Verification of Computer Switching Networks: An Overview

Shuyuan Zhang[1], Sharad Malik[1], and Rick McGeer[2]

[1] Department of Electrical Engineering, Princeton University, Princeton, NJ
{shuyuanz,sharad}@princeton.edu
[2] HP Laboratory, Palo Alto, CA
rick.mcgeer@hp.com

Abstract. Formal verification has seen much success in several domains of hardware and software design. For example, in hardware verification there has been much work in the verification of microprocessors (e.g. [1]) and memory systems (e.g. [2]). Similarly, software verification has seen success in device-drivers (e.g. [3]) and concurrent software (e.g. [4]). The area of network verification, which consists of both hardware and software components, has received relatively less attention. Traditionally, the focus in this domain has been on performance and security, with less emphasis on functional correctness. However, increasing complexity is resulting in increasing functional failures and thus prompting interest in verification of key correctness properties. This paper reviews the formal verification techniques that have been used here thus far, with the goal of understanding the characteristics of the problem domain that are helpful for each of the techniques, as well as those that pose specific challenges. Finally, it highlights some interesting research challenges that need to be addressed in this important emerging domain.

1 Introduction

Today's computer networks have become extremely large and complicated. The increased scale is observed in datacenters, as well as enterprise networks which can have hundreds of thousands of networking devices. The increased complexity is due to multiple kinds of networking devices (routers, switches, Network Address Translators or NATs, firewalls) that need to work together to execute the diverse network functions such as routing, access control, encryption and network address translation. Some of these devices need to support multiple protocols to make the network safer and faster. Further, the implementations of the protocols and network devices differ across vendors. Thus, it is non-trivial to make the system work correctly and efficiently.

One of the difficulties in managing such a complex system is the correct configuration of the network devices. Misconfiguration of the network counts for more than half of network downtime [5]. The misconfiguration bugs result in different kinds of network errors, among which are reachability failures, forwarding loops, blackholes, access control failures, and isolation guarantee failures. These errors

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 1–16, 2012.

can violate the security requirements, fail the correct delivery of packets, and degrade the efficiency and performance of the network. Thus, it becomes critical that we can detect network errors and verify network properties as we run and maintain the network system.

Recent years have seen the application of formal verification techniques in network configuration verification. These attempts span a range of techniques, from graph-based analysis [6] to the use of various verification engines such as ternary symbolic simulation [7], model checking [8,9] and propositional logic property checking [10]. In this paper we provide a review of these techniques, including our recent work on using propositional property checking [11,12] with the goal of understanding the characteristics of the problem domain that are helpful for each of the techniques, as well as those that pose specific challenges. Finally, it highlights some interesting research challenges that need to be addressed in this important emerging domain.

This paper is organized as follows. Section 2 provides some general background for the problem domain. The next two sections provide an overview of the two main classes of approaches, based on finite-state machine verification and Boolean satisfiability. Finally, Section 5 discusses some directions for future work in this domain.

2 Network Systems and Properties: Background

2.1 Network System States

The network systems we consider are packet switching networks. The network components can be a variety of devices such as routers, switches, bridges, Network Address Translators (NAT), firewalls, and even OpenFlow switches [13]. These devices are connected by links. In this paper, we refer to these devices as middleboxes or simply switches. Figure 1 provides an illustrative sketch of a network comprising of switches connected by links. The switches process the packets and the links transfer them between switches. The processing can vary depending on the switch, e.g. a firewall decides which packets are allowed to go through and which are blocked based on a set of rules. A router decides which of its output ports (and thus the link connected to that port) an incoming packet should be routed to based on its routing table. The flow of packets is referred to as the network traffic. Each packet consists of a header and a payload. The header captures the information needed to process the packets, e.g. source/destination addresses and the time-to-live field which indicates how long the packet may continue to stay in the network. The header may be modified as a packet is processed by a switch. The payload contains the application data.

Historically, networking traffic is thought of as operating on two levels or *planes*. In general, these refer to classes of message exchanged between switches. In classic networking, switching/routing decisions are made by switches individually, on the basis of information captured in a number of on-switch data structures, known as the *forwarding information base* (FIB) and the *routing information base* (RIB). The FIB and the RIB are updated regularly by the switch

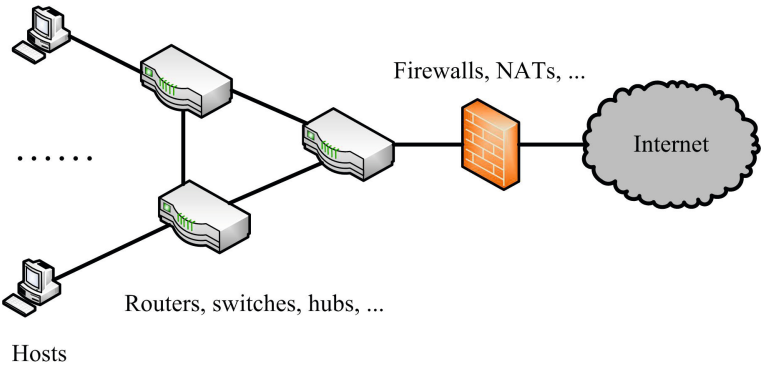

Fig. 1. Sketch of a Packet Switching Network

or router on the basis of messages received in the course of its operation. See, for example [14], among many others.

- The *Forwarding Data Plane* consists of application traffic, and is simply passed by the switch on the appropriate port as directed by the FIB and the RIB, with appropriate rewrites to the various header fields.
- The *Control Plane* consists of traffic used by the network to compute information in the RIB and the FIB. Classic examples of messages exchanged on the forwarding plane compute shortest paths to specific destinations, determine the location of devices with specific Media Access Control (MAC) or Internet Protocol (IP) addresses, and explicit control messages sent between devices on the network and from external control sources. Examples of the latter are *Border Gateway Protocol* [15] messages, which concern the handling of packets destined for locations outside the local area network and *Simple Network Management Protocol* [16]message.

The network states we are interested in here are the forwarding/blocking rules extracted from *Routing Information Base (RIB)* and *Forwarding Information Base (FIB)* in routers, *Access Control List (ACL)* in firewalls, and *Forwarding Table* in switches. We define the *switch state* as the collection of all the RIBs, FIBs, ACLs, forwarding tables, configuration policies stored in the switch at a single instance of time (see Figure 2). The *network state* is the collection of the switch states in all switches. These rules comprising the switch state usually have two fields, one matching field which specifies the packet header information for packets which should be processed using this rule, and one action field, which specifies what actions will be taken on the matching packets, i.e. how the packet is to be processed. This varies with the switch, e.g. a firewall rule will indicate if the matching packet should be dropped or allowed, a router will decide which output port (and connected link) a matching packet should be forwarded to. A payload of a packet is typically not considered in the rule matching as it does not determine the packet processing. The action field can be forwarding actions such as: blocking the packet, forwarding the packet to specific ports, flooding the packet (forwarding a copy to all but the incoming port), forwarding the packet

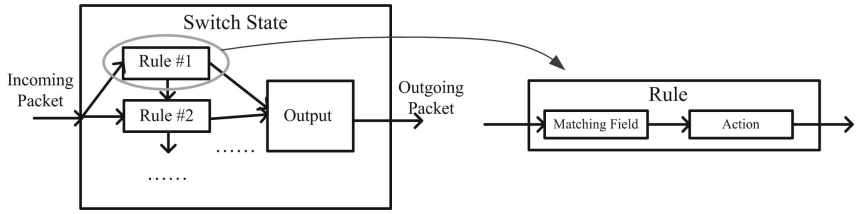

Fig. 2. The Switch State

to the incoming port, rewriting rules which rewrite some of the header fields of the packet (as in NATs), and packet encapsulation which includes an existing packet header in a new header (used for network security).

These network states we focus on here are completely static. They are snapshots of dynamic networks at a single instant in time and do not change during verification. Gude argued that changes in the network rules are on the order of tens of events per second for a network with thousands of hosts while packets arrive on the order of millions of arrivals per second for a 10Gbps link [17]. As network rule updates are much slower than the packet arrival rate, the network can be largely regarded as a static system. Consequently, we assume that the network system is stateless as it is completely fixed during verification and no packet can modify the network state. While some verification techniques consider dynamic system states such as the graph-based analysis by Xie *et al.* [6], *the focus of this paper is on formal verification techniques that consider a single snapshot of the network system state.* . We note that there has been recent work on safe update protocols for OpenFlow networks, which aim to ensure that verification certificates given by static techniques are not invalidated by network updates [18,19,20].

2.2 Network Properties

In this paper, the properties of interest are those related to functional correctness. We do not study properties related to speed metrics such as congestion, latency, and bandwidth. The following properties have been the subject of interest in the various verification efforts.

Reachability. *Reachability* is concerned with whether the network always successfully delivers packets to the intended end hosts. There are various flavors of reachability. A loosely defined reachability property can be that a packet P can get to the end host A. Although it specifies that the packet will reach A, it does not specify whether a copy of the packet may also get to other hosts. A stricter definition of reachability requires that P will always go to A and nowhere else.

Forwarding Loop. A network is said to have a *forwarding loop* if the same packet returns to a location that it has visited before. Again, there are several flavors of this property, e.g. returning to the same location with exactly the same header, or returning to the same location with a possibly different header. The former case indicates the presence of an infinite loop, since this packet

will repeatedly return to this location. The latter case may also be undesirable since there is usually no reason for a packet to return to the same location. The classic defense against forwarding loops is the `Time-To-Live` field in the packet header, which sets the maximum hop count for a packet: each switch decrements the TTL field, and discards those whose TTL reaches zero. While TTL fields preserve network resources against infinite loops, TTL discards still represent a forwarding bug that a verification system must diagnose.

Packet Destination Control. Another class of properties specifies details of fine-grained packet handling. Examples include requiring that packets of a specific class be dropped -*blacklisting*, or requiring some packets to traverse specific devices or edges - *waypointing*. Blacklisting often arises in security applications, and waypointing in audit requirements.

Slice Isolation. Multi-tenant networks must be able to guarantee that mutually-untrusting users are guaranteed privacy. A virtual, isolated private network in a shared network is referred to as a *slice* of the network [21,7]. Slice isolation specifies that it is impossible for one slice to eavesdrop or interfere with another.

2.3 OpenFlow and Stateless Networking

The recent rise in interest in network verification [7,11] has been due to the introduction of the OpenFlow protocol [13,22,23,24,25,26,27,28]. OpenFlow centralizes the control plane into a centralized controller. In the OpenFlow protocol, the various control plane sensing and command messages become controller inputs and the various routing tables to the switches are controller outputs. An important implication of this change to centralized control is that while a distributed control plane may result in non-deterministic ordering of network state updates, a centralized controller will result in deterministic network state updates thus making it easier to reason about system states and enabling analysis of a single snapshot of the system state.

Verification of the controller, which in its most general sense is Turing-complete, remains undecidable. However, the *output* of the controller is another matter. It was observed in [11,12] that this could be transformed into a graph of state-free, combinational logic elements. Though the graph is cyclic, by a technique of unrolling the graph in time, an acyclic graph can be derived. An acyclic graph of state-free, combinational logic elements is a simple logic network; its verification is $\mathcal{NP}$-complete, and in fact [11,12] demonstrated that network verification problems could be transformed into satisfiability problems on logic networks.

The recognition that OpenFlow offered a logic abstraction of the network dovetailed with an emerging literature on declarative, state-free network specification [29,30,31]. FML [30] explicitly coupled a formal verification procedure to a formal declarative specification of network behavior. The class of specifications anticipated by [30] was relatively weak however; FML specifications were verifiable in poly-time, indicating either weak verification or limited specification.

3 Finite-State Machine Based Approaches

Three techniques borrowing heavily from the ideas of Finite-State Machine (FSM) verification were recently proposed: [7,8,9,19]. All three treat the packet as a finite state machine. The state is the tuple (h, p) where h is the packet header and p is its location. The transition function is encapsulated in the rules of the network devices. The initial state of the packet is an arbitrary function of the header bits and location, as required by the specific verification obligation; a completely unspecified initial state is possible. All possible evolutions of this state graph are enumerated by symbolically propagating the packet through the network, with the header bits set as necessary for each possible propagation.

The techniques vary in the structures and methods used to propagate packets, the domain of the transfer functions, and the representation of the verification obligation. [7] permits arbitrary functions of the header bits, and represents them as Boolean functions in sum-of-products form (disjunctive-normal form or DNF). [19] specifies network properties using Computation Tree Logic (CTL) [32], and uses model checking with NuSMV [33] to verify them. [8,9] use a similar formulation of network states, but explicitly build Binary Decision Diagrams (BDDs) for the network states and verify them using SMV [34,35].

When the set of all possible state vectors have been enumerated, i.e. a fixed point is reached, the iteration ceases and the set of reached states examined to determine if an undesirable state (bad combination of header bits and location) has been reached. The Header Space Analysis (HSA) approach [7] refers to the Boolean space of header bits as the *Header Space*; the *Network Space* is the space of the tuples (h, p). The packet location is an integer encoded as a Boolean in the usual fashion.

The behavior of each network device, including packet forwarding by routers and switches, packet header modification by NATs, and packet blockage by firewalls, is modeled as a transfer function over the Network Space. This function takes the current packet state as input, and outputs the possible new states, (h', p'), where h' is the new header and p' is the new location, exactly modeling the action of the network device. In general, the function is non-deterministic, even given a fully-specified network state, e.g., a multicast packet will have multiple new locations.

The Network Transfer Function is simply the disjunction of all the device transfer functions. Let (h, p) be the packet state. The Network Transfer Function Ψ is

$$\Psi(h,p) = \begin{cases} T_1(h,p) & \text{if } p \in switch_1 \\ \ldots & \ldots \\ T_n(h,p) & \text{if } p \in switch_n \end{cases}$$

If two devices are physically connected to each other, the outgoing packets from one device will directly reach the incoming port of the other. To represent this direct connection, the *Topology Transfer Function* Γ defined as

$$\Gamma(h,p) = \begin{cases} (h,p^*) & \text{if } p \text{ connected to } p^* \\ (h,NULL) & \text{if } p \text{ is not connected} \end{cases}$$

Here NULL refers to a dummy port.

The joint use of network transfer function and topology transfer function is used to emulate the propagation of packets through the network. Let function $\Phi = \Psi(\Gamma(.))$ and recursively applying function Φ to packet (h, p) for k times or $\Phi^k(h, p)$ simply gives us the packet headers and locations after the network processes the original packet (h, p) for k hops.

Both [8,9] and [19] use formulations equivalent to [7]. [8,9] represent the explicit transition relation $T(\mathbf{current}, \mathbf{next})$ as a BDD, using only a small subset of the header bits as packet state: source and destination IP address. Here **current** and **next** are the current and next values of the packet state respectively. Matching rules are implemented in strict priority: if rule C_i and rule C_j for $j > i$ both match packet header h, then C_i matches and C_j is discarded. The exact match function for C_i is then effectively $(\neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_{i-1} \wedge C_i)$. If C_i matches h then the packet state changes from (h, p) to the next state (h', p') based on the action of rule C_i. The transition relation is simply the disjunction of these selections over all the devices in the network. This formulation of the transition relation can be easily modified to accommodate more header bits in the matching rules, and non-strict priority among matching functions. [19] formulates the network transfer function as a NuSMV program in CTL, and uses NuSMV to represent the network states and perform the standard analyses.

Since [7] does not use an available verifier, this is the only system which explicitly employs a specific propagation and solution technique. As mentioned above, transition functions, reachable states, and verification properties are expressed as Boolean functions in DNF over the network space; the state reachability iteration is performed by propagating the reached states through the transfer function until a fixpoint is reached, a technique known as Ternary Symbolic Simulation [36].

3.1 Property Checking

Reachability Analysis. Network reachability for location a is obtained by setting the initial state to `dest_ip` $= a$, then performing the classic reachable-states iteration. Reachable states with a location set to an internal device are discarded, and the remaining reachable states must all have `location` $= a$.

To check the reachability between switch a and switch b, [7] first enumerates all the paths that connect a and b. $T(.)$ is iteratively applied to the packet for every switch along the path. Then reachability from a to b is defined as

$$R_{a \to b} = \bigcup_{a \to b \text{ paths}} \{T_n(\Gamma(T_{n-1}(...(\Gamma(T_1(h, p))...)))\}$$

where h is the *header space*, p is a port of a, and $T_i(.)$ $(i \in [1, n])$ is the transfer function of switch S_i along the path $a \to S_1 \to ... \to S_{n-1} \to S_n \to b$. $R_{a \to b}$ are the final packets that will end up in b.

Forwarding Loop. Forwarding loops occur when a packet transits the same switch twice. Ternary symbolic simulation [7] checks for forwarding loops by

manually injecting the whole *header space* into one of the ports and propagating through the network until every packet either 1) gets out of the network; 2) returns to a port already visited; or 3) returns to the port that the original *header space* is injected to. To incorporate the packet history, [7] extends the original packet state with the visit history of the switches a packet has passed through. If case 3) occurs, it has found a generic forwarding loop and it ignores case 2) to avoid finding the same loop.

[19] and [8,9] use explicit model checking, and formulate the forwarding loop condition as the following temporal logic formula for the underlying verifier: $(loc = a_1) \wedge \mathbf{EX}(\mathbf{EF}(loc = a_1))$. This captures the property that the same packet gets back to switch a_1 twice.

Packet Destination Control. Reitblatt proposed using CTL to check whether a packet can get out of the network, get dropped, go through certain switches, or never pass through certain links [19]. Al-Shaer's model can also verify those properties. All of these are easily verified through the reachable-states iteration; a packet being dropped is signified by simply assigning a new location for dropped packets.

Slice Isolation. A *slice* of a network or distributed system is an isolated virtual network; in the case of a network, it refers to the ability of two or more independent users to write non-conflicting rules for packet destinations. Though the two users share the network substrate, each directs its own packets individually. In HSA or the model checking based approaches, the slices are isolated if the rules in each slice are written over disjoint network spaces. If $T_1(x, y), T_2(x, y)$ are the transition relations for the two slices, the isolation condition is simply

$$(\exists y T_1(x, y)) \cap (\exists y) T_2(x, y)) = \emptyset$$

3.2 Discussion

When the notation and terminology of [7] is translated into conventional terms, it is apparent that the method is simply checking safety properties of finite-state machines, using state propagation based upon symbolic simulation of implicants. This is nothing more than the familiar reachable-states iteration:

$$R_n(x) = R_{n-1}(x) \cup \exists_y [T(y, x) \cap R_{n-1}(y)]$$

iterated to the fixpoint $R^* = R_n = R_{n-1}$, and the verification step is simply to apply the appropriate safety check:

$$R^*(x) \cap F(x) = \emptyset$$

where $F(x)$ represents the collection of bad states. The forwarding loop verification procedure detailed above could be done far more efficiently than is described in [7], using the insights gained by recognizing the isomorphism with finite-state machine traversal. The question of whether a forwarding loop exists is essentially a question of whether the location bits in the packet state are repeated. With the symbolic ternary simulation of HSA with a DNF representation, and writing

the state vector as $x_H x_L$, where x_H are the values of the header bits and x_L the values of the location bits, the new implicants introduced into R_n are

$$\exists_y [T(y, x_H x_L) \cap R_{n-1}(y)]$$

The location bits are repeated if and only if the following condition holds:

$$(\exists_{x_H} R_{n-1}(x_H x_L)) \cap (\exists_{y x_H} [T(y, x_H x_L) \cap R_{n-1}(y)]) \neq \emptyset$$

This is clearly a polynomial computation (satisfiability and existential quantification for DNF is trivial, and it can be shown [37] that $\exists_{y x_H} [T(y, x_H x_L) \cap R_{n-1}(y)]$ is bounded above by the size of $T(y, x_H x_L)$, as is $R_{n-1}(x_H x_L)$). Of course, the size of $T(y, x_H x_L)$ may be exponential in the size of the network.

The isomorphism of the approach of [7] to the finite-state-machine verification method of [38] is striking: the same representation is used for the reachable state set, and the same algorithm is used for the iteration. [7] reports good experimental results on real networks; in contrast, researchers in verification have long since abandoned symbolic simulation of finite-state machines using DNF to represent state sets, because the size of the state sets explodes rapidly. The success of [7] suggests that the FSMs derived from networks are quite well-behaved under ternary symbolic simulation using DNF. A variety of theoretical and empirical observations support this view. In particular:

- **FSMs derived from networks are, in practice, shallow.** In [11], it was observed that a primary goal in network design is that packets traverse the network in as few hops as possible, and in particular do not traverse the same device twice. Bugs can occur – that is why a primary verification task is detecting forwarding loops – but there are a variety of safeguards built in to ensure this property. Chief among them is the *Time-To-Live* field in the packet header, which enforces a maximum hop count for a packet.
- **The network transition function is small compared to transition functions for computational FSMs.** Computational FSMs are typically expressed as the tensor product of component FSMs: because of the combinatorial properties of the tensor product, the resulting FSM transition function grows larger very rapidly. However, the network transition function in DNF is the *disjunction* of the transition functions of the component devices; disjunctions of sum of products forms grow the like the sum of the sizes of the component functions.
- **The network transition functions rely on relatively few header bits.** In principle, network devices may switch a packet on a large subset of the header bits. The Open Flow 1.1 specification identifies 15 separate fields as possibilities for switching. In practice, rules use a small subset of the header bits, for technological reasons. There is a high premium on making routing decisions using the "fast path" of network devices, which uses specialized hardware resources for matching. These resources generally offer only exact matches on specific field values, or matching on prefixes of field values, or single matches on hash functions. Ternary Content-Addressable Memories

(TCAMs) offer general matching, but these are expensive; so, typically, small TCAMs are used to buttress large, cheap, exact-match memories, which work off a few common fields: layer-2 and layer-3 addresses, source and destination port number, VLAN, protocol, type of service. Rule designers know this, and so they tend to craft rules which match on specific, fully-specified field values and prefixes of fields. It can be shown [37] that functions of this form grow slowly under various compositions, which offers a heuristic argument for the observed slow growth of network transition functions and reachable states.

– **The network transition relation is close to the network transition function in size.** [37] showed that the depth of an FSM was bounded above by the DNF size of its transition relation, as were the DNF sizes of its reachable states function R^*. The characteristic function of the transition relation of an FSM is given by $T(x, y) = 1$ iff $y = F(x)$, where $F(x)$ is the transition function. Usually, the transition relation is very large compared to the transition function, since equality across input and output fields results in a long list of enumerated implicants. However, if fields are either absent or fully specified – in other words, if implicants are all minterms over a suitably chosen subspace – then the transition relation is not much larger than the transition function, and the bounds given by [37] become relevant. That this property holds is suggested by the technological bias to minterm-heavy rulesets discussed above.

The methods of both [19] and [8,9] are isomorphic in formulation to the method of [7]; the only difference is the structures used to represent the state sets, the transition relations, and the verification obligations. [7] used a DNF representation; [8,9] used Binary Decision Diagrams; [19] relied on the internal structures of NuSMV. All three methods showed experimental success. In fact, each classic technique from the formal verification literature performed better on these examples than in verification of general hardware or software systems. We conjecture that this is due to the rulesets that network administrators write; as [37] observed, these rulesets have properties that lead to small BDDs and DNFs.

In sum, though [7] [19] [8,9] model the network verification problem as $\mathcal{P}$-Space complete FSM verification problems, as the network state is fixed, in practice it is an $\mathcal{NP}$-complete problem, with constraints that force the instances to be particularly easy to solve. In view of these, the superior results of weak techniques in this domain are explicable, though it is possible/likely that the use of stronger techniques would yield better results.

4 Boolean Satisfiability Based Approaches

Every form of verification discussed above expresses network properties as logic formulae, and uses the standard techniques of formal verification to solve them. One fundamental property of switching networks is bidirectionality: if there is a connection between port i and port j on a network device, data flows both from i to j and j to i. Dealing with cyclic graphs is therefore a core issue in network verification. In the methods of Section 3, this was done by modeling

packet traversal of networks as the evolution of a finite state machine: transit of the packet from place to place (potentially with concomitant rewrite of header bits) was modeled as a change in state of the FSM. The negative aspect of this approach is that FSM verification is a $\mathcal{P}$-space complete problem.

When a finite state machine is known to be loop-free (evolution of the machine results in a fixed point of the state in a bounded number of iterations), then the FSM is isomorphic to a combinational logic network, polynomially related to the original FSM. Verifying combinational logic networks is far easier than verifying FSMs: combinational logic verification problems all reduce to satisfiability problems which are $\mathcal{NP}$-complete.

The technique which transforms a cyclic into an acyclic combinational network is *loop unrolling*, which in the case of a network is termed *unrolling in time* [11]. This technique, used in [11][10], assumes that verification issues concern the transit of a single packet [11] (this was also the fundamental implicit assumption underlying the various formulations in Section 3). In it, each network device at time t forwards a packet to a neighbor device at time $t + 1$. Here, "time" is in fact hop count, instead of real time. The resulting graph is acyclic. The transfer functions at each device are precisely the transfer functions given in the finite state machine formulations; however, in this technique, rather than modifying the state vector they simply are connected to the appropriate successor nodes in the next time slot. [11] makes the unrolling in time explicit; [10] implicitly does this by formulating verification properties as a two-dimensional matrix of functions, where the columns are logic functions at the switches, the rows are times or levels, and all connections go from level i to $i + 1$.

Mai's system, Anteater, has some similarities with Xie's static reachability analysis. Both [10] and [11] argue that the fundamental verification problem in networking is reachability; all other problems can be phrased as minor variants of reachability. Indeed, the principal difference is that while [6] represented properties as explicit sets of packets, [10][11] represent the set as characteristic functions. Verification of reachability, forwarding loop and packet destination control is isomorphic, once the appropriate translation is made between sets of packets and characteristic functions. [11] is essentially simply a thorough analysis of a method identical to [10], and heuristic arguments which suggest that the resulting satisfiability instances will be easy to solve. [12] provides additional detail including techniques for compact representation that minimize the unrolling needed and detailed property specification that permits for a rich set of properties. The methods of [10][11][12] are simply combinational versions of the stateful methods of Section 3.

In order to represent the transfer function of the network as a whole, the network function at each switch is realized as a logic network. There are two fundamental actions: transport of the input from port i to port j, and determining the values of the header bits on port j. The former is computed from the propagation rules of the switch; simultaneously, the output bits on port j as determined by the input bits on port i are computed. The propagation functions are then used to select the appropriate values of the input port.

4.1 Property Specification and Verification

Reachability. Reachability of switch `s` from `a` in $t+1$ hops is determined by dynamic programming in the obvious way as the conjunction of the reachability of each neighbor `u` of `s` from `a` in t hops with the propagation function from `u` to `s`, and the disjunction of these functions over all neighbors. Endpoints `b` are simply terminal cases of this function. Formally, if $P(a, u, t)$ is the propagation function indicating that u is reachable from a in t hops, and $p(u, s)$ is the switch transfer function indicating that s is reachable from u, we have

$$P(a, s, t+1) = \bigvee_u P(a, u, t) \wedge p(u, s)$$

Forwarding Loop. Two sorts of forwarding loops are of interest. In one restrictive case, a packet cannot traverse a switch twice at different times. This is encapsulated as $P(s, s, t) \equiv 0$ for each t, s. The second case incorporates packet header information: one cannot traverse the same switch with the same values. This adds additional constraints on the packet header values.

Packet Destination Control. Packet destination control is written simply as a function of the reachability propagation functions, e.g.packet blacklisting checks that a packet cannot exit the network, i.e. it must be dropped by the network.

Slice Isolation. Slice isolation is worth a separate discussion, because it is time-invariant and does not rely on network unrolling. At each switch, the propagation functions written by each user must be disjoint; since this is time-invariant, it is easily checked for each switch in isolation.

4.2 Discussion

Much of the discussion of finite-state machine based approaches apply here as well. Fundamentally, FSM based approaches iteratively construct a logic function by composition, and test the resulting logic function for satisfiability; different verification obligations lead to different functions. When an FSM is isomorphic to a combinational logic function, as the global transition functions on switching networks generally are, the FSM-based approaches essentially build up the same logic network as would be found directly by calculation. As a result, in practice the FSM and combinational logic based approaches will wind up doing the same calculation over mathematically-equivalent objects; all that differs is the data structures used to represent these objects and the calculation techniques.

A thorough analysis and discussion of this single underlying object can be found in [11]. The analysis of the underlying functions in the Boolean space strongly suggested that the resulting underlying functions are amenable to the standard techniques of logic verfication, specifically ternary symbolic simulation, model checking, and propositional logic verification using SAT.

The approach in [11] dispenses with much of the complexity of other approaches, which maintain additional structures and information for various calculations. For example, [10] maintains a switch history list for each packet, and

uses this to check for forwarding loops. This is something which is calculated directly by the pure-functional approach. There may be advantages in either practical computational efficiency or in computing side propositions in the construction and maintenance of side data structures: for example, keeping track of the actual path traversed by a packet. This can be done by the pure-functional approach, but it is awkward.

This pure-functional approach is under development and test; if succesful and practical, it opens up a rich new domain for future directions in verification and switching research,

5 Conclusions and Future Directions

This paper makes a case that FSM and SAT based techniques can be successfully deployed for the verification of static snapshots of computer switching networks. The techniques that we have covered have been shown to be useful for practical sized networks. Specifically, the HSA method has been deployed for the verification of the Stanford backbone network [7]. The success of the FSM method is, in part, due to the small size of the packet state and the small depth of its state space. It is unclear if state instrumentation to encode more complex properties will retain these favorable characteristics. Increasing the number of state-bits will push the capacity limits of both BDD and SAT based model checkers. With the SAT-based propositional logic verification approach, as long as the properties can be expressed as propositional logic formulas, the size of the property formula is relatively small compared to the network state formula. Thus, its scalability is less dependent on the type of property being checked.

These initial results are based on modest sized systems. However, overall, both FSM and SAT based approaches will need to be tested for larger scale systems, e.g. entire datacenters or large scale enterprise networks. This will likely need development of new ideas in their solutions, or at the least adaptation of scaling techniques used in other domains. For example, large datacenters are likely to have symmetry in their structure. This may enable the use of parametric model checking techniques [39], or symmetry reduction in model-checking [40] and SAT-based techniques [41]. Their application will open up new challenges.

A further, more ambitious, goal is to use the verification techniques as a core for network synthesis. Logic synthesis techniques have been used successfully for synthesis of the rules for a single switch [42]. The basic technique for checking the equivalence of two firewalls has been used to formulate the problem of the synthesis of optimal firewalls [12]. Synthesis, of course, is of greater complexity than verification, and the optimal firewall synthesis problem is formulated as a Quantifed Boolean Formula (QBF) Satisfiability problem. Besides the increased theoretical complexity of QBF, practical QBF solvers have only shown limited success. Thus, the practical success of such synthesis formulations is not clear. However, high-quality synthesis can have tremendous benefits, both in improving network performance, as well as reducing design complexity.

Finally, this paper has focused on stateless switching networks. While this focus is well justified, including the network state transitions, for example through

the specification of an OpenFlow controller that sets the switch rules, will allow for full system verification. This will involve interesting modeling of the allowable network state transitions. Critical to this is careful choice of models for the specification of OpenFlow controllers. Many current controller implementation methods (e.g. Nox[17], Floodlight[43]) are Turing-complete: while these models offer computational power, verification certificates that can be offered are very weak. Computational power and verifiability are competing qualities. As a result, in any field of computation, one wants the *weakest* computational model that will accomplish the task. Some controller specification methods are written over weak computational models (e.g., [30]); others such as [44] offer some possibility of strong verification, by restricting the domain of discourse of potentially powerful computational elements.

In conclusion, this paper highlights an important emerging verification domain that can benefit from the advances in formal verification, discusses initial successes of both FSM and SAT based approaches and highlights important areas of work for extending the impact of formal verification in this domain.

References

1. Burch, J.R., Dill, D.L.: Automatic Verification of Pipelined Microprocessor Control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994)
2. Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D.E., McMillan, K.L., Ness, L.A.: Verification of the futurebus+ cache coherence protocol. Formal Methods in System Design 6, 217–232 (1995), doi:10.1007/BF01383968
3. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
4. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation OSDI 2008, pp. 267–280. USENIX Association, Berkeley (2008)
5. Alimi, R., Wang, Y., Yang, Y.R.: Shadow configuration as a network management primitive. SIGCOMM Comput. Commun. Rev. 38(4), 111–122 (2008)
6. Xie, G.G., Zhan, J., Maltz, D.A., Zhang, H., Greenberg, A., Hjalmtysson, G., Rexford, J.: On static reachability analysis of ip networks. In: INFOCOM Comput. Commun. Societ. Preceedings IEEE, vol. 3 (2005)
7. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: static checking for networks. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI 2012, pp. 9–9. USENIX Association, Berkeley (2012)
8. Al-Shaer, E., Marrero, W., El-Atawy, A., ElBadawi, K.: Network configuration in a box: towards end-to-end verification of network reachability and security. In: 17th IEEE International Conference on Network Protocols, ICNP 2009, pp. 123–132 (October 2009)
9. Al-Shaer, E., Al-Haj, S.: Flowchecker: configuration analysis and verification of federated openflow infrastructures. In: Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration, SafeConfig 2010, pp. 37–44. ACM, New York (2010)

10. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P.B., King, S.T.: Debugging the data plane with anteater. In: Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM 2011, pp. 290–301. ACM, New York (2011)
11. McGeer, R.: Verification of switching network properties using satisfiability. In: ICC Workshop on Software-Defined Networks (June 2012)
12. Zhang, S.: Model checking/boolean satisfiability in switch network verification and synthesis. Princeton University, Department of Electrical Engineering, Ph.D. Research Seminar Examination Report (May 2012)
13. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38(2), 69–74 (2008)
14. Moy, J.: RFC 2328: OSPF Version 2. Technical report, IETF (1998)
15. Hares, S., Rekhter, Y., Li, T., Addresses, E.: A Border Gateway Protocol 4 (BGP-4). Technical Report 4271, RFC Editor, Fremont, CA, USA (January 2006)
16. Harrington, D., Presuhn, R., Wijnen, B.: An architecture for describing simple network management protocol (snmp) management frameworks. Technical report, RFC Editor, United States (2002)
17. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., Shenker, S.: Nox: towards an operating system for networks. SIGCOMM Comput. Commun. Rev. 38(3), 105–110 (2008)
18. Reitblatt, M., Foster, N., Rexford, J., Walker, D.: Software updates in openflow networks: Change you can believe in. In: Proceedings of HotNets (2011)
19. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., David, W.: Abstractions for network update. SIGCOMM Comput. Commun. Rev. (August 2012)
20. McGeer, R.: A safe, efficient update protocol for openflow networks. In: Proceedings of Hot SDN (2012)
21. Sherwood, R., Gibb, G., Yap, K.K., Casado, M., Appenzeller, G., McKeown, N., Parulkar, G.: Can the production network be the testbed. In: OSDI (2010)
22. Foundation, T.O.N.: The openflow switch specification, `http://OpenFlowSwitch.org`
23. Casado, M., McKeown, N.: The virtual network system. In: ACM SIGCSE (2005)
24. Casado, M., Garfinkel, T., Akella, A., Freedman, M., Boneh, D., McKeown, N., Shenker, S.: Sane: A protection architecture for enterprise networks. In: Usenix Security (2006)
25. Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N., Shenker, S.: Ethane: Taking control of the enterprise. In: Proceedings of ACM SIGCOMM (August 2007)
26. Casado, M., Koponen, T., Moon, D., Shenker, S.: Rethinking packet forwarding hardware. In: Proc. Seventh ACM SIGCOMM HotNets Workshop (2008)
27. Casado, M., Freedman, M.J., Pettit, J., Luo, J., Gude, N., McKeown, N., Shenker, S.: Rethinking enterprise network control. Transactions on Networking (ToN) 17(4), 1270–1283 (2009)
28. Casado, M., Koponen, T., Ramanathan, R., Shenker, S.: Virtualizing the network forwarding plane. In: PRESTO (2010)
29. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. CACM 52(11), 87–95 (2009)
30. Hinrichs, T., Gude, N., Casado, M., Mitchell, J., Shenker, S.: Practical declarative network management. In: Proceedings of ACM SIGCOMM Workshop: Research on Enterprise Networking, WREN (2009)

31. Voellmy, A., Hudak, P.: Nettle: Taking the Sting Out of Programming Network Routers. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 235–249. Springer, Heidelberg (2011)
32. Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. Journal of Computer and System Sciences 30(1), 1–24 (1985)
33. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
34. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 1020 states and beyond. Information and Computation 98(2), 142–170 (1992)
35. McMillan, K.L.: Symbolic Model Checking, 1st edn. Kluwer Academic Publishers (1993)
36. Bryant, R., Seger, C.-J.: Formal Verification of Digital Circuits Using Symbolic Ternary System Models. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 33–43. Springer, Heidelberg (1991)
37. McGeer, R.: New results on bdd sizes and implications for verification. In: Proceedings of the International Workshop on Logic Synthesis (June 2012)
38. Devadas, S., Ma, H.K.T., Newton, A.R.: On the verification of sequential machines at differing levels of abstraction. IEEE Trans. on CAD of Integrated Circuits and Systems 7(6), 713–722 (1988)
39. Emerson, E., Namjoshi, K.: On model checking for non-deterministic infinite-state systems. In: Proceedings of Thirteenth Annual IEEE Symposium on Logic in Computer Science, pp. 70–80 (June 1998)
40. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design 9, 105–131 (1996), doi:10.1007/BF00625970
41. Aloul, F., Sakallah, K., Markov, I.: Efficient symmetry breaking for boolean satisfiability. IEEE Transactions on Computers 55(5), 549–558 (2006)
42. McGeer, R., Yalagandula, P.: Minimizing rulesets for tcam implementation. In: Proceedings IEEE Infocom (2009)
43. The floodlight openflow controller, `http://floodlight.openflowhub.org/`
44. Foster, N., Harrison, R., Meola, M.L., Freedman, M.J., Rexford, J., Walke, D.: Frenetic: A high-level language for openflow networks. In: ACM PRESTO 2010 (2010)