A Survey on Network Verification and Testing With Formal Methods: Approaches and Challenges

Yahui Li, Xia Yin, Zhiliang Wang, Jiangyuan Yao, Xingang Shi, Jianping Wu, Han Zhang, *Member, IEEE*, and Qing Wang

***Abstract*—Networks have grown increasingly complicated. Violations of intended policies can compromise network availability and network reliability. Network operators need to ensure that their policies are correctly implemented. This has inspired a research field, network verification and testing, that enables users to automatically detect bugs and systematically reason their network. Furthermore, techniques ranging from formal modeling to verification and testing have been applied to help operators build reliable systems in electronic design automation and software. Inspired by its success, network verification has recently seen increased attention in the academic and industrial communities. As an area of current interest, it is an interdisciplinary subject (with fields including formal methods, mathematical logic, programming languages, and networks), making it daunting for a nonprofessional. We perform a comprehensive survey on well-developed methodologies and tools for data plane verification, control plane verification, data plane testing and control plane testing. This survey also provides lessons gained from existing solutions and a perspective of future research developments.**

***Index Terms*—Network verification, network testing, formal methods, network reliability, software-defined network.**

I. Introduction

MODERN enterprise networks are complicated. A single service request in Google or Azure is responded to by hundreds of servers. Networking devices modify packets quite differently, e.g., firewalls, which interdict messages based on various rules, and load balancers, which spread traffic using hash functions. Hundreds of devices perform complex network functions, which is quite challenging and hence error prone. Errors (e.g., configuration errors, software implementation bugs, and unexpected protocol interactions) can lead to numerous security vulnerabilities, including network outages, router oscillations and forwarding blackholes.

A survey [1] conducted by the North American Network Operators Group revealed that 35 percent of all respondents encountered more than 25 tickets per month. Other surveys revealed that Microsoft Azure suffers from more than 10,000 cores being down per hour. Unfortunately, network administrators still use primitive tools, e.g., Ping [2] and TraceRoute [3], to manage networks. In addition, a new network architecture, Software-Defined Networking (SDN), is attracting increasing attention from industry and academia [4]. The innovative features of centralized control and programmability dramatically simplify network management and enable network innovations. However, the programmability of SDN increases the risk of network errors. Traditional network testing techniques cannot be directly applied to SDN. Therefore, we urgently need automated analysis of network systems.

Network operators manage networks with experience and intuition. They have been regarded as "masters of complexity" [5]. Reasoning networks manually is challenging fundamentally because of the scale and diversity of large networks and the rapidity of deployments. Looking for a bug in a bad access control list or simple questions about a network are notoriously difficult tasks. We do not know which packet headers from host A can reach host B, let alone answer security questions such as whether group X is isolated from group Y [6]. Certain quantitative questions, such as "is my load balancer distributing evenly?" or "why is my backbone utilization poor?", are also difficult to answer.

Inspired by the well-honed automated reasoning technologies for software and hardware, networking researchers have proposed a new research paradigm, called network-level verification and testing[1] [5]. They regard the whole network system as a software program that takes packets from the network input edges and outputs packets of the output edge after rewriting them [2]. Previous research in networks focused on verifying the design and implementation of protocols [7], [8], rather than network verification and testing that formally verify or test the correctness of the entire network

Manuscript received November 6, 2017; revised May 11, 2018 and June 26, 2018; accepted August 23, 2018. Date of publication August 31, 2018; date of current version February 22, 2019. This work was supported in part by the National High Technology Research and Development Program of China 863 Program under Grant 2015AA016105, and in part by the National Natural Science Foundation of China under Grant 61402253. *(Corresponding author: Zhiliang Wang.)*

Y. Li, X. Yin, J. Wu, and Q. Wang are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China.

Z. Wang and X. Shi are with the Institute for Network Sciences and Cyberspace, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: wzl@cernet.edu.cn).

J. Yao is with the College of Information Science and Technology, Hainan University, Haikou 570228, China.

H. Zhang is with the School of Cyber Science and Technology, Beihang University, Beijing 100191, China.

Digital Object Identifier 10.1109/COMST.2018.2868050

[1]In this paper, we use the terms 'network-level verification and testing' and 'network verification and testing' interchangeably.

TABLE I
DATA PLANE VERSUS CONTROL PLANE

	Data Plane (DP)	***Control Plane (CP)***
Definition	Collection of forwarding tables and logic that forward packets.	Program builds forwarding tables with topology and lived links.
Formula	$(FIB * Packet) \rightarrow FwdResult$	$(Config * Environment) \rightarrow FIB$
Program types	Forwarding table in IP network. Flow table in SDN network	RIP, OSPF, BGP etc. in IP network
Verification tasks	For all packets, invariants hold with the forwarding tables.	For all packets, invariants hold with the configurations.

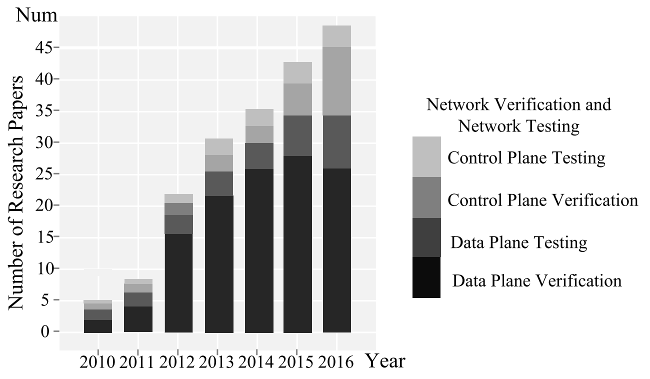


Fig. 1. Number of research papers on network verification and testing in this survey.

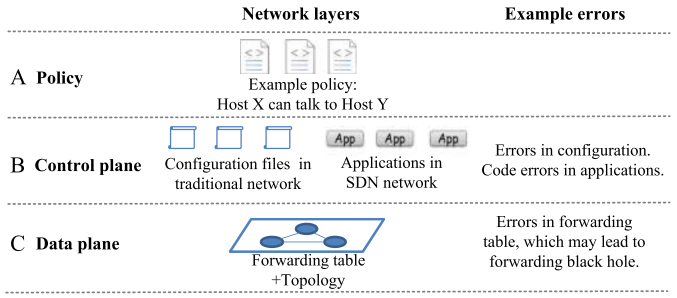

Network layers
Example errors
A Policy
Example policy:
Host X can talk to Host Y
B Control plane
Configuration files in traditional network
App
Applications in SDN network
Errors in configuration.
Code errors in applications.
C Data plane
Forwarding table +Topology
Errors in forwarding table, which may lead to forwarding black hole.

Fig. 2. Network layers and example errors. In traditional network, users implement policies (A) by configuring device-specific configurations (B). Then devices generate FIBs (C) using routing protocols, which determine the forwarding behavior of packets. In SDN network, polices (A) are implemented via SDN applications (B) in a programmable way. Then controller translates the logic of applications into flow entries installed in the switches (C).

system by collecting network data plane or control plane information.

Recently, network verification and testing have seen increased popularity in the verification and programming language community [5], [9], [10]. Many companies and universities, including Microsoft, Stanford, and UIUC, have set up network verification research groups [11], [12]. Fig. 1 shows that research in this field has intensified. Some efforts have been made to introduce the development of formal verification techniques (e.g., model checking) [13]. This is orthogonal to our goal, which is to introduce the application of formal verification to network systems. Despite some work referring to the application of formal methods to networking [14]–[18], a detailed survey on this topic, network-level verification and testing, has not yet be conducted. Specifically, a survey conducted by Qadir and Hasan [19] is the closest to our survey. The survey introduces the broad use of formal verification in networking, including checking security properties for different protocols (e.g., medium access control protocols, routing protocols, reliability protocols and other protocols) and verifying network systems. We focus on the application of formal method techniques to control and data plane verification (testing), therein attempting to prove or check network-wide properties rather than properties for individual protocols. Qadir and Hasan [19] provide detailed background of formal methods, but brief introduction of the works on data and control plane verification. In this paper, we focus on summarizing the developments of the techniques, and discussing future research directions in this domain field. In addition, as a new research paradigm, network verification and testing have seen increased popularity in the verification and programming language community since 2011. Some recent progresses in this area are not involved in the survey [19].

We introduce the background of this survey in Section II. Section III introduces formal methods that are the foundation of network verification. In Section IV, we provide a brief overview of techniques in the area. Then, we provide detailed introduction of data plane verification in Section V, control plane verification in Section VI, data plane testing in Section VII, and control plane testing in Section VIII. Section IX gives a comprehensive comparison of network verification and network testing. Sections X–XI discuss the challenges and potential research directions in this area. Finally, concluding remarks are given in Section XII.

II. BACKGROUND

A. Data Plane and Control Plane

Fig. 2 illustrates the layers of network, including policy (A), control plane (B) and data plane (C). Users usually need to implement a set of policies (A), such as whereby host *X* can talk to host *Y*. In traditional networks, users use low-level configurations (B) to realize high-level policies (A) manually. Devices in networks such as routers and switches then run control protocols, e.g., routing protocols, whose parameters are defined by these configurations, finally generate forwarding information base (C). The incoming packets to these devices are handled (possibly modified or forwarded to a neighboring router) according to the forwarding state. While in SDN networks, the polices (A) are implemented via SDN applications (B) in a programmable way. Then the controller translates the control logic of the applications into flow entries to be installed in the switches (C), which are responsible for packet forwarding.

Before discussing the focus of the sub topic (data plane and control plane), we introduce the following term, *invariant*, to facilitate subsequent presentation. In verification community, an *invariant* is a property, held by a class of objects, which remains unchanged when transformations of a certain type are applied to the objects. Given a model of the system, we can check or prove whether the model meets a set of invariants.

TABLE II
ABBREVIATION NOTATIONS

Abbreviation	*Definition*
ACL	Access control list
BDD	Binary decision diagram
CTL	Computation tree logic
CP	Control plane
DP	Data plane
EFSM	Extended finite state machines
FIB	Forwarding information base
FSM	Finite state machines
LTL	Linear temporal logic
PCTL	Probabilistic computation tree logic
PSL	Property specification language
SAT	Satisfiability theories
SDN	Software-Defined networking
SMT	Satisfiability modulo theories
TA	Timed automata

Invariants of the network specify the correctness properties of the forwarding or routing behavior. For example, the invariant, *no forwarding loops*, asserts that packets do not encounter forwarding loops in the network.

The *data plane*, such as the flow table in SDN networks [20] and the forwarding information base (FIB)[2] in traditional networks [21], is the snapshot of the forwarding table in network devices. As shown in Table I, the inputs of data plane verification tool are the data plane's snapshot and network invariants (e.g., no forwarding loops and no black holes.). It models the snapshot as logical facts, which determine whether a packet should be forwarded to a neighbor in each router or dropped. Then, it formally checks whether the snapshot is consistent with the invariants. However, it cannot detect failed links/routers or performance problems caused by network congestion. To address these challenges, researchers have proposed data plane testing. With the forwarding table and topology information, such testing generates abstract test cases via a formalized model, and converts abstract test cases into real test traffic to detect forwarding errors and performance issues [1].

The *control plane* refers to the network program that establishes the data plane forwarding tables integrated with topology information – (e.g., which router ports are interconnected) and the environment (e.g., route advertisements) [22]. The control plane in traditional networks includes the algorithm protocols (e.g., OSPF [23] and BGP [24]) distributed in configuration files scattered among thousands of network devices. The control plane in SDN networks is concentrated in the controller and in the application on top of it [25]. The inputs for control plane verification and testing are route announcements, link states and other information.

The policy is the reference of con trol plane and data plane verification. To ensure the configuration and SDN program is well designed, control plane verification checks consistency between policies and the configuration or the SDN program. To ensure the network behaves as designed, data plane verification checks consistency between policies and the forwarding states of network data plane. How do users

[2]All used abbreviation notations in this survey are listed in Table II.

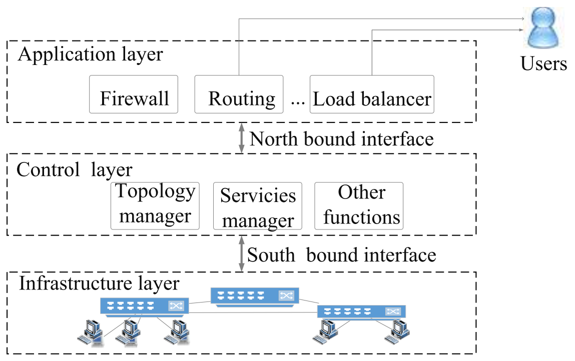


Fig. 3. A high-level overview of the software-defined networking architecture [4].

correctly check whether policies satisfy business requirements before the policies are converted in to configuration? This type of work is not within the scope of this paper, as this survey introduces verification works that check the network wide properties. We focus on whether the data plane or control plane of the actual network satisfies the policies that correspond to the invariants of the users.

B. Software-Defined Networking

This section introduces the basic concepts of *Software-Defined Networking*. Automatic reasoning technologies in traditional networks and SDN networks are quite different.

SDN decouples the control logic from forwarding devices. The prototype of SDN is preliminarily implemented based on the OpenFlow protocol [21]. The foundation of SDN proposed by the standardization organization the Open Network Foundation (ONF) is shown in Figure 3. The foundation defines three conceptual layers: the infrastructure layer, the control layer and the application layer [21]. (I) The infrastructure layer is generally referred to as a forwarding device and is responsible for packet forwarding. It offers programmable flow tables, which consist of many flow table entries. The controller configures the switch through the Southbound API [21]. (II) For the control layer, the controller can be seen as the 'network brain', which undertakes massive compute and storage tasks. It maintains a global network view. (III) In the application layer, applications are the programs on top of the controller and allow users to define network behaviors in a programmable way [26]. The communication interface between the controller and the applications is called the Northbound API. Generally speaking, the controller and applications represent the control logic, which forms the control plane. The control plane translates the control-logic policy of the applications into rules installed in the data plane.

The entire structure of traditional networks is highly decentralized and operates in an entirely distributed manner, making it difficult to realize automatic network verification tools. In contrast, SDN networks are built on top of standard interfaces. This enables the controller to configure and communicate among different underlying devices. These features (e.g., the centralized control logic and standard interface) enable users to verify the correctness and security network properties with formal methods.

TABLE III
SUMMARY OF REPRESENTATIVE TOOLS BASED ON MODEL CHECKING

Tool	***Model language***	***Specification language***	***Brief summary***
SPIN [27]	Promela	LTL	Verifies distributed and concurrent systems.
NuSMV [28]	Verilog	LTL, CTL, and PSL	Extends the symbolic model checking tool SMV based on BDD.
Alloy [29]	First-order logic	Alloy	Analyzes user-specified properties based on a partial model.
UPPAAL [30]	Timed automata	TCTL	Used for the verification of real-time systems.
PRISM [22]	PRISM	CSL, LTL, and PCTL	Models systems that exhibit random or probabilistic behaviors.

TABLE IV
SOME APPLICATIONS OF FORMAL METHODS IN NETWORKING

Project	***What it does***	***Based on***
Model Checking		
Sethi et al. [48]	Present a model-checking-based method to test SDN controllers with customized model language.	Murphi [53]
NICE [25]	Performs symbolic execution and model checking to check the entire network state space.	-
Kuai [49]	Combines partial order reduction and abstraction to perform model checking on SDN controllers.	Murphi [53]
Verificare [36]	Utilizes the Verificare modeling language to perform model checking on SDN controllers.	Murphi [53]
Flowlog [50]	Proposes a declarative finite-state SDN programming language that supports model checking.	Alloy [29]
FlowChecker [51]	Verifies the correctness of OpenFlow networks based on symbolic model checking.	NumSVM [28]
Theorem Proving		
VeriCon [54]	Apply theorems proving the verification of infinite-state SDN programs.	-
Guha et al. [55]	Propose a language for SDN controller programming, Netcore, which supports verification.	Coq [56]
Reitblatt et al. [57]	Ensure consistency of network updates using the Coq prover.	Coq [56]
NATKAT [58]	Utilizes a sound program logic for verifying SDN controller based on Kleene Algebra.	Coq [56]
Symbolic Execution		
BUZZ [59]	Generates abstract test cases for stateful data planes based on Klee.	Klee [60]
HSA [6]	Models packets as Ternary bit-vectors. Performs symbolic simulation for finding data plane bugs.	-
SymNet [61]	Statically analyzes the network data plane based on optimized symbolic execution.	-
Dobrescu et al. [52]	Apply symbolic execution to verify the data plane.	S2E
NICE [25]	Applies symbolic execution to exercise code paths of controllers.	-
SAT solver		
Anteater [9]	Verifies invariants in the data plane by an SAT solver.	-
NATSAT [62]	Employs an SAT solver to verify whether the properties of the data plane are violated.	-
FLOVER [63]	Verifies flow policies in an SDN network.	Yices [64]
NoD [65]	Implements a new SAT-based solution for the reachability set predicate in the data plane.	Z3 [60]

III. FOUNDATION OF FORMAL VERIFICATION AND TESTING

Broadly speaking, formal methods are based on mathematically techniques, which can be used in the specification and modeling of systems. This section discusses the terminology of formal methods used throughout this survey.

A. Model Checking

Model checking was first developed independently by Clarke *et al.* [31] and Baier and Katoen [32]. This technique checks whether the system *model* satisfies the *specifications*. The *system model* is defined by a finite state model such as automata or finite state machines (FSM). *Specifications* are described as temporal logic formulas. Model checking checks whether the system model satisfies a specification [33], [34].

Generally speaking, model checkers typically consist of three components: (i) a method that describes the state transition of the system (S) to be verified, where S is typically written in a state machine [35], a prepositional logic language [36], the Datalog language [37] etc.; (ii) a specification that describes the properties of the system with prepositional temporal logic formula (F), such as computation logic formula (CTL) [38], temporal logic, and linear time formula (LTL) [39]; and (iii) a checking procedure that checks whether the system satisfies the desired invariants. It is formulated as a mathematical problem of whether the state transition system model S satisfies formula F. It converts the model of the system into logical formulas, and then, it computes the satisfiability of the formulas. After finding a violation, it produces a counterexample. The counterexamples allow users to diagnose and repair errors in the system. Various model checking tools have been proposed (e.g., SPIN [27], NuSMV [28] and Alloy [29]), as illustrated in Table III.

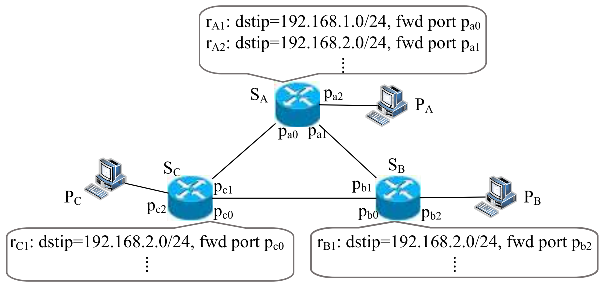
Since the checking procedure exhaustively searches the system state space, the size of the system state space increases exponentially. Many optimized approaches have been proposed. Bounded model checking leverages a fast SAT solver [40]. Symbolic model checking represents state transitions symbolically [41]–[44] using a Binary Decision Diagram (BDD). Other techniques (e.g., partial order reduction [45] and abstraction) reduce the size of the state space that needs to be searched.

Model checking has been widely used to find errors in software and hardware systems [46], [47]. Recently, it has attracted significant attention in the network verification and testing research community [25], [36], [48]–[51]. As illustrated in Table IV, Flowchecker, proposed by Al-Shaer and Al-Haj [51], detects network configuration bugs using the NuSMV tool. NICE [25] combines symbolic execution and model checking to test SDN applications [25]. Sethi *et al.* [48] presented another test SDN controller based on a model checking method.

Fig. 4 presents an example network with three switches. Each switch has forwarding rules which indicate the packets forwarding behaviour. For example, forwarding rule

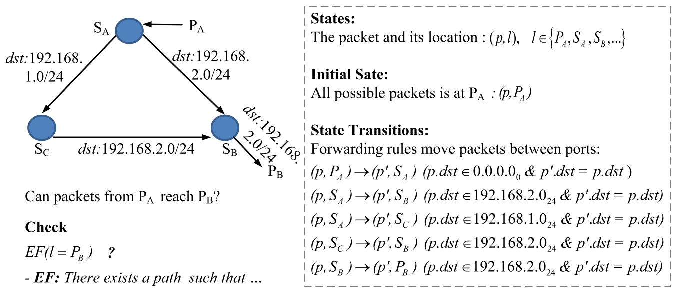
TABLE V
SUMMARY OF REPRESENTATIVE TOOLS BASED ON FORMAL METHODS

Technology	*Tool*	*Brief summary*
Theorem Proving	Coq [56]	An interactive theorem prover that assists in extracting a certified program from the constructive proof.
	HOL [66]	Represents a family of interactive theorem provers based on higher order logics and strategies.
	Isabelle [55]	A popular LCF-style theorem prover (written in Standard ML) that can work with various logics.
Symbolic Execution	Klee [60]	Performs automatic generation for programs. The supported languages are C and LLVM.
	SAGE [67]	Dynamic symbolic execution engine with whitebox fuzzing. The supported languages include C and C++.
	Pex [68]	A dynamic symbolic execution engine. The supported language is NET.
	JPF [69]	Combines symbolic execution and model checking. The supported language is Java.
SAT/SMT Solver	Z3 [70]	A state-of-the-art SMT solver from Microsoft Research.
	Yices2 [64]	An efficient SMT solver that can also act as an SAT or a MaxSAT solver.
	CVC4 [71]	Works with a version of first-order logic.
	MiniSAT [72]	Supports incremental SAT and has mechanisms for adding non-clausal constraints.
	Kodkod [73]	An SAT-based constraint solver that can work with first-order logic



r_{A1}: dstip=192.168.1.0/24, fwd port p_{a0}
r_{A2}: dstip=192.168.2.0/24, fwd port p_{a1}
S_A
p_{a2}
P_A
p_{a0} p_{a1}
S_C
S_B
P_C
p_{c1}
p_{b1}
P_B
p_{c2}
p_{c0}
p_{b0}
p_{b2}
r_{C1}: dstip=192.168.2.0/24, fwd port p_{c0}
r_{B1}: dstip=192.168.2.0/24, fwd port p_{b2}

Fig. 4. A simple network with three switches. Switch S_A has a forwarding rule r_{A1} that forwards packet whose destination is 192.168.1.0/24 via port p_{a0}, a forwarding rule r_{A2} that forwards packet whose destination is 192.168.2.0/24 via port p_{a1}. Rules in switch S_B and switch S_C are similarly defined.



S_A
P_A
*dst:*192.168. 1.0/24
*dst:*192.168. 2.0/24
*dst:*192.168. 2.0/24
S_C
*dst:*192.168.2.0/24
S_B
P_B
Can packets from P_A reach P_B?
Check
$EF(l = P_B)$?
- ***EF:*** *There exists a path such that ...*
States:
The packet and its location : (p,l), $l \in \{P_A, S_A, S_B, ...\}$
Initial Sate:
All possible packets is at P_A : (p, P_A)
State Transitions:
Forwarding rules move packets between ports:
$(p, P_A) \rightarrow (p', S_A)$ $(p.dst \in 0.0.0.0_0$ & $p'.dst = p.dst)$
$(p, S_A) \rightarrow (p', S_B)$ $(p.dst \in 192.168.2.0_{24}$ & $p'.dst = p.dst)$
$(p, S_A) \rightarrow (p', S_C)$ $(p.dst \in 192.168.1.0_{24}$ & $p'.dst = p.dst)$
$(p, S_C) \rightarrow (p', S_B)$ $(p.dst \in 192.168.2.0_{24}$ & $p'.dst = p.dst)$
$(p, S_B) \rightarrow (p', P_B)$ $(p.dst \in 192.168.2.0_{24}$ & $p'.dst = p.dst)$

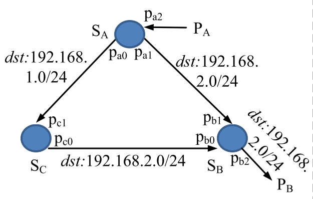
Fig. 5. A simple model checking encoding of the network in Fig. 4. The initially state represents all possible packets are at host P_A. The forwarding rules on switches moving packets between ports can be viewed as state transitions.

r_{A1} on switch R_A forwards packets whose destination ip are 192.168.1.0/24 via port p_{a0}. The *states* of the network system are defined by the *packet and its location*. All possible packets are at host P_A, which is the initial state. The forwarding rules on switches moving packets between ports can be viewed as state transitions. The corresponding model checking encoding of the network is shown in Fig. 5. The state is encoded as (p, l), where l is the location of the packet p. The initial state is encoded as (p, P_A), which represents all possible packets are at host P_A. The relevant transitions are defined according to the forwarding rules. For example, the formula, $(p, S_A) \rightarrow (p', S_B)$ $(p.dst \in 192.168.2.0_{24}$ & $p.dst' = p.dst)$, represents the forwarding logic of r_{A2}. We introduce a convenient way to write the matching condition in forwarding rules. For example, $dst \in 192.168.2.0_{24}$ is a boolean formula testing the equality between the first 24 bits of destination ip and 192.168.2.0. After encoding initial state and state transition, we define a property, there exists a path such that packets from P_A to P_B can be expressed as $EF(l = P_B)$ in CTL. In syntax of CTL, E means that there exists at least one path starting from the current state where the property holds, and F means that the property eventually has to hold (somewhere on the subsequent path). The model and property can be input to a model checker engine (e.g., NuSMV [28]). If the engine returns pass, the property holds on the system model. If the model violates the property, the system will return a counterexample. In this simple example, the model satisfies the input property.

B. Theorem Proving

Theorem proving is another important formal verification method. As depicted in [74] and [75], it consists of formulas that represent the implementation and that describe the system properties. The formulas (also known as formalized mathematical statements) consist of a set of axioms and derivation rules. This technique checks whether the property is valid with the axiom and derivation rules [74]. In early research, engineers manually designed a verification framework and then manually verified the reasoning correctness in the sketch. Recently, procedures (e.g., decision procedures) have been processed automatically by the machine, named, theorem prover. Theorem proving can be classified into automated theorem proving and interactive theorem proving. The former addresses proving mathematical theorems with computer programs. Despite the complexity, automated reasoning over algebraic proofs greatly improves developments in computer science. Interactive theorem proving handles the proof problem with human assistance. Users have to support the axioms and proof strategies with expert knowledge, which is the most creative part of the verification [75]. Table V lists various theorem provers that are routinely used in verification projects.

Unlike model checking, theorem proving does not need to exhaustively check the entire state space. This technique takes a mathematical statement and checks the proof. When checking the properties of a network system with theorem proving, users can check all admissible network topologies specified with the logic. However, scaling theorem proving to



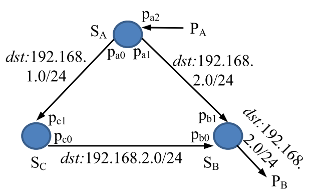
Can packets from P_A reach P_B?

Prove:

$P_A.sent(p.dst \in 0.0.0.0_0)$
$\Rightarrow P_B.recv(p.dst \in 192.168.2.0_{24})$

Topology Relations:
$link(S_A, p_{a2}, P_A)$
// Host P_A is connected to switch S_A via port p_{a2}
$link(S_A, p_{a1}, p_{b1}, S_B)$
// Port p_{a1} of switch S_A is connected to port p_{b1} of
// switch S_B
⋮
Forwarding Relations:
$S_B.recv(p.dst \in 192.168.2.0_{24}, p_{b1}) \wedge S_B.ft(p.dst \in 192.168.2.0_{24}, p_{b2}) \Rightarrow S_B.sent(p.dst \in 192.168.2.0_{24}, p_{b2})$
⋮
$link(S_A, p_{a1}, p_{b1}, S_B) \wedge S_A.sent(p.dst \in 192.168.2.0_{24}, p_{a1}) \Rightarrow S_B.recv(p.dst \in 192.168.2.0_{24}, p_{b1})$
$link(S_B, p_{b2}, P_B) \wedge S_B.sent(p.dst \in 192.168.2.0_{24}, p_{b2}) \Rightarrow P_B.recv(p.dst \in 192.168.2.0_{24})$
⋮

Fig. 6. A simple theorem proving encoding of the network in Fig. 4. The statements on the right of the figure define the forwarding logic of the network. The formula on the lower left of the figure defines the invariant: packets from P_A can reach P_B.



Can packets from P_A reach P_B?

```
1  Packet A[20];
2  // Model of a switch
3  Packet Switch(SWId id, Packet  inPacket ){
4      outPacket = lookUp(id, inPacket);
5      return outPacket;}
6  ⋮
7  main(){
8  // Model of the data plane
9     initializeTags(A[]);
10 ⋮
11   for each injected A[i]{
12       while (!DONE(A[i])){
13          Forward A[i] on current link;
14          A[i] = Switch (swid, A[i]);
15          assert(
16          (!(A[i].d_Tag == hostId[PB]))
17          || (!(A[i].s_Tag == hostId[PA]));
18 }}}
```

Fig. 7. A simple symbolic execution encoding (pseudocode) of the network in Fig. 4. Lines 3-5 in the pseudocode indicate the behaviour of switch. Lines 11-14 model the forwarding behavior of data plane. Lines 16-17 define an assertion, which indicates the network property.

complex networks is challenging. It heavily depends on the users' knowledge of the system. In general, the underlying logics are very complex and difficult to understand. In addition, the process is slow and fallible because it requires user interaction in most cases.

Theorem proving is popular in network verification [54], [55], [57], [58]. As shown in Table IV, VeriCon [54] applies theorem proving to verify infinite-state SDN programs. Reitblatt *et al.* used the Coq prover to ensure the update consistency of SDN networks [76]. Guha *et al.* [57] proposed Netcore for SDN controller programming, which also uses the Coq tool.

Fig. 6 shows a theorem proving encoding of the network in Fig. 4, which is motivated by VeriCon [54]. To verify the properties of the system, the theorem prover receives three inputs, including the forwarding rules, the toplogy of the network system, and an invariant formula such as a first-order logic formula. Topology relations define the links in the topology. For example, $link(S_A, p_{a1}, p_{b1}, S_B)$ means that port p_{a1} of switch S_A is directly connected to port p_{b1} of switch S_B. Forwarding relations model the logic of existing forwarding rules on switches moving packets between ports. For instance, $S_A.ft(p.dst \in 192.168.2.0_{24}, p_{b2})$ defines a matching condition in a forwarding rule r_{B2} in the network. $S_B.recv(p.dst \in 192.168.2.0_{24}, p_{b1})$ defines that the packet p can be received on switch S_B at ingress port p_{b1}. $S_B.sent(p.dst \in 192.168.2.0_{24}, p_{b2})$ defines that the packet p can be sent on switch S_B at port p_{b2}. To check whether packets from P_A can reach P_B, we can prove whether the formula on the lower left of the Fig. 6 is true. After users inputs the deduce rules, a theorem prover can prove whether the property holds.

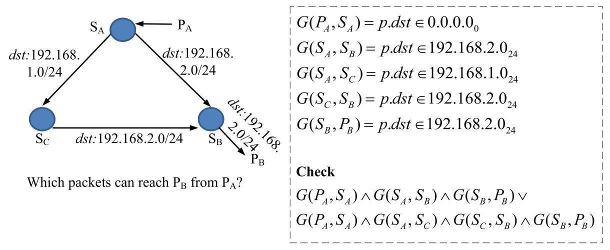
C. Symbolic Execution

Symbolic execution is a popular approach for analyzing a software program and represents program inputs with symbolic values instead of concrete values [77]. On the surface, the network does not resemble software code. It consists of routers, network interface cards, cabling etc. Moreover, the entire network can be considered as the "program". For instance, a forwarding table of the router can be considered as a program statement that forwards packets to the destination address. Therefore, the symbolic execution concept can be applicable to the analysis of the network. Instead of running with the normal inputs, it uses symbolic variables representing arbitrary values as inputs to run a program. An interpreter follows the program and logically forks and follows both branches at each code branch. It exercises all possible paths and records the constraints of the symbolic variables on each path, referred to as path conditions. Then, it solves path constraints using a constraint solver, and it obtains inputs that follow the associated path during an execution [78].

Unfortunately, symbolic execution does not scale to large programs because of path explosion. The number of feasible paths increases exponentially with increasing program size. This is unacceptable when applied to actual network code [52], [79]. The various efforts shown in Table V have made symbolic execution practical for verification such as by merging similar paths and parallelizing independent paths.

Symbolic execution has been widely used in network verification and testing [6], [25], [52], [59], [61], as shown in Table IV. SymNet [61] analyzes the network data plane based on symbolic execution [80]. Dobrescu and Argyraki [52] applied symbolic execution to verify the data plane code. NICE [25] also employs symbolic execution to the exercise code paths of the controller [25]. BUZZ generates abstract test cases for stateful data planes based on Klee [59].

Fig. 7 shows a symbolic execution encoding of the simple network in Fig. 4, which is motivated by BUZZ [59]. The program snippet indicates how forwarding rules move packets between ports. In each iteration, a packet is processed (line 12) in two steps: (1) it is forwarded to the other end of the current link (line 13), (2) it is then passed as an argument to the switch connected at this end (line 14). The output packets are then processed in the next iteration. We can define an assertion: packets from P_A can be moved to P_B (lines 16-17).



$$G(P_A,S_A)=p.dst \in 0.0.0.0_0$$
$$G(S_A,S_B)=p.dst \in 192.168.2.0_{24}$$
$$G(S_A,S_C)=p.dst \in 192.168.1.0_{24}$$
$$G(S_C,S_B)=p.dst \in 192.168.2.0_{24}$$
$$G(S_B,P_B)=p.dst \in 192.168.2.0_{24}$$

Check

$$G(P_A,S_A)\wedge G(S_A,S_B)\wedge G(S_B,P_B)\vee$$
$$G(P_A,S_A)\wedge G(S_A,S_C)\wedge G(S_C,S_B)\wedge G(S_B,P_B)$$

Fig. 8. A simple SAT encoding of the network in Fig. 4. The forwarding rules on switches move packets between ports, which can be formulated as boolean formulas on the upper right of the figure.

$A[i].s\_Tag$ records the original source ip of the packets, and $A[i].d\_Tag$ records the current location of the packets. This will allow the symbolic execution engine to find a "violation". We input the program and assertion into a symbolic execution engine (e.g., Klee [60]). The engine injects symbolic packets and tracks their evolution through the program of the network. The engine ultimately returns whether the assertion holds.

D. SAT/SMT Solvers

Many practical problems in network verification have been reduced to SAT problems and solved by SAT solvers. The satisfiability problem is as known as SAT and can be applied to all logic formulas [81]. If the given propositional formula is satisfiable, we can obtain values of the boolean variables that make the formula logically true. A Boolean formula consists of Boolean variables and operators. The input of the SAT solver is a formula expressed in propositional logic theories. The solver automatically decides on the satisfiability in the corresponding syntax.

The satisfiability modulo theories (SMT) problem generalizes pure the SAT problem. To express design and verification conditions, SMT provides first-order theories [82]–[84]. The express ability of first-order logic (e.g., difference logic, arrays, bit-vectors, and linear arithmetic) is much greater (richer) than that of propositional logic [85] in SAT. Both SAT and SMT have proven to be typical NP-complete problems, which are difficult to solve in theory. Fortunately, various SAT/SMT tools, as shown in Table V, have been proposed and seen rapid progress [13], [86], [87].

SAT/SMT solver has attracted significant attention in network verification and testing [9], [62], [63], [65]. FLOVER verifies the properties of SDN networks based on the SMT Yices solver [63]. Anteater employs an SAT solver to determine whether the properties of the network data plane are violated [9]. NetSAT is another example that verifies SDN networks based on an SAT solver [62].

A simple SAT encoding of a simple network is shown in Fig. 8. It is motivated by Anteater [9]. Similarly, we encode the matching condition in forwarding rules in a convenient way. For example, the matching condition in rule r_{A2} in Fig. 8 is represented as $dst \in 192.168.2.0_{24}$. The forwarding rules on switches move packets between ports, which can be formulated as a boolean formulas. For example, the forwarding rule at S_A which forwards packets to S_B is formulated as a boolean formula $G(S_A, S_B) = p.dst \in 192.168.2.0_{24}$. The formula $G(P_A, S_A) \wedge G(S_A, S_B) \wedge G(S_B, P_B)$ encodes the constraints on the physical path $P_A \to S_A \to S_B \to P_B$ in Fig. 8. The constraints on the path $P_A \to S_A \to S_C \to S_B \to P_B$ are similarly defined. If we want to check whether packets can be delivered from P_A to P_B (which is defined by the formula on the lower of Fig. 8), we can query the formula with an SAT solver (e.g., Z3 [70]). After we input the formulas, if the solver returns true, then the property holds; otherwise, it returns a counterexample.

TABLE VI
SUMMARY OF FORMAL METHODS USED IN NETWORKING

Technique	***Model Checking***	***Interactive Theorem Proving***	***Automated Theorem Proving***	***Symbolic Execution***	***SAT/SMT Solver***
Soundness	√	√	√	√	√
Automation	√	×	√	√	√
Generic	×	√	√	×	×
Counterexample	√	×	×	√	√
State-explosion	√	×	×	√	×

E. Conclusion on Formal Methods

We provide a comparison of well-known formal methods. As shown in Table VI, the first characteristic is soundness, which indicates that the results verified by the technique are always true. A technique is termed sound if the results verified by it are always true. All techniques in Table VI can be termed sound. Note that symbolic execution can be sound in principle. For symbolic execution, users simply provide their program, and the symbolic execution engine will examine all the feasible paths to generate test inputs or check assertions. In fact, the number of possible inputs of the system is usually very large and cannot be covered even in symbolic form. In order to tackle path explosion, techniques like approximations are often introduced [88]. However, those approximations techniques can make the execution unsound. The second characteristic is automation: a technique is automatic if it does not require user guidance. As well known, model checking is a automatic verification technique for finite-state systems. Logic in theorem proving is expressive for defining many formal language semantics and concepts in mathematics. As introduced earlier, theorem provers consist of automated theorem provers and interactive theorem provers. The main challenge of interactive theorem provers is proof automation. They always require explicit user guidance in the verification process [55]. The third characteristic refers to whether the context of the problem being modeled is generic. In terms of tackling a wide range of problems, among all techniques in Table VI, theorem proving is the only technique that supports verifying generic problems. Theorem proving supports verifying generic theorems, as it does not require an instantiation. For example, theorem proving is able to verify that an SDN program is correct on all admissible topologies and for all possible (infinite) sequences of network events [54]. While model checking usually requires a specific instantiation (e.g., a particular network topology and a specific sequence of network events) [25]. Similarly, symbolic execution cannot verify the general property for all possible networks (e.g., the correctness property of the SDN program for all possible sequences of network events) [25]. SAT/SMT solvers decide whether logic formula is satisfiable, and they also require an instantiation (e.g., the correct property holds on a specific topology [89]). The fourth characteristic is the provisioning of illustrative counterexamples upon finding a bug. Both model checkers, symbolic execution engines and SAT/SMT solvers provide one counterexample when invariant violation occurs. However, theorem proving does not provide an example with a negative connotation, which restricts the ability to debug the system. The final characteristic refers to state-explosion that must be addressed to solve most real-world problems. Symbolic execution engine suffers the explosion problem as the growth of code path. SAT/SMT solvers decide whether logic formula has a solution. They do not have a notion of state. Model checkers face a combinatorial blow up of the state-space. If we use model checking to check that the systems (e.g., finite state SDN programs) behave correctly, scaling the method to large networks is highly nontrivial [49]. While, theorem proof does not have a notion of state, and it does not model the state-space of the system. Users usually try to do theorem proof as the main verification to avoid the state-explosion problem.

IV. Overview

In this section, we introduce the shortcomings of traditional analysis approaches for networks, and summarize how the verification techniques in each plane attempt to address the challenges. In addition, we briefly explain how each plane in traditional and SDN networks impacts the choice of formal verification techniques.

As introduced earlier, the complexity of a network can inevitably lead to errors. Clearly, manually reasoning, such as checking mailing lists or naively using ping or traceroute, is inefficient. Some works check for problems in the network in a black-box manner. For instance, they send probes or review logs, which is very slow for detecting failures. Some analysis approaches which are based on the information of network configurations have been also proposed. For instance, FIREMAN [90] automatically analyzes ACLs in configuration files using a rule graph model. However, they are developed on customized models for limited aspects of the configuration. Other methods check specific correctness properties of the network to handle complexity. In addition, because they do not analyze all aspects of the configuration, they are inefficient at helping users determine how the discovered errors impact the final packet forwarding of the network.

To address these challenges, data plane verification approaches have been proposed to *check the combined impact of all configuration aspects* [9]. The first application of formal verification techniques to the data plane was Anteater [9]. Anteater avoids modeling complex protocol behaviors in configurations by analyzing the FIB in networks. It can verify network invariants such as reachability, isolation and loop freedom. Similar to Anteater, various data plane verification works (e.g., HSA [6]) focus on verifying forwarding tables. However, most works can only support off-line verification at low speeds. Some works, such as Veriflow [91], improve and optimize the calculation, and they can verify in near real time. Moreover, there are various middleboxes (e.g., firewalls) in the data plane whose forwarding behaviors are determined by previously observed traffic. Some tools, including SymNet [61] and VMN [92], utilize algorithms to verify such stateful data planes. Other works, such as SLA-verifier [93], attempt to use algorithms to verify performance properties. While these works represent great progress for data plane verification, there are a number of open problems in this area. For example, one of the main problems is scalability in the verification of stateful data planes.

The only difference is the data collection process when we verify the data plane in traditional vs SDN networks. When we verify traditional networks, we can collect FIBs from networking devices through SNMP, or terminals. When verifying SDN networks, with the Southbound Interface, we can obtain the forwarding rules by monitoring the rules, which are inserted, modified or deleted in switches by the controller. SDN alone does not make the data plane verification problem easy. It makes data collecting much easier so that we can verify the data plane in real time by monitoring rule updates. The details of data plane verification are shown in Section V.

Control logic in SDN is logically centralized in the controller, in contrast to distributed protocols in devices in traditional networks. Therefore, we introduce control plane verification work in traditional networks and in SDN networks. In SDN networks, the task can be converted into how to verify the program code of the SDN controller and applications. Therefore, formal verification techniques, such as model checking, theorem proving and symbolic execution, which are widely used in code verification, can also be used to verify SDN programs. In traditional networks, due to the complexity of routing protocols and their interactions, it is not an easy task to verify all aspects of a configuration. Some standard formal verification methods (e.g., model checking and theorem proving) cannot be applied to this verification task. For example, due to the huge state space of the distributed devices in network, verification via model checkers is not scalable (also known as state explosion problem). Because of the complex logic of the control plane, the network invariants also cannot be proved by standardized theorem provers.

Unlike prior configuration analysis work that customizes models for limited aspects of the configuration and checks specific correctness properties, some work has made progress in configuration verification. Batfish [94] can simulate the behavior of all distributed protocols to obtain the forwarding tables; then, it verifies the network properties with a data plane verifier. ARC [95] abstracts the control plane as abstract weighted graphs. ERA [96] and Minesweeper [89] analyze the control plane via the abstraction of a special data structure: the route record. This is significant progress; however, scalability is an open problem in this field. The details of control plane verification in traditional networks are shown in Section VI-A.

Although SDN can be tailor-made to suit the user's needs, its programmability amplifies the opportunity for errors. The control plane in SDN networks is the program of the centralized controller and applications. These verification works can be classified into verifying the SDN programs and developing verified controller. First, certain tools (e.g., Kuai [49] and Veificare [36]) check the SDN program using model checkers. In contrast, VeriCon [54] uses first-order logic to formulate the network and invariants, and it then proves the correctness of the program. Second, various verification-friendly languages have been proposed for SDN controllers (e.g., Flowlog [50] and NATKAT [58]). They can prevent problematic rules from being installed on the data plane. The details of control plane verification in SDN networks are shown in Section VI-B.

Data plane verification and control plane verification cannot be used to determine if there is a hardware failure or congestion problems. Network testing is complementary to network verification. Network testing tools detect these types of errors by systematically generating packet probes. They observe whether the actual forwarding violates the intentions of the network operator. In SDN networks, testing focuses on checking the behavior of each individual behavior of each switch, as the controller has an overview of each switch. Monocle [97] and RuleSope [98] were developed to check the flow tables in the switch. In traditional networks, testing probes are usually designed in an end-to-end manner. For instance, ATPG [99] can check both failure rules and performance failures such as congestion. The details of data plane testing are shown in Section VII.

Most control plane testing works have focused on the control plane in SDN networks. Although some efforts have been made to analyze and verify configurations, no related work on control plane testing in traditional networks has been performed. Most control plane testing approaches have been proposed to detect design and implementation errors. Certain white-box testing approaches (e.g., NICE [25]) assume that the sources code of the controller can be obtained. They extract formal models from the source codes and perform model-based testing to find faults. Because these methods depend on the application's source codes, they cannot test controllers developed in languages that they do not support. To address this challenge, various black-box testing approaches have been proposed. For instance, Yao *et al.* [100], [101] described the system behaviors with parallel component models in a novel manner. The details of control plane testing in SDN networks are given in Section VIII.

V. Techniques for Network Data Plane Verification

This section provides a detailed description of data plane verification techniques. Compared with the control plane, the data plane has well-understood semantics and reflects the combined impact of all configuration aspects. The data plane does not need to unify diverse configuration languages from different vendors (e.g., Cisco and Arista) or model dynamic behaviors across various protocols. Fig. 9 summarizes a high-level image of data plane verification. Given a topology and network data plane snapshot, it derives a logical formula that models the entire network. It then verifies logical formulas derived from the specified invariants. The invariants specify the correctness conditions of the forwarding behavior in the network, including the absence of loops, packet reachability and bidirectional forwarding.

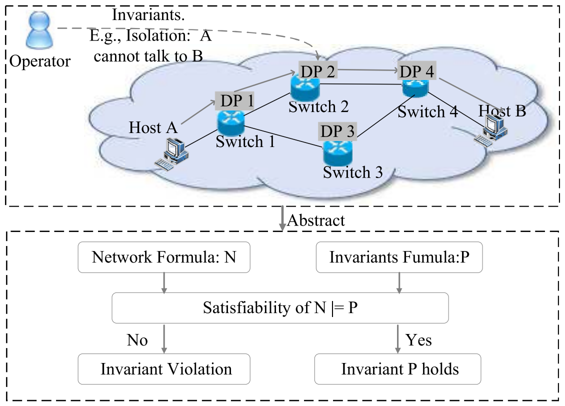


Fig. 9. Workflow of data plane verification. DP refers to data plane (forwarding information bases). The FIBs and user's invariants are collected. Then they are expressed with formal formulas. With novel algorithms, we can verify whether the network model stratify these invariants.

For data plane verification, the only difference between traditional and SDN networks is the data collection process. When verifying traditional networks, we can collect FIBs through SNMP, terminals, or control sessions. For SDN networks, we can obtain the forwarding rules by monitoring the rules that are inserted, modified or deleted with a layer controller and switches.

A. Static Analysis and Off-Line Verification

Significant effort has been made to verify the actual behavior of a network through formal analysis of data plane states. In early research, such tools collected the FIBs and then verified the properties off-line. If the network state violated an invariant, these tools gave counterexamples to help users find the errors. Anteater [9] realized the first verification system of the data plane based on the reachability algorithm proposed by Le *et al.* [102]. FlowChecker [51], NetSAT [62], and HSA [6] are also based on similar ideas. These techniques indeed make data plane verification of widespread interest.

1) Verification Based on SAT Solver: Anteater [9] was the first data plane verification system used to detect errors in networks including ACLs, VLAN tags, etc. First, it obtains the data plane snapshot by collecting forwarding tables. Second, the operator defines the invariants (e.g., forwarding, connectivity, or consistency) to be checked against the network. Third, it encodes the snapshot into Boolean expressions and translates the invariants into SAT instances. Then, Anteater checks the invariants with an SAT solver. The system derives a counterexample (e.g., a packet header or FIB entries) to assist in diagnosis once violations occur. Anteater finds various bugs such as forwarding loops caused by configuration errors in the campus network and packet loss in router software (Quagga).

NetSAT [62], proposed by Zhang and Malik, is another approach that reduces the data plane verification problem to an SAT problem. NetSAT also provides a framework for modeling networks and verifies a set of network invariants. Compared with Anteater, it achieves better performance. For instance, loops are caught in Anteater by making two copies of every switch and checking if each switch can reach its copy. Each switch must be checked separately, which is extremely inefficient. In contrast, NetSAT builds a single formula and checks the loop invariant. NetStat is more modular because it separates the network from the invariant formula.

However, both Anteater and NetSAT have limitations. First, if the FIBs dynamically change while being collected, an inconsistent data plane snapshot will be produced, resulting in false positives. Second, if the network suffers from reachability failures, it is difficult to collect the FIB snapshots. Third, these systems suffer from poor scalability and minimal time efficiency problems. For instance, Anteater spends two hours checking three standard invariants in the campus network. In addition, these systems only provide a single counterexample if an invariant violation occurs, which makes it difficult to locate which rule leads to the violation.

2) Verification Based on Model Checking: Flowchecker [51] applies model checking to SDN network. In fact, Al-Shaer *et al.* have been working on firewalls networks with model checking for 10 years [103]. FlowChecker encodes forwarding rules into Boolean expressions. In addition, network invariants are expressed with Computation Tree Logic (CTL). FlowChecker uses the binary decision diagram (BDD) to model a state machine that encodes an inter-connected network of OpenFlow switches behavior, where BDD is the data structure that can compress the state space. In addition, then it uses symbolic model checking tool NuSVM [28] to check network invariants. Although Flowchecker could check the correctness of the configuration deployed by new protocols, it suffers a serious scalability problem. Evaluation shows that it can be only applied to the small-scale network.

3) Verification Based on Symbolic Simulation: Unlike the Anteater, HSA [6] proposed by Kazemian *et al.* can find all counterexamples, when a violation occurs. HSA is a vendor-independent and protocol agnostic verification framework. It is a novel methodology that combines formal methods (model checking, symbolic simulation) with network domain features together [5]. First, packet headers are modeled as subsets of a geometrical space. Each bit in the geometrical space is commensurate with one dimension in space and does not have a protocol-specific (associated) meaning. Second, the network topology is modeled using a topology transfer function γ. Meanwhile, the networking boxes (e.g., routers and firewalls) are abstracted as transfer functions γ on sets of headers. ϕ consists of an ordered set of rules. HSA computes reachability sets from A to B by composing ϕ and γ along all paths. It can check against network invariants (e.g., reachability failures, routing loops and slice isolation) with several algorithms. However, when the head space is 80 bits, the cost of exploring the head space state of 2^{80} packets is huge. HSA then uses some optimizations based on the cube compression technique to mitigate the state space explosion issue.

4) Verification Based on SMT Solver: Frameworks such as Anteater and HSA cannot detect non-direct network violations (such as OpenFlow's *Set-Field* or *Goto-Field* modification actions) or perform stateful monitoring. FLOVER [63] extends these techniques. FLOVER models and verifies intermediate actions (e.g., 'goto' and 'set' actions). FLOVER demonstrates the conformance of flow rules against network invariants. It translates flow tables into many Yices (an SMT solver) assertions. Finally, it detects whether they are inconsistent with network invariants.

5) Verification Based on Symbolic Execution: Dobrescu *et al.* proposed a verification tool [52] that checks whether a software data plane satisfies invariants (e.g., bounded-execution, crash-freedom, and filtering). It is challenging to check these invariants in general software; the problem is even unsolvable with existing tools. Dobrescu *et al.* sidestepped the challenge by combining symbolic execution and compositionality with domain optimization. As long as the pipeline meets some standards, it can be utilized to validate the entire data plane of the network.

6) Conclusion on Off-Line Verification: The above-mentioned approaches statically analyze network data plane snapshots. Anteater makes data plane verification practical by converting it into an SAT problem. HSA [9] addresses static checking using functional simulation. FlowChecker [51] encodes flow tables into BDDs and uses model checking to verify security invariants. FLOVER [63] leverages the ability of Yices to efficiently verify flow rule sets. Table VII compares these tools.

Frameworks such as Anteater and NetSAT provide a single counterexample if an invariant violation occurs. In contrast, HSA can find the full set of failed packet headers, which is useful for detecting failures. A counterexample is detrimental because operators analyze the invariant and the error for the network location. Outputting all counterexamples makes it easier to locate which rule leads to the counterexample under HSA. For example, if an ACL erroneously drops packets that are sent to 192.168.0.0/16, reachability fails with a counterexample (e.g., 192.168.1.200). We need substantially more insight to detect all dropped packets. If it outputs the set of packets being dropped, it can suggest the ACL bug more directly (e.g., packets with destinations that match 192.168.* .*). HSA can output all counterexamples because HSA moved beyond finding reachability predicates to finding reachability sets and for arbitrary protocols.

These tools find problems after bugs occur in the data plane, which may potentially damage the network. Despite this, these tools could play a similar role as post-layout verification tools do in hardware design but in networks.

B. Real-Time Online Verification

Because of rule insertion and deletion performed by protocols, networks change over time. Previous tools are not adequate for checking the correctness of every network update such as the migration of fast virtual machines. The structure of traditional networks is highly decentralized and operates in an entirely distributed manner. Therefore, it can be impossible to obtain the FIBs in real time. Fortunately, in SDN networks, we can obtain the forwarding rules by monitoring the rules that are inserted, modified or deleted with a layer controller and switches. However, it has been proved in [65] that packet

TABLE VII
SUMMARY OF REPRESENTATIVE PROJECTS OF OFF-LINE VERIFICATION

Project	***Technology***	***Counterexamples***	***Comparison***
Anteater [9]	SAT Solver	One	Cannot scale to large networks.
NATSAT [62]	SAT Solver	One	More modular than Anteater.
FLOVER [63]	SMT Solver	One	Models *set* and *goto table* action commands.
Dobrescu et al. [52]	Symbolic Execution	One	Combines symbolic execution with domain knowledge.
HSA [6]	Symbolic Simulation	All	Provides a protocol-agnostic verification framework.

filters make reachability checks NP-Complete. The problem of checking the properties in real time is more difficult. However, some efforts have been made to verify the data plane in real time, therein attempting to obtain a quick response to failures.

1) Heuristic Verification Based on Equivalence Classes: VeriFlow [91], [109] is the first verification system that can check network invariants within a few hundred microseconds. It can be viewed as an improved version of Anteater. VeriFlow observes state changes between the control plane and the switches. Therefore, a new rule can be verified before it is installed in the switch. First, the network is sliced into equivalence classes (ECs), in which packets experience the same forwarding actions throughout the network. Second, it builds individual forwarding graphs for every EC. Then, it traverses these graphs to check invariants. When the network changes (e.g., a forwarding rule is inserted), a very small number of ECs are affected. VeriFlow searches rules via a trie structure and updates the graph. However, it is not appropriate to verify the network of multiple controllers because of the difficulty in obtaining a complete view of the network state.

2) Verification Based on Incremental Computation: NetPlumber [107] is the most closely related work with VeriFlow. It incrementally checks for compliance of state changes. It models the network box as the node and establishes a dependency graph, the Rule Dependency Graph (RDG), between the rules. Network invariants are equivalently converted into reachability assertions. Once the network changes (a message goes through a network box), it updates the corresponding network dependency graph and redoes all forms of verification. Upon detecting a violation, NetPlumber blocks the change. The change must be determined by a rule, and adding a rule does not significantly affect the forwarding rule equivalence class. Moreover, it provides a policy query language, FlowExp, which is similar to FML. Therefore, it can not only define simple invariants (packet loss and loops) but also support flexible policy definitions.

However, NetPlumber takes a long time to process link access and does not apply to networks with frequent link changes. VeriFlow and NetPlumber achieve similar runtime performance. Similar to VeriFlow, NetPlumber is protocol independent and can additionally verify arbitrary header modifications, including rewriting and encapsulation.

3) Verification Based on Equivalence Classes: Atomic Predicates Verifier [104] (AP) is more efficient than previous tools (e.g., NetPlumber and Veriflow) for network verification. Packet filters are represented as a set of predicates. Yang and Lam developed a novel algorithm to calculate atomic predicate sets. Each predicate can be expressed as the disjunction of atomic predicates, which speeds up computation. Atomic predicates can be stored as integers, and the disjunction is computed as the union of integer sets. Therefore, packet sets can be calculated quickly, with atomic predicates being the minimum. Redundancy in forwarding and ACL is eliminated by AP Verifier. In particular, AP encodes the state model with BDDs, and the use of BDDs is more efficient than BDDs in other tools (e.g., FlowChecker [51]).

4) Verification Based on MapReduce: Although tools such as Anteater and Veriflow can find forwarding errors, they cannot scale to large data centers. In data center networks, the forwarding state makes it difficult to obtain an accurate snapshot (e.g., different routing processes update their switches using unsynchronized clocks). An inaccurate data plane snapshot can result in false positives. To address these challenges, Zeng *et al.* [105] proposed a new method for verifying very large networks, called Libra. On the one hand, Libra records network event streams from routing processes and provides an algorithm to capture table and consistent data plane snapshots in large-scale networks. On the other hand, it substantially improves scalability by exploiting the scaling properties of MapReduce. It reduces the verification task into smaller, parallel computations (e.g., partitioning based on switches or subnets). For the evaluation, Libra can verify the forwarding behavior of a network with 10,000 switches in less than a minute by harnessing 50 servers. This is the beginning of distributed computing for network verification.

5) Verification Based on Symmetry: Previous checkers face difficulties verifying large data centers with millions of routing rules. Most data center networks are highly regular by design. Plotkin *et al.* [106] proposed a new method, network transformations, to verify large networks by exploiting these regularities. In other words, the network snapshot and the invariants to be verified are transformed into simpler versions. If the transformed invariant is valid in the transformed network, the original invariant is valid in the original network. Network transformations consist of network symmetry and network surgery. It exploits the domain structure w of packet headers, packet locations, and rules distributed in devices. Irrelevant or redundant headers, rules, or ports are "sliced" away with network surgery. Experiments show that this technique speeds up the verification task in a large data center network of 100,000 virtual machines by 65x. The time needed to calculate the all-pair reachability of the virtual machines was reduced from 5.5 days to 2 hours.

6) Verification Based on Similarity: Most tools exploit two observations: (i) only small parts of a network tend to be affected by typical changes to the data plane, and (ii) many different packets tend to share the same forwarding behavior across the entire network. Delta-net [108] shows how to

TABLE VIII
SUMMARY OF REPRESENTATIVE PROJECTS FOR REAL-TIME VERIFICATION

Project	***Optimizations***	***Comparison***
AP [104]	Computes equivalence classes	Exploits a heuristic strategy to discover invariance violations.
Libra [105]	Computes via MapReduce	Leverages distributed computing in network verification.
Plotkinet al. [106]	Exploits symmetries	Exploits regularities in rules and topology (not headers).
VeriFlow [91]	Computes equivalence classes	Exploits optimal strategy to discover invariance violations.
NetPlumber [107]	Exploits incrementally	Verifies only small parts of model which is caused by rule changes.
Delta-net [108]	Exploits similarity	Exploits similarity among forwarding behaviors of packets.

effectively exploit the “similarity among forwarding behavior of packets through parts of the network”. The method is proposed as the first provably amortized quasi-linear algorithm to do so. In experiments with SDN-IP, Delta-net checks a rule insertion or removal in 40 microseconds on average; this is a 10x improvement over previous tools.

7) Conclusion on Real-Time Verification: The above-mentioned approaches can verify the network data plane in real time. Table VIII shows that they perform optimizations from various aspects, including (I) incremental computation. A single rule (e.g., an ACL) change does not change the network state significantly. NetPlumber [107] leverages that fact to perform incremental computation. It only performs small modifications to incorporate the rule change. (II) Equivalence Classes. Although the verification complexity is proportional to the number of headspace and forwarding rules, the number of header equivalence classes is small [110]. Khurshid *et al.* [91] and Yang and Lam [104] leveraged this fact. They proposed two strategies (a heuristic strategy in Veriflow [91] and an optimal strategy in AP [104]) to obtain the equivalence classes. (III) Symmetries. Plotkin *et al.* [106] observed that many rules and boxes are repeated. For instance, some backup routers have many redundancy rules. They exploit symmetry to increase the verification speed. (IV) Similarity. Delta-net [108] effectively exploits another characteristic, the similarity among the forwarding behaviors of packets through parts of the network, rather than its entirety. (V) Distributed computing. These approaches, except for NetPlumber and Libra [105], assume centralized computing. NetPlumber introduces a ‘rule clustering’ technique to enhance scalability. Libra is more efficient and is based on MapReduce, therein scaling linearly with both rules and subnets.

C. Optimization via Adding Functionality

The above-mentioned tools all assume that the forwarding behavior is defined by the control plane. However, networks contain not only routers but also middleboxes (e.g., caches and stateful firewalls) whose forwarding behavior is dependent on the previous traffic. In other words, the forwarding behavior of the middleboxes can be altered by the previously transmitted traffic. Fig. 10 shows a usage of reflexive ACLs. Traffic is allowed if it belongs to a TCP connection initiated by an internal host. The above-mentioned verification tools, which only take FIBs into consideration, cannot verify a stateful data plane with middleboxes. In this section, we introduce optimization techniques based on adding verification functionality such as the support of a stateful data plane.

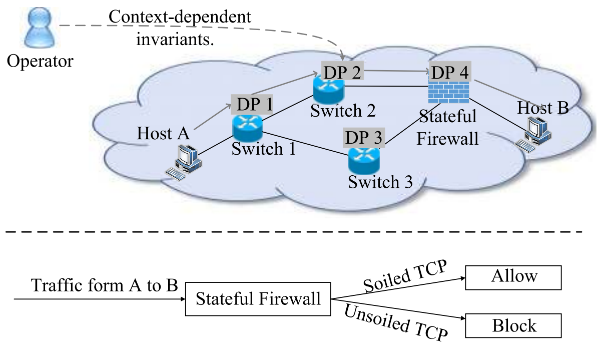


Fig. 10. Invariants in stateful networks [59]. DP refers to data plane (forwarding information bases). The invariant here specifies that only traffic belonging to a TCP connection initiated by host *B* inside the department (i.e., if traffic context is “solicited”) be allowed. The state of stateful firewall depends on the history traffic.

1) Verification Based on the Datalog: Existing tools, including Anteater [9], VeriFlow [91], and NetPlumber [107], assume fixed packet headers and forwarding rules. However, new mechanisms continuously result in new forwarding behaviors (e.g., adding VXLAN). Lopes *et al.* [65], [111] proposed an approach for automatically verifying dynamic networks, called Network Optimized Datalog (NoD). This approach uses a Datalog to specify network invariants and model network forwarding behaviors. Compared with the regular expression language, Datalog is more expressive. By adding new rules, NoD can check dynamic networks (e.g., add new packet headers without changing the internal structure of the tool). However, the Datalog engine suffers from poor scalability. To address this challenge, NOD modifies the Z3 implementation by adding new optimizations (e.g., a new combined Filter-Project operator).

The SecGuru [112] tool is an early version of NOD, deployed in the Microsoft production cloud Azure. It precisely encodes policies (policies and semantic diffs) and the analysis questions as bit-vector logic formulas. A key design aspect of SecGuru is an algorithm for compactly enumerating symbolic diffs. SecGuru requires a database of predefined common beliefs. These policies rarely change, which makes it usable as a regression test. It can also be used as a regression test suite before network policies are deployed. SecGuru proactively detects and avoids policy miscalculations that lead to security and availability issues. Each check in SecGuru takes 150-600 ms in Azure.

2) Verification Based on SMT Solver: Velner *et al.* [98] address the complexity of verifying safety invariants in stateful networks. Reasoning about reachability policies is

TABLE IX
SUMMARY OF REPRESENTATIVE PROJECTS THAT OPTIMIZE DATA PLANE VERIFICATION BY ADDING FUNCTIONALITY

Project	***Technology***	***Optimization (Support for)***	***Comparison***
NOD [65]	Optimized Datalog	Dynamic networks and imperfect specifications	Nearly as fast as HSA but substantially more expressive.
VMN [92]	SMT Solver	Stateful boxes	Extends verification to stateful boxes such as NAT.
SymNet [61]	Symbolic execution	Dynamic networks and stateful boxes	More expressive and faster than NOD.
Sung et al. [113]	SMT Solver	Quantified invariants	Extends verification to support non-Boolean invariants.

TABLE X
SUMMARY OF REPRESENTATIVE PROJECTS FOR DATA PLANE VERIFICATION

	HSA [6]	Anteater [9]	NetPlumber [107]	VeriFlow [91]	Delta-net [108]	NOD [65]	VMN [92]	SymNet [61]
Scalability	High	Low	Middle	Middle	High	Low	Low	High
Online	×	×	√	√	√	√	×	√
Coverage								
Packet reachability	√	√	√	√	√	√	√	√
Packet header changes	×	√	√	√	√	√	√	√
Modeling								
Model independence	√	√	√	√	√	×	√	√
Model language	Impressive	Impressive	Impressive	Impressive	Declarative	Declarative	Declarative	Impressive
Expressiveness								
Routers	√	√	√	√	√	√	√	√
Dynamic tunnel	×	×	×	×	×	×	√	
NATs	×	×	×	×	×	√	√	√
IP fragmentation	×	×	×	×	×	×	×	×
Accuracy								
False positives	√	√	√	×	×	√	√	√
False negative	√	√	×	√	√	√	√	√

undecidable because of unbounded ordered channels among middleboxes. They proved that checking reachability policies is EXPSPACE-complete in a stateful network.

Generally speaking, the code of most commonly deployed middleboxes cannot be obtained by the operators. Standard techniques cannot be naively used to analyze the middleboxes. For example, model checking a simple middlebox for very simple invariants would not scale. Panda *et al.* [92], [114] proposed a new approach that extends the benefits of verification to networks with middleboxes, called Verification for Middlebox Networks (VMN). It uses a simple abstract forwarding model and abstracts packet classes to model middleboxes. The forwarding models can typically be derived from a general description of the middlebox's behavior. VMN has some significant realizations: (I) It first separates the middlebox classification and middlebox forwarding behaviors. Then, it models network behaviors with logical formulas and generates other formulas corresponding to the network invariants. VMN then uses Z3 [70], an SMT solver, to check whether network invariants hold. (II) To make the verification scalable, it takes advantage of network topology symmetry and policy symmetry. Since the results depend on network symmetry, they cannot be applied to all types of networks. Moreover, it is based on the SMT solver, whose state space grows exponentially as the scale of the network increases. These problems reduce the verification speed.

3) Verification Based on Symbolic Execution: SymNet extends verification to handle mutable datapath elements [61], [115]. First, it generates an abstract data plane model encoded in SEFL models for the network configurations (e.g., router tables, firewalls and arbitrary Click modular router configurations). Then, it injects symbolic packets with the symbolic execution technique. To ensure that the model accurately represents real code, it performs fidelity testing. SEFL represents the novelty of the work, which incorporates numerous features (e.g., built-in map data structures, bounded loops, and dedicated path control instructions). SymNet can verify the packet header memory safety of networks with complex functionalities (e.g., dynamic tunneling, encryption, and stateful processing). It also scales to networks with thousands of prefixes. However, it can only be applied to certain middleboxes, which can produce issues.

4) Conclusion on Adding Functionality: The approaches in this section improve the verification performing by adding functionality. Table IX compares these approaches based on different characteristics: (I) NOD [65] applies to dynamic networks and the definitions of the incomplete properties and supports a richer high-level attribute definition based on the Datalog language while simultaneously increasing the verification speed. (II) Work at Berkeley and CMU extends analysis to stateful rules such as NAT. Panda *et al.* [92] used an SMT solver to verify stateful networks. SymNet's middlebox model [61] is similar to the proposal of Panda. Symnet uses optimized symbolic execution to verify dynamic and stateful networks. (III) Sung *et al.* [113] proposed methods to verify a few quantization invariants with the CoS configuration. Although these approaches represent significant progress, problems, such as the lack of verification of the network performance invariants, remain.

D. Quantitative Invariants Verification

The above-mentioned verification work focuses on the Boolean properties of the network such as reachability. The reachability property is very important to ensure the correctness of network functions. However, performance properties

TABLE XI
SUMMARY OF CHALLENGE AND SOLUTIONS IN NETWORK VERIFICATION

Challenge	***Solution***	***Project***
Cannot verify in real time	Verifies by equivalence class	VeriFlow [91], NetPlumber [107]; ddNF [116]
Cannot verify stateful data plane only	Verifies by adding functionality	NOD [65], VMN [92], SymNet [61]
Cannot verify routing computations	Performs control space analysis	Batfish [94], ARC [95], era [96]
Cannot verify router implementation faults	Exploits data plane tester	ATPG [99] (in Microsoft clouds)

such as bandwidth, latency, and the rate of packet loss, are also important. Performance property verification is beyond the scope of the aforementioned verification tools. Existing tools (e.g., NOD [65], VMN [92] and SymNet [61]) can only verify Boolean invariants. To verify performance properties, some efforts have been made to support the verification of quantitative invariants.

Sung *et al.* [113] extended verification techniques to support quantitative invariants. The popular service differentiation in Virtual Private Networks (VPNs) drives the business. Configuration is error prone because of the complex nature of Class of Service (CoS) configurations. Sung *et al.* propose an approach to find all class of service treatments (e.g., marking, queues, and rates) for any flow *F* for which a user queries. Similar to reachability, the technique accumulates QoS actions in a path. Using a novel formal representation of policies, *rule-sets*, it can support the analysis of the quantitative properties of an arbitrary set of flows. BDDs are leveraged to represent access control lists to improve performance. They also develop a prototype to analyze CoS configurations. Although it represents progress in quantitative property verification, it is limited to the analysis of CoS configurations and scales poorly. SLA-verifier [93] uses a quantitative model of the network. Their model describes not only how forwarding rules move the packets but also performance operations such as how to compute the end-to-end performance metrics of the packets. Using a set of algorithms, it supports verifying performance properties on the flow space and states. It converts the verification task into the analysis of a graph, which can increase the speed of verification.

Although various works represent significant progress, the verification of quantitative properties remains very challenging. Because the actual performance of the traffic is heavily dependent on the actual complex states of the network, the analysis of the quantitative properties with the models, which are modeled manually and coarsely, may be imprecise.

E. Conclusion on Data Plane Verification

In Table X, we provide a comparison of the most well-known data plane verification tools. These tools have a few desirable characteristics: (I) Coverage. The properties that can be verified include reachability and packet modifications. (II) Expressiveness. This describes the ability to analyze network functionalities. For example, HSA cannot model NAT, and SymNet [61] cannot model IP fragmentation. (III) Ease of modeling. This describes the difficulty in generating models among tools. SEFL develops an imperative language, which is easy to use. (IV) Model independence. NOD [65] does not have this property. (V) Scalability. Can the technique scale to enterprise and operator networks? HSA [6] and SymNet scale very well on optimized models. (VI) Real time. Tools such as HSA and Anteater cannot perform checking in real time. NetPlumber [107], Veriflow [91] and Delta-net [108] perform checking in milliseconds. Delta-net [108] incrementally maintains a compact representation about the flows of all packets in the network. Existing tools cannot simultaneously achieve all goals. Moreover, tools may lack expressiveness or do not scale to large networks (e.g., Anteater). HSA [6] and NOD [65] can capture reachability in stateless networks. Among these techniques, NOD [65] is the most complete tool for network models and constraints. These techniques have achieved useful but limited results. For a small university network, it took approximately 1 day to verify reachability for all stations [94]. Some hardware failures had no impact on the FIB, which cannot be checked by these data plane verification tools. The approaches in Table XI can facilitate the checking of hardware failures, quantitative metrics (e.g., bandwidth), etc.

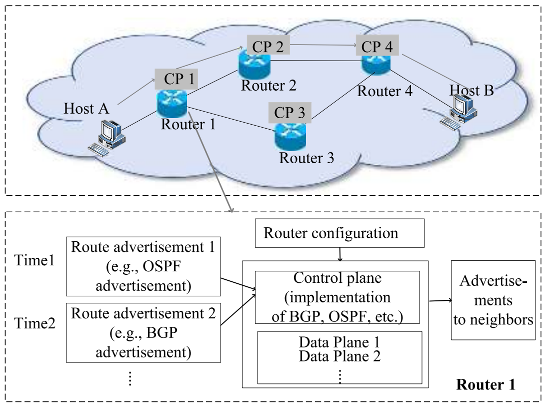

CP 1
CP 2
CP 3
CP 4
Host A
Router 1
Router 2
Router 3
Router 4
Host B
Router configuration
Time1
Route advertisement 1 (e.g., OSPF advertisement)
Time2
Route advertisement 2 (e.g., BGP advertisement)
Control plane (implementation of BGP, OSPF, etc.)
Data Plane 1
Data Plane 2
Advertisements to neighbors
Router 1

Fig. 11. CP refers to control plane. For example, the control plane on R1 is the program that takes configuration files and the network environments (i.e., route advertisements) and generates data planes.

VI. Techniques of Network Control Plane Verification

Data plane verification cannot proactively prevent errors before the configuration is deployed to the network. Once the problem is found in the data plane, the operators still need to find the configuration snippets corresponding to the problem. The relationship between forwarding behaviors and configuration fragments are really complex. On the other hand, the snapshot of the data plane is always changing. When verifying the data plane, we are required to perform calculations by repeatedly collecting data plane information, as depicted in Fig. 11. Therefore, moving from data plane verification to control plane verification is significant.

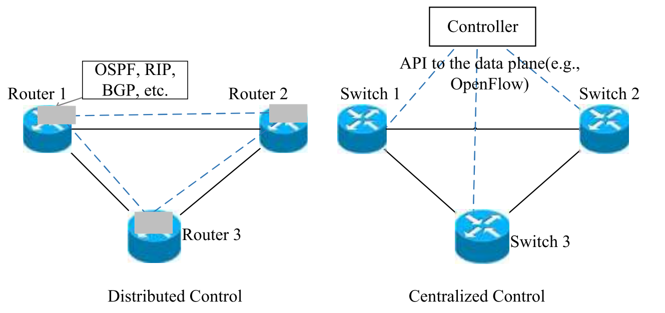


Fig. 12. Control plane in traditional networks: distributed in configuration files. Control plane in SDN: logically centralized in controller and applications. Dark lines are the data information. Blue lines are the control information.

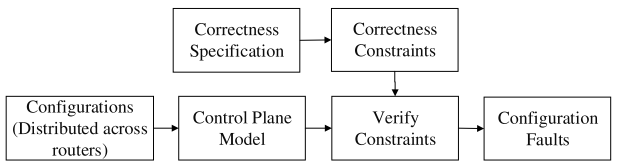


Fig. 13. Workflow of control plane verification in traditional networks. Configurations in the router define which protocol it can use, what is the link cost, which neighbor should be sent route advertisements etc. Follow the RFCs of routing protocols, the control model is built.

The control plane is the program that integrates the network topology and link state information and establishes the forwarding table of the data plane. As shown in Fig. 12, in traditional networks, the control plane is the implementation of the algorithm protocols distributed in thousands of network devices. While in SDN networks, the control plane is concentrated in the controller and the applications on top of it. In traditional networks, certain standard formal verification methods (e.g., model checking) cannot be used for this verification task. For example, due to the state explosion issue facing distributed devices, verification via model checkers is not scalable. The control logic in SDN is logically centralized to the controller via the state of the program. Because formal verification techniques such as model checking, theorem proving and symbolic execution have made great progress in software code verification, they can also be used to verify SDN programs with certain optimizations. Therefore, we categorize related work into two branches: (i) control plane verification in traditional networks and (ii) control plane verification in SDN networks.

A. Control Plane Verification in Traditional Networks

Traditional networks continue to be dominant. The implementation of a network permits a great deal of flexibility through its various modes of operation. Configuring a network is difficult because of the complex policy requirements and low-level configuration languages. Certain critical errors may arise only during failures, e.g., when upgrading network protocols or replacing legacy devices or because of the hardware interconnections between different vendors [117], [118]. Verifying network configurations can avoid errors before the configurations are deployed onto the network.

It is challenging to detect configuration faults. Govindan *et al.* and Griffin-Wilfong *et al.* theoretically proved that BGP loops are possible [119]–[121]. Gao-Rexford found sufficient conditions for avoiding loops [80], and Le *et al.* [102] proved that route redistribution can cause loops. However, these results are strongly based on human analysis and theoretical proofs. Some recent tools make control plane verification for traditional networks work. Fig. 13 shows the general process of control plane verification in a traditional network. The inputs of control plane verification are the topology environment information of the link failure and configuration files such as the route announcement of the configuration.

1) Verification Based on Static Analysis: The router configuration checker (rcc [122]) was the first static analysis tool that can automatically detect faults in BGP configurations in real networks. rcc synthesizes and represents the configuration in a unified form. It is difficult to define high-level correctness specifications for BGP, and specifications must can be tested against actual configurations. rcc addresses this challenge and defines two high-level aspects of correctness: path visibility and path visible faults. Route validity is whereby a route correctly propagates routes for existing, usable IP-layer paths. Path visibility denotes that routers learn routes that exist in the network. It uses these specifications to derive constraints, and it enables users to detect network-wide configuration faults, which can improve the Internet routing infrastructure.

2) Verification Based on Datalog: Existing configuration analysis tools, including rcc, develop customized models for specific protocols. Therefore, the scope of what can be checked is limited. Batfish [94] overcomes this challenge. It verifies control plane configurations in four steps. First, it takes the network configuration file as input; then, the inputs are encoded as a set of logical facts. Next, it generates a logical control plane model in LogiQL [37]. With environments (each link status and route announcement), it executes the LogiQL program. Batfish can obtain the data plane model. The third step enables users to check the data plane model. Batfish employs NoD [65] to perform data-plane analysis. Finally, it simulates the behavior of counterexample packets via the data plane model. This step can help operators understand property violations and repair the network configuration.

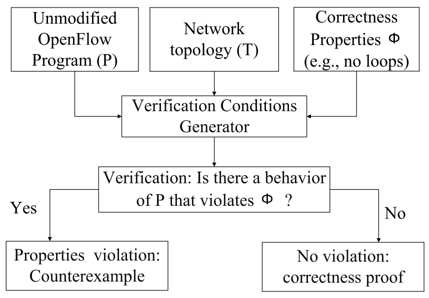
Verifying a simple configuration file with Batfish [94] takes nearly two hours because it requires as much simulation of the network links as when it generates the data plane. In other words, Batfish [94] spends a substantial amount of time on the conversion process from the configuration file to the data plane forwarding logic. Nevertheless, its input remains a subset of the actual topological environment. Furthermore, the control plane model in Batfish [94] does not consider modifications of the MPLS, resource competition etc.

3) Verification Based on Abstract Interpretation: Batfish [94] performs proactive configuration analyses by faithfully deriving the data plane. However, detailed data plane generation is never necessary. On the one hand, proactive analysis tasks often do not require the paths themselves. Some invariants focus on the existence of paths or depend on path sets taken. On the other hand, network protocols only interact in certain ways in data centers. By leveraging the above factors, Gember-Jacobson *et al.* developed a new abstract representation, ARC, for control planes [95]. This technique consists of a series of weighted digraphs, and these digraphs are network-protocol independent. It abstracts routing protocols and captures the collective impact on routing and forwarding. The technique then verifies key invariants by computing simple graph characteristics in polynomial time. ARC uses algorithms to accurately model common mechanisms (ACLs, ECMP and static routes) and protocols (e.g., OSPF, RIP and eBGP) used in data center networks. Thus, ARC is an order of magnitude faster than previous tools, e.g., Batfish [94].

4) Verification Based on BDD: Control plane verification faces two challenges: expressiveness (capturing the diverse behaviors of routing network protocols) and scalability (exploring the model with respect to the environment). To address these challenges, efficient reachability analysis (ERA) [96] builds a model for the network control plane and uses a repertoire of techniques for scalable exploration of this model. ERA first designs a unified abstraction, route, which succinctly captures the diverse behaviors of various protocols and their interactions. With a route announcement, a router function produces route announcements for its neighbors. It then uses BDDs [123] to compactly represent the route announcements and shrinks the representation using the equivalence classes of the announcements [104]. ERA also uses some techniques (e.g., K-map [124], with VXM2 [125]). Consequently, it takes a few seconds to identify bugs in various polices that can be turned into reachability relationships (e.g., valley-free routing) for a network with more than one thousand routers. However, ERA cannot detect reachability errors in transient states or convergence errors.

5) Verification Based on SMT: Minesweeper [89] has both high network design coverage in that it works for a large collection of network protocols, features and topologies as well as high data plane coverage in that it can verify a large number of properties for all possible data planes that might emerge from the control plane. It uses a graph-based model, where rich logical constraints on its edges and nodes encode all possible interactions of route messages. It encodes the stable states of a network as a satisfying assignment to an SMT formula. To improve the scalability, they also propose combinational search (not message set computation). It also designs a range of highly effective optimizations (e.g., slicing and hoisting) that reduce the number of constraints in the formulas. Users use Minesweeper on a collection of real and synthetic configurations, therein showing that it is effective at finding issues in real configurations and can scale to large networks.

6) Conclusion on Control Plane Verification of Traditional Networks: This section focuses on the configuration-based paradigm. Although SDN has become popular, many networks remain configuration based. The approaches in this section can be used to successfully find errors proactively before the configuration is deployed. Similar to the tools in [90], [126]–[128], rcc [122] only focuses on a single routing protocol. ARC [95] focuses on a limited set of routing protocol features. It cannot capture multiple routing protocols or complex features. In contrast, Batfish [94] can simulate the behavior of individual protocols to obtain the data plane, although this process is computationally expensive. To address the scalability challenge, ERA [96] provides a route abstraction based on BDD. It can scale enterprise networks with hundreds of devices. Minesweeper [89] achieves both high network design coverage and high data plane coverage while remaining sufficiently scalable to enable the verification of many real-world networks. These tools require users to provide assumptions on the environment; they thus can produce false positives.



Unmodified OpenFlow Program (P)
Network topology (T)
Correctness Properties Φ (e.g., no loops)
Verification Conditions Generator
Verification: Is there a behavior of P that violates Φ ?
Yes
No
Properties violation: Counterexample
No violation: correctness proof

Fig. 14. Verification of SDN Program via code verification. The code of applications, and topology and user's invariants are collected. When they are expressed with formal formulas, we can verify whether the model stratify these invariants.

B. Verification of SDN Program

In SDN networks, the behavior is determined by the controller, which enables users to easily verify network invariants. Furthermore, SDN is distributed and asynchronous in nature, making it easy to lead to inconsistencies [131]. Routing software developed by Cisco and Juniper has been tested for many years in real-world networks [5]. It is not wise for user-defined programs to be blindly deployed in networks [132]. Recent research on SDN control plane verification can be classified into two branches: (i) verifying the SDN programs and (ii) developing verified controllers.

The SDN architecture facilitates innovation and the development of applications. Applications running over the controller manage network devices through the uniform northbound interface. They specify network policies, e.g., routing and access control policies. Similar to normal software, these control programs may suffer from design or implementation errors. Therefore, significant effort has been invested in checking the correctness of the applications. Fig. 14 shows the workflow of the verification on application code. They verify or prove whether the program satisfies the properties with formal methods.

1) Verification Based on Model Checking: Numerous works apply finite state model checking to check whether programs are correct in SDN networks. Skowyra *et al.* [129], [130] proposed Verificare, a verification tool for SDN-enabled applications. They developed the Verificare Modeling Language (VML), which combines the characteristics of SDN networks. VML is more suitable to modeling SDN networks. First, the network model defined in the VML is compiled into LTS [124]. The model and network invariants to be verified are translated into logical formulas. Then, the invariants can

TABLE XII
SUMMARY OF REPRESENTATIVE PROJECTS IN SDN PROGRAM VERIFICATION

Project	*Technology*	*Based on*	*Comparison*
Verificare [129, 130]	Model checking	SPIN [27]	Proposes the Verificare modeling language.
Sethi et al. [48]	Model checking	Murphi [53]	Handles arbitrary numbers of packets.
Kuai [49]	Model checking	Murphi [53]	Combines partial order reduction and abstraction.
VeriCon [54]	Theorem proving	Z3 [70]	Guarantees the absence of errors. However, it is not automatic.

be verified using various model checkers (e.g., SPIN [27] and Alloy [29]) automatically. Finally, Verificare outputs traces leading up to the violation. Unlike earlier SDN-verification tools, Verificare models both SDN controllers and applications compositionally.

Sethi *et al.* [48] presented abstractions for SDN controllers that can perform model checking on an arbitrary number of packets. Existing approaches that verify SDN controllers can only handle a small number of packets. To address the state explosion issue, they first construct a data state abstraction. This abstraction significantly reduces the model size. Although it can be used to verify larger topologies, it cannot scale to data centers.

Kuai [49] is a distributed enumerative model checker for SDNs. The controller model and invariants to be verified are written in Murphi [53]. Kuai implements a counter-abstraction for tracking. This abstraction enables users to check the system with finite states. The evaluation shows that partial order reduction techniques can reduce state spaces by many orders of magnitude.

2) Verification Based on Theorem Proving: In general, applying the above-mentioned scaling methods based on model checking to large networks is challenging. Clients can generate packets in an unbounded manner, and these packets can be processed in arbitrary interleaved orders. Thus, the state space remains unbounded, and the topology is fixed with several clients. Moreover, these methods cannot prove the absence of errors. VeriCon [54] addresses the challenges outlined above. This method verifies whether an SDN program is correct for all possible sequences of network events and all admissible topologies. It first expresses programs in a simplified imperative event-driven programming language, CSDN, that manipulates relations. The method specifies admissible network topologies and network invariants with first-order logic. It then implements classical Floyd-Hoare-Dijkstra deductive verification via the Z3 SMT solver. It quickly outputs a concrete counterexample if an invariant violation occurs. These two choices guarantee that verification conditions are simple enough to be expressible, which enables VeriCon to rapidly verify invariants.

3) Conclusion on SDN Program Verification: As show in Table XII, much effort has been invested to applying formal methods (e.g., state model checking and theorem proving) to check whether SDN applications behave correctly. Verificare [129], [130], and Kuai [49] leverage finite-state model checking to check programs. SDN applications are modeled on the state transition system of the events (e.g., two links connect). These approaches, in general, suffer two main problems. First, scaling these tools to large networks is highly nontrivial. These approaches cannot handle infinite-state SDN programs because of the state space explosion problem. Second, although they can be easily used, they cannot prove the absence of errors. In contrast, VeriCon [54] provably verifies network invariants of programs at compile time. It symbolically reasons about potentially infinite network states and verifies whether network invariants are preserved. We note, however, that VeriCon relies on user-provided invariants.

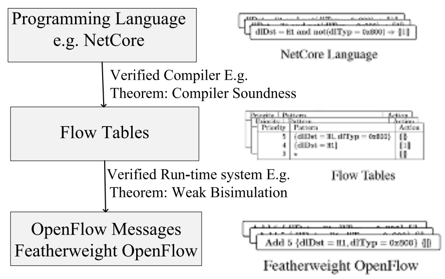


Fig. 15. Workflow of a verified controller [57]. The compiler translates programs written into low-level packet-processing rules. Then we can verify the generic network invariants before rules are installed into switches.

C. Verified Controller in SDN Network

Rather than verifying SDN programs, some researchers have developed a verified SDN controller with the help of formal methods. These approaches check whether the controller correctly installs rules with a formal specification and a detailed operational model of an SDN. The verified controllers are generally developed based on high-level programming languages. Fig. 15 shows the workflow of a verified controller. It supports verifying properties at runtime in the compiler. Because the programming language supports invariant verification, they can avoid problematic rules being installed on the switches.

1) Programing Languages That Supports Theorem Proving: Guha *et al.* [57] designed and implemented the first machine-verified SDN controller, called NetCore. They formalized a detailed operational OpenFlow model using the Coq proof assistant. They developed a verified compiler with this model. The controller is established correctly once and for all, which obviates the need for run-time or post-hoc verification as in most tools. It can prove that the system is absent of bugs because the behavior of the application is verified in Coq. NetCore gives network programmers robust static guarantees backed by machine-checked proofs against a foundation's model.

Chen *et al.* [133] proposed another approach to programming and verifying SDNs. The approach is based on Network Datalog (NDLog), a declarative language that provides an

TABLE XIII
SUMMARY OF REPRESENTATIVE PROJECTS IN VERIFIED CONTROLLERS

Project	*Technology*	*Based on*	*Comparison*
NetCore [57]	Theorem proving	Coq [56]	Develops a model for OpenFlow and formalizes it in the Coq.
NetKAT [58]	Theorem proving	Coq [56]	Develops a sound program logic based on Kleene Algebra.
NDLog [133]	Theorem proving	Coq [56]	Develops a program logic to verify the NDLog program.
Flowlog [50]	Model checking	Alloy [29]	Enables cross-tier verification via a tierless language.

encoding of network functionalities. NDLog can encode certain SDN applications succinctly. To verify invariants of the NDLog program, they proposed a sound program logic. Furthermore, the system verify invariants compositionally.

NetKAT [58] is another programming language for SDNs. It is based on Kleene algebra with tests (KAT), a sound equational theory. It consists of primitives for filtering and transmitting packets, composition operators, and a Kleene star operator that iterates programs. NetKAT is a synthetic technique for checking reachability. It can also ensure isolation between programs by proving non-interference properties. NetKAT's denotational semantics describe network programs as functions from packet histories. The equational theory in NetKAT is complete and sound with respect to the model. Programmers can create compositional and expressive network programs. They can also reason about their semantics (e.g., reachability, traffic isolation, and access control) effectively.

2) Programming Languages That Support Model Checking: Flowlog [50] is a stateful rule-based language for SDNs. It is a tierless language that simplifies SDN programming. It performs SDN verification in a cross-tier manner. Flowlog is reminiscent of both SQL and rule-based languages and supports programming with mutable states. Flowlog enables the minimum amount of necessary traffic between switches and the controller. It can verify several properties (e.g., topology-independent properties).

3) Conclusion on Verified Controller: There is an increasing trend in programming languages to support network verification [50], [57], [58], [76], [133], [134]. In addition to the work mentioned in Table XIII, FatTire [76], Flog [76] etc. can also transform a program into verifiable code. The main insight of these approaches is that the compiler and run-time system translate programs written in this language into low-level packet-processing rules. These methods then verify the generic network invariants before rules are installed into the switches. Because its behavior is verified using formal tools, it establishes the correct controller once and for all, obviating the need for run-time or post-hoc verification, as in most tools. However, these languages are ad hoc, and the invariants supported by each language are different. Furthermore, a language is limited to its own controller and cannot be used for the verification of other controllers.

D. Conclusion on Control Plane Verification

Configuration verification can flag errors proactively. We provide a comparison of the most well-known control plane verification tools in traditional networks in Table XIV. These tools have a few desirable properties: (I) Model expressiveness and model tractability. These tools reference various behaviors of various protocols. A naive expressive model is impractical to explore. Batfish [94] derives fully declarative logical models of the network's control plane. Although it is expressive, it is too complex to explore. For most tools, generating the data plane is ineffective. On the other extreme, ARC [95] offers a high-level control plane model and is orders of magnitude faster than Batfish. However, it cannot capture many properties of the control plane. (II) Scalability. Scalability represents scalable control plane exploration to identify violations. (III) Accuracy. We analyze the accuracy based on the false positive or false negative metrics. (IV) Generic. Being generic means that it checks whether the feature of general protocols (e.g., BGP and OSPF route advertisements) are captured. rcc [122] develops customized models for specific configuration aspects. This selective focus limits its checking scope. ERA efficiently checks reachability in certain large symbolic environments (e.g., the environment with all possible eBGP advertisements). Minesweeper [89] can verify configurations in all environments. Finally, we note that no approaches in configuration verification can detect bugs in router hardware and software.

Control plane verification in SDN networks has made significant achievements: (I) Various systems (e.g., Kuai [49] and Veificare [130]) use model checking to check SDN programs. They model the program as a state-transition event system. The main challenge to these approaches is the number of messages. Furthermore, they cannot guarantee an absence of errors. In contrast, VeriCon [54] uses first-order logic to model networks and invariants. It can potentially handle infinite topologies, and it can verify the absence of errors. However, the Hoare-style verification in VeriCon requires inductive invariants. (II) Many SDN programming languages have been developed (e.g., Flowlog [50] and NetKat [58]). References [135] and [136] introduced abstractions for programming controllers. The compiler of the high-level language in NetCore [57] generates semantically equivalent code. Reference [137] defined a declarative language to ease the task of verifying SDN programs. Language abstractions can prevent problematic rules from being installed on the data plane, which is orthogonal in verifying SDN programs. As discussed earlier, SDN control plane verification has achieved great progress. However, the task of verifying complex properties remains challenging because of the highly dynamic nature of SDN network.

VII. TECHNIQUES FOR NETWORK DATA PLANE TESTING

Data plane verification can ensure that the data plane respects the actual policy. Control plane verification can help check configuration errors and problematic controller codes. However, implementation bugs in switch failures can still

TABLE XIV
SUMMARY OF REPRESENTATIVE PROJECTS OF CONTROL PLANE VERIFICATION IN TRADITIONAL NETWORK

Project	***Model input/output***	***Expressiveness***	***Scalability***	***What it does***
rcc [122]	Actual protocol's messages	Low	Low	Checks common BGP faults (mostly syntactical).
Batfish [94]	Actual protocol's messages	High	Low	Computes the data plane for one BGP environment.
ARC [95]	Protocol-agnostic I/O	Low	High	Reasons a rich class of BGP operators across all failures.
ERA [96]	Route as a compact bit vector	Middle	Middle	Reasons across a subset of maximal environments.
Minesweeper [89]	Route as a compact bit vector	Middle	Middle	Reasons across all environments to find bugs.

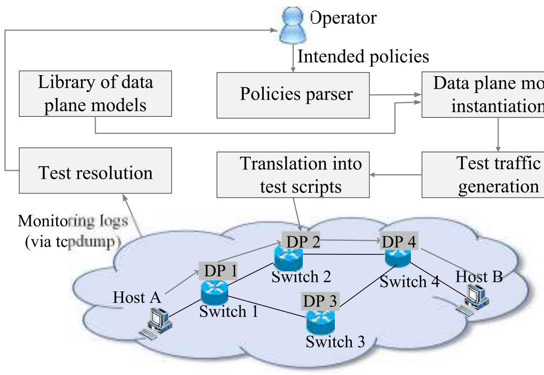


Fig. 16. Workflow of data plane testing. With the information of data plane, they systematically find test inputs that trigger certain behaviors of the data plane with model.

manifest in obscure manners. Network testing can address the above challenges by systematically probing packet generation. This concerns the correctness of the entire network system, e.g., whether the network implementation violates the intentions of the network operator, whether there are hardware failures, or whether there are congestion problems.

SDN networks allow the controller to configure and communicate among different underlying devices. In SDN networks, data plane testing work focuses on checking the behavior of each behavior of each switch, as the controller has an overview of each switch. In traditional networks, it can be impossible to monitor or collect data on every switch. Therefore, testing probes are usually designed in an end-to-end manner.

Fig. 16 shows a general workflow of data plane testing in traditional networks. The data plane testing tool first collects the topology information and the forwarding states. With the information, it constructs data plane model and generates the test packets which satisfies the testing policies input by the operator. Finally, it periodically sends the test packets by the test terminals and monitors the network. Once an error is detected, the testing tool reports to the operator.

A. Stateless Data Plane Testing

In stateless data planes, the results of packet forwarding depend on the current router or switch rules, regardless of historical packets. There has been much work on testing stateless data planes. We classify these work into three branches: white-box testing, black-box testing and gray-box testing.

1) White-Box Testing Based on HSA: Automatic Test Packet Generation (ATPG) [1], [99] is designed to discover inconsistencies between forwarding tables and actual forwarding states. It first collects the topology information and the forwarding states, including FIBs and ACLs. ATPG generates a device-independent model of the data plane with the approach in Header Space Analysis [6]. Then, it computes a minimum abstract test packet set to maximally check each rule/link with that model. Finally, it periodically sends the test packets by the test terminals and monitors the network. Once an error is detected, ATPG triggers a mechanism to localize the fault. It complements but also extends previous tools that can detect performance faults or can only locate errors based on faulty results [99].

Klee [60], in symbolic execution and programming languages, is closely related to work on ATPG. Klee attempts to simply find a minimal test to cover code lines, and ATPG proves minimal packets to traverse each link/rule. Unlike Klee, ATPG can also find links or queues that cause performance problems. Klee solves constraints with an SMT solver. ATPG simulates the forward path of a packet, and it is more effective.

2) White-Box Testing Based on SAT Solver: Monocle [97], [138], [139] verifies whether the data plane satisfies the view of the controller in an SDN network. First, the verification of forwarding tables can be formulated as an SAT problem. Then, probe packets targeting a particular rule are systematically probed in the switch data plane. To improve the scalability, Monocle optimizes the conversion of the constraints. To minimize overhead, it formulates and solves a graph vertex coloring problem. Monocle also provides more details on how the SAT solution is translated into a real packet. Therefore, in dynamic networks with frequent flow table changes, it can detect a misbehaving rule of the data plane.

In addition to Monocle [138], RuleScope [140] checks the priority failures of flow tables in addition to missing faults. In line with established systems [138], RuleScope inspects the forwarding behavior through probing. Generating probe packets and processing probing results are challenging tasks. Monocle solves this problem by reducing the probe generation to the SAT problem. It also uses algorithms to detect and troubleshoot rule faults at limited computational costs. It leverages packet tracing tools (e.g., NetSight [141]) for probing. However, RuleScope cannot address rule updates. Moreover, rules are distributed frequently in SDN networks, thereby restricting the deployment of RuleScope.

3) Gray-Box Testing Based on Planned Tomography: Various solutions (e.g., ATPG [99]) require details of the configuration. However, sometimes, we can only obtain coarse forwarding information about the network, for example, multipath configuration without knowledge of router-internal hash functions. To address this challenge, NetSonar [142] detects performance problems with only coarse information of the system. NetSonar is a system that generates test packets to probabilistically cover the network (e.g., changing paths and multipath routing). It uses planned tomography, with input coming from a novel test technique that maximizes component coverage while minimizing probing overhead. By computing probabilistic and diagnosable covers, NetSonar utilizes partial forwarding information and addresses the remaining uncertainty. By computing probabilistic path coverage, it addresses nondeterminism in multipath routing. NetSonar localizes faults with minimal test overhead via diagnosable link covers. It has been deployed in a Microsoft inter-data center network. It is a first step toward building tools that balance what can be known about networks with their unknowns.

4) Black-Box Testing Based on Performance Measurements: Pingmesh [143] is a system that detects networks with inter-server network latency measurements as an indicator. To reduce overhead, it measures inter-server latencies at three levels (top-of-rack switch, intra-data center and inter-data center). Pingmesh creates complete latency graphs at each level. Its controller creates the list of servers to ping, and its agents at each server collect the latency data. The analysis of the data can provide an increase in performance over historical baselines. Pingmesh uses a simple technique to diagnose complicated problems in data center networks. Pingmesh can also detect network SLA violations, packet drops, black holes etc. and locate the source of the network problem. However, simply measuring latencies in Pingmesh cannot be used to detect certain types of general problems (e.g., identifying a faulty spine switch).

5) White-Box Testing Based on Symbolic Execution: A key component to networks is switches. In SDN networks, the rapidly changing OpenFlow specification can lead to multiple implementations that behave differently. Kuzniar *et al.* proposed SOFT [144] to test the interoperability of the implementation of OpenFlow agents running on the switch. It first leverages symbolic execution to explore each agent in isolation. Then, it can derive which set of inputs causes which outputs. To improve the scalability of symbolic execution, SOFT combines symbolic and concrete inputs. It then cross-checks all distinct behaviors across different switch agents, and it uses a constraint solver to calculate the common input subset that causes inconsistent behaviors. However, it requires the cooperation of various equipment manufacturers to access the source code.

6) White-Box Testing Based on Model Checking: NOT NICE [145] uses a method to verify whether the OpenFlow switch satisfies the OpenFlow specification [146]. It first presents a generic OpenFlow switch model based on first-order logic. It then leverages Alloy to model and verify the properties of OpenFlow switches. Alloy transforms constraints into Boolean formulas and then solves them using an external SAT solver. It can support a set of invariants (e.g., non-loop forwarding), but it inherits the shortcomings of model checking (e.g., state space explosion).

7) Black-Box Testing Based on Formal Model: Previous approaches rely on the source code, which is often difficult to obtain. To address this challenge, Yao *et al.* presented a black-box testing method [147] to test the data plane of an SDN network. They defined a pipelined extended finite state machine model. The model adds extended finite state machines (EFSM) with communication channels and shared variables. It provides an easier model for the multilevel pipelines in switches.

8) Black-Box Testing Without Formal Models: OFTest [148] uses thousands of Python test cases for OpenFlow specifications (e.g., OpenFlow 1.3). All test cases are developed manually with each OpenFlow switch version. It cannot achieve a good coverage of the system behavior because the test case generation is not based on formal models. Similarly, OFLOPS [149] performs performance testing for OpenFlow switches (e.g., the device is heavily loaded). It can discover inconsistencies between the control plane and the data plane that can be undetectable in other approaches. It tests the capacity and bottlenecks of forwarding equipment, the consistency of flow tables, the granularity of flow spaces, and the type of message modification. OFTEN [150] is a systematic testing framework for network behavior. It combines a state-space exploration technique, model checking, with the execution of the actual switch. It checks whether there exists any erroneous conditions via the interaction of the controller execution and real switches. Kuzniar *et al.* [151] tested the flow table update rates of hardware OpenFlow switches. Compared with other approaches, it tests along many different dimensions (e.g., rule installation latency). It also reports several new types of anomalous behaviors, which advances the understanding of the OpenFlow switch performance.

9) Conclusion on Stateless Data Planes: This section introduces data plane testing techniques, as shown in Table XV. Although Monocle [97] and RuleSope [98] are more fine grained than ATPG [99], neither can be used to perform performance testing similar to ATPG. ATPG takes more time to generate the monitoring probes. Monocle can observe the switch reconfiguring data plane in dynamic networks. Based on Monocle, RuleScope can also check whether a rule's priority is correct. Beyond these ideas, other methods introduce techniques for testing OpenFlow switches, as switches are the key component in the network data plane. For example, NOT NICE [145] checks whether the implementation of the switch conforms to the specifications from a model checking perspective. Yao *et al.* [101], [147] proposed a black-box testing method for switches with a formal model. However, they check each individual switch each time. To ensure the correctness of the entire data plane, operators need to test the switches one by one, which is inefficient.

B. Stateful Data Plane Testing

Complex context-dependent policies are implemented in stateful networks (e.g., some packets need to be sent to an intrusion detection system). Unfortunately, early approaches to network testing faced challenges in handling such scenarios. Therefore, in this section, we introduce work that can support testing for the data plane with middleboxes.

1) Black-Box Testing Based on Symbolic Execution: BUZZ is a model-based testing framework for stateful data planes. It first instantiates an expressive-yet-scalable data plane model

TABLE XV
TOOLS FOR STATELESS DATA PLANE TESTING

Project	*Based on*	*What it checks*	*Model with source code*
ATPG [100]	HSA [6]	Failed rules	√
Monocle [98]	SAT Solver	Missing rules	√
RuleScope [99]	SAT Solver	Incorrect rules	√
NetSonar [145]	-	Failed rules/links	√
Pingmesh [146]	-	Performance errors	-
BUZZ [59]	Klee [60]	Errors in Middlebox	×
SOFT [147]	Cloud9 [146]	Error codes in switch	√
NOT NICE [148]	Alloy [29]	Errors in switch	√
Yao et al. [150]	PITSv3 [158]	Error codes in switch	×
OFTest [151]	-	Error codes in switch	-

via a traffic unit abstraction: the BUZZ Data Unit (BDU). Network functions can be modeled as an ensemble of finite-state machines. Abstract test traffic is generated to trigger the policy. It optimizes symbolic execution to reduce the scope and number of symbolic variables. Then, it develops custom translation mechanisms to translate abstract test traffic into concrete test traffic. However, the model process in BUZZ is complex and error prone, as it is built based on expert knowledge. To address this challenge, NFactor [152] uses a method to automatically extract the model from the source code using program analysis techniques. Furthermore, the symbolic execution in BUZZ cannot completely cover all network states. In addition, SFC-Checker [93], [153] tests the validity of the stateful data plane from other aspects.

2) Conclusion on Stateful Data Plane Testing: This section presents the techniques used to test networks with middleboxes. BUZZ uses symbolic execution to generate packets to test whether a network enforces an invariant. The work most similar to BUZZ is ATPG, which does not capture stateful behaviors. We analyze the work from the following aspect. (I) Modeling stateful data planes. FlowTest [155], SymNet [61], and VMN [92] model stateful behaviors. FlowTest's models do not scale beyond 4-5 node networks. Works such as FlowTest, Symnet, and VMN are different from BUZZ in terms of both techniques and goals. (II) Model synthesis. The model used in BUZZ is generated by hand. NFactor generates models from the source code automatically via program analysis [156], [157]. (III) Completeness and soundness. BUZZ favors soundness over completeness. However, it still manually models the data plane based on expert knowledge, and it cannot handle middleboxes containing uncertain internal states.

C. Conclusion on Data Plane Testing

This section introduces techniques for data plane testing. As shown in Table XV, ATPG [99], NetSonar [142] and Pingmesh [143] can all be used to test the network performance (e.g., network congestion and network delay). In particular, Pingmesh is suitable to large data centers, and the results are more accurate. ATPG and NetSonar need detailed network forwarding information, which is possible to obtain in small and medium-sized network. RuleScope [140] is the finest grained among ATPG, Monocle and RuleScope. BUZZ [59] can uncover policy violations in stateful data planes. SOFT [144], NOT NICE [145], Yao *et al.* [101], [147], OFTest [148], etc. can test the correctness of the data plane switches. To check the entire data plane, they need to check each switch entity one by one, which is not efficient, especially for large networks. Although data plane testing has made great progress, current testing methods cannot handle large networks (e.g., data centers) well. In the future, white-box or black-box fuzzy testing technology may help alleviate this problem.

VIII. TECHNIQUES FOR NETWORK CONTROL PLANE TESTING

SDN controls a network using a central software program, which can alleviate the risk of bugs. Theses errors in the software can result in erroneous network behaviors. In the networking community, there is burgeoning interest in network control plane testing. Up to now, most control plane testing works have focused on the control plane in SDN networks. These approaches have attempted to detect design and implementation errors in SDN programs. Some approaches extract formal models from the source codes and perform model-based testing to find faults. Other approaches attempt to test the gram in a black-box manner. In addition, despite some effort being been made to verify configurations, no related work on control plane testing has been done for traditional networks. We classify the works for SDN control plane into two branches: white-box testing and black-box testing.

1) White-Box Testing Based on Model Checking: As is well known, model checking can efficiently verify the correctness of distributed systems. However, it suffers from the state explosion problem, which makes it difficult to check SDN programs. Yakuwa *et al.* [158] proposed a novel method: SDPOR-DS. They reduced the state space (e.g., by orders of network events) using dynamic partial-order reduction (DPOR) [159]. They also proposed a symbolic state transition model. The execution time of this method is less than that of state-of-the-art tools. NICE [25] performs symbolic execution and model checking to systematically explore the network state space (the controller, the switches, and the hosts). NICE applies the symbolic execution of event handlers to exercise code paths of applications. To reduce the state space, NICE simplifies the switch model and uses effective strategies (e.g., generating event interleavings). NICE efficiently uncovers eleven bugs in real applications.

2) White-Box Testing Based on Automata: The behaviors of SDN applications depend intimately on the absolute and relative timings of inputs. To systematically handle time, DeLorean [160] uses a new technique to systematically explore the behavior of control programs. It models SDN programs with timed automata (TA) [161]. To avoid the state-space explosion problem, they reduce the number of regions by reducing the number of clock variables. They explore a program as multiple, independent control loops and predict the response of the program to certain events. TA-based model checking has been widely used in the real-time-testing community [38], [123], [162], [163]. To our knowledge, DeLorean is the first case that applies TA to SDN programs.

3) Black-Box Testing Without Formal Model: Scott *et al.* [164] proposed a method that triggers a given bug with a

TABLE XVI
TOOLS IN CONTROL PLANE TESTING

Project	*Expressiveness*	*Scalability*	*Implementation errors*	*Model with source code*
NICE [25]	High	Low	√	√
DeLorean [160]	High	Low	√	√
STS [164]	Low	High	√	×
Taste [166]	Middle	Middle	×	×
Yao et al. [100]	High	High	√	×
Armageddon [167]	Low	High	√	-

sequence of responsible inputs. They designed and implemented a system: the SDN Troubleshooting System (STS). STS first leverages fuzzy testing to randomly inject interfering network events (e.g., data packet transmission and link failures) into the simulation network [165]. Then, it checks the network invariants (e.g., end-to-end reachability and a lack of forwarding black holes) based on Hassel [6]. STS can test multiple controller platforms without any requirements on their source codes. However, it cannot systematically cover the behaviors of the applications because of the absence of formal models.

Encouraged by the success of testing techniques in distributed systems, Shelly *et al.* [167] designed and implemented Armageddon [168], which introduces systematic chaos into SDN networks. Armageddon introduces failures (e.g., some ports being down) without violating network invariants. Because failures need to guarantee some concept of coverage (e.g., complete coverage), Armageddon uses efficient algorithms to compute failure scenarios. Armageddon induces the failures in production networks and monitors whether the SDN controller can sustain these failures. A controller is resilient if it sustains all failures. When testing real-world networks, Armageddon achieves maximum link coverage within only a few iterations. However, it is challenging to compute what failed and when it failed.

4) Black-Box Testing With Formal Model: To model the SDN network more efficiently, Lebrun *et al.* [166] defined a new language named data path requirement language (DPRL), which can define arbitrary constraints on a data path. It extends the FML language and adds a regular grammar definition. They utilized DPRL to describe the data path requirements of the data paths and generate corresponding test packets. They transform DRPL statements into automata and check whether test packets are accepted by the corresponding automata. They also implemented a prototype tool to track the data paths followed by test packets and check whether requirements are satisfied. However, the coarse-grained model limits the testing capability. To overcome this challenge, Yao *et al.* [100] described the system behaviors with parallel component models in a novel manner. They modeled an application as component models, where data are passed. This approach is more efficient than [166], as its model can describe the details of the SDN applications.

5) Conclusion on Control Plane Testing: Many studies contribute to white-box testing with formal models for SDN applications. Some approaches based on while-box testing (e.g., NICE [25] and DeLorean [160]) assume that the sources code of the controller can be obtained. Most controllers are open source; this assumption is always true for open-source controllers. They extract formal models from the source codes and perform model-based testing to find faults. As shown in Table XVI, they are usually expressive because of the formal models. Because these methods depend on the application's source codes, they cannot test controllers developed in languages that they do not support. Because of the massive numbers of packets and network events, such methods always suffer from the state-explosion problems. To address the scalability challenge, these methods always perform domain optimization. For instance, Yakuwa *et al.* [158] optimized their model checking with dynamic partial-order reduction.

Other methods use black-box testing methods for SDN applications. Reference [164] presented a black-box fuzzy testing method that can test multiple controller platforms. Their method avoids the state-space explosion problem via testing without formal models. However, it cannot ensure systematic coverage due to the absence of formal models. To improve the coverage, others have proposed black-box testing methods with formal models. For instance, Lebrun *et al.* [166] presented the formal language DPRL to describe the requirements of data paths and to generate test packets.

The above methods model or simulate networks to detect errors via model checking, symbolic execution, or unit testing. However, they suffer from certain limitations. For example, they can only detect a subset of possible errors. They always run control software in a virtual environment (e.g., Mininet [176]). However, the behavior of an actual device is often quite different from that of a virtual device. Many errors (e.g., race conditions) can only be found when the applications are being deployed on real hardware [144]. Despite these limitations, these methods remain useful for detecting bugs and errors in SDN networks.

IX. Relationship Between Network Verification and Network Testing

Network verification is useful when initially designing a network. It can be used to evaluate whether designs uphold desirable invariants. However, verification results hold if and only if implementations are correct (e.g., protocol implementations in switches). Network testing is complementary to network verification, which can check errors in implementations. However, testing cannot prove the absence of bugs in a network. The completeness and soundness of testing results are determined by the method used, test conditions etc. Combining a verification tool such as VMN [92], [114] (HSA [6]) with a testing tool such as BUZZ [59] (ATPG [99]) allows users to guarantee network reliability more efficiently.

A. Data Plane Verification and Testing

Data plane verification has been well studied. This technique verifies network snapshots via reductions to SAT, model checking or symbolic simulation. It is intrinsically a state machine verification technique. Verifying reachability in a finite network is PSPACE-complete. The techniques used in Anteater [9] and HSA [6] work well in practice. They exploit domain optimization of the network, which reduces the verification problem to being closer to NP-complete. These

TABLE XVII
DATA PLANE VERIFICATION VERSUS DATA PLANE TESTING

	Data Plane Verification	***Data Plane Testing***
Input	Data plane snapshots: FIBs + Topology	Data plane snapshots: FIBs + Topology
What it does	Constructs formal models from network snapshots and checks network invariants against the model	Generates test cases with a data plane model or no model and detects hardware failures or performance errors
Technique	Reduces to Datalog/SAT or Model checking	Model-based testing; Fuzzy testing
Tools	Anteater [9] NOD [65] VMN [92] SymNet [61] etc.	ATPG [99] Monocle [97] RuleScope [98] BUZZ [59] etc.
Comparison	Data plane testing can check implementation errors (e.g., broken links and crashed devices), which cannot be checked by verification	

TABLE XVIII
CONTROL PLANE VERIFICATION VERSUS CONTROL PLANE TESTING

	Control Plane Verification		***Control Plane Testing***
Supported network	Traditional network	SDN network	SDN network
Input	Configuration + Environment	Applications + Controller	Applications + Controller
What it does	checks the correctness of configurations	Verifies SDN programs	Generates test cases from source codes or models
Technique	Static analysis, graph analysis	Model checking, theorem proving	Finite-state model checking, model-based testing
Tools	Batfish [94] ARC [95] etc.	VeriCon [54] Kuai [49] etc.	NICE [25] Yao et al. [100] etc.
Comparison	Control plane testing can find implementation bugs in applications. However, it may discover bugs too late with run-time overhead.		

TABLE XIX
SOME EXAMPLES OF NETWORK BUGS AND ERRORS THAT CAN BE DETECTED

Examples	***Description***	***Symptom***	***Checking Tool***
Errors in traditional networks			
Upgrade errors due to new devices	A router is problematically introduced to offload traffic [9].	Forwarding loop	Data plane
Lack of default routes in routers	Due to a manual configuration, there is a lack of default routes in routers [9].	Packet loss	Data plane
Blocking traffic toward unused IP	Due to network evolution, a user does not update the ACLs [9].	Forwarding loop	Data plane
Software errors in routers	After receiving BGP updates, Quagga 0.96.5 cannot update the routing tables [9].	Packet loss	Data plane
Errors in network maintenance	A problematic configuration is accidentally created during switch testing [99].	Forwarding loop	Data plane
Incorrect state migration	OpenNF [169] does not migrate the session count after hosts are removed [59].	Packet loss	Data plane
Software errors in switches	The programmer forgets to add a process's IP options in Click [52].	Forwarding loop	Data plane
Hijacked prefix in configuration	Because of an error in the configuration, some source nodes have routes to the victim's prefix through either the adversary or the victim. [94]	Multipath inconsistency	Control plane
Errors in SDN networks			
Design errors in learning switches	The program forgets to send a message when the destination is known [54].	Packet loss	Control plane
Excess flooding	Pyswitch does not construct a spanning tree for topologies with cycles [25].	Forwarding loop	Control plane
Reactive control in Kinetic	Malicious packets are incorrectly let through in a Kinetic dynamic firewall [59].	Incorrect reachability	Data plane
Host unreachable after moving	Soft timeout does not expire, resulting in an unreachable new location [25].	Forwarding blackhole	Control plane
ARP packets forgotten	Loadbalancer neglects discarding some request packets during resolution [25].	Forgotten Packets	Control plane
Inconsistent program and switch	The data structure of the program AuthNoFlowRemoval is inconsistent with the forwarding tables of the switches [54].	Packet loss	Control plane

techniques represent a good first step toward making networks more reliable. However, some tools assume that switches will process instructions emitted by the controller in sequence, even though actual switches often reorder messages. This means that the invariants that they verified do not always actually hold.

Hardware may not follow the configuration in an actual network such as ASIC errors, link failures, and link congestion. As show in Table XVII, data plane testing (e.g., ATPG [99]) can be used to discover various properties (e.g., packet loss, congestion, or faulty hardware), whereas network verification tools cannot. This technique is orthogonal to data plane verification tools since it focuses on actual network correctness. The main challenge is whether the testing process is completed without degrading performance. Both data plane verification and data plane testing are fundamentally limited. Because networks are always in constant churns (e.g., a single route advertisement results in changes to data planes [94]).

B. Control Plane Verification and Testing

Control plane verification provides more powerful analysis that can reason about more than the current "incarnation" of the network. The control logic in a traditional network is distributed in the configuration file. Current verification tools support many routing protocol features. The analysis in Batfish [94] is expensive because it simulates the routing protocols' running processes. ERA [96] uses an abstract encoding of the control plane information, making it more scalable than Batfish. However, these tools cannot check for router implementation errors. The control logic in SDN is concentrated on the controller and applications. Table XVIII shows that some methods verify SDN controllers at run time with code verification, whereas other methods use verified SDN controllers assisted by formal operational models of SDN. However, verification cannot be used to find software implementation bugs in real SDN programs. Control plane testing uses formal-model-based approaches to find implementation bugs in controller codes.

X. DISCUSSION

In this section, we discuss the lessons that we have learned from current solutions.

A. What Network Bugs Can Be Detected

In Table XIX, we list some example network bugs and errors that can be detected by the methods reviewed in the survey. It

TABLE XX
RESEARCH IDEAS VIA EXEMPLARS FROM PROGRAMMING LANGUAGE AND HARDWARE VERIFICATION

Earlier Exemplar	***Examples in Network Verification and Testing***
Ternary simulation, symbolic execution in software [170]	Header space analysis [6] proposed by Kazemian et al.
Certified development of an OS Sel4 [171]	Certified development of a controller [58] proposed by Guha et al.
Specification mining for hardware [172]	Mining for enterprise policies proposed by Benson et al. [173].
Exploit symmetry in model checking [174]	Exploit symmetry in data center verification proposed by Plotkin [106].
Google's Netflix in distributed system [175]	Armageddon [167] proposed by Nick et al.
Symbolic Execution in software [60]	Software data plane verification method proposed by Dobrescu et al [52].

briefly gives the descriptions, effects, and other details of the network bugs. As long as the forwarding behavior of the FIBs violates network invariants, it can be detected by data plane verification tools such as Anteater [9]. For instance, when upgrading a network, a router is problematically introduced to offload the traffic, resulting in forwarding loops. Some network errors in configurations, such as a lack of default routes in routers and blocking traffic toward unused IPs, can be detected by data plane verification tools. Even the software errors in routers can be checked by these tools. Data plane testing tools such as ATPG [99] can detect outage problems accidentally created during switch testing. Configuration verification tools can help detect configuration errors before deployment. For instance, because of a single hijacked prefix in a configuration, some source nodes can have routes to a victim's prefix through either an adversary or a victim. It violates the property of multipath inconsistency.

In SDN networks, verification and testing tools can be used to check for design errors in applications. For instance, Pyswitch does not construct a spanning tree for topologies with cycles, which can lead to forwarding loops. With tools such as NICE [25], we can discover the errors that violate network properties. Some consistency errors between a program and switch can also be discovered by tools such as Vericon [54]. For instance, in the program AuthNoFlowRemoval, the data structure is inconsistent with the forwarding tables of the switches. More example errors, such as host unreachable after moving, can be found in the table.

B. Ideas That We Can Discover

Various methods in other areas (e.g., software programs) have been borrowed for network verification and testing. Similar to ternary symbolic simulation in hardware verification, HSA [6] performs symbolic execution to analyze network data planes. Similar to certified operating systems, Guha *et al.* first proposed machine-verified controllers. Inspired by the symmetry in model checking, Plotkin *et al.* proposed to simplify redundancy rules in a data center network to improve scalability. Similar to real-time post-deployment failure injection tools in large-scale distributed systems (e.g., Google's Netflix), Armageddon systematically introduces failures (e.g., link failures and network failures) to test the robustness and reliability of SDN controllers. Dobrescu and Argyraki [52] leveraged symbolic execution to find errors in the data plane. Table XX lists more examples. Learning from successful cases in software and hardware verification can help us solve problems in networks.

TABLE XXI
SUMMARY OF EXPLOITED DOMAIN STRUCTURES IN NETWORKING VERIFICATION AND TESTING

Project	***Description***
	Exploit with domain knowledge
HSA [6]	Symbolic simulation by defining header-space structures.
NetPlumber [107]	Incremental verification by defining rule dependencies.
ATPG [99]	Network graph limits size of state space.
SymNet [61]	Optimized symbolic execution by defining SEPL language.
Dobrescu et al. [52]	Optimized symbolic execution with bounded loops.
NOD [65]	Network-optimized with new fused operators.
VeriFlow [91]	Incremental verification by defining Tries structure.
	Naively using methods form the verification community
FlowChecker [6]	Naively performs symbolic model checking with NuSMV [28]; Suffers from the state explosion problem.
NOT NICE [145]	Naively performs model checking with Alloy [29]; Suffers from the state explosion problem.
Anteater [9]	Naively applies an SAT solver with Yices [63]; Poor network scalability.

C. What Optimization Can We Make

As discussed earlier, network verification has many similarities to software and hardware verification. However, the naive use of formal methods suffers from problems concerning model expression and scalability. Networks have their own characteristics, which are different from software and hardware. Throughout this survey, formal methods combined with network domain features are found to be more efficient in many aspects. Table XXI shows some examples: (I) In contrast to traditional model checking [42] and binary decision diagrams [165] in hardware verification, HSA exploits the network features to provide a vendor-independent model. Note that frameworks based on naive model checking and SAT solvers are limited to providing a single counterexample when detecting a violation of the invariants. HSA outputs all packets of every pair of nodes, which is easier to do than incremental verification. (II) The idea that ATPG [99] generates the minimum test set to cover all links coming off in software testing. Rather than solving constraints using an SMT solver, ATPG directly simulates the packet forwarding path. Therefore, ATPG is 10-times faster than the naive attempt using KLEE [60]. (III) Datalog solvers produce all solutions but scale poorly. NOD [65] optimizes the Datalog implementation engine to scale to large header spaces by defining a fused Select-Project operator. (IV) SymNet employs symbolic execution on network models instead of the code itself. Previous works program the models in C [6], but models in C are too complex to analyze. To improve scalability, SymNet proposes a new language, SEPL, to define the behaviors of the data plane. However, naively applying standard program verification techniques to networks cannot help us solve domain

TABLE XXII
TOOLS AVAILABLE IN NETWORK VERIFICATION AND TESTING

Technique	***Tool and Dataset***
Symbolic execution	KLEE [60], JPF [69], SAGE [67]
Proof assistants	Coq [56], Isabelle [55]
Model checking	SPIN [27], NuSMV [28]
SAT/SMT solver	CVC4 [71], Yices [64], Z3 [70]
BDD packages	Buddy [177]
Network-specific verification	Hassel [6], VeriFlow [91], NoD [65]
Network-specific testing	ATPG [99], BUZZ [59]

problem as well. For example, FlowChecker [6] naively performs symbolic model checking based on NuSMV [28]. However, it suffers from the state-space explosion problem and scales poorly.

D. What Tools Are Available

Network verification and testing are now a well-established topic with a series of available tools, as shown in Table XXII. Some tools have been released publicly. Researchers can build their own tools with these engines. However, selecting an inappropriate tool will lead to poor verification results. A set of tools can support verification, but each tool is optimized in different respects (e.g., HOL-Light, a high-level logic theorem prover, does not support probabilistic reasoning, whereas HOL supports probabilistic reasoning). It is critical to select appropriate tools according to the given characteristics.

XI. FUTURE WORK AND OPEN QUESTIONS

Although the technologies of network verification and testing have seen progress, there are major issues confronting future research.

A. Stateful Data Plane Verification

Stateful data planes will become more popular in the future, e.g., SDN and NFV enable richer network data processing services. There is substantial research on modeling network functions [59], [61], [92]. However, most works model the stateful data plane manually based on specialized knowledge. In addition, the expressiveness and scalability of the model are poor. It is wise to automatically generate models from the source code. Therefore, we need to develop more scalable verification and testing techniques to explore stateful data planes.

B. Verification of Quantitative Invariants

Network verification has been used to check network reachability. Current technologies focus on verifying Boolean invariants (e.g., no forwarding loops and reachability). Although these invariants are fundamental, quantity invariants (e.g., latency, packet loss, and bandwidth) are also important to operators. Verifying quantitative invariants is another direction for future work.

C. Control Plane Verification in Complex Routing Scenarios

Current methods for control plane verification in traditional networks only support simple routing scenarios. They are not applicable to modeling other protocols (e.g., iBGP and MPLS TE [178]). In addition, the expressiveness and scalability of the control plane model faces many challenges. Synthesizing an expressive and scalable model of a control plane will be necessary in the future.

D. Testing Multiple Applications in SDN Control Plane

Applications in SDN work together to offer flexible network functionality. With the emergence of SDN app stores, an increasing number of SDN applications are being deployed by various control domains. Unfortunately, multiple applications can produce unintentional interference, even though each individual application may be well developed. However, current methods are designed to check the correctness of each individual application. We argue that we need systematic and automated methodologies to check the correctness of multiple SDN applications.

E. Fault Location

Once invariant violations are detected, we need to automatically locate and generate repairs (e.g., changing or adding configuration lines). Current tools for verification and testing do not facilitate the debugging and repair of detected errors. Debugging tools complement verification and testing. Current debugging tools (e.g., OFRewind [179] and Ndb [164]) are limited to specific scenarios [180]–[182].

F. Automatic Synthesis of Network Specifications

Network invariants (e.g., no forwarding loops and no forwarding black holes) define the correct behavior of a network. The understanding of different network administrators is different. As Engler noted [183], determining which specification to check is a major obstacle. These invariants are in the minds of network operators. How can one use current network verification techniques when one does not know the reachable pairs? Hence, another natural direction of future work is to automatically synthesize network specifications from the network system.

XII. CONCLUSION

Recent works have shown a growing interest in network verification and testing in academic and industrial research. The number of research projects on this subject has grown significantly since the first data plane verification work, Anteater, in 2011. Some of these tools have been deployed in real production networks (e.g., SecGuru in Microsoft Azure and Anteater in UIUC). This paper has presented a comprehensive review of this research. We categorize existing research across different conceptual planes of the network: data plane verification and testing and network control plane verification and testing. We draw the following conclusions: (i) The best techniques for reachability (data plane) verification are currently fast enough for large networks. (ii) New network features (e.g., standardizing the interfaces for network programs) can potentially provide a foundation for network reasoning. Researchers have developed and verified controllers and proposed formal methods to verify SDN program. (iii) Data plane testing could help operators find hardware failures and performance errors at minimal cost. (iv) Traditional networks remain dominant, and configuration verification can offer opportunities to find latent errors. Even techniques in network verification and testing have made great progress. However, certain open issues, including verifying stateful data planes, verifying quantitative invariants, and the automatic synthesis of network invariants, are challenging. We hope that this survey will offer readers an opportunity to rethink existing techniques for exploiting domain structures and find innovative research ideas.

Acknowledgment

The authors thank the anonymous reviewers for their suggestions.

References

[1] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, Nice, France, 2012, pp. 241–252.

[2] M. Muuss. (1983). *The Story of the Ping Program*. [Online]. Available: http://mirrors.pdp-11.ru/_vax/www.bandwidthco.com/whitepapers/netforensics/icmp/The%20Story%20of%20the%20PING%20Program.pdf

[3] *Traceroute*. Accessed: 2004. [Online]. Available: ftp://ftp.ee.lbl.gov/traceroute.tar.gz

[4] *Software-Defined Networking: The New Norm for Networks*, Open Netw. Found., Menlo Park, CA, USA, 2012.

[5] G. Varghese. Accessed: 2015. *Vision for Network Design Automation and Network Verification*. [Online]. Available: http://cseweb.ucsd.edu/∼varghese/networkdesignautomationvision.pdf

[6] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. 9th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, San Jose, CA, USA, Apr. 2012, pp. 113–126.

[7] F. Babich and L. Deotto, "Formal methods for specification and analysis of communication protocols," *IEEE Commun. Surveys Tuts.*, vol. 4, no. 1, pp. 2–20, 1st Quart., 2002.

[8] K. Bhargavan, D. Obradovic, and C. A. Gunter, "Formal verification of standards for distance vector routing protocols," *J. ACM*, vol. 49, no. 4, pp. 538–576, 2002.

[9] H. Mai *et al.*, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, Toronto, ON, Canada, 2011, pp. 290–301.

[10] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proc. 1st Symp. Netw. Syst. Design Implement. (NSDI)*, San Francisco, CA, USA, Mar. 2004, pp. 155–168.

[11] M. Muuss. Accessed: 2014. [Online]. Available: www.microsoft.com/en-us/research/project/network-verification/

[12] V. Jacobson. Accessed: 2015. [Online]. Available: conferences.sigcomm.org/sigcomm/2015/tutorial-nwverif.php

[13] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 2, pp. 156–173, 2005.

[14] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1617–1634, 3rd Quart., 2014.

[15] D. Kreutz *et al.*, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[16] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and OpenFlow: From concept to implementation," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 2181–2206, 4th Quart., 2014.

[17] H. Farhady, H. Y. Lee, and A. Nakao, "Software-defined networking: A survey," *Comput. Netw.*, vol. 81, pp. 79–95, 2015.

[18] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha, "A survey of securing networks using software defined networking," *IEEE Trans. Rel.*, vol. 64, no. 3, pp. 1086–1097, Sep. 2015.

[19] J. Qadir and O. Hasan, "Applying formal methods to networking: Theory, techniques, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 256–291, 1st Quart., 2015.

[20] K. Nichols, S. Blake, F. Baker, and D. Black, "Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers," Internet Eng. Task Force, Fremont, CA, USA, RFC 2474, 1998.

[21] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[22] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd Int. Conf. Comput.-Aided Verification (CAV)*, Snowbird, UT, USA, Jul. 2011, pp. 585–591.

[23] J. Spragins, "OSPF: Anatomy of an Internet routing protocol [book reviews]," *IEEE Netw.*, vol. 12, no. 6, p. 4, Nov./Dec. 1998.

[24] Y. Rekhter and T. Li, "A border gateway protocol 4 (BGP-4)," Internet Eng. Task Force, Fremont, CA, USA, RFC 1654, 1994.

[25] M. Canini, D. Venzano, P. Perešíni, D. Kostic, and J. Rexford, "A NICE way to test openflow applications," in *Proc. 9th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, San Jose, CA, USA, Apr. 2012, pp. 127–140.

[26] N. Gude *et al.*, "NOX: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[27] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[28] A. Cimatti *et al.*, "NuSMV 2: An OpenSource tool for symbolic model checking," in *Proc. 14th Int. Conf. Comput.-Aided Verification (CAV)*, Copenhagen, Denmark, Jul. 2002, pp. 359–364.

[29] D. Jackson, *Software Abstractions—Logic, Language, and Analysis*. Cambridge, MA, USA: MIT Press, 2006. [Online]. Available: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[30] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. Softw. Tools Technol. Transfer*, vol. 1, nos. 1–2, pp. 134–152, 1997.

[31] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[32] C. Baier and J. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[33] E. M. Clarke, "The birth of model checking," in *25 Years of Model Checking—History, Achievements, Perspectives*. Heidelberg, Germany: Springer-Verlag, 2008, pp. 1–26.

[34] D. Marker, *Model Theory: An Introduction*. New York, NY, USA: Springer-Verlag, 2002.

[35] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines—A survey," *Proc. IEEE*, vol. 84, no. 8, pp. 1090–1123, Aug. 1996.

[36] P. Norvig, "Artificial intelligence: A modern approach," in *Applied Mechanics Materials*, vol. 263, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2003, pp. 2829–2833.

[37] S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: An interactive tutorial," in *Proc. ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, Athens, Greece, Jun. 2011, pp. 1213–1216.

[38] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*. Amsterdam, The Netherlands: Elsevier, 1990, pp. 995–1072.

[39] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, 1994.

[40] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods Syst. Design*, vol. 19, no. 1, pp. 7–34, 2001.

[41] E. M. Clarke, K. L. McMillan, S. V. A. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *Proc. 8th Int. Conf. Comput.-Aided Verification (CAV)*, New Brunswick, NJ, USA, Jul./Aug. 1996, pp. 419–427.

[42] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992.

[43] K. L. Mcmillan, "The SMV system," in *Symbolic Model Checking*, vol. 38. Boston, MA, USA: Springer, 1992, pp. 161–165.

[44] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 1999, pp. 193–207.

[45] A. P. Sistla, "Symmetry reductions in model-checking," in *Proc. 4th Int. Conf. Verification Model Checking Abstract Interpretation (VMCAI)*, New York, NY, USA, 2003, p. 25.

[46] J. M. Wing and M. Vaziri-Farahani, "Model checking software systems: A case study," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, 1995, pp. 128–139.

[47] G. Singh and S. Shukla, *Model-Checking Based Verification for Hardware Designs Specified Using Bluespec System Verilog*, CiteSeerX, Jan. 2008, pp. 39–43.

[48] D. Sethi, S. Narayana, and S. Malik, "Abstractions for model checking SDN controllers," in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, Portland, OR, USA, Oct. 2013, pp. 145–148.

[49] R. Majumdar, S. D. Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, Lausanne, Switzerland, Oct. 2014, pp. 163–170.

[50] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *Proc. USENIX Conf. Netw. Syst. Design Implement.*, Seattle, WA, USA, 2014, pp. 519–531.

[51] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *Proc. ACM Workshop Assurable Usable Security Configuration*, 2010, pp. 37–44.

[52] M. Dobrescu and K. J. Argyraki, "Software dataplane verification," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Seattle, WA, USA, Apr. 2014, pp. 101–114.

[53] D. L. Dill, "The murphi verification system," in *Proc. Int. Conf. Comput.-Aided Verification*, 1996, pp. 390–393.

[54] T. Ball *et al.*, "VeriCon: Towards verifying controller programs in software-defined networks," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 282–293, 2014.

[55] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL—A Proof Assistant for Higher-Order Logic* (LNCS 2283). Heidelberg, Germany: Springer, 2002.

[56] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions* (Texts in Theoretical Computer Science. An EATCS Series). Heidelberg, Germany: Springer, 2004.

[57] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controllers," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Seattle, WA, USA, Jun. 2013, pp. 483–494.

[58] C. J. Anderson *et al.*, "NetKAT: Semantic foundations for networks," in *Proc. ACM SIGPLAN SIGACT Symp. Principles Program. Lang.*, 2014, pp. 113–126.

[59] S. K. Fayaz, Y. Tobioka, S. Chaki, and V. Sekar, "Scalable testing of context-dependent policies over stateful data planes with armstrong," *CoRR*, vol. abs/1505.03356, pp. 1–16, Jun. 2015. [Online]. Available: http://arxiv.org/abs/1505.03356

[60] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Conf. Oper. Syst. Design Implement.*, San Diego, CA, USA, 2008, pp. 209–224.

[61] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, Florianópolis, Brazil, 2016, pp. 314–327.

[62] S. Zhang and S. Malik, "SAT based verification of network data planes," in *Automated Technology for Verification and Analysis*, vol. 8172. Cham, Switzerland: Springer Int., 2013, pp. 496–505.

[63] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *Proc. IEEE Int. Conf. Commun.*, 2013, pp. 1974–1979.

[64] L. De Moura and B. Dutertre, "Yices 1.0: An efficient SMT solver," *Satisfiability Modulo Theories Competition*, vol. 54, no. 8, p. 1, 2006.

[65] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. USENIX Conf. Netw. Syst. Design Implement.*, 2015, pp. 499–512.

[66] M. J. C. Gordon, *HOL: A Proof Generating System for Higher-Order Logic*. Boston, MA, USA: Springer, 1987.

[67] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[68] N. Tillmann and J. D. Halleux, "Pex—White box test generation for .NET," in *Proc. 2nd Int. Conf. Tests Proofs (TAP)*, Prato, Italy, Apr. 2008, pp. 134–153.

[69] C. S. Păsăreanu *et al.*, "Symbolic pathfinder: Integrating symbolic execution with model checking for Java bytecode analysis," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013.

[70] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. 14th Int. Conf. Tools Algorithms Construct. Anal. Syst. (TACAS)*, Mar./Apr. 2008, pp. 337–340.

[71] T. Liang *et al.*, "An efficient SMT solver for string constraints," *Formal Methods Syst. Design*, vol. 48, no. 3, pp. 206–234, 2016.

[72] N. Een, "MiniSat: A SAT solver with conflict-clause minimization," in *Proc. Int. Conf. Theory Appl. Satisfiability Test. (SAT)*, 2005, pp. 502–518. [Online]. Available: https://ci.nii.ac.jp/naid/10027365050/en/

[73] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Proc. TOOLS Algorithms Construct. Anal. Syst. Int. Conf. (TACAS)*, 2007, pp. 632–647.

[74] C. L. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving* (Computer Science Classics). Boston, MA, USA: Academic, 1973.

[75] M. Davis, G. Logemann, and D. W. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[76] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined (NETWORKING)*, 2013, pp. 109–114.

[77] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[78] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[79] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "Buzz: Testing context-dependent policies in stateful networks," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 275–289.

[80] L. Gao, T. G. Griffin, and J. Rexford, "Inherently safe backup routing with BGP," in *Proc. IEEE INFOCOM*, vol. 1, 2001, pp. 547–556.

[81] S. Malik and L. Zhang, "Boolean satisfiability from theoretical hardness to practical success," *Commun. ACM*, vol. 52, no. 8, pp. 76–82, 2009.

[82] L. M. de Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[83] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. Amsterdam, The Netherlands: IOS Press, 2009, pp. 825–885.

[84] Y. Hamadi, L. Bordeaux, and L. Zhang, "Propositional satisfiability and constraint programming: A comparative survey," *ACM Comput. Surveys*, vol. 38, no. 4, p. 12, 2006.

[85] M. Ben-Ari, *Mathematical Logic for Computer Science*, 3rd ed. London, U.K.: Springer, 2012.

[86] *SAT-Based Verification Framework*. Boston, MA, USA: Springer, 2007.

[87] C. P. Gomes, H. A. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," in *Handbook of Knowledge Representation*. Amsterdam, The Netherlands: Elsevier, 2008, pp. 89–134.

[88] P. Godefroid and J. Kinder, "Proving memory safety of floating-point computations by combining static and dynamic program analysis," in *Proc. Int. Symp. Softw. Testing Anal.*, 2010, pp. 1–12.

[89] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 155–168.

[90] L. Yuan *et al.*, "FIREMAN: A toolkit for firewall modeling and analysis," in *Proc. IEEE Symp. Security Privacy*, 2006, pp. 199–213.

[91] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 467–472, 2012.

[92] A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker, "Verifying isolation properties in the presence of middleboxes," *CoRR*, vol. abs/1409.7687, 2014. [Online]. Available: http://arxiv.org/abs/1409.7687

[93] Y. Zhang, W. Wu, S. Banerjee, J.-M. Kang, and M. A. Sanchez, "SLA-verifier: Stateful and quantitative verification for service chaining," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Atlanta, GA, USA, 2017, pp. 1–9.

[94] A. Fogel *et al.*, "A general approach to network configuration analysis," in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Oakland, CA, USA, 2015, pp. 469–483.

[95] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proc. Conf. ACM SIGCOMM Conf.*, Aug. 2016, pp. 300–313.

[96] S. K. Fayaz *et al.*, "Efficient network reachability analysis using a succinct control plane representation," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 217–232.

[97] P. Peresini, M. Kuzniar, and D. Kostic, "Rule-level data plane monitoring with monocle," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 595–596, 2015.

[98] Y. Velner *et al.*, "Some complexity results for stateful network verification," in *Proc. Int. Conf. TOOLS Algorithms Construct. Anal. Syst.*, 2016, pp. 811–830.

[99] H. Zeng, P. Kazemian, G. Varghese, and N. Mckeown, "Automatic test packet generation," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 554–566, Apr. 2014.

[100] J. Yao *et al.*, "Model based black-box testing of SDN applications," in *Proc. CoNEXT Student Workshop (CoNEXT)*, Sydney, NSW, Australia, Dec. 2014, pp. 37–39. [Online]. Available: http://doi.acm.org/10.1145/2680821.2680828

[101] J. Yao *et al.*, "Testing black-box SDN applications with formal behavior models," in *Proc. IEEE MASCTOS*, 2017, pp. 110–120.

[102] F. Le, G. G. Xie, and H. Zhang, "Instability free routing: Beyond one protocol instance," in *Proc. ACM Conf. Emerg. Netw. Exp. Technol. CoNEXT*, 2008, p. 9.

[103] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *Proc. IEEE Int. Conf. Netw. Protocols*, 2009, pp. 123–132.

[104] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *Proc. 21st IEEE Int. Conf. Netw. Protocols (ICNP)*, 2013, pp. 1–11.

[105] H. Zeng *et al.*, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. USENIX NSDI*, 2014, pp. 87–99.

[106] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *Proc. ACM PLDI*, 2014, pp. 69–83.

[107] P. Kazemian *et al.*, "Real time network policy checking using header space analysis," in *Proc. Usenix Conf. Netw. Syst. Design Implement.*, 2013, pp. 99–111.

[108] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time network verification using atoms," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 735–749. [Online]. Available: https://www.usenix.org/conference/nsdi17/technicalsessions/presentation/horn-alex

[109] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," presented at the 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2013, pp. 15–27.

[110] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 203–214, 1999.

[111] N. P. Lopes, N. Bjørner, P. Godefroid, and G. Varghese, "Network verification in the light of program verification," MSR, Rep., 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/network-verification-in-the-light-of-program-verification/

[112] N. P. Lopes, N. Bjørner, P. Godefroid, and G. Varghese, "Automated analysis and debugging of network connectivity policies," MSR, Seattle, WA, USA, Rep. MSR-TR-2014-102, Sep. 2013.

[113] Y.-W. E. Sung, C. Lund, M. Lyn, S. G. Rao, and S. Sen, "Modeling and understanding end-to-end class of service policies in operational networks," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Architect. Protocols Comput. Commun.*, 2009, pp. 219–230.

[114] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 699–718.

[115] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Static checking for stateful networks," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization (HotMiddlebox)*, 2013, pp. 31–36.

[116] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese, "ddNF: An efficient data structure for header spaces," in *Hardware and Software: Verification and Testing*, R. Bloem and E. Arbel, Eds. Cham, Switzerland: Springer Int., 2016, pp. 49–64.

[117] F. Le, G. G. Xie, D. Pei, J. Wang, and H. Zhang, "Shedding light on the glue logic of the Internet routing architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 39–50, 2008.

[118] D. A. Maltz *et al.*, "Routing design in operational networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 27–40, 2004.

[119] X. Qie and S. Narain, "Using service grammar to diagnose BGP configuration errors," in *Proc. 17th Conf. Syst. Admin. (LISA)*, San Diego, CA, USA, 2003, pp. 237–246.

[120] K. Varadhan, R. Govindan, and D. Estrin, "Persistent route oscillations in inter-domain routing," *Comput. Netw.*, vol. 32, no. 1, pp. 1–16, 2000.

[121] T. G. Griffin, F. B. Shepherd, and G. T. Wilfong, "The stable paths problem and interdomain routing," *IEEE/ACM Trans. Netw.*, vol. 10, no. 2, pp. 232–243, Apr. 2002.

[122] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis (awarded best paper)," in *Proc. 2nd Symp. Netw. Syst. Design Implement. (NSDI)*, 2005, pp. 43–56.

[123] M. Hendriks and K. G. Larsen, "Exact acceleration of real-time model checking," *Electron. Notes Theor. Comput. Sci.*, vol. 65, no. 6, pp. 120–139, 2002.

[124] R. G. Bennetts, "Introduction to switching theory and logical design," *IEE Proc. E Comput. Digit. Techn.*, vol. 128, no. 6, p. 261, Nov. 1981.

[125] *The Intel Intrinsics Guide*. Accessed: 2016. [Online]. Available: http://intel.ly/24sk3uz

[126] E. S. Al-Shaer and H. H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *Proc. IEEE 23rd Annu. Joint Conf. Comput. Commun. Soc. (INFOCOM)*, Hong Kong, Mar. 2004, pp. 2605–2616.

[127] C. R. Kalmanek, S. Misra, and R. Yang, *Guide to Reliable Internet Services and Applications*. London, U.K.: Springer, 2010.

[128] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis," in *Proc. Int. Conf. Large Install. Syst. Admin.*, San Jose, CA, USA, 2010, pp. 1–8.

[129] R. W. Skowyra, A. Lapets, A. Bestavros, and A. J. Kfoury, "Verifiably-safe software-defined networks for CPS," in *Proc. 2nd ACM Int. Conf. High Confidence Netw. Syst. CPS Week (HiCoNS)*, Philadelphia, PA, USA, Apr. 2013, pp. 101–110.

[130] R. Skowyra, A. Lapets, A. Bestavros, and A. J. Kfoury, "A verification platform for SDN-enabled applications," in *Proc. IEEE Int. Conf. Cloud Eng.*, Boston, MA, USA, Mar. 2014, pp. 337–342.

[131] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM Conf.*, Helsinki, Finland, 2012, pp. 323–334.

[132] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Lombard, IL, USA, Apr. 2013, pp. 1–13.

[133] C. Chen, L. Jia, W. Zhou, and B. T. Loo, "Proof-based verification of software defined networks," in *Proc. Open Netw. Summit Res. Track (ONS)*, Santa Clara, CA, USA, Mar. 2014, pp. 1–2.

[134] N. P. Katta, J. Rexford, and D. Walker, "Logic programming for software-defined networks," in *Proc. Workshop Cross*, 2013, pp. 1–3.

[135] N. Foster *et al.*, "Languages for software-defined networks," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 128–134, Feb. 2013.

[136] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, Hong Kong, Aug. 2013, pp. 87–98.

[137] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, analyzable controller programming," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Hong Kong, 2013, pp. 79–84.

[138] P. Perešíni, M. Kuzniar, and D. Kostic, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, Heidelberg, Germany, Dec. 2015, pp. 1–32.

[139] M. Kuzniar, P. Perešíni, and D. Kostić, "Providing reliable FIB update acknowledgments in SDN," in *Proc. ACM Int. Conf. Emerg. Netw. Exp. Technol.*, 2014, pp. 415–422.

[140] K. Bu *et al.*, "Is every flow on the right track? Inspect SDN forwarding with RuleScope," in *Proc. IEEE INFOCOM IEEE Int. Conf. Comput. Commun.*, San Francisco, CA, USA, 2016, pp. 1–9.

[141] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. Mckeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX Conf. Netw. Syst. Design Implement.*, Seattle, WA, USA, 2014, pp. 71–85.

[142] H. Zeng *et al.*, "Measuring and troubleshooting large operational multipath networks with gray box testing," Mountain Safety Res., Seattle, WA, USA, Rep. MSR-TR-2015-55, Jun. 2015.

[143] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM Conf. Spec. Interest Group Data Commun. (SIGCOMM)*, London, U.K., Aug. 2015, pp. 139–152.

[144] M. Kuzniar, P. Perešíni, M. Canini, D. Venzano, and D. Kostic, "A SOFT way for OpenFlow switch interoperability testing," in *Proc. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, Nice, France, Dec. 2012, pp. 265–276.

[145] N. Ruchansky and D. Proserpio, "A (not) NICE way to verify the OpenFlow switch specification: Formal modelling of the OpenFlow switch using alloy," in *Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM)*, New York, NY, USA, 2013, pp. 527–528.

[146] *OpenFlow Switch Specification*, OpenFlow Consortium, Denmark, SC, USA, 2015.

[147] J. Yao, Z. Wang, X. Yin, X. Shi, and J. Wu, "Formal modeling and systematic black-box testing of SDN data plane," in *Proc. 22nd IEEE Int. Conf. Netw. Protocols (ICNP)*, Raleigh, NC, USA, Oct. 2014, pp. 179–190.

[148] *Oftest LC Validating OpenFlow Swtiches*. Accessed: 2014. [Online]. Available: http://www.projectfloodlight.org/oftest/

[149] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation," in *Proc. Passive Active Meas. 13th Int. Conf. (PAM)*, Vienna, Austria, Mar. 2012, pp. 85–95.

[150] M. Kuzniar, M. Canini, and D. Kostic, "OFTEN testing OpenFlow networks," in *Proc. Eur. Workshop Softw. Defined Netw. (EWSDN)*, Darmstadt, Germany, Oct. 2012, pp. 54–60.

[151] M. Kuzniar, P. Perešíni, and D. Kostic, "What you need to know about SDN flow tables," in *Proc. Passive Active Meas. 16th Int. Conf. (PAM)*, New York, NY, USA, Mar. 2015, pp. 347–359.

[152] W. Wu, Y. Zhang, and S. Banerjee, "Automatic synthesis of NF models by program analysis," in *Proc. 15th ACM Workshop Hot Topics Netw. (HotNets)*, Atlanta, GA, USA, Nov. 2016, pp. 29–35.

[153] B. Tschaen *et al.*, "SFC-checker: Checking the correct forwarding behavior of service function chaining," in *Proc. IEEE Conf. Netw. Function Virtual. Softw. Defined Netw. (NFV-SDN)*, Nov. 2016, pp. 134–140.

[154] X. Yin, Z. L. Wang, C. M. Jing, and X. G. Shi, "A TTCN-3-based protocol testing system and its extension," *Sci. China Inf. Sci.*, vol. 51, no. 11, pp. 1703–1722, 2008.

[155] S. K. Fayaz and V. Sekar, "Testing stateful and dynamic data planes with FlowTest," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, Chicago, IL, USA, Aug. 2014, pp. 79–84.

[156] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. 12th Int. Conf. Comput.-Aided Verification (CAV)*, Chicago, IL, USA, Jul. 2000, pp. 154–169.

[157] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proc. SIGSOFT/FSE 19th ACM SIGSOFT Symp. Found. Softw. Eng. (FSE) 13th Eur. Softw. Eng. Conf. (ESEC)*, Szeged, Hungary, Sep. 2011, pp. 267–277.

[158] Y. Yakuwa, N. Tomizawa, and T. Tonouchi, "Efficient model checking of OpenFlow networks using SDPOR-DS," in *Proc. 16th Asia–Pac. Netw. Oper. Manag. Symp. (APNOMS)*, Hsinchu, Taiwan, Sep. 2014, pp. 1–6.

[159] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2005, pp. 110–121.

[160] J. Croft, R. Mahajan, M. Caesar, and M. Musuvathi, "Systematically exploring the behavior of control programs," in *Proc. USENIX Annu. Tech. Conf. USENIX ATC*, Santa Clara, CA, USA, Jul. 2015, pp. 165–176.

[161] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[162] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—A tool suite for automatic verification of real-time systems," in *Proc. DIMACS/SYCON Workshop Hybrid Syst. III Verification Control*, 1996, pp. 232–243.

[163] C. Daws and S. Yovine, "Reducing the number of clock variables of timed automata," in *Proc. Real Time Syst. Symp.*, Washington, DC, USA, 1996, pp. 73–81.

[164] C. Scott *et al.*, "Troubleshooting blackbox SDN control software with minimal causal sequences," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, Chicago, IL, USA, Aug. 2014, pp. 395–406.

[165] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[166] D. Lebrun, S. Vissicchio, and O. Bonaventure, "Towards test-driven software defined networking," in *Proc. IEEE Netw. Oper. Manag. Symp. (NOMS)*, Kraków, Poland, May 2014, pp. 1–9.

[167] N. Shelly *et al.*, "Destroying networks for fun (and profit)," in *Proc. 14th ACM Workshop Hot Topics Netw.*, Philadelphia, PA, USA, Nov. 2015, p. 6.

[168] M. A. Chang, B. Tschaen, T. Benson, and L. Vanbever, "Chaos monkey: Increasing SDN reliability through systematic network destruction," in *Proc. ACM Conf. Spec. Interest Group Data Commun. (SIGCOMM)*, London, U.K., Aug. 2015, pp. 371–372.

[169] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 163–174.

[170] A. Dill and T. Sun, "Synergistic derepression of gibberellin signaling by removing RGA and GAI function in arabidopsis thaliana," *Genetics*, vol. 159, no. 2, pp. 777–785, 2001.

[171] G. Klein *et al.*, "SeL4: Formal verification of an OS kernel," in *Proc. 22nd ACM Symp. Oper. Syst. Principles (SOSP)*, Big Sky, MT, USA, Oct. 2009, pp. 207–220.

[172] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. Design Autom. Conf.*, Jun. 2010, pp. 755–760.

[173] T. Benson, A. Akella, and D. A. Maltz, "Mining policies from enterprise network configuration," in *Proc. ACM SIGCOMM Conf. Internet Meas. Conf.*, Chicago, IL, USA, 2009, pp. 136–142.

[174] A. P. Sistla and P. Godefroid, "Symmetry and reduced symmetry in model checking," in *Proc. Comput.-Aided Verification 13th Int. Conf. (CAV)*, Paris, France, Jul. 2001, pp. 91–103.

[175] X. Amatriain and J. Basilico. (2015). *Recommender Systems in Industry: A Netflix Case Study*. [Online]. Available: https://conferences.oreilly.com/velocity/vl-ca-2018/public/schedule/detail/66606/

[176] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. 9th ACM Workshop Hot Topics Netw. (HotNets)*, Monterey, CA, USA, Oct. 2010, p. 19.

[177] *Buddy: A BDD Package*. Accessed: 2016. [Online]. Available: http://buddy.sourceforge.net/manual/main.html

[178] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. Mcmanus, "Requirements for traffic engineering over MPLS," *J. Evidence Based Soc. Work*, vol. 10, no. 2, pp. 63–72, 1999.

[179] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling record and replay troubleshooting for networks," in *Proc. USENIX Conf. USENIX Tech.*, 2011, p. 29.

[180] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "Differential provenance: Better network diagnostics with reference events," in *Proc. ACM Workshop Hot Topics Netw.*, 2015, p. 25.

[181] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, "Diagnosing missing events in distributed systems with negative provenance," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 383–394.

[182] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "The good, the bad, and the differences: Better network diagnostics with differential provenance," in *Proc. Conf. ACM SIGCOMM*, Florianópolis, Brazil, 2016, pp. 115–128.

[183] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proc. 4th Conf. Symp. Oper. Syst. Design Implement. (OSDI)*, 2000, Art. no. 1.

Yahui Li received the B.S. degree from the College of Software, Jilin University, China, in 2015. She is currently pursuing the Ph.D. degree with Tsinghua University. Her research concerns network verification, network testing, and formal methods.

Xia Yin received the B.E., M.E., and Ph.D. degrees in computer science from Tsinghua University in 1995, 1997, and 2000, respectively, where she is a Full Professor with the Department of Computer Science and Technology. Her research interests include future Internet architectures, formal methods, protocol testing, and large-scale Internet routing.

Zhiliang Wang received the B.E., M.E., and Ph.D. degrees in computer science from Tsinghua University, China, in 2001, 2003, and 2006, respectively, where he is currently an Associate Professor with the Institute for Network Sciences and Cyberspace. His research interests include formal methods and protocol testing, next-generation Internet, and network measurement.

Jiangyuan Yao received the B.E. degree in engineering mechanics from the Shandong University of Science and Technology, the M.E. degree in network information and security from Beijing Jiaotong University, and the Ph.D. degree in computer science from Tsinghua University. He is an Associate Professor with the College of Information Science and Technology, Hainan University. His research concerns formal methods, protocol testing, and cyberspace security.

Xingang Shi received the B.S. degree from Tsinghua University and the Ph.D. degree from the Chinese University of Hong Kong. He is currently with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include network measurement and routing protocols.

Jianping Wu received the B.S., M.S., and Ph.D. degrees from Tsinghua University. He is a Full Professor and the Director of the Network Research Center and the Ph.D. Supervisor with the Department of Computer Science, Tsinghua University. Since 1994, he has been in charge of the China Education and Research Network. He is a member of the Information Advisory Committee, Office of National Information Infrastructure, the Secretariat of State Council of China and is also a Vice President of the Internet Society of China. His research interests include next-generation Internet, IPv6 deployment and technologies, and Internet protocol design and engineering.

Han Zhang (M'14) received the B.S. degree in computer science and technology from Jilin University and the Ph.D. degree from Tsinghua University. He is currently with the School of Cyber Science and Technology, Beihang University. His research concerns computer networks, network security, and AI.

Qing Wang received the B.S. degree in computer science and technology from Tsinghua University, China, in 2015, where she is currently pursuing the M.S. degree. Her research concerns interdomain routing and routing scalability.