Модел. и анализ информ. систем. Т. **0**, № **0** (2000) 0–13

УДК 519.681

A formal model and verification problems for Software Defined Networks

E.V. Chemeritsky, R.L. Smelyansky[1], V.A. Zakharov[2]

Applied Research Center for Computer Networks
Lomonosov Moscow State University

e-mail: zakh@cs.msu.su

получена ????

Ключевые слова: software defined network, switch, controller, forwarding rule, packet, formal model, specification, model checking

Software-defined networking (SDN) is an approach to building computer networks that separate and abstract data planes and control planes of these systems. In a SDN a centralized controller manages a distributed set of switches. A set of open commands for packet forwarding and flow-table updating was defined in the form of a protocol known as OpenFlow. In this paper we describe an abstract formal model of SDN, introduce a tentative language for specification of SDN forwarding policies, and set up formally model-checking problems for SDN.

1. Software Defined Networks and OpenFlow protocol

Since the very beginning of computer-based telecommunication engineering networks have been built out of special-purpose devices (routers, switches, firewalls, gateways). Each of these units runs sophisticated distributed algorithms to provide such functionalities as topological discovering, routing, traffic monitoring and balancing, access control, etc. A typical enterprise network might include hundreds or thousands of such devices, and in the most of them hardware and software components are closed and proprietary. These networks are managed through a set of complex heterogeneous interfaces that are used to configure separately the network devices. As the size of networks increases and network protocols become more involved, management of traditional networks tends to be remarkably complex and error-prone activity. The difficulty of tuning coherently a considerable amount of network devices operating independently is one of the most severe obstacle in the development of novel network technologies such as data centers and cloud computing.

[1]This research is supported by the Skolkovo Foundation Grant N 79, July, 2012
[2]This research is supported by the RFBR grant 12-01-00706

To cope with these principal difficulties a new kind of network architecture referred to as Software Defined Networks (SDNs) has emerged recently. SDNs have two distinguished features: 1) data flows (data plane) are separated from control flows (control plane), and 2) multiple network units can be managed by the same control program that regulates data flow propagation. Therefore, a network becomes far more susceptible to the outside control which leads to a more coordinated behavior of its components. In a SDN data flows are forwarded by the switches via communication channels just as in traditional networks. But, unlike traditional networks, control flows are transmitted via dedicated channels that connect switches and controllers. In fact, such a control flow channel may be virtual in the case when it shares the same communication link with some data flow channel. A controller is a general purpose machine (server) capable of performing application programs that manage packet switching and routing in a network. In 2011 the Open Networking Foundation defines OpenFlow protocol [1] as the first standard communications interface between the control and forwarding layers of an SDN architecture. The key points of this protocol are described below.

SDN is a real-time distributed system whose main components are divided into two classes — *switches* and *controllers*.

A switch is a network unit supplied with a number of *ports*; each port has its *ingress* and *output buffers*. Some ports of a switch are linked with ports of other switches by *data flow channels*. Data packets are transmitted through these channels. Every port of a switch has its unique identifier — *port name*. In addition, the control flow channel joins some distinguished port of a switch to the controller. A switch sends packets and statistic data to the controller and receives commands in response. Every switch is supplied with a *flow table* which is a list of *packet forwarding rules* (they are also called *flow entries*). After arriving to the ingress buffer of a switch port a data packet is processed by a flow table. The packet is matched against the the forwarding rules to select an appropriate rule. When a forwarding rule is selected then it either forwards the packet to the output buffer of a certain port, or drops the packet. If the packet is passed to the output buffer of a data flow port then it will be transmitted through the data flow channel to the corresponding ingress port on the other side of the channel. If the packet is passed to the output buffer of the control flow port then it will be sent to the controller.

A packet is an elementary formatted unit of data carried by a packet-switched network. Every packet carries two kinds of information: *header* (control information) and *payload* (user data). When packets are processed by switches and controllers their payloads are not taken into account and remain intact. A header of a packet may be divided into several fields. These fields contain source and destination addresses, the information about network protocol, type of service, etc. Packet forwarding rules are able to modify some fields, but when packets are transmitted through the channels their headers are not changed.

Each forwarding rule consists of a list of *match fields*, a list of *actions*, a *priority*, *timeouts*, and a *counter*.

A match field is a pair $(field, pattern)$, where $field$ is an identifier of a header field, and *pattern* is a string composed of 0,1 (binary symbols) and * (the wildcard symbol). A packet header matches a pair $(field, pattern)$ if all binary symbols of the string *pattern* are the same as the corresponding bits in the field $field$ of the header. A forwarding rule is applicable only to those packets whose header matches all match fields of the rule.

An action is a basic operation that processes a packet. Two kinds of actions are possible: *forwarding actions* and *header modification actions*. A forwarding action passes a packet to the output buffer of a certain port of the switch. A header modification action changes a specified field of a packet header. The actions are applied in the order specified by the list. If the list of actions includes a forwarding action then a copy of the packet processed by the previous header modification actions is passed immediately to the corresponding port of the switch. If the list of actions does not end with a forwarding action then a packet is dropped after processing by this rule.

A priority indicates subordination level of a rule.

Timeouts indicate maximum amount of time before the forwarding rule is expired by the switch. Each rule has an *idle timeout* and a *hard timeout* associated with it. An idle timeout causes the rule to be removed when it has matched no packets within a certain time. A hard timeout causes the rule to be removed after a certain time regardless of how many packets it has processed.

A counter is updated whenever the rule is applied to a packet.

Every open flow table consists of forwarding rules. It is supplied with an algorithm (procedure) for selecting appropriate rule to process the packets that arrive to the ingress buffers of the switch ports. On receipt of a packet the switch starts by performing the table look-up. Packet match fields are extracted from the packet. In addition to packet headers, matches can also be performed against the ingress port. When several forwarding rules match some packet header only the highest priority rule is selected. The counter associated with the selected forwarding rule must be updated and the list of actions included in the selected forwarding rule must be applied to the packet. OpenFlow does not specify explicitly how to resolve the case when there are multiple matching rules with the same highest priority. Every flow table must support a table-miss rule; it specifies how to process those packets that do not match any forwarding rule of the table.

A flow table can be changed in the following cases.

1. Expiration of a timeout of some forwarding rule. In this case the rule is deleted from the table and the controller is notified about this event.

2. A switch receives a command to add a new rule to the flow table. In this case the rule which is the parameter of this command is inserted to the table. If the table already contains the rule to be added then the counter of the rule is dropped and its timers are reset.

3. A switch receives a command to delete all rules that includes certain match fields. In this case all such forwarding rules are excluded from the table.

A controller manages all those switches that are connected with it by control flow channels. Through this channel the control sends commands to switches. OpenFlow standard admits the following types of messages and commands.

- Read-State commands are used by the controller to collect various information from a switch, such as current configuration, statistics (the number of packets that match specified rules) and capabilities.

- Modify-State commands are sent by the controller to add, delete and modify forwarding rules in OpenFlow tables of switches.
- Packet-out commands are used by the controller to send a certain packet out of a specified port of a switch; this command must contain a full packet and a list of action to be applied to this packet.

By turn, a switch sends asynchronous messages to a controller in the following cases:

- to denote a packet arrival to the control flow port of the switch; the control of this packet is transferred to the controller,
- to inform the controller about the removal of a forwarding rule from a flow table as the result of a controller flow delete command, or the expiring of a forwarding rule time-out.

SDNs can both simplify existing network applications and serve as a platform for developing new ones (see [2]). The main advantage of this network architecture is that programmers are able to control the behavior of the whole network by configuring coherently the flow tables in the switches. Nevertheless, bugs are likely to remain problematic since the complexity of software will increase. Moreover, SDN allows multiple applications operate simultaneously on the same controller and, hence, manage the same network. This opportunity may result in conflicting rules that spoil the forwarding policy of the whole system. The solution is to develop a toolset which could be able 1) to check correctness of a separate application operating on the controller w.r.t. a specified forwarding policy, 2) to check consistency of forwarding policies implemented by various applications, and 3) to monitor and check correctness and safety of the entire SDN.

Strange as it may seem, but only few researchers made attempts to apply formal techniques to verification of SDN behavior. The authors of [3] introduced a relational model of communication networks and developed a BDD-based toolkit to verify reachability properties of packet routing. Similar models of networks were considered in [4, 5, 6, 7]; the authors of these papers used other techniques (SAT solving, manipulations with DNFs or binary strings) to verify the same class of reachability properties. The main deficiency of these models is that they do not take into account controllers; instead, they are aimed at checking only network configurations — the snapshots of the data plane. In the models developed in [8, 9, 10] SDN is regarded as automaton (transition system) which passes from one state to another as the switches forward packets, send messages to controllers, or update their flow-tables. Automata-theoretic models are verified by means of extended static analysis. However, such models are in rather poor agreement with symbolic techniques which is unavoidable when the analysis of even local networks is concerned. As for specification language, the authors of all these papers used temporal logics (CTL or LTL) just to specify paths in the data plane routed by switches. We think that when reachability properties are concerned, transitive closure operator is more suitable for this purpose.

In this paper we study the verification problem for SDNs. Our main contribution to the study of verification problems for SDNs includes:

- introduction of a combined (relational and automata based) formal model which captures the most essential features of SDN behavior;
- introduction of a tentative language for specification of SDN forwarding policies which uses transitive closure operator to specify reachability properties of packet forwarding relation and temporal operators to specify behavior of SDN as a whole;
- formal setting of the model checking problem for SDNs.

2. A formal model of Software Defined Networks

In this section we present a formal model of SDNs. Unlike all known models introduced so far our model makes it possible to specify and analyze packet forwarding relations (relational component of the model) as well the behavior of controllers interacting with network switches (automata component of the model).

The model of SDN introduced in this paper is a finite discrete time model; i.e. our abstraction misses such issues as arithmetic operations and real time. As a consequence, it does not capture such features as requests and messages that refer to counters and deletion of forwarding rules from flow-tables at the expiration of the respective time-outs. To simplify the presentation of the model we also ignore the priorities of forwarding rules, but this is not a principal limitation. Unlike the SDN models introduced in [8, 9, 10] our model deals with paths in the data plane routed by forwarding rules (per flow model) rather than individual packets that traverse a network of switches (per packet model). The semantics of the SDN model is defined in terms of a *packet forwarding relation* on *packet states*. A packet state is specified by the header of the packet and the location of the packet in the network. A packet forwarding relation specifies how packet states can be changed while packets traverse the network. When applied to a packet, a forwarding rule changes the packet state by modifying its header and by transmitting the packet from one port of the switch to another. Packet states also change when packets are sent from one switch to another via a network channel. The packet forwarding relation allows some packets to be sent to the controller via control channels. Such packet states are regarded as messages addressed to the controller which is viewed as a transducer with a set of packet states for the input alphabet and a set of commands for the output alphabet. Upon receiving a message (which is just a packet state) from a switch the controller moves to another control state and outputs a finite sequence of commands to switches. These commands update the flow tables in some switches and, thus, modify the packet forwarding relation. An observer may consider the behavior of SDN as an alternating sequence of messages delivered to the controller (events) and packet forwarding relations computed by the network. These concepts and principles are defined formally as follows.

To avoid ambiguity we use the term "network" for a set of switches communicating with each other via data flow channels, and the term "SDN" for a distributed system which consists of a network and a controller communicating via control channels.

Packet header is a Boolean vector $\mathbf{h} = (h_1, h_2, \dots, h_N)$. All headers are assumed to have the same length N. The set of all packet headers is denoted by $\mathcal{H}$, $\mathcal{H} = \{0,1\}^N$. Components of the header $\mathbf{h}$ are denoted by $\mathbf{h}[i]$, $1 \leq i \leq N$.

Port of a switch is a Boolean vector $\mathbf{p} = (p_0, p_1, p_2, \ldots, p_k)$. Its components are denoted by $\mathbf{p}[i]$, $0 \leq i \leq k$. If $\mathbf{p}[0] = 1$ then $\mathbf{p}$ is an input port, otherwise it is an output port. All switches in the network are assumed to be identical and have the same number of ports. The set of all (input,output) ports of a switch is denoted by $\mathcal{P}(\mathcal{IP}, \mathcal{OP})$ respectively. The output port $\mathbf{p}$ such that $\mathbf{p}[i] = (0, 0, \ldots, 0)$ is a drop port. It is denoted by *drop*; at arriving to this port the packets are dropped. The output port $\mathbf{p} = \langle 0, 1, 1, \ldots, 1\rangle$ is the control output port. It is denoted by *octr*; at arriving to this port the packets are sent to a controller. The input port $\mathbf{p} = \langle 1, 1, 1, \ldots, 1\rangle$ is the control input port. It is denoted by *ictr*; only commands from the controller are passed to this port.

All switches of a network are enumerated and the name of each switch is a Boolean vector $\mathbf{w} = (w_1, w_2, \ldots, w_m)$. Its components are denoted by $\mathbf{w}[i]$, $0 \leq i \leq m$. The set of such vectors is denoted by $\mathcal{W}$.

A pair $\langle \mathbf{h}, \mathbf{p}\rangle$, $\mathbf{h} \in \mathcal{H}$, $\mathbf{p} \in \mathcal{P}$, is called a *local packet state.* A pair $\langle \mathbf{p}, \mathbf{w}\rangle$, $\mathbf{p} \in \mathcal{P}$, $\mathbf{w} \in \mathcal{W}$, is called a *node.* A triple $\langle \mathbf{h}, \mathbf{p}, \mathbf{w}\rangle$, $\mathbf{h} \in \mathcal{H}$, $\mathbf{p} \in \mathcal{P}, \mathbf{w} \in \mathcal{W}$, is called a *packet state.* The set of all packet states is denoted by $\mathcal{S}$.

A *header pattern* is a vector $\mathbf{z} = (\sigma_1, \sigma_2, \ldots, \sigma_N)$, where $\sigma_i \in \{0, 1, *\}$, $1 \leq i \leq N$. A *port pattern* is a vector $\mathbf{y} = (\delta_1, \delta_2, \ldots, \delta_k)$, where $\delta_i \in \{0, 1, *\}$, $1 \leq i \leq k$. Patterns are used for the selection of appropriate rules from flow tables as well as for the updating of packet headers.

Any *action* a is either a forwarding action $OUTPUT(\mathbf{y})$, where $\mathbf{y} \in \mathcal{OP}$, or a header modification action $SET\_FIELD(\mathbf{z})$, where $\mathbf{z}$ is a header pattern. An *instruction* is any finite sequence of actions.

A *flow entry* is a tuple $\mathbf{r} = \langle(\mathbf{z}, \mathbf{y}), \alpha\rangle)$, where $\mathbf{z}, \mathbf{y}$ are header and port patterns, and α is an instruction. A *flow-table* of a switch is a finite set of forwarding rules.

Both the topology and the functionality of a net components are defined by means of binary relations on packet states and nodes. These relations are specified by Quantified Boolean Formulae. When dealing with patterns we use two auxiliary functions $U_\sigma(u, v)$ and $E_\sigma(u)$, where $\sigma \in \{0, 1, *\}$, and u, v are Boolean vectors, such that

- if $\sigma = *$, then $U_\sigma(u, v) \ = \ u \equiv v$ and $E_\sigma(u) \ = \ 1$,
- if $\sigma \in \{0, 1\}$, then $U_\sigma(u, v) \ = \ u \equiv \sigma$ and $E_\sigma(u) \ = \ u \equiv \sigma$.

An action $a = OUTPUT(\mathbf{y})$ sends packets without changing their headers to all output ports that match a pattern $\mathbf{y} = (\delta_1, \delta_2, \ldots, \delta_k)$. An action $b = SET\_FIELD(\mathbf{z})$ modifies headers of packets following a pattern $\mathbf{z} = (\sigma_1, \sigma_2, \ldots, \sigma_N)$: a bit $\mathbf{h}[i]$ in a header remains intact if $*$ is in the position i of $\mathbf{z}$, otherwise it is changed to $\mathbf{z}[i]$. The semantics of both actions is specified by the binary relations

$$\begin{aligned} R_a(\langle \mathbf{h}, \mathbf{p}\rangle, \langle \mathbf{h}', \mathbf{p}'\rangle) &= \bigwedge_{i=1}^{N} (\mathbf{h}[i] \equiv \mathbf{h}'[i]) \ \wedge \ \bigwedge_{i=1}^{k} U_{\delta_i}(\mathbf{p}'[i], \mathbf{p}[i]) \\ R_b(\langle \mathbf{h}, \mathbf{p}\rangle, \langle \mathbf{h}', \mathbf{p}'\rangle) &= \bigwedge_{i=1}^{N} U_{\sigma_i}(\mathbf{h}'[i], \mathbf{h}[i]) \ \wedge \ \bigwedge_{i=1}^{\ell} (\mathbf{p}[i] \equiv \mathbf{p}'[i]) \ . \end{aligned}$$

on the set of local packet states $\mathcal{H} \times \mathcal{P}$.

An instruction α computes a sequential composition of its actions. If α is empty then a packet by default have to be dropped, i.e. sent to the port *drop*. Therefore, we assume that every instruction always ends with a forwarding action. The semantics of the instruction α is specified by the binary relation R_α which is defined as follows:

1. if α is empty then $R_\alpha = \mathbf{false}$;
2. if $\alpha = a, \beta$ then the relation R_α is defined by one of the formulae below depending on a:
   (a) if a is a forwarding action then
   $$R_\alpha(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle) \;=\; R_a(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle) \vee R_\beta(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle)\ ,$$
   (b) if a modifies packet headers then
   $$R_\alpha(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle) \;=\; \exists \mathbf{h}''\ (R_a(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}'', \mathbf{p} \rangle) \wedge R_\beta(\langle \mathbf{h}'', \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle))\ .$$

A packet forwarding rule is a triple $r = (\mathbf{y}, \mathbf{z}, \alpha)$, where $\mathbf{y} = (\delta_1, \delta_2, \dots, \delta_\ell)$ is a port pattern, $\mathbf{z} = (\sigma_1, \sigma_2, \dots, \sigma_N)$ is a header pattern, and α is an instruction. This rule applies the instruction α to all packets whose port and header match the patterns. Its effect is specified by the binary relation R_r on the set of local packet states $\mathcal{H} \times \mathcal{P}$

$$R_r(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle) \;=\; \mathit{precond}_r(\langle \mathbf{h}, \mathbf{p} \rangle) \;\wedge\; R_\alpha(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle)\ ,$$

where

$$\mathit{precond}_r(\langle \mathbf{h}, \mathbf{p} \rangle) \;=\; \bigwedge_{i=1}^{\ell} E_{\delta_i}(\mathbf{p}[i]) \;\wedge\; \bigwedge_{j=1}^{N} E_{\sigma_j}(\mathbf{h}[j])$$

is a *precondition* of the rule r.

A flow table *tab* is a pair (D, β), where $D = \{r_1, r_2, \dots, r_n\}$ is a set of forwarding rules and β is a default instruction. A switch applies rules from its flow table to those packets which arrive to the input ports of a switch. If all rules from the set D are inapplicable to a packet then the default instruction β takes effect. The semantics of the flow table *tab* is specified by a binary relation

$$\begin{aligned} R_{tab}(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle) \;=\; & \bigvee_{i=1}^{n} R_{r_i}(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle) \vee \\ & \vee\ (\neg(\bigwedge_{i=1}^{n} \mathit{precond}_{r_i}(\langle \mathbf{h}, \mathbf{p} \rangle)) \;\wedge\; R_\alpha(\langle \mathbf{h}, \mathbf{p} \rangle, \langle \mathbf{h}', \mathbf{p}' \rangle))\ . \end{aligned}$$

on the set of local packet states $\mathcal{H} \times \mathcal{P}$. The set of all possible flow tables is denoted by Tab.

The topology of a network is completely defined by a *packet transmission relation* $T \subseteq (\mathcal{OP} \times \mathcal{W}) \times (\mathcal{IP} \times \mathcal{W})$. Although our model admits an arbitrary transmission relation of the type specified above, in practice T is an injective function. Nodes that are involved in the relation T are called *internal nodes* of the network; others are called *external nodes*. We denote by In and Out the sets of all external input nodes and external output nodes of a network respectively. External nodes of a switch are assumed to be connected to outer devices (controllers, servers, gateways, etc.) that are out of the scope of the SDN controller. Packets enter a network through the input nodes and leave a network through its output nodes.

When a set of switches $\mathcal{H}$ and a topology T are fixed then a *network configuration* is a total function $Net : \mathcal{W} \to Tab$ which assign flow-tables to the switches of the network. Finally, the semantics of a network at a given configuration Net is specified by the *packet forwarding relation* relation

$$\begin{aligned} R_{Net}(\langle \mathbf{h}, \mathbf{p}, \mathbf{w}\rangle, \langle \mathbf{h}', \mathbf{p}', \mathbf{w}'\rangle) \;=\; & (C_{Net}(\langle \mathbf{h}, \mathbf{p}, \mathbf{w}\rangle, \langle \mathbf{h}', \mathbf{p}', \mathbf{w}'\rangle) \wedge Out(\mathbf{p}', \mathbf{w}')) \vee \\ & \vee\ \exists \mathbf{p}''(C_{Net}(\langle \mathbf{h}, \mathbf{p}, \mathbf{w}\rangle, \langle \mathbf{h}', \mathbf{p}'', \mathbf{w}\rangle) \wedge \\ & \qquad \wedge\ T(\langle \mathbf{p}'', \mathbf{w}\rangle, \langle \mathbf{p}', \mathbf{w}'\rangle)). \end{aligned}$$

on the set of (global) packet states $S = \mathcal{H} \times \mathcal{P} \times \mathcal{W}$, where

$$C_{Net}(\langle \mathbf{h}, \mathbf{p}, \mathbf{w}\rangle, \langle \mathbf{h}', \mathbf{p}', \mathbf{w}'\rangle) \;=\; \big(\bigvee_{\mathbf{w} \in \mathcal{W}} R_{Net(\mathbf{w})}(\langle \mathbf{h}, \mathbf{p}\rangle, \langle \mathbf{h}', \mathbf{p}'\rangle)\big) \;\wedge\; \bigwedge_{j=1}^{k}(w_j \equiv w_j')\ .$$

When $R_{Net}(\mathbf{s}, \mathbf{s}')$ holds for a pair of packet states $s = \langle \mathbf{h}, \mathbf{p}, \mathbf{w}\rangle$ and $s' = \langle \mathbf{h}', \mathbf{p}', \mathbf{w}'\rangle$ then every packet with a header $\mathbf{h}$ which comes to a port $\mathbf{p}$ of a switch $\mathbf{w}$ is forwarded in one hop either to an input port $\mathbf{h}'$ of a switch $\mathbf{w}'$ or to an outer device connected to an external output port $\mathbf{h}'$ of a switch $\mathbf{w}$.

A controller is a reactive program which receives messages from switches via their control ports and generates response commands that change the content of flow-tables. A switch sends a message to the controller as a request for updating its flow-table: a message indicates that the flow-table of a switch has no appropriate rules to process a packet which arrives on an input port of the switch. Therefore, a message may be viewed as a packet state. A controller generates two types of commands to add and delete forwarding rules. A command $add(\mathbf{w}, r)$, where $\mathbf{w} \in \mathcal{W}$ and r is a forwarding rule, installs the rule r in the flow-table of the switch $\mathbf{w}$. We denote by $\mathcal{C}$ the set of all possible commands. A command $del(\mathbf{w}, \mathbf{z}, \mathbf{y})$, where $\mathbf{w} \in \mathcal{W}$, and $\mathbf{z}, \mathbf{y}$ are header and port patterns, deletes from the flow-table of the switch $\mathbf{w}$ all forwarding rules $\mathbf{r} = \langle(\mathbf{z}', \mathbf{y}'), \alpha\rangle$ when the patterns $\mathbf{z}', \mathbf{y}'$ match the patterns $\mathbf{z}, \mathbf{y}$ respectively. Commands of both types change network configurations; we write $Net' = update(cmd, Net)$ to indicate that a command cmd changes a network configuration Net to a network configuration Net'. If $\omega = cmd_1, cmd_2, \dots, cmd_n$ is a finite sequence of commands then we write $update(\omega, Net)$ for $update(cmd_n, update(\dots, update(cmd_2, update(cmd_1, Net))))$.

A formal model of a controller is a transducer $A = (\mathcal{H}, \mathcal{C}, Q, q_0, \Delta)$, where

- $\mathcal{H}$ and $\mathcal{C}$ are input and output alphabets respectively,
- Q is a set of control states,
- $q_0,\ q_0 \subseteq Q$, is an initial control state, and
- $\Delta,\ \Delta \subseteq Q \times \mathcal{H} \times C^* \times Q$ is a transition relation.

A quadruple $(q, \mathbf{s}, \omega, q')$ from Δ means that a controller A when receiving a message $\mathbf{s}$ at the control state q can generates a finite sequence of commands ω and transits to the control state q'.

At every network configuration Net a controller A may receive only such messages that are triggered by packets incoming to the network via external nodes. In this cases a message includes a modified header of such packet and an input node of the switch which sends the message to the controller. To specify the set of messages $Event(Net)$ admissible at a network configuration Net we consider the transitive-reflexive closure R^*_{Net} of the one-hop forwarding relation R_{Net}. Then

$$\begin{aligned} Event(Net) = \{\langle \mathbf{x}_0, \mathbf{y}_0, \mathbf{z}_0\rangle \ : \ \exists\, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{x}' \ \ & (\langle \mathbf{y}, \mathbf{z}\rangle \in In \wedge \\ & \wedge\ R^*_{Net}(\langle \mathbf{x}, \mathbf{y}, \mathbf{z}\rangle, \langle \mathbf{x}', \mathbf{y}_0, \mathbf{z}_0\rangle) \wedge \\ & \wedge\ R_{Net}(\langle \mathbf{x}', \mathbf{y}_0, \mathbf{z}_0\rangle, \langle \mathbf{x}_0, octr, \mathbf{z}_0\rangle))\} \end{aligned}$$

A formal model of SDN is specified by the sets $\mathcal{W}, \mathcal{P}, \mathcal{H}$ of switches, their ports and packet headers, a packet transmission relation T, and a control A. A *partial run* of SDN $M = (\mathcal{W}, \mathcal{P}, \mathcal{H}, T, A)$ is a sequence (finite or infinite)

$$run = (Net_0, q_0) \xrightarrow{\mathbf{s_1}} (Net_1, q_1) \xrightarrow{\mathbf{s_2}} \cdots \xrightarrow{\mathbf{s_i}} (Net_i, q_i) \xrightarrow{\mathbf{s_{i+1}}} (Net_{i+1}, q_{i+1}) \xrightarrow{\mathbf{s_{i+2}}} \cdots \ (*)$$

where for every $i,\ 0 \le i$,

1. Net_i is a network configuration, q_i is a control state of A, and s_i is a packet state,

2. $\mathbf{s}_{i+1} \in Event(Net_i)$,

3. a transition relation of A includes a quadruple $(q_i, \mathbf{s_{i+1}}, \omega_i, q_{i+1})$ such that $Net_{i+1} = update(\omega_i, Net_i)$.

Pairs (Net_i, q_i) are viewed as the states of SDN and packet states $\mathbf{s}_{i+1}$ play the role of messages sent to the controller. A *complete run* is a partial run which is either infinite or ends with a state of SDN (Net_i, q_i) such that $Event(Net_i) = \varnothing$. Given a SDN M and network configuration Net_0 we write $Run(M, Net_0)$ to denote the set of all complete runs of M which begin with a pair (Net_0, q_0).

3. Specification of forwarding policies

Usually a wide range of requirements is imposed upon communication networks to guarantee their correct, safe and secure behavior. We consider only those requirements that concern the reachability properties. Certain packets have to reach their destination, whereas some other packets have to be dropped. Certain switches are forbidden for some packets, whereas some other switches have to be obligatorily traversed. Loops are not allowed. These and some other requirements constitute a *forwarding policy.* One of the aims of network engineering is to provide such a loading of switches with forwarding rules as to guarantee compliance with the forwarding policy. Since flow-tables of switches are updated by the controller, this raises two problems that are fundamental in software engineering:

1. *verification of SDN against a forwarding policy*: given a SDN M and a set of initial network configurations $\mathcal{N}$ check that for every network configuration Net, $Net \in \mathcal{N}$, all runs from $Run(M, Net)$ satisfy a given forwarding policy;

2. *implementation of forwarding policy*: given a forwarding policy and a set of initial network configurations $\mathcal{N}$ build a controller A such that for every network configuration Net, $Net \in \mathcal{N}$, every run run, $run \in Run(M, Net)$ of the corresponding SDN M satisfies this policy.

In order to apply formal methods to these problems one needs a formal language to specify forwarding policies.

In this section we present a tentative variant of a specification language for SDN forwarding policies. Since the behavior of a SDN evolves in time and all the states of this process may be significant for the forwarding policy, it is reasonable to use temporal logics to specify the properties of the SDN behaviors. Yet the forwarding policies also refer to properties of network configurations at some stages of the SDN behavior. These properties mostly concern the paths routed in a network by packet forwarding rules; they can be expressed in terms of one-hop packet forwarding relation R_{Net}. To this end we choose first-order logic with transitive closure operator (FO[TC] in symbols) to specify the properties of network configurations. Our choice was determined by the following considerations.

1. FO[TC] is a far more expressive formalism than propositional temporal logics or regular expressions that are used as forwarding policy specification languages in [3, 4, 5, 6, 7]; in fact, the temporal logics are strictly embedded in FO[TC] (see [12, 13]);

2. in contrast to temporal logics FO[TC] allows one to operate explicitly with packet forwarding relation; this alleviates both the writing and the understanding of specifications;

3. finite model checking problem for FO[TC] is decidable within logarithmic space (see [11]).

But the properties of network configurations are formulated in terms of relationships between packet states. Therefore, we need also a simple language for expressing such relationships. Since packet states are thought of as Boolean vectors, the best way is to choose Boolean formulae for this purpose. Now we consider this multi-level language for specification of forwarding policies in some more details.

Let $Var = \{X_1, X_2, \dots\}$ be a set of variables; they are evaluated over the set $\mathcal{S} = \mathcal{H} \times \mathcal{P} \times \mathcal{W} = \{0,1\}^{N+k+m}$ of packet states. A *packet state specification* is any Boolean formula φ constructed from a set of Boolean variables $\{X_i[j] \ : \ X_i \in Var, \ 1 \le j \le N+k+m\}$ and connectives $\neg$, $\wedge$. The set of such formulae is denoted $\mathcal{L}_0$.

A language for specification of network configurations $\mathcal{L}_1$ uses only three predicate symbols $R^{(2)}, I^{(1)}, O^{(1)}$ for the signature. It is the smallest language which satisfies the following rules:

1. if $\varphi \in \mathcal{L}_0$ then $\varphi \in \mathcal{L}_1$;

2. if $X, Y \in Var$ then the atomic formulae $R(X, Y)$, $I(X)$, $O(Y)$ are in $\mathcal{L}_1$;

3. if $\psi(X, Y)$ is a formula in $\mathcal{L}_1$ and it includes exactly two free variables then $TC(\varphi(X, Y)) \in \mathcal{L}_1$;

4. if ψ_1 and ψ_2 are formulae in $\mathcal{L}_1$ and $X \in Var$ then the formulae $(\neg\psi_1)$, $(\psi_1 \wedge \psi_2)$, $(\exists X\ \psi_1)$ are in $\mathcal{L}_1$.

The semantics of $\mathcal{L}_1$ is defined as follows. Let Net be a network configuration, and $\mathbf{s} = \langle \mathbf{h}, \mathbf{p}, \mathbf{w} \rangle$ and $\mathbf{s}' = \langle \mathbf{h}', \mathbf{p}', \mathbf{w}' \rangle$ be a pair of packet states. Then

1. $Net \models R(X, Y)[\mathbf{s}, \mathbf{s}']$ iff $(\mathbf{s}, \mathbf{s}') \in R_{Net}$;

2. $Net \models I(X)[\mathbf{s}]$ iff $\langle \mathbf{p}, \mathbf{w} \rangle \in In$;

3. $Net \models O(X)[\mathbf{s}]$ iff $\langle \mathbf{p}, \mathbf{w} \rangle \in Out$;

The satisfiability relation for other formulae in $\mathcal{L}_1$ is defined straightforward.

Some facts are worthy to be mentioned with respect to FO[TC]. As it follows from the results of [11], model checking problem for FO[TC] is NLOG-complete. Moreover, as it was shown in [12, 13] both μ-calculus and PDL can be translated in FO[TC] (although the size of formulae may grow exponentially). As for network model checking against $\mathcal{L}_1$ specifications, we proved

Theorem. The model checking problem $Net \models \psi$ for closed formulae in $\mathcal{L}_1$ is PSPACE-complete.

Proof. The state space $S = \mathcal{H} \times \mathcal{P} \times \mathcal{W}$ of a model Net is at most exponential of the size of Net. Therefore, due to the results from [11], model checking problem $Net \models \psi$ is decidable within polynomial space. The proof of PSPACE-hardness of network model checking problem is based on the fact that packet headers may be viewed as configurations of linear bounded Turing machine. In this case the commands of such a machine can be simulated by some appropriate flow table rules. In fact, it is sufficient to have only one network switch with a loop to simulate by means of such a network any Turing machine operating on the tape whose size is bounded by the size of packet header. Thus, the Halting problem for linear bounded Turing machine that is known to be PSPACE-complete (see [14]) can be reduced to the problem of checking whether a given network Net drops a given packet that arrives at a given ingress port of a definite switch.

A forwarding policy, i.e. desirable properties of SDN behavior, can be specified by means of propositional temporal logics where formulae from $\mathcal{L}_1$ serve as atomic propositions.

Let $\mathcal{L}_0(X)$ and $\mathcal{L}_1(X)$ be the set of all those formulae from $\mathcal{L}_0$ and $\mathcal{L}_1$ respectively which have the only free variable X. Then $LTL(\mathcal{L}_1)$ is the smallest language which satisfies the following rules:

1. if $\psi(X) \in \mathcal{L}_1(X)$ then $\psi(X) \in LTL(\mathcal{L}_1)$;

2. if $\Phi(X)$, $\Psi(X) \in LTL(\mathcal{L}_1)$ then $(\neg\Phi(X))$, $(\Phi(X) \wedge \Psi(X))$, $(\mathbf{X}\, \Phi(X))$ and $(\Phi(X)\ \mathbf{U}\ \Psi(X))$ are in $LTL(\mathcal{L}_1)$.

Formulae from $LTL(\mathcal{L}_1)$ are evaluated on infinite sequences of network configurations $\{Net_i\}_{i=1}^{\infty}$ for a given packet state $\mathbf{s}$ as follows:

- if $\psi(X) \in \mathcal{L}_1(X)$ then $\{Net_i\}_{i=1}^{\infty} \models \psi(\mathbf{s})$ iff $Net_1 \models \psi(\mathbf{s})$,
- $\{Net_i\}_{i=1}^{\infty} \models \mathbf{X}\ \Phi(\mathbf{s})$ iff $\{Net_i\}_{i=2}^{\infty} \models \Phi(\mathbf{s})$,
- $\{Net_i\}_{i=1}^{\infty} \models \Phi(\mathbf{s})\mathbf{U}\Psi(\mathbf{s})$ iff $\{Net_i\}_{i=k}^{\infty} \models \Psi(\mathbf{s})$ for some $k,\ 1 \le k$, and $\{Net_i\}_{i=j}^{\infty} \models \Psi(\mathbf{s})$ for every $j,\ 1 \le j < k$,
- the semantics of connectives $\neg$ and $\wedge$ is defined in the usual way.

A language for specification of forwarding policies $\mathcal{L}_2$ is the set of expressions $\varphi(X) \Rightarrow \Phi(X)$, where $\varphi(X) \in \mathcal{L}_0(X)$, and $\Phi(X)$ is a temporal formula from $LTL(\mathcal{L}_1)$. The semantics of these expressions is defined through the satisfiability relations on the runs of formal models of SDNs. Suppose that $(*)$ is a run of SDN M and $\varphi(X) \Rightarrow \Phi(X)$ is an expression from $\mathcal{L}_2$. Then $run \models \varphi(X) \Rightarrow \Phi(X)$ iff $Net_{i=n}^{\infty} \models \Phi(\mathbf{s_n})$ for every $n,\ 1 \le n$, such that $\varphi(\mathbf{s_n}) = 1$.

A forwarding policy FP can be specified by a constraint ψ on initial network configurations which is a closed formula from $\mathcal{L}_1$, and a finite set $\{\varphi_1(X) \Rightarrow \Phi_1(X), \dots, \varphi_n(X) \Rightarrow \Phi_n(X)\}$ of expressions from $\mathcal{L}_2$. We say that SDN M *implements a forwarding policy* FP iff for every network configuration Net_0 such that $Net_0 \models \psi$ every run $(*)$ from the set $Run(M, Net_0)$ satisfies all requirements $\varphi_i(X) \Rightarrow \Phi_i(X)$, $1 \le i \le n$. Thus, the model checking problem for SDNs is that of checking whether a given formal model of SDN M satisfies a specification of given forwarding policy FP.

4. Conclusion

It is worth noticing that if a controller of SDN is a finite state machine then the model checking problem, as defined above, is decidable for such models of SDN. This is due to the fact that the network has only finitely many configurations and, hence, all runs of SDN can be united in a finite state transition system. Thus, the model checking problem for SDN can be reduced to a finite model checking problem for PLTL. The main difficulty in using this consideration in practice is that the size of the statespace of this transition system may be double exponential on the size of respective SDN description. Till now we do not know how to cope with this problem. Nevertheless, we have built a BDD-based toolset for model checking network configurations against their specifications, i.e. closed formulae ψ from $\mathcal{L}_1$. Using this toolset we are able to check on-the-fly the simple forwarding policy specifications of the form $true \Rightarrow \mathbf{G}\ \psi$, i.e. safety invariants of SDN behavior.

We would like to thank the anonymous referee for the valuable comments that help the authors to improve the paper.

Список литературы

1. OpenFlow Switch Specification. Version 1.4.0, August 5, 2013, www.opennetworking.org.

2. H. Kim, N. Feamster. Improving network management with software defined networking. Communications Magazine, IEEE, 2013, p. 114-119.

3. E. Al-Shaer, W. Marrero, A. El-Atawy, K. El Badawi. Network Configuration in a Box: Toward End-to-End Verification of Network Reachability and Security. In the 17th IEEE International Conference on Network Protocols (ICNP'09), Princeton, New Jersey, USA, 2009, p. 123-132.

4. H. Mai, A. Khurshid, R. Agarwal, M. Caesar, R.B. Godfrey, S.T. King. Debugging of the Data Plane with Anteater. In the Proceedings of the ACM SIGCOMM conference, 2011, p. 290-301.

5. P. Kazemian, G. Varghese, N. McKeown. Header space analysis: Static checking for networks. In the Proceedings of 9-th USENIX Symposium on Networked Systems Design and Implementation, 2012.

6. A. Khurshid, W. Zhou, M. Caesar, P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In the Proceedings of International Conference "Hot Topics in Software Defined Networking"(HotSDN), 2012, p. 49-54.

7. S. Gutz, A. Story, C. Schlesinger, N. Foster. Splendid isolation: A Slice Abstraction for Software Defined Networks. In the Proceedings of International Conference "Hot Topics in Software Defined Networking"(HotSDN), 2012, p. 79-84.

8. M. Reitblatt, N. Foster, J. Rexford, D. Walker. Consistent updates for software-defined networks: change you can believe in!. HotNets, v. 7, 2011.

9. M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger. D. Walker. Abstractions for Network Update. In the Proceedings of ACM SIGCOMM conference, 2012, p. 323-334.

10. M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford. A NICE way to Test OpenFlow Applications. In the Proceedings of Networked Systems Design and Implementation, April 2012.

11. N. Immerman. Languages that capture complexity classes. SIAM Journal of Computing, v. 16, N 4, 1987, p. 760-778.

12. N. Immerman, M. Vardi. Model checking and transitive closure logic. Lecture Notes in Computer Science, 1997, p. 291-302.

13. N. Alechina, N. Immerman. Reachability logic: efficient fragment of transitive closure logic. Logic Journal of IGPL, 2000, v. 8, N 3, p. 325-337.

14. Z. Galil. Hierarchies of Complete Problems. Acta Informatica, 1976, N 6, p. 77-88.

???

???

Keywords: ???

???

Сведения об авторе: ???