Tang *et al. Journal of Internet Services and Applications* (2016) 7:1
DOI 10.1186/s13174-016-0043-y

Journal of Internet Services and Applications
a SpringerOpen Journal

RESEARCH Open Access

# Automatic belief network modeling via policy inference for SDN fault localization

Yongning Tang[1*], Guang Cheng[2], Zhiwei Xu[3], Feng Chen[4], Khalid Elmansor[5] and Yangxuan Wu[6]

**Abstract**

Fault localization for SDN becomes one of the most critical but difficult tasks. Existing tools typically only address a specific part of the problem (e.g., control plane verification, flow checker). In this paper, we propose a new approach to tackle SDN fault localization by automatically Modeling via Policy Inference (called *MPI*) the causality between SDN faults and their symptoms to a belief network. In the *MPI* system, a service oriented high level policy language is used to specify network services provisioned between end nodes. *MPI* parses each service provisioning policy to a logical policy view, which consists of a pair of logical end nodes, a traffic pattern specification, and a list of required network functions (or a service function chain). An SDN controller takes the policies from multiple parties and provisions the requested services on its orchestrated SDN network. *MPI* queries the controller about the network topology and retrieves flow rules from all SDN switches. *MPI* maps the policy view to the corresponding implementation view, in which all the logical components in the policy view are mapped to the actual system components along with the actual network topology. Referring to the component causality graph templates derived from SDN reference model, the implementation view of the current running network services can be modeled as a belief network. A heuristic fault reasoning algorithm is adopted to search for the most likely root causes. *MPI* has been evaluated in both a simulation environment and a real network system for its accuracy and efficiency. The evaluation shows that *MPI* is a highly scalable, effective and flexible modeling approach to tackle fault localization challenges in a highly dynamic and agile SDN network.

**Keywords:** SDN, Belief network, Policy, Fault localization

## 1 Introduction

Software-defined infrastructure is revolutionizing the way that large-scale data centers and service provider networks are built and operated [1]. This revolution is driven by the fast-paced improvements of virtualization technologies and programmability interfaces addressing both network and computing resources. High reliability in Software-Defined Networks (SDN) becomes a fundamental requirement as being deployed for high critical network infrastructures, such as data centers and WANs [2, 3].

SDN provides new abstractions and open interfaces to enable network programmability to multiple SDN layers, including the high-level policy layer as well as the logical and physical control layers. On the other hand, Network virtualization technologies shift traditional network functions to virtual ones called Virtual Network Functions (VNF), and embed them into commoditized hardware. Comparing to the common operations in traditional network environment, SDN paradigm empowers network operators (1) deploy, configure and update network functions in a much faster and easier way; (2) maximize resource utilization when using commodity hardware and software-based functions; and (3) dynamically scale resources based on service requirements and traffic patterns.

The emergence of SDN provides both an opportunity and a challenge for all aspects of network management, especially for fault localization. In a traditional network, it is practically impossible to unambiguously discern the intent of a network operator (or tenant), which is implicitly expressed as the combination of all protocol configurations over all distributed network devices. To infer the intent or the expected network

*Correspondence: ytang@ilstu.edu
[1] School of Information Technology, Illinois State University, Normal, IL 61790, USA
Full list of author information is available at the end of the article

behavior, one must gather this state from every network device, understand multiple vendor-specific and protocol-specific configuration formats, and implement logic to infer intended network behavior from configuration. In order to conduct effective network fault localization, a properly designed network monitoring system needs to be deployed to collect the status of a set of specific predefined monitored objects and correlate observed network abnormal to its root causes. SDN makes network flexible and agile. However, these attracting SDN features also make troubleshooting SDN highly challenging, and many successfully adopted fault localization techniques in traditional networks (e.g., Belief Network) become inapplicable.

One of the attractive promises of cloud service is cloud automation, which is the capability of service provisioning to quickly allocate data center resources in order to accomplish a job or request from customers that calls for heavy-duty batch processing. Many tools available that can automate setting up, scaling up and down, storing the corresponding results, and tearing down jobs or requested resources (e.g., shutting down the pay-as-you-go process) in cloud.

The main advantage of Software Defined Networks (SDN) is the separation between control and data forwarding planes to allow for a flexible management of the network resources and the network itself. An SDN network is orchestrated by a logically centralized controller. Moving control logic out of vendor proprietary hardware and into software enables concise policy specifications and unambiguous understanding on their intent. Software-Defined Networks (SDN), Network Function Virtualization (NFV) and cloud automation allow multiple parties (network operators, application admins, cloud tenants) jointly control network behavior via high-level network policies. Highly flexible SDNs and various virtualization techniques greatly improve network scalability and agility.

The new network paradigm also brings unprecedent challenges on managing a highly dynamic and large scale SDN network. Among various SDN management tasks, fault localization is one of the most critical but difficult tasks [4]. In the context of SDN, faults are any root causes that trigger the mismatch between underlying network behavior and high-level network policies. Many tools [5–10] have been developed to facilitate SDN fault localization. However, these tools typically only address a specific part of the problem (e.g., control plane verification, flow checker).

In this paper, we propose a new approach called *MPI* to tackle SDN fault localization by automatically *Modeling* via *Policy Inference* the causality between the faults in SDN and their symptoms to a belief network, a probabilistic graphical model that represents a set of observable symptoms and their root causes via a directed acyclic graph (DAG). In the *MPI* system, a service oriented high level policy language is used to specify network services provisioned between end nodes. *MPI* converts each service provisioning policy to its *Policy View*, which consists of a pair of logical end nodes, a traffic pattern specification, and a list of required network functions (or a service function chain). An SDN controller takes the policies from multiple parties and provisions the requested services on the controlled SDN network. *MPI* then queries the controller about the network topology and retrieves flow rules from all SDN switches. Based on the flow rules and the configuration of end nodes and network function nodes (also called network functions for simplicity), *MPI* maps the *Policy Views* to their corresponding *Implementation Views*, in which all the logical components in the *Policy Views* are mapped to the actual system components (including physical hardware and logical software components) along with the actual network topology. Referring to the component causality graph templates derived from SDN reference model, each *Implementation View* of a provisioned network service can be modeled to a *Service Belief Network*. According to the physical and logical relationships among the involved components of each service, the *Service Belief Networks* can be integrated to be a *Comprehensive Belief Network*. Finally, a heuristic fault reasoning algorithm is adopted to search for the most likely root causes that are the best explanation of the observed symptoms.

In this paper, we make the following contributions:

- We develop a new service oriented policy language that can concisely and unambiguously express user intent in a requested network service, and can be easily mapped to its actual implementation on an SDN network.
- We design a practical and feasible approach to automatically model symptom-fault causality and develop an algorithm to dynamically construct a *Service Belief Network* for each policy (and thus its defined network service).
- We implement the system and evaluate it for its accuracy and efficiency in both a simulation environment and a real network system.

This rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 overviews the *MPI* system and briefly introduces its functional modules. Section 4 describes the policy language used in *MPI*. Several main ideas and their related technical details are presented in Section 5. We show *MPI* implementation and its performance evaluation in Sections 6 and 7, respectively, Section 8 concludes the paper.

## 2 Related work

Fault localization in a traditional network relies on certain diagnosis model. A dependency graph is commonly used to represent the relationships among physical and logical components, as a bridge to correlate root causes (i.e., faults) with observable symptoms. A dependency graph can be generated in a relatively static network either manually or using some automation tools. However, in a dynamic and elastic network (e.g., network topologies and services can keep changing in an order of magnitude of minutes) provisioned via SDN and NFV, the traditional approach cannot meet the requirement for achieving efficient and accurate network fault localization.

In the following, we first briefly review several conventional fault localizations techniques built upon bipartite causality graphs. Then we discuss the efforts developed for SDN troubleshooting. Finally, we highlight the difference of our proposed *MPI* approach.

### 2.1 Bipartite causality graphs

In the past, numerous paradigms were proposed upon which fault localization techniques were based. These paradigms derive from different areas of computer science, including techniques derived from the field of artificial intelligence (rule-, model-, and case-based reasoning tools as well as decision trees, and neural networks), model-traversing techniques, graph-theoretic techniques, and the codebook approach.

Among those techniques, a bipartite causality graph is a special form of a fault propagation model that encodes direct causal relationships among faults and symptoms. Many fault localization techniques proposed in the literature [11–13] use bipartite causality graphs.

One of the techniques tailored toward a bipartite causality graph based fault propagation model is Active Integrated fault Reasoning (AIR) [14], which is designed to minimize the intrusiveness of investigation actions (e.g., probings) via incrementally enhancing the fault hypothesis and optimizing the investigation action selection process. AIR is incremental, which allows a network administrator to initiate recovery actions sooner, and allows additional testing procedures to be performed. The fact that multiple alternative hypotheses are available makes it easier to replace a solution when the most probable one proves to be incorrect.

### 2.2 SDN troubleshooting

SDN as a relatively new network paradigm has changed the way how a network service could be defined and deployed. We all hope SDN can make our network more predictable, controllable and manageable. There are several tools developed to evaluate the forwarding behavior of data plane, whereby it ensures that the data plane's forwarding behavior is as specified by the network policy. Please note that fault localization is important even in a self-healing SDN system [15] to maintain the self healing capability in the future.

#### *2.2.1 SDN data plane testing*

ATPG [8], an automatic test packet generation tool, reads router configurations to create a network model. This model is traversed in order to generate a minimum amount of testing packets such that each link in the network is minimally tested once and maximally every rule is traversed once. The main goal of ATPG is to detect network failures in a stable state and not during policy changes.

NetSight [6] offers a platform which records packet histories and enables applications to retrieve packet histories of interest. The authors implemented several applications on top of NetSight that automatically retrieve the packet histories leading to specified events, e.g., reachability errors or loops

Lebrun et al. [16] present an expressive requirement formalization language called Data Path Requirement Language (DPRL) that extends the Flow-based Management Language (FML) [17]. DPRL supports arbitrary constraints on data paths. The authors implement a requirement checker that automatically generates and injects test packets in order to verify the specified path behavior. However, the developed tool is used to conduct pre-deployment tests in an emulated network.

OFRewind [10] enables recording and replaying packets collected in an SDN network. This allows to reproduce network errors and localize problems. OFRewind provides an interface for controlling the topology, timeline and specifying traffic for collecting and replaying in a debug run.

#### *2.2.2 Verification of network policy in SDN*

In network invariant checking, network requirements (e.g., reachability) are specified. The debugging tools then verify that the network policy does not circumvent those requirements. However, the actual network behavior is never tested. This is in contrast to the presented *MPI* tool that compares the policy state with the actual network forwarding behavior. We assume that the network policy is semantically correct. The network invariant checking tools on the other hand assume that the data plane is working correctly. By only using network invariant checking the network failures would never have been discovered.

Header space analysis [5] is a general and protocol-agnostic framework. It allows to statically check network specifications and configurations to identify network requirements such as reachability failures or forwarding loops. The authors model the network topology as well as the forwarding behavior of each network box as a

function. This function allows them to track headers as they are transformed by the successive network boxes along the path.

Anteater [9] collects the network topology and the forwarding information from the network devices. The operator may then specify an invariant which is verified by Anteater by using a boolean satisfiability problem (SAT) solver.

Various formal methods based verification tools are developed to check the status after modeling network behaviors using a binary logic. ConfigChecker [18] convert network rules (configuration and forwarding rules respectively) into boolean expressions in order to check network invariants. They use Binary Decision Diagram (BDD) to model the network state, and run queries using Computation Tree Logic (CTL). VeriFlow [19] uses graph search techniques to verify network-wide invariants, and handles dynamic changes in real time. Moreover, unlike previous solutions, VeriFlow can prevent problems from hitting the forwarding plane. It is designed as a layer between the controller and the network devices. VeriFlow intersects each rule installation message issued by the controller. It then checks the network topology for a specified network invariant. Only if the invariant is guaranteed, the new rule is installed.

NetPlumber [7] is closely related to VeriFlow. It also checks network updates in real-time. Contrary to VeriFlow that only allows for the verification of forwarding actions, NetPlumber additionally verifies arbitrary header modifications.

Beckett et al. [20] implemented an assertion language for SDN applications on top of VeriFlow. The assertion language allows verifying and debugging of SDN applications with dynamically changing verification conditions. Thus, they enable the operator to describe the desired behavior of the evolution of the network rather than a fixed network behavior

### 2.3 Symptom-fault modeling via policy inference

SDN brings both new challenges and opportunities to the task of fault localization. This work we proposed in *MPI* advances the state of the art by describing a self-modeling based fault localization to discover at runtime the dependency model of SDN/NFV infrastructures via policy inference. *MPI* uses component dependency graph templates to automatically model end-to-end services into a symptom-fault bipartite belief network. *MPI* assumes a continuous changing network topology and service provisioning, and update the belief network by newly obtained beliefs (i.e., verified faults and their related symptoms).

Due to the flexibility and agility in SDN, pre-creating fault diagnosis model becomes infeasible for SDN. Comparing to the related work above, The main contribution of *MPI* is the dynamic and automatic creation of the belief network based fault diagnosis model directly derived from high-level policies.

## 3 System overview

Figure 1 provides an overview of the *MPI*, including its system function modules and their interactions with external components. In this paper, a policy from a stakeholder may consist of one or multiple policy rules, and each policy rule defines an end-to-end network service. In the following paper, we may use *policy* and *policy rule* interchangeably if no ambiguity in the context.

In an SDN network, the network behavior is controlled by a set of high-level policy rules. In *MPI*, each policy rule describes an end-to-end network service. All policies are submitted to an SDN controller, which translates them into the corresponding flow rules to specific SDN switches in order to provision the requested services. At the meanwhile, the same policies are fed to ***Policy Parser***, a functional module in *MPI*.

***Policy Parser*** interprets each policy defined network service to its ***Policy View*** (Fig. 3b), which consists of a

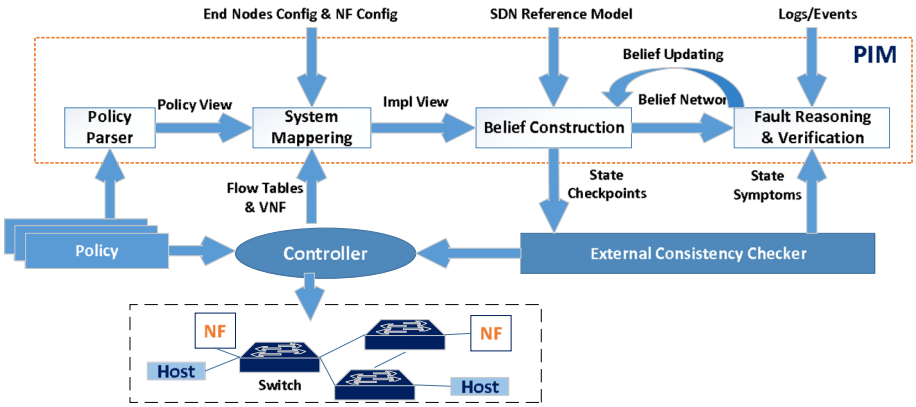


**Fig. 1** System architecture

pair of logical end nodes, traffic pattern specification, and a list of required network functions (or a service function chain). Network Functions are required mainly for meeting with various requirements in network connectivity, security and performance.

A ***Policy View*** is further passed to the ***System Mapping*** module in *MPI*. Thanks to the central view from the controller, the ***System Mapping*** module could query the controller and map the ***Policy View*** of a network service to its ***Implementation View*** (Fig. 3c), in which all the logical components described in a policy are mapped to the actual system components (including physical hardware and logical software components) with the actual underlying network topology.

Using component causality graph templates derived from the SDN reference model, the ***Belief Construction*** module models each service provisioning policy to the corresponding ***Service Belief Network*** (Fig. 3d). According to the physical and logical relationships among the involved components of each service, multiple ***Service Belief Networks*** can be integrated to a ***Comprehensive Belief Network***, where the observations related to each individual service can be accumulated to amplify their causal relationship to the root causes. Previous fault localization experience (i.e., the verified faults and their related symptoms) is fed back to the ***Belief Construction*** module to update the existing belief network.

The purpose of the entire SDN paradigm is to translate high-level human intent to low-level network behavior via multiple logical and physical layers, and each layer performs part of this translation process [21]. The layering used in SDN is to present different levels of abstraction for the purpose of virtualization and central control. In *MPI*, we adopt the commonly used 5-layer SDN reference model as shown in Fig. 4, which is essentially the same as the one proposed in [21]. The task of translation between layers is conducted by various software components [5–10].

The basic assumption in SDN fault localization is: as long as a high-level policy is defined correctly, translated correctly across layers, and executed correctly by hardware, then the network should behave normally and the related service should be provisioned correctly. In other words, any mistranslation between layers by either faulty software components or faulty hardware components could result to network misbehavior or a failure of service provisioning. The translation and execution result by each layer is presented by a corresponding network state (e.g., flow rules in a flow table). In *MPI*, two types of symptoms are used to identify and reason about network faults: mismatched states (i.e., high-level specification or low-level configuration of network behavior) between layers, and system logs of different hardware components (e.g., virtual or physical interfaces, links, disks).

The verifiable symptoms related to network state mismatching are sent as checkpoints to the external ***Consistency Checker***, in which several off-the-shelf tools (e.g., VeriFlow [19], Anteater [9]) are used to verify the related symptoms. Actively (i.e., from Consistency Checker) and passively (i.e., from system logs) observed symptoms are provided to the ***Fault Reasoning & Verification*** module to identify the most likely root causes, which are further verified later through some testing tools (e.g., ATPG [8]).

## 4 Service-oriented policy language

For a network service in an SDN network, in additional to providing basic network connectivity between end nodes, it often needs to meet specific user requirements, which are primarily related to network security and performance. Various Network Functions (NFs) are designed for satisfying user requirements. Commonly used network functions include NAT (network address translation), FW (firewall), BC (byte counter), LB (load balancing), DPI (deep packet inspection), and many others provided as value-added services (VAS) in a network infrastructure (e.g., in cloud data centers). Recently, the concept of Service Function Chaining (SFC) [22] is proposed as a traffic steering technology in directing the traffic flows of network functions.

In the following, we discuss a policy language used in *MPI* to facilitate service provisioning and fault localization. As discussion above, we believe that a high-level network service policy language should have the following features to meet the requirements of defining and managing network policies in a large-scale enterprise network:

- End-to-end service oriented: The language should focus the user's intent on the network requirements of network services rather than low-level configurations.
- Modular and reusable: The language objects are self contained and can be reused and extended to scale to large networks.
- Conflict free: The compiled policies should not have conflicts for a single user (intra-policy) or across multiple users (inter-policy).

Figure 2 shows the grammar of *MPI* policy language in EBNF syntax. The recent research [23] shows that high-level policies for SDN networks could be categorized into three types of elements: end nodes (ENs), network functions (NFs), and traffic specifications or patterns (TPs) between these elements.

Our policy language is conflict free with the help of the concept of logical user collision domain, which can clearly restrict the scope of policy effect. However, the design of conflict free language is not the focus of this work, and thus not presented here. It is worth noting that potential semantic conflicts from different stakeholders' policies

```
<endnode_def> ::= "endnode" <endnode_name> "=["<endnode> { "," <endnode> }"]"
    <endnode> ::= <ip_address> | <ip_range>| <endnode_name> { <operator> <endnode> }
      <traffic> ::= <protocol> ".["<predicates>"]"
        <nf> ::= <nf_name> "= match{"<field_name> <cmp_op> <value>{","<field_name> <cmp_op> <value>}"}"
                                  "action{"<field_name><act_op> <value>{","<field_name> <act_op> <value>}"}"
    <protocol> ::= "tcp" | "udp" | "icmp" | "ip"
    <cmp_op> ::= "=" | ">" | ">=" | "<" | "<=" | "<>"
     <act_op> ::= "modify" | "inc" | "dec"
  <predicates> ::= <predicate> { ("," | "OR") <predicate>
   <predicate> ::= <field_name> <operator> <value>
     <service> ::= <service_name> "="<endnode>"." <traffic>"."<sfc>"."<endnode>
        <sfc> ::= {<nf>}
```

**Fig. 2** The syntax of *MPI* policy language

can also be reconciled [23]. In this paper, we assume there is no conflict among different policies.

There are several important elements in the policy language: endnode, network function, traffic, and service. A service is composed of two logical endnodes and a traffic pattern between them with optionally a set of netowrk functions. Next, we elaborate each language element with examples.

**Endnode** An endnode is a logical unit which contains the network addresses of entities (workstations, servers and network devices) that share the same pattern (e.g., in the same subnet or providing same type of server). The entities in one endhost can come from different physical subnets. An endhost can be defined by a set of IP ranges, IP addresses with wildcard or host names. Also, one endhost can be constructed by combining other endhost. An endhost can be viewed as a special set contains only network address, so we can apply the set operations on endhost: intersection (*), union (+), subtraction (-). The syntax of endhost definitions shown in Fig. 2 is in EBNF. As usual square braces indicate optional items and curly braces indicate potentially empty repetition. The following example defines a endhost labmachines:

$$labmachines = [\,172.16.1.1 - 172.16.1.100]$$
$$webserver = [\,10.0.0.1 - 10.0.0.2]$$

**Traffic** A traffic is defined as the combination of a protocol name and a set of properties associated with that protocol. Each property of a protocol is a predicate which is defined by the field name in that protocol header, the operator and value of that field. The field names in common protocol headers have been predefined in *MPI* language and the supporting protocols can be extended if needed in the future. Predicates can be linked together using the logical operator AND and OR, where the comma represents AND. The syntax of service definition is shown in Fig. 2.

The policy language defines high-level users' intent, which can be interpreted by a network application and instantiated by an SDN controller. The SDN controller will populate Flow Tables of selected SDN switches to connect all required network functions to implement the demanded network services. According to the corresponding network service graph, which is a list of service functions that must be traversed along with a list of required intermediate switches.

For example, $tcp.[\,port = 80]$ means http traffic. One service can represent multiple traffic flows in the network as long as those traffic flows can satisfy the conditions defined in the properties set. For example, $tcp.[\,port > 2045, port < 3078]$ represents all tcp traffics with destination port between 2045 and 3078. We can define the yahoo instant messaging (yahoo msg) and Bit Torrent (torrent) service as follow:

$$trafficyahoo_msg = tcp.[\,port = 5050],$$
$$torrent = tcp.[\,port >= 6881, port <= 6999];$$
$$http = tcp.[\,port = 80];$$

**NF** A nf (network function) is defined as the combination of a "match" operation and an "action" operation. For example, we can define a commercial L7-aware network function load balancer (LB) as "match(dstip = Web.virtIP) modify(dstip = Web.RIPs)", where Web.RIPs is a set of real IP addresses of destination web server EPs and Web.virtIP is the exposed virtual address of the web service. We can define the network function Load Balancer as follow:

$$nf\ \ LB = match(dstip = 192.168.1.1)$$
$$action(dstip\ \ modify\ \ 10.0.0.1,\ 10.0.0.2)$$

**Service** A service is a network service provisioning between two end nodes with a specified traffic pattern and a service function chain consisting of a set of various network functions. For example, a web service between "labmachines" and "webserver" through a "LB" via TCP port 80 is simply presented as:

$$webservice = labmachines.http.LB.webserver$$

A policy rule from a stakeholder is essentially a service specifications, and structured and expressed as a conceptual network diagram. One one hand, such design

in the policy language facilitates the mapping from the high-level service policy to the real provisioning network system. On the other hand, such a policy language has been proven to be an applicable policy language for network service provisioning in a data center or WAN network [23]. In the following, we focus on showing a feasible approach to automating the process of fault localization in a highly flexible, dynamic and large scale SDN network.

## 5 System design

*MPI* aims to explore a practically feasible solution to help network admins to automate network troubleshooting by automatically modeling the causality relationship between network faults in SDN and their related symptoms. The challenge we are tackling is to localize network faults in a large-scale SDN network with: (1) a high-level policy driven and centrally controlled network architecture; and (2) the need for dynamic and agile service provisioning. *MPI* uses Belief Network [24] to model the correlation between the symptoms (e.g., state mismatching) and the corresponding hardware and software faults in an SDN network. In the following, we elaborate the major steps used in *MPI*.

### 5.1 Mapping the state views across layers

Since the *Policy View* abstraction (e.g., service, endnodes, traffic and nf) have been incorporated into the *MPI* policy language (Fig. 3a), it is straightforward for the *Policy Parser* module to parse a policy specified network service to its corresponding *Policy View* as shown in Fig. 3b.

The *System Mapping* module in *MPI* is to map a given *Policy View* to its actual implementation in a network. The *System Mapping* module treats a controller as a black box, and thus its function doesn't rely on a specific controller implementation. The *System Mapping* module first queries the controller about the network topology and the forwarding rules of each switch (via the flow table on each switch). The controller responds this request by providing the network topology in a specific topology data structure depending on the actual controller implementation. For example, OpenDaylight [25] and a Floodlight [26] controller provide two different data structures, differing in the number and type of fields and the field names.

We adopt the same algorithm as [7] to create the plumbing graph. A plumbing graph captures all possible paths of flows forwarded by SDN switches through an entire network. Nodes in the graph correspond to the forwarding rules in the network and directed edges represent the next hop dependency of these rules.

The configuration (e.g., IP addresses, locations) related to the endnodes and Network Functions are provided to the *System Mapping* module. In the case when the endnodes and Network Function (NF) nodes are configured (e.g., using DHCP) or provisioned (e.g., by the controller) dynamically, the *System Mapping* module would query the related service nodes (e.g., DHCP server, controller) about the related configuration information.

In order to map a *Policy View* to its *Implementation View*, the *System Mapping* module attaches the source (e.g., labmachines) and sink (e.g., webserver) end nodes from each provisioned service as well as all NF nodes

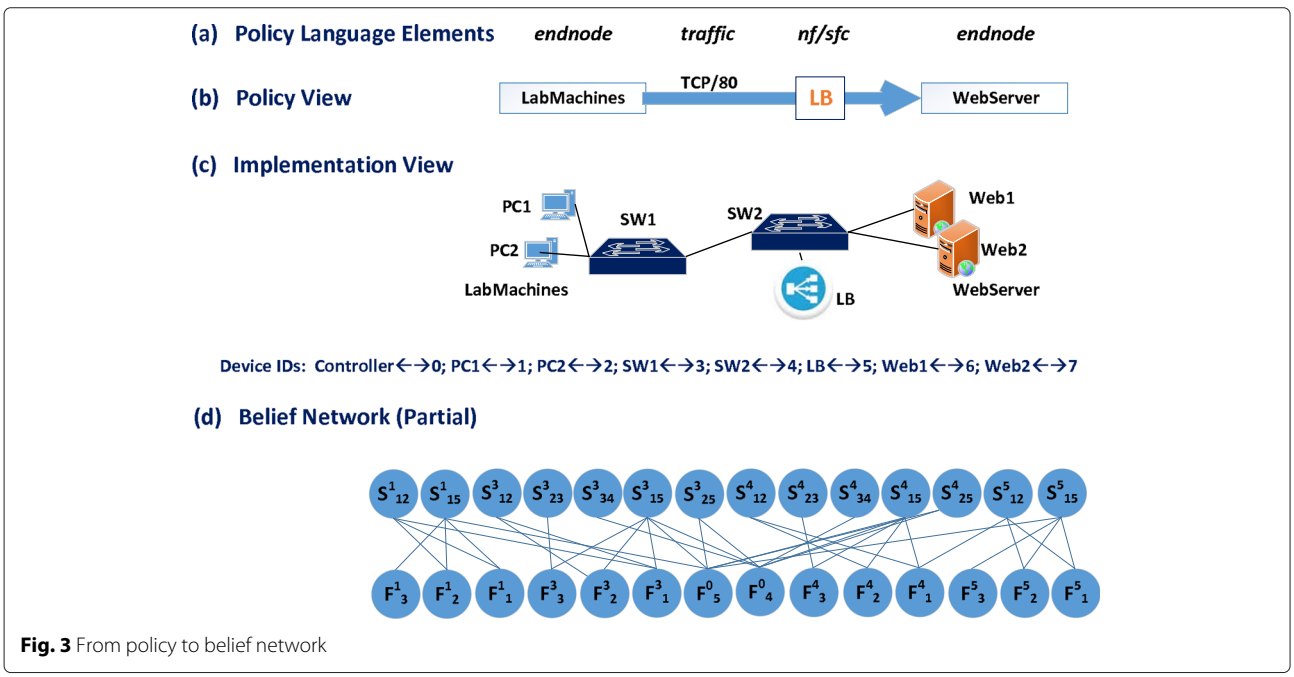


**Fig. 3** From policy to belief network

(e.g., LB) onto the plumbing graph, and then identify the necessary pipe(s) (i.e., the network topology for this service) connecting all these endnodes and NF nodes. The *Implementation View* renders the service topology with all related nodes (e.g., endnodes, NF nodes) and the intermediate forwarding switches as an example shown in Fig. 3c.

### 5.2 SDN symptom-fault reference model

SDN paradigm presents a new logical layering structure, and has changed the original dependency relationships among network components defined in OSI network model. In the new SDN model, several original OSI layers are merged (e.g., data forwarding could be jointly based on information from data link, network, and transport layers), and additional new layers are added (e.g., network hypervisor layer). Moreover, the original fully distributed network paradigm has shifted to a central (at least logically) control mechanism. Thus, network behavior in SDN could be related only to specific network nodes, or orchestrated by the central controller.

In *MPI*, we develop an SDN Symptom-Fault Reference Model based on a commonly referred SDN architectural reference [21, 27] as shown in Fig. 4. This model depicts an SDN network into five architectural State Layers (SL), and in parallel, five Root Cause (i.e., Fault) Layers (FL). The five architectural State Layers from the top down direction are: Policy State Layer $\left(SL_5^c\right)$; Logical State Layer $\left(SL_4^c\right)$; Physical State Layer $\left(SL_3^n\right)$; Device State Layer $\left(SL_2^n\right)$; and Hardware State Layer $\left(SL_1^n\right)$. Here, $c$ is the SDN controller, and $n$ represents any SDN switch or virtual network function box in a SDN system. The five Fault Layers from the top down direction are: Application Fault $(F_5^c)$; NetHypervisor Fault $(F_4^c)$; NetOS Fault $(F_3^n)$; Firmware Fault $\left(F_2^n\right)$; and Hardware Fault $\left(F_1^n\right)$.

It has been observed [4] that most errors in an SDN network are mistranslations between architectural state layers. In the context of this work, we define a fault in SDN as a violation at certain layer of the user intent defined by a high-level policy, which could be caused by both hardware issues (e.g., malfunctioning interface) and/or policy misinterpretation by certain software component(s).

The symptoms can be observed accordingly by checking the inconsistency between the behavior at a lower layer and the user intent carried from a higher layer. There are several tools [5, 7–9] developed to check the intent and state inconsistency between layers. We classify the faults in SDN on five different state layers on a device $n$. We use $F = \left\{f_i^n\right\} (1 \leq i \leq 5)$ to denote the fault set, and $f_i^n$ is a faulty component at layer $i$ on device $n$. We use $S = \left\{s_{ij}^n\right\} (i < j, 1 \leq i,j \leq 5)$ to denote the symptom set. A symptom $s_{ij}^n (i < j)$ indicates the mismatched states between a lower layer $i$ and a higher layer $j$ on device $n$. The state (e.g., flow forwarding rules at the Device View layer) of a layer represents its own interpretation on the user intent defined in one or more policies passed from its upper layer.

A symptom can be caused by one or multiple faults in $F$. Causality matrix $P_{F\times S} = \left\{p(s_{ij}^m|f_i^n)\right\}$ is used to define causal relationships between a fault $f_i^n$ and a symptom $s_{ij}^m$, here $m, n$ stand for devices $m$ and $n$. $p\left(s_{ij}^m|f_i^n\right)$ stands for the probability that the symptom $s_{ij}^m$ could be observed when the fault $f_i^n$ is localized. If $m = n$, it means the symptom and the fault are related to the same device. If $n = c$, the device is the controller $c$. In our current *MPI* implementation, both SDN northbound application and network hypervisor are running on the same server as the controller.

In *MPI*, we use several out-the-shelf tools [5–10] to check the symptoms. The common approach used in these tools is to model the different network states and behaviors using a verifiable or computable representation, such as Binary Decision Diagram (BDD). In the prototyped *MPI* system with the current availability of state verification tools, we check symptoms $s_{12}$, $s_{23}$, $s_{34}$, $s_{15}$, and $s_{25}$. In this work, we assume the policy defined by a network operator or network tenants is semantically correct.

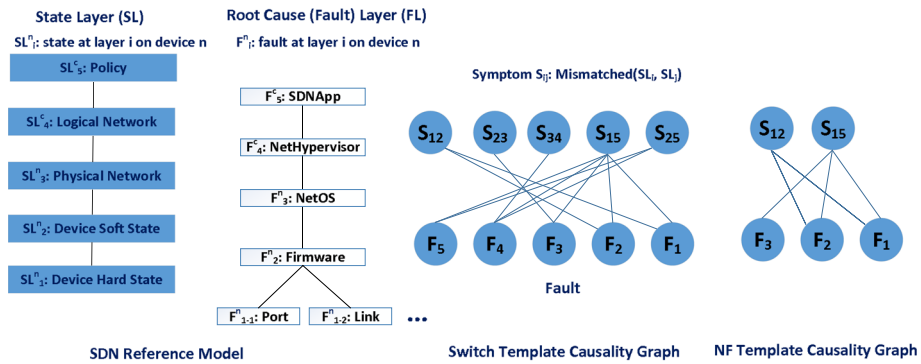


**Fig. 4** SDN reference model and component causality graph template

Otherwise, the policy needs to be redeveloped. However, checking the semantics of a given policy is out of the scope of this work.

### 5.3 Causality graph template

In the following, we introduce the concept of causality graph template, which is essentially a belief network for a specific type of individual network entity.

A causality graph template is the building block of a belief network constructed for fault reasoning in a given SDN network. As defined in a *Policy View*, we have two types of network entities: *Endnode*, *Network Function node* (i.e., a machine providing certain network function) or simply NF. Once a *Policy View* mapped to its *Implementation View*, there are other two types of entities introduced, namely *Switch* and *Controller*. In this paper, we concentrate on these four network entities only. A nice feature in *MPI* is its extensibility. If a new network entity introduced later to SDN paradigm, it can be modeled and added to the system as a new causality graph template.

A causality graph template represents the causal correlation between symptoms and faults for a specific network entity. As an example, we describe how to construct a causality graph template for NF as shown in Fig. 4. Please note that a causality graph template for a SDN switch is constructed differently due to the additional involved logical layers. For instance, symptom $s_{15}$ could be caused by mismatched state translations between any two layers in the context of SDN switches.

In the current *MPI* implementation, we choose to observe two types of symptoms $s_{12}$ and $s_{15}$ for a NF entity. Therefore, we add two symptom vertices $s_{12}$ and $s_{15}$ to the template. Since when $s_{12}$ observed, the root causes could be either $f_1$ or $f_2$, or both. Accordingly, we add two fault vertices to the template with one edge between $s_{12}$ and $f_1$ and another edge between $s_{12}$ and $f_2$ to indicate their correlation. Similarly, if $s_{15}$ observed, the root causes could be any combinations among $f_1$, $f_2$, and $f_3$. Correspondingly, three edges between $s_{15}$ and $f_1, f_2, f_3$ are added to the template, respectively.

It is worth noting that the granularity of the faults and their observable symptoms only depends on the SDN symptom-fault reference model, and the availability of various state mismatching (i.e., symptom) checking tools. The model can be extended and continuously updated to improve its accuracy and achieve finer diagnosing granularity.

### 5.4 Constructing belief network

From the *Implementation View* of a service, we firstly find all relevant network components. Then referring to the causality graph template of each type of components, we create the corresponding belief network instance for each network component. A causality graph template represents a local perspective of symptom-fault causality related a specific network entity. A global or comprehensive belief network can be constructed by connecting all related belief network instances of all components based on the architectural and topological relationships among software and hardware components from a single or multiple network entities.

From physical perspective, there are essentially three types direct connectivity between network entities: an end node and a switch; one switch to another switch; and one switch to one NF. From architectural standpoint, all symptoms $s_{i3}$, $s_{i4}$ and $s_{i5}$ ($i < 3$) from different entities indicate their common correlation to some central logical components (e.g., a controller, a policy server) that may be on the same physical hardware.

The following shows the algorithm to construct a service belief network for a specific service $i$ specified by a given policy rule $P$.

In Algorithm 1, for each network component or entity (including endnodes, network functions, and intermediate switches) shown in the implementation view, we construct a component belief network based the corresponding causality graph template (line 1–5); and then based on the logical and topological relationships among the entities, construct a *Service Belief Network* (line 6–10). Further based on the logical and topological relationships among the services, multiple Service Belief Networks can be integrated to be a *Comprehensive Belief Network*.

There are many heuristic algorithms developed to search for the most likely root causes in a given bipartite belief network. In *MPI*, we adopt our previously developed solution [14] to model the process of root cause selection as a weighted set-cover problem.

## 6 Prototype

The *MPI* prototype system is implemented in Python 2.7 and Erlang 16.2.1 [28]. As an extension to Pyretic [29], we use open source Python packages for mapping different views across layers. For concurrently parsing multiple

**Algorithm 1** *Service belief network* construction

**Require:** The implementation view $IV_p$ of service $i$ specified by a policy rule P

1: **for all** entity $e_j \in IV_p$ **do**
2: $t_j = getType(e_j)$
3: $bn_j = instanceOfCausalityTemplate(t_j)$
4: **end for**
5: find all adjacent entity pairs $EP = \{p_{a,b} =< e_a, e_b >\}$
6: **for all** entity pair $p_{a,b} \in EP$ **do**
7: $ct_{a,b} = getConnectionType(p_{a,b})$
8: $bn = joinCausalityTemplate(ct_{a,b}, bn_a, bn_b)$
9: **end for**
10: merge all common control entities in $bn$
11: **return** *Service Belief Network* $bn_i$ for service $i$

policies from different stakeholders, Open source functional programming language Erlang is used as a backend engine for belief network construction.

In this section, we describe two key implementation challenges of our design. We start with a description of the symptom monitoring module. Then we provide some details on the population of prior and conditional probabilities in a *Service Belief Network*, and the incremental belief updating over a deployed SDN infrastructure.

As *MPI* starts the fault localization process based on the observation of symptoms over the related network entities (e.g., switches, end hosts, NF nodes) according to the *Implementation View* of the service provisioning policies, it is crucial to tradeoff the overhead and performance of such a fault localization process triggered by observing symptoms. *MPI* starts with less intrusive actions on coarse-grained symptoms by comparing the view of the network state offered by a logical centralized controller to the descriptive high level policy. *MPI* uses a configurable and adaptable timer to control the frequency of verification actions to tradeoff the intrusiveness and fault detection efficiency.

It is non-trivial to obtain network statistics and populate the belief network with the prior probabilities of identifiable components (both hardware and software) and their conditional probabilities of observable symptoms given related faulty components. The bipartite belief network can be constructed according to the implementation view of a given policy. The prior failure probability of a type of hardware (e.g., port, link) or software (e.g., POX, SDN apps) is computed by dividing the number of components of a given type that observe failures by the total component population of the given type. This gives the probability of failure in a selected measurement period (e.g., one month, six months, or one year). The conditional probability $p(s|f)$ between a fault $f$ and an observable symptom $s$ is not easy to obtain. It is possible sometimes to evaluate conditional probabilities from empirical data obtained from the past behavior of a network service provider. In our prototype, we compute $p(s|f)$ based on Bayesian formula: $p(s|f) = p(f|s) \times p(s)/p(f)$. $p(f|s)$ is the posterior probability that shows how likely a fault $f$ occurred if a symptom $s$ observed. $p(s)$ is the likelihood a symptom $s$ may be observed. At the initial phase, *MPI* assigns identical initial probabilities to all unknown objects based on *Principle of Indifference* [30]. The information of verified faults and observed symptoms from the current fault localization process is used to update the belief probabilities to increase the accuracy of the model.

## 7 System evaluation

In this section, we present performance results of our *MPI* implementation. For our evaluations, *MPI* is deployed on a Dell Optiplex 9020 machine with an Intel Core i7-4790 Processor Quad Core at 3.6 GHz and 32 GB of RAM running Linux kernel 3.13.0.

We emulate switches and hosts with mininet 2.1.0 and openvswitch 2.0.2 on the machine in order to create a topology and specify policies between end hosts. Since the focus here is not to optimally design an SDN network, the results described use two simple but representative network topologies: linear and fat tree topologies [2]. For a test deployment, we randomly generate service provisionings and security policies among end hosts. A POX controller running on the same machine for *MPI* to parse the applied policies.

In our evaluation, we focus on two important aspects of *MPI* performance: system overhead and viability. We verify the system overhead by showing the model construction time under different scale of networks and policies, and its fault localization time. We demonstrate *MPI* is a viable fault localization approach in SDN via its accuracy evaluation.

### 7.1 Belief network construction time

We study the construction time of the modeling algorithm as a function of (1) the number of network entities ($N_e$), and (2) the number of policies ($N_p$).

Our primary results show that due to concurrent belief network construction using Erlang, the runtime overhead of *MPI* is linearly increasing even for a network orchestrated by a very large policy pool. *MPI* is practical and scalable, being able to parse and construct belief networks for up to a thousand of policies in under 100 s that produce nearly half million correlation edges.

We launched the modeling algorithm (Algorithm 1 for both linear and fat-tree topologies [2], ranging from 10 up to 5000 network entities. We averaged the construction time 10 times per topology to obtain more reliable results. Figure 5 shows a slow linear increase trend in the growth of constructing time with the number of network entities for both type of topologies. When the network size increased 500 times bigger, the construction time is only increased roughly 10 times longer. Linear topologies scale better than tree topologies, but in both cases the modeling time remains less than 100 s seconds.

In addition to the scale of network topologies and policies, the construction time in *MPI* is also affected by the complexity of the network and the policy as shown in Fig. 5. In general, a network with more complicated interconnection and interactions among nodes (e.g., fat-tree vs. linear topology, with NF nodes vs. without NF nodes), *MPI* takes more time to parse a network topology and map the views. Here, we use different policy update frequencies to represent different levels of network agility, and validate the impact of network agility on the performance of *MPI*. When given the same set (100 policies as

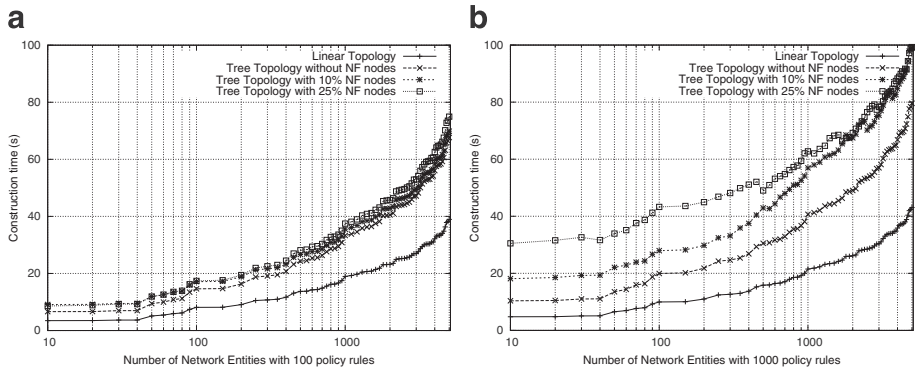


**Fig. 5** Belief network construction time

shown Fig. 5a and 1000 policies Fig. 5b) of policies, the construction time is increased 10–30 % on average.

We evaluate the impact of the policy size on the incremental updating agility and the consumed system memory via another experiment. In this experiment, we use a typical university campus network topology with 57 routers to create an SDN testbed. Then we dynamically changing the number of policies from 10 up to 10,000 to check (1) the required system memory for storing and constructing the belief network based on the policy input; and (2) the agility of *MPI* on its required updating time upon its previous belief network when receiving new policies.

The evaluation results show that the *MPI* system consumes more memory when new correlation relationships introduced by the newly added policies to the constructed belief network. However, when the increased policies start reuse the previously formed correlations among system components, there are less and less memory required and the reused components increase its exposure to the system observation. From our evaluation result as shown in Fig. 6a, the consumed memory is increased quickly until the number of policies reach to 400; after that turning point, the consumed memory still increases but in much slower pace. In our experiment, *MPI* only requires less than 1.3 GB memory when handling 10,000 policies, which is clearly practically feasible and actually efficient considering the typical configuration of a modern server.

Another important factor to check is the agility of *MPI* in responding to the possible rapid change on the service provisioning policies. In the same experiment, we introduce 100 new policies each test until reach the maximum limit 10,000, and measure the time required for updating the previous constructed belief network. As shown in Fig. 6b, the updating time is pretty stable around 20 s when paring the addition of new policies. This is because the random policy generation makes the newly introduced causality among components close to a constant rate. At the same time, the concurrency capability introduced by Erlang also increases the system scalability.

### 7.2 System viability

We evaluate the viability of *MPI* using a real SDN system. In this system, we deploy a controller, four hardware SDN switches, four software switches, a load balancer, and eight end hosts and two servers in a linear topology.

In our experiments, we use a known buggy POX load balancer implementation, and manually inject synthetic

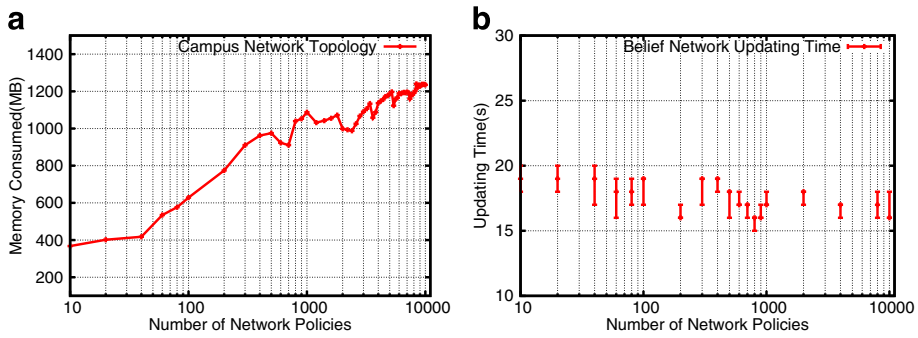


**Fig. 6** Memory overhead and updating time

trigger events and hardware failures to check the viability of *MPI* in terms of its detection accuracy and detection time.

It has been reported that POX applications do not always check error messages sent by switches rejecting invalid packet forwarding commands [31], which has been used to trigger a bug in POX's load balancer application when it starts randomly load balancing each subsequent packet for a given flow over the servers. In our current prototype, the problem was observed by the symptom $s_{15}$, and pinpointed to the controller quickly in less than 5 s after the problem triggered.

For the injected hardware failure, *MPI* can detect it in also less than 5*s* with 100 % accuracy even when the system works at its initial phase when the identical initial probabilities assigned to a constructed *Service Belief Network*.

## 8 Conclusion

Our goal in this work is to explore a feasible approach called *MPI* to automate a fault localization process by automatically modeling symptom-fault causality in a highly dynamic and large-scale SDN network. Our approach is suitable to any SDN network topology and independent from the controller implementation (e.g. POX or OpenDaylight). The concept of service-oriented policy and component causality template makes *MPI* highly scalable and extensible. Our simulations and real network experiments show that *MPI* is a viable and effective fault localization approach.

In our future work, we will increase the diagnosis granularity by incorporating more finer-grained network state checking tools. We also plan to extend the expressiveness of the current service language to better represent high-level user intents.

**Competing interests**
The authors declare that they have no competing interests.

**Authors' contributions**
YT, GC and KE developed a new service oriented policy language that can concisely and unambiguously express user intent in a requested network service, and can be easily mapped to its actual implementation on an SDN network. YT, GC, ZX, FC and KE designed a practical and feasible approach to automatically model symptom-fault causality and develop an algorithm to dynamically construct a Service Belief Network for each policy (and thus its defined network service). YT, GC, ZX, FC, KE and YW implemented the system and evaluate it for its accuracy and efficiency in both a simulation environment and a real network system. All authors read and approved the final manuscript.

**Author details**
[1]School of Information Technology, Illinois State University, Normal, IL 61790, USA. [2]School of Computer Science and Engineering, Southeast University, National Computer Network Key Laboratory, Nanjing, P. R. China. [3]Department of Computer and Information Science, University of Michigan-Dearborn, Dearborn MI 48128, USA. [4]Department of Automation, University of Science and Technology of China, Hefei 230027, P. R. China. [5]College of Mathematical Sciences, University of Khartoum, Khartoum, Sudan. [6]China Telecom Corp. Ltd., Hefei, P. R. China.

Received: 15 October 2015 Accepted: 30 December 2016
Published online: 20 January 2016

**References**

1. Batista DM, Blair G, Kon F, Boutaba R, Hutchison D, Jain R, Ramjee R, Rothenberg CE (2015) Perspectives on software-defined networks: interviews with five leading scientists from the networking community. J Internet Serv Appl 6:22
2. Al-Fares M, Loukissas A, Vahdat A (2008) A Scalable, Commodity Data Center Network Architecture. In: Proceedings of the ACM SIGCOMM 2008 Conference
3. Hong C-Y, Kandula S, Mahajan R, Zhang M, Gill V, Nanduri M, Wattenhofer R (2013) Achieving high utilization with software-driven WAN. In: Proceedings of the ACM SIGCOMM 2013 Conference
4. Scott RC, Wundsam A, Zarifis K, Shenker S (2012) What, Where, and When: Software Fault Localization for SDN. EECS Department, University of California, Berkeley. Tech. Rep. UCB/EECS-2012-178
5. Kazemian P, Varghese G, McKeown N (2012) Header Space Analysis: Static Checking for Networks. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. USENIX NSDI
6. Handigol N, Heller B, Jeyakumar V, Mazi'eres D, McKeown N (2014) I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI 2014, USA. pp 71–85
7. Kazemian P, Chang M, Zeng H, Varghese G, McKeown N, Whyte S (2013) Real Time Network Policy Checking Using Header Space Analysis. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI 2013, CA, USA. pp 99–112
8. Zeng H, Kazemian P, Varghese G, McKeown N (2012) Automatic Test Packet Generation. In: Proceedings of the 8th International Conference on Emerging Networking, CoNEXT 2012, New York, NY, USA. pp 241–252
9. Mai H, Khurshid A, Agarwal R, Caesar M, Godfrey PB, King ST (2011) Debugging the Data Plane with Anteater. In: Proceedings of the ACM SIGCOMM 2011 Conference, New York, NY, USA. pp 290–301
10. Wundsam A, Levin D, Seetharaman S, Feldmann A (2011) Ofrewind: Enabling record and replay troubleshooting for networks. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2011, USA. pp 29–29
11. Katzela I, Schwartz M (1995) Schemes for fault identification in communication networks. IEEE/ACM Trans Networking 3(6):733–764
12. Steinder M, Sethi AS (2002) End-to-end service failure diagnosis using belief networks. In: Proc. Network Operation and Management Symposium, Florence, Italy. pp 375–390. [91]
13. Yemini SA, Kliger S, Mozes E, Yemini Y, Ohsie D (1996) High speed and robust event correlation. IEEE Commun Mag 34(5):82–90
14. Tang Y, Al-Shaer E, Boutaba R (2008) Efficient Fault Diagnosis Using Incremental Alarm Correlation and Active Investigation for Internet and Overlay Networks. IEEE Trans Netw Serv Manag 5(1):36–49
15. Sanchez J, Yahia IGB, Crespi N (2014) Self-Healing Mechanisms for SoftwareDefined Networks. In: Proceeding of the 8th International Conference on Autonomous Infrastructure, Management and Security (AIMS)
16. Lebrun D, Vissicchio S, Bonaventure O (2014) Towards Test-Driven Software Defined Networking. In: IEEE Network Operations and Management Symposium. pp 1–9
17. Hinrichs TL, Gude NS, Casado M, Mitchell JC, Shenker S (2009) Practical declarative network management. In: Proceedings of the 1st ACM Workshop on Research on Enterprise Networking, WREN 09, New York, NY, USA. p 1C10
18. Al-Shaer E, Marrero W, El-Atawy A, ElBadawi K (2009) Network configuration in a box: Towards end-to-end verification of network reachability and security. In: ICNP
19. Khurshid A, Zhou W, Caesar M, Godfrey PB (2012) VeriFlow: Verifying Network-wide Invariants in Real Time. In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN 2012, New York, NY, USA. pp 49–54
20. Beckett R, Zou XK, Zhang S, Malik S, Rexford J, Walker D (2014) An Assertion Language for Debugging SDN Applications. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN 2014, New York, NY, USA. pp 91–96

21. Heller B, Scott C, McKeown N, Shenker S, Wundsam A, Zeng H, Whitlock S, Jeyakumar V, Handigol N, McCauley J, Zarifis K, Kazemian P (2013) Leveraging SDN Layering to Systematically Troubleshoot Networks. HotSDN
22. Halpern J, Pignataro C (eds) (2015) Service Function Chaining (SFC) Architecture. draft-ietf-sfc-architecture-11
23. Prakash C, Lee J, Turner Y, Kang J-M, Akella A, Banerjee S, Clark C, Ma Y, Sharma P, Zhang Y (2015) PGA: Using Graphs to Express and Automatically Reconcile Network Policies
24. Pearl J (1988) Probabilistic Reasoning in Intelligent Systems. Morgan-Kaufmann
25. OpenDaylight Platform. https://www.opendaylight.org/. Last access 01 Sept 2015
26. Project Floodlight. http://www.projectfloodlight.org/. Last access 01 Sept 2015
27. Sanchez J, Grida Ben Yahia I, Cresp I (2015) Self-Modeling based diagnosis of Software-Defined Networks. In: Proceedings of the 2015 IEEE Conference on Network Softwarization (NetSoft)
28. Armstrong J, Virding R, Williams M (1996) Concurrent Programming in Erlang. 2nd edition. Prentice Hall International, UK
29. Monsanto C, Reich J, Foster N, Rexford J, Walker D (2013) Composing Software Defined Networks. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI 2013, USA. pp 1–13
30. Jaynes ET (2003) Probability Theory: The Logic of Science. Cambridge University Press. ISBN 0-521-59271-2
31. Scott C, Wundsam A, Raghavan B, Panda A, Liu Z, Whitlock S, El-Hassany A, Or A, Lai J, Huang E, Acharya HB, Zarifis K, Shenker S (2014) Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. Proceedings of the ACM SIGCOMM 2014 Conference