q$
$\mathsf{FWD}_Q(R,S)$	forward from all ports in R to those in S where $R,S \subseteq Q$
$\mathsf{SFW}_Q(S,t)$	stateful firewall where ports in S are internal, t is a soft timeout on how long traffic to a source socket is allowed absent traffic from the socket and $S \subseteq Q$
$\mathsf{LB}_Q(R,S)$	round-robin load balance TCP connection requests arriving at a port in R to ports in S where $R,S \subseteq Q$
$\mathsf{SLB}_Q(R,S)$	round-robin server load balancing of TCP connection requests arriving at a port in R to ports in S where $R \subseteq Q$ and $S \subseteq Q \times$ *HWaddr*

Suppose we want a virtual switch capable of supporting more ports than any one physical switch can support. The switches are arranged in a stack and connected by a trunk link. Switches at the top and bottom of the stack have one trunk port apiece while each interior switch has two. The virtual switch architecture is shown in Fig. 5. Each switch performs

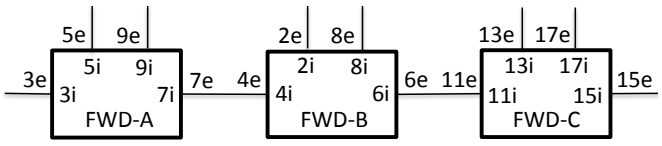

Fig. 5. Virtual switch

MAC address learning at its non-trunk ports only. That way the number of MAC addresses learned at any physical switch is independent of stack size. A unicast frame received at a non-trunk port like $2i$ should be forwarded to both trunk ports ($4e$ and $6e$) unless it's destined for a host behind another one of its non-trunk ports (here just $8e$). In general, each device forwards to its trunk ports unless the frame is destined for a host behind one of its non-trunk ports. Broadcasts are forwarded out all ports except the originating port.

To give the architecture this desired semantics, a tensor product of λ-SFA is computed for each of the devices. The product computed for the FWD-B device is shown in Table II (FWD-A and FWD-C are similar). MAC address learning

TABLE II
THE λ-SFA FOR VIRTUAL SWITCH DEVICE FWD-B

Device	Q	Tensor Product
FWD-B	$\{2,4,6,8\}$	$\mathsf{FWD}_Q(\{2i\},\{4e,6e,8e\})\times$ $\mathsf{FWD}_Q(\{4i\},\{2e,6e,8e\})\times$ $\mathsf{FWD}_Q(\{8i\},\{2e,4e,6e\})\times$ $\mathsf{FWD}_Q(\{6i\},\{2e,4e,8e\})\times$ $\mathsf{ML}_Q(\{2i,8i\},120\mathsf{sec})$

occurs at ports $2i$ and $8i$ only. Each ingress port can forward any frame to every other port. So how is a unicast frame arriving at say $8i$ prevented from being forwarded to trunk ports $4e$ and $6e$ if the frame is destined for a host that was learned at $2i$? If the host's MAC address was learned within the last 120 seconds, $\mathsf{ML}_Q(\{2i,8i\},120sec)$ will prevent it.

A. Virtual Stateful Firewall

Consider a virtual stateful firewall comprised of two physical stateful firewalls. Together they appear as just one firewall to any client on an internal network. The virtual firewall must ensure that no packets sent to a client in response to a TCP connection the client opened are dropped by either firewall as a result of being unaware of the connection. The desired behavior can be specified using our repository of λ-SFA. The idea is to steer all traffic in a single connection through one of the two physical firewalls for the lifetime of the connection. Thus there is no need for the firewalls to communicate connection state so the virtualization scales out.

A virtual stateful firewall architecture is shown in Fig. 6. The internal network connects to the virtual firewall via

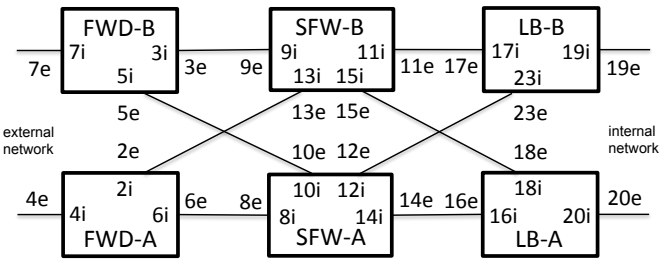

Fig. 6. Virtual stateful firewall

two devices LB-A and LB-B. They balance TCP connection requests from the internal network to the external network through two physical stateful firewalls SFW-A and SFW-B. Requests are forwarded on the other side of the firewalls. The semantics of the A devices are given by the products shown in Table III (B devices are similar). Each device is a switch even though the firewalls learn ports for TCP/IP source sockets.

How does the virtual firewall work? Suppose egress interfaces $4e$, $7e$, $19e$ and $20e$ are each connected to a different gateway router and that the internal gateways behind $19e$ and $20e$ have been statically configured to know the hardware addresses of the external gateways. Suppose a TCP SYN packet arrives at say $20e$ framed with a destination hardware address matching one of the external gateway routers. Forwarding at LB-A allows the frame to be forwarded out $16e$ or $18e$. Which one is determined by load balancing, specifically $\mathsf{LB}_{\{16,18,20\}}(\{20i\},\{16e,18e\})$. Suppose it's forwarded to SFW-A. By learning MAC addresses at ports $8i$ and $10i$, SFW-A will have learned the port ($8e$ or $10e$) to which the frame should be forwarded. Suppose it is $8e$. Then FWD-A receives the frame at $6i$ and learns the source socket of the packet at port 6 as guaranteed by SL. It then forwards the frame out $4e$ to the gateway.

Now suppose a SYN-ACK packet arrives in response at $4i$ framed with the hardware address of an internal gateway. FWD-A will forward it out $6e$ assuming the packet's destination socket matches the source socket learned there from the SYN packet (enforced by $\mathsf{SL}_{\{2,4,6\}}(\{2i,6i\},120sec)$). Since SFW-A processed the original SYN, it will permit the SYN-ACK to be forwarded but only to port $14e$ as guaranteed by SFW. LB-A then forwards it out $20e$. Thereafter connection traffic is forwarded to the right physical firewall by socket learning at the LB and FWD devices.

If FWD-A has not learned the destination socket of a packet it receives at $4e$ then its forwarding behavior as prescribed by its definition in Table III allows it to forward the packet to SFW-A or SFW-B. But since it hasn't learned the socket, neither of these devices will have seen a packet with a matching source socket so SFW will prevent them from forwarding the packet to the internal side.

B. Virtual Direct-return Server Load Balancer

One type of commercial load balancer is a physical device that distributes traffic arriving at an ingress interface to multiple egress interfaces based on some distribution algorithm like round-robin scheduling. A direct-return balancer gives the ingress interface a public IP address called the virtual IP address or VIP. Each server connected to an egress interface of the device has an interface also with the virtual IP address. The balancer rewrites the destination hardware address of an incoming frame to that of one of the server interfaces depending on the distribution algorithm. Therefore the balancer is transparent to the server which can reply with a frame destined for a router as opposed to the balancer. The balancer merely has to bridge such frames, hence direct return.

A virtual direct-return server load balancer architecture is given in Fig. 7. TCP connection requests to VIP arrive at interfaces $2i$ and $7i$. Suppose each of these interfaces is connected to a gateway to a different subnet. Because each is connected to its own gateway, replies are not bottlenecked by a single device so the architecture can scale out. The architecture also has the property that the load distribution to the servers remains invariant under changes to the load distribution at $2i$ and $7i$. TCP requests are bridged via two tiers of load balancing to one of four servers. The LB tier balances TCP connection requests between the SLB devices, each of which balances them between two servers. Unlike the LB devices, the SLB devices also rewrite destination hardware addresses to match those of the servers. We can give the architecture this desired behavior by the products for A devices shown in Table IV (B devices are similar).

TABLE III
λ-SFA FOR A DEVICES OF VIRTUAL STATEFUL FIREWALL

Device	Q	Tensor Product
LB-A	$\{16,18,20\}$	$\mathsf{LB}_Q(\{20i\},\{16e,18e\})\times$ $\mathsf{FWD}_Q(\{16i,18i\},\{20e\})\times$ $\mathsf{FWD}_Q(\{20i\},\{16e,18e\})\times$ $\mathsf{SL}_Q(\{16i,18i\},\mathsf{120sec})\times$
SFW-A	$\{8,10,12,14\}$	$\mathsf{FWD}_Q(\{12i,14i\},\{8e,10e\})\times$ $\mathsf{FWD}_Q(\{8i,10i\},\{12e,14e\})\times$ $\mathsf{SFW}_Q(\{12i,14i\},\mathsf{120sec})\times$ $\mathsf{ML}_Q(\{8i,10i\},\mathsf{120sec})$
FWD-A	$\{2,4,6\}$	$\mathsf{FWD}_Q(\{2i,61\},\{4e\})\times$ $\mathsf{FWD}_Q(\{4i\},\{2e,6e\})\times$ $\mathsf{SL}_Q(\{2i,6i\},\mathsf{120sec})$

TABLE IV
λ-SFA FOR A DEVICES OF VIRTUAL SERVER LOAD BALANCER

Device	Q	Tensor Product
LB-A	$\{2,4,6\}$	$\mathsf{LB}_Q(\{2i\},\{4e,6e\})\times$ $\mathsf{FWD}_Q(\{2i\},\{4e,6e\})\times$ $\mathsf{FWD}_Q(\{4i,6i\},\{2e\})\times$ $\mathsf{SL}_Q(\{4i,6i\},\mathsf{120sec})\times$ $\mathsf{ML}_Q(\{2i\},\mathsf{120sec})$
SLB-A	$\{8,10,12,14\}$	$\mathsf{FWD}_Q(\{12i,14i\},\{8e,10e\})\times$ $\mathsf{FWD}_Q(\{8i,10i\},\{12e,14e\})\times$ $\mathsf{SLB}_Q(\{8i,10i\},\{(12e,M1),$ $(14e,M2)\})\times$ $\mathsf{SL}_Q(\{12i,14i\},\mathsf{120sec})\times$ $\mathsf{ML}_Q(\{8i,10i\},\mathsf{120sec})$

IV. Tensor Products

The tensor product of two λ-SFA is another λ-SFA that describes their intersection. It is how elementary behaviors are refined to produce more detailed and complex behaviors. For example, consider the product of the forwarding and

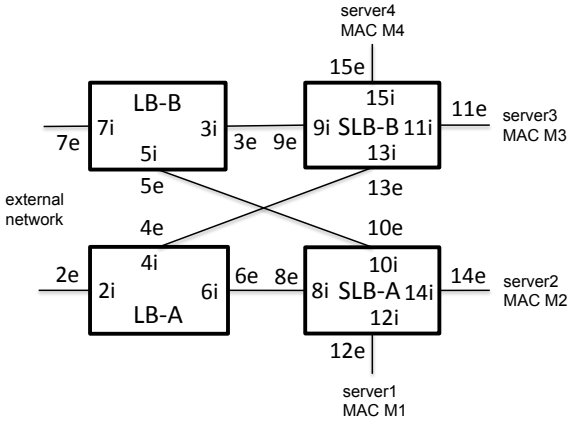

Fig. 7. Virtual direct-return server load balancer

TABLE V
$\mathsf{FWD}_{\{2,4,6\}}(\{2i\},\{4e,6e\}) \times \mathsf{ML}_{\{2,4,6\}}(\{4i\},15\mathsf{sec})$

State	Transition	
A1	A1:	$loc \neq 2i \wedge loc \neq 4i$
	A2:	$\lambda g.\, loc = 4i$
	B1:	$\lambda f.\, loc = 2i$
A2	A1:	$loc \neq 2i \wedge loc \neq 4i \wedge (ct - g.ts > 15sec)$
	A2:	$loc \neq 2i \wedge (loc \neq 4i \vee cf.sa \neq g.sa) \wedge (\neg egress \vee cf.da \neq g.sa \vee loc = 4e) \wedge (ct - g.ts < 16sec)$
	A2:	$\lambda g.\, loc = 4i \wedge (cf.sa = g.sa \vee (ct - g.ts > 15sec))$
	B2:	$\lambda f.\, loc = 2i \wedge (ct - g.ts < 16sec)$
	B1:	$\lambda f.\, loc = 2i \wedge (ct - g.ts > 15sec)$
B1	A1:	$cf = f \wedge (loc = 4e \vee loc = 6e)$
	B1:	$(cf \neq f \vee \neg egress) \wedge loc \neq 4i$
	B2:	$\lambda g.\, loc = 4i$
B2	A1:	$cf = f \wedge (loc = 4e \vee loc = 6e) \wedge (ct - g.ts > 15sec)$
	B2:	$\lambda g.\, loc = 4i \wedge (cf.sa = g.sa \vee (ct - g.ts > 15sec))$
	A2:	$cf = f \wedge (loc = 4e \vee loc = 6e) \wedge (cf.da \neq g.sa \vee loc = 4e) \wedge (ct - g.ts < 16sec)$
	B1:	$(cf \neq f \vee \neg egress) \wedge loc \neq 4i \wedge (ct - g.ts > 15sec)$
	B2:	$(cf \neq f \vee \neg egress) \wedge (loc \neq 4i \vee cf.sa \neq g.sa) \wedge (\neg egress \vee cf.da \neq g.sa \vee loc = 4e) \wedge (ct - g.ts < 16sec)$

learning machines $\mathsf{FWD}_{\{2,4,6\}}(\{2i\},\{4e,6e\})$ in Fig. 1 and $\mathsf{ML}_{\{2,4,6\}}(\{4i\},15sec)$ in Fig. 2. We would expect it to be a smarter forwarding machine since it is mixed with learning. Indeed we see this in their product given in Table V (states C and 3 are ignored in the product since these are states from which no final state can be reached). Notice start state A1 doesn't transition to B2. That's because $loc = 2i \wedge loc = 4i$ is unsatisfiable. It does transition however to A2 when a frame arrives at $4i$. The frame is then bound to g. In state A2, if a frame arrives at $2i$ then it will be bound to f and a transition is made to B2 or B1 depending on whether knowledge of g's location has expired. Suppose it hasn't and we proceed to B2. From there a transition will be made to A2 if the current frame is f, it's located at egress port $4e$ or $6e$, and knowledge of g's location, namely port 4, has not expired. In this case, notice the remaining condition on the transition requires that the egress port be $4e$ if the current frame's destination hardware address $cf.da$ matches the learned MAC address $g.sa$. This illustrates the refinement of forwarding in light of learning.

A. Verification

Proofs about elementary behaviors can be leveraged in proofs about their products. To illustrate, we formulate invariants for the forwarding and learning machines of Figs. 1 and 2. They are then used directly to get invariants for states of the product shown in Table V, the proofs of which follow from those for the individual forwarding and learning machine invariants. We begin with some definitions that help to express the invariants.

A frame received in an input sequence w at a port in set R can be buffered in a buffer of size m for R if at most $m-1$ frames were received at R and not forwarded before the frame was received. We say a frame received in w at a port in R is *pending* in w in a buffer of size m for R if it can be buffered in a buffer of size m for R and is not forwarded in w. Let $Fwd_Q(R,m)$ be true for sequence w if every frame received in w at a port in R that can be buffered in a buffer of size m for R is forwarded in w to a port in $Q - R$.

For a λ-SFA with transition function δ, we define a multi-step transition function called $\hat{\delta}$ in the standard way. If s is a state and σ maps λ variables to frames then $\hat{\delta}(s,w,\sigma)$ is a pair consisting of the state reached from s on input w using one or more transitions of δ and the mapping σ updated with bindings for λ variables according to these transitions. Then the state invariants for $\mathsf{FWD}_{\{2,4,6\}}(\{2i\},\{4e,6e\})$ become

$S_A(w)$: $\exists \sigma, \sigma'.\ \hat{\delta}(A,w,\sigma) = (A,\sigma')$ iff $Fwd_{\{2,4,6\}}(\{2\},1)$ is true for w.

$S_B(w)$: $\exists \sigma, \sigma'.\ \hat{\delta}(A,w,\sigma) = (B,\sigma')$ iff w has exactly one frame $\sigma'(f)$ pending in w in a buffer of size 1 for port 2 and $Fwd_{\{2,4,6\}}(\{2\},1)$ holds for w without $\sigma'(f)$.

$S_C(w)$: $\exists \sigma, \sigma'.\ \hat{\delta}(A,w,\sigma) = (C,\sigma')$ iff $Fwd_{\{2,4,6\}}(\{2\},1)$ is false for w.

Proof proceeds by mutual induction on the length of w.

We now formulate invariants for $\mathsf{ML}_{\{2,4,6\}}(\{4i\},15sec)$. Arriving at provable invariants for it is more challenging. The most recent source MAC address seen at port 4 may not be learned if another source MAC is currently learned there. So the current MAC address learned there if any is the one ending the longest *unexpired path* for port 4.

Let a path for MAC address b in w be a subsequence of ingress frames whose source addresses are b and for no two consecutive frames in the subsequence with arrival times t and t' is $t' - t > 15sec$. A path for address b in w is unexpired for port k if it ends with an ingress frame at k at time t' and $t - t' < 16sec$ if w ends with a frame at time t. Address b is learned at port k in w, denoted $L(b,w,k)$, if there's a path for b in w that is unexpired for k. Finally, we say there's proper forwarding in w with respect to learning at port k, denoted $PF(k,w)$, if for all prefixes $u.(h,je,t)$ of w, $L(h.da, u.(h,je,t), k)$ implies $j = k$. The state invariants become

$S_1(w)$: $\exists \sigma, \sigma'.\ \hat{\delta}(1,w,\sigma) = (1,\sigma')$ iff $PF(4,w)$ and for no address is there an unexpired path in w for port 4.

$S_2(w)$: $\exists \sigma, \sigma'.\ \hat{\delta}(1,w,\sigma) = (2,\sigma')$ iff $PF(4,w)$ and $\sigma'(g)$ ends the longest unexpired path for $\sigma'(g.sa)$ in w for port 4 from the end of the last expired path in w for port 4.

$S_3(w)$]: $\exists \sigma, \sigma'. \hat{\delta}(1,w,\sigma) = (3,\sigma')$ iff $\neg PF(4,w)$.

Now we can construct invariants for the states of the tensor product in Table V. For instance, the invariant for state B2 is the conjunction of S_B and S_2:

$S_{B2}(w)$: $\exists \sigma, \sigma'.\ \hat{\delta}(A1, w, \sigma) = (B2, \sigma')$ iff w has exactly one frame $\sigma'(f)$ pending in w in a buffer of size 1 for port 2, $Fwd_{\{2,4,6\}}(\{2\}, 1)$ holds for w without $\sigma'(f)$, $PF(4, w)$ and $\sigma'(g)$ ends the longest unexpired path for $\sigma'(g.sa)$ in w for port 4 from the end of the last expired path in w for port 4.

V. Code Synthesis

A λ-SFA expects an input sequence and accepts or rejects it. The sequence might be produced by a switch or middlebox and the λ-SFA could serve to check either offline or online whether the device behaves according to the λ-SFA. While this is useful, the long-term goal is to synthesize code from the λ-SFA that is guaranteed to produce only acceptable behaviors. A λ-SFA can make transitions on either ingress or egress activity. A received frame might cause a transition to a new state but its transmission to a particular egress port can also cause a transition. Thus generated code cannot merely "run" a λ-SFA on a stream of received frames. Some of its transitions occur when frames are transmitted from egress ports but no such transmissions are part of the input stream. It's up to the code to produce them in response to ingress activity. Further, while propositions like $cf = f$ suggest when to output a frame, there may be a choice of transitions within a state such as reading or writing a frame. So while every λ-SFA is deterministic, choices arise in the steps the code can take, making code generation a code synthesis problem. In some cases, a heuristic may guide us, say for example, forwarding before buffering more input. But this would reduce latency at the expense of throughput so there are tradeoffs to consider.

VI. Related Work

There has been extensive work done in programming networks through higher-level languages [6], [9], [11], [12], [14] however the focus has not been on identifying reusable primitives. Such focus can be seen more in the early work around kernel network stack development and later in extensible routers [5], [7]. Among the more influential efforts in this area is *Click* [8], a Linux-based platform for building a router from reusable modules that are linked together to form a packet-processing chain. *Click* modules are C++ classes that can reflect arbitrary computation. Thus they are limited to chaining and not amenable to verification.

NetKat [3] is a language for formally describing packet paths through a network. Extensions have been developed to handle paths determined by more than just forwarding tables in the network. *Temporal NetKat* [4] can describe paths that depend on a packet's history in the network while *WNetKat* [13] can handle paths that depend on link latency, bandwidth and device state. The focus here is on describing paths in order to answer queries about the network or configure it. The work though is unsuitable for specifying the semantics of devices in the network since real-time constraints are not considered.

VII. Conclusions

NFV gives network operators the ability to rapidly provision a network function normally found in a proprietary middlebox on a much larger scale using commodity hardware. Efforts to support NFV have focused primarily on implementation, specifically, building scalable VM architectures and interfaces to configure them in order to virtualize some network function. This paper takes a different approach. The emphasis is on building verifiably-correct virtualizations starting with automata for elementary component behaviors and combining them to get the desired behavior. Reasoning is done incrementally and the resulting proofs can be leveraged when these automata are combined. Executable code whose performance rivals that of custom-built software has yet to be synthesized for λ-SFA. But we believe synthesizing efficient code from a provably-correct λ-SFA will be easier than proving non-synthesized code correct.

References

[1] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

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

[3] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKat: Semantic Foundations for Networks. In *Proceedings of 41st ACM Symposium on Principles of Programming Languages*, pages 113–126, 2014.

[4] R. Beckett, M. Greenberg, and D. Walker. Temporal NetKAT. In *Proceedings of 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 386–401, 2016.

[5] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings SIGCOMM'98*, pages 229–240, 1998.

[6] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *Proceedings of 1st ACM Workshop on Research on Enterprise Networking*, Barcelona, Spain, 2009.

[7] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *Proceedings IFIP-TC6 4th Int'l Working Conference on Active Networks*, pages 20–31, 2002.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. on Computer Systems*, 18(3):263–297, 2000.

[9] B. T. Loo and W. Zhou. *Declarative Networking*. Morgan & Claypool Publishers, 2012.

[10] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of NSDI'14*, pages 459–473, 2014.

[11] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *Proceedings of 39th ACM Symposium on Principles of Programming Languages*, 2012.

[12] S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative Infrastructure Configuration Synthesis and Debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

[13] S. Schmid, K. Larsen, and B. Xue. WNetKat: A Weighted SDN Programming and Verification Language. In *Proceedings of 20th Int'l Conf. on Principles of Distributed Systems*, 2016.

[14] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Provisioning Network Resources. In *Proceedings CoNEXT'14*, 2014.

[15] R. Stoenescu, V. Olteanu, M. Popovivi, M. Ahmed, J. Martins, R. Bifulco, F. Manco, F. Huici, G. Smaragdakis, M. Handley, and C. Raiciu. In-Net: In-Network Processing for the Masses. In *Proceedings of 10th European Conference on Computer Systems*, pages 23:1–23:15, 2015.