International Journal of Electrical and Computer Engineering (IJECE)
Vol. 9, No. 4, August 2019, pp. 2627~2636
ISSN: 2088-8708, DOI: 10.11591/ijece.v9i4.pp2627-2636

A VNF modeling approach for verification purposes

Guido Marchetto, Riccardo Sisto, Matteo Virgilio, Jalolliddin Yusupov
Department of Control and Computer Engineering, Politecnico di Torino, Italy

Article Info

Article history:

Received Apr 27, 2018
Revised Jan 28, 2019
Accepted Mar 4, 2019

Keywords:

Formal verification
Model extraction
Modeling
Network functions
Parser

ABSTRACT

Network Function Virtualization (NFV) architectures are emerging to increase networks flexibility. However, this renewed scenario poses new challenges, because virtualized networks, need to be carefully verified before being actually deployed in production environments in order to preserve network coherency (e.g., absence of forwarding loops, preservation of security on network traffic, etc.). Nowadays, model checking tools, SAT solvers, and Theorem Provers are available for formal verification of such properties in virtualized networks. Unfortunately, most of those verification tools accept input descriptions written in specification languages that are difficult to use for people not experienced in formal methods. Also, in order to enable the use of formal verification tools in real scenarios, vendors of Virtual Network Functions (VNFs) should provide abstract mathematical models of their functions, coded in the specific input languages of the verification tools. This process is error-prone, time-consuming, and often outside the VNF developers' expertise. This paper presents a framework that we designed for automatically extracting verification models starting from a Java-based representation of a given VNF. It comprises a Java library of classes to define VNFs in a more developer-friendly way, and a tool to translate VNF definitions into formal verification models of different verification tools.

Corresponding Author:

Jalolliddin Yusupov,
Department of Control and Computer Engineering,
Politecnico di Torino,
Corso Duca degli Abruzzi, 24, TO, Italy.
+39 345 5170753.
Email: jalolliddin.yusupov@polito.it

1. INTRODUCTION

The telecommunications world is exhibiting a very rapid change in its various aspects such as service flexibility, architectural design, and the way services are created, sourced, deployed, and supported. New studies in this sector are coming out every day to change the entire structure of the system by introducing dynamic adjustment of the network resources, custom configuration on a per-user basis, network programmability, etc. The expectation for significant cost savings is frequently mentioned as one of the primary benefits of these studies.

The virtualization technology has emerged as a way to decouple software applications from the underlying hardware and enable software to run in a virtualized environment, with a consequent increase of the service flexibility. In particular, the notion of Network Functions Virtualization (NFV) [1] is evolving to remedy the static nature of traditional networks by promoting innovation in network management and deployment of network services. In an NFV environment, a network is comprised of software-based applications called Virtual Network Functions (VNF) that take on the responsibility of handling specific network functions that run on one or more virtual machines (VMs) or in containers, on top of the physical networking infrastructure.

The idea that now almost anyone can introduce complex VNF software, in today's modern networks, increases the impact of possible network configuration errors. As a result, a substantial amount of effort is required to ensure networks' correctness, safety, and security. Therefore, verification of networks is key to eliminate errors and build robust infrastructures. With this respect, mechanized formal techniques have proved to be powerful engines for a formal verification of the network behavior in many different contexts [2-4].

In the networking panorama, most existing verification tools - model checking, SAT solvers, and theorem provers - rely on a formal model provided according to a given description language. The main challenge providers of NFV software have to face in order to enable formal verification of virtualized networks is the model construction: there is a large semantic gap between the artifacts produced by software developers and those accepted by current verification tools. For example, powerful approaches such as [2] and [5] have already evolved from research and are now being rolled into production, but this gap might be a significant hurdle for their wide adoption in real network environments. In essence, these tools are based on a complex modeling technique, tend to lock the user into a single kind of checking technology, require to accurately model network functionality, which relies on expert input, and usually oblige developers to learn a whole new language (e.g., SEFL in [5]).

This motivates the work presented in this paper, i.e., a framework for a user-friendly VNF modeling that developers can use to provide a formal description of their functions to be used in a verification process. The major highlight of our framework is its simplicity and we develop it targeting three specific objectives:
- To simplify the definition of a network function forwarding model in a well-known language.
- To leave some general concepts and flexibility to developers in such a way that they could define the desired behavior for all their network functions.
- To provide an automatic translation from the function model definition into an abstract formal model for verification tools.

In order to meet the above-mentioned principles, we select Java as a well-known and wide spread language that developers find simple and easy to grasp. The specific library we propose in this Java-based framework represents the typical set of high-level operations commonly used for describing the network function's behavior. Starting from a skeleton class definition of a generic network function, a VNF developer can easily extend the provided artifacts to inherit basic properties, data types, and methods and customize function behavior. Our framework also includes a parser that analyzes the Java source code and produces an abstract formal model of the network function that can be automatically translated into the input language proper of a given verification tool. This second step is clearly tool dependent and we plan to enrich our framework in order to support the vast majority of the existing tools. Currently, the parser operation is oriented to formal verification tools based on the analysis of logic formulas and in particular the VeriGraph [6] tool is adopted as a use case. Verigraph requires to model complex network scenarios as sets of First Order Logic (FOL) formulas and uses an SMT solver, Z3[7], to verify satisfiability of these formulas. We hence developed a translator for converting the output of the parser to the proper set of FOL formulas. To check the correctness of the models obtained, the developer can run tests to verify network properties in terms of reachability between different nodes in several simple graphs that include the developed VNF model. We first introduce our modeling technique in Section 3. The use case and the obtained results are presented in Section 4. In addition, Section 2 discusses related work, while Section 5 concludes the paper.

2. RELATED WORK

There has been a significant amount of activity in the past years on attempting to provide a proper support for the translation of software system descriptions to the input models for verification tools. Among the others, we can mention Bandera [8] and JavaPathFinder [9]. The two approaches are based on model checking, and the models they extract are models of Java software. The main difference with respect to the proposed framework is that they consider general-purpose Java programs and their main target is the identification of programming errors and bugs. Here instead we deal with the forwarding behavior of VNFs. We are not interested in all the details of the VNF code execution. Furthermore, we want our analysis to be extremely fast.

Moreover, many approaches and methods for static network analysis have been proposed [2, 5, 10]. Network Optimized Datalog [2] relies on Datalog both for network models and policy constraints. BUZZ [10] uses hand-generated models of network functions in a domain specific language. As we discussed earlier, modeling network functionality for using these tools is difficult and requires a detailed understanding of the verification tool's semantics. Therefore, our automated approach to generate models eliminates the necessity of having detailed domain knowledge and helps network engineers to quickly determine the behavior of a network function.

On the other hand, SymNet [5] constructs models using an imperative, modeling language, called Symbolic Execution Friendly Language (SEFL). While the way this language has been designed has similarities with the Javabased library we propose, this approach lacks the idea of ease of modeling, by introducing a new language. Despite the fact that they provide parsers to automatically generate SEFL models from real network functions, this generation only covers routers and switches. Our approach, instead, is based on the well-known user-friendly Java language and can be used to verify any virtual network function.

A proposal more similar to our target is, NFactor [11], which provides a solution to automatically analyze the source code of a given network function to generate an abstract forwarding model, motivated by network verification applications. While relying on advanced tools [12] and techniques [13, 14] from the program analysis community, they do not require a specific structure of the source code of the function to be analyzed. This feature is considered as an advantage from a generality point of view. Unfortunately, creating a model that captures all code paths of a network function is challenging, because the state processing may be hidden deep in the code. This may cause the analysis to miss certain state changes. For example, implementations might use pointer arithmetic to access state variables, which is difficult to trace, and NFactor does not seem to deal with these language features appropriately. Another limitation of the approaches based on the extraction of models from source code is that the code of most network functions is typically proprietary. Instead of relying on vendors to release their code, the aim of our framework is to give developers the opportunity to implement their network functions resembling as much as possible the real one’s behavior.

3. MODELING TECHNIQUE

Currently, there is no standard or modeling language to accurately represent the diversity and complexity of network functions. Most of the research efforts in proposing VNF models are focused on network verification and gained popularity in the verification community. In this section, we list the open problems that we have encountered while looking at the proposed VNF models in the verification context.

3.1. Overview and problem statement

Modeling of VNFs is useful in a number of ways ranging from finding scalability issues in applications to finding network configuration bugs, in particular by means of formal verification tools. However, formal modeling of network functionalities is difficult and requires a detailed understanding of the specific verification tool’s internals, semantics, and modeling language. With this respect, an automated approach to generate models eliminates the necessity of having detailed knowledge in the formal verification domain and helps engineers to quickly determine the behavior of a VNF-based network, starting from a more user-friendly description of the involved VNFs. In particular, the possibility to describe VNFs by means of a Java-like modeling language would significantly lower barriers to entry for these powerful verification approaches.

An imperative language such as Java focuses on describing how a program operates. A VNF developer can write a code that describes in exact detail the steps that the VNF must make when a packet is received from one of its interfaces. In contrast, declarative languages adopted in logic-based formal verification tools do not specify a step or sequence of steps to execute, but rather predicates that must hold.

The conceptual gap between these two representations is the important challenge solved by our approach. The proposed framework provides a Java library and a parser. The library can be used for modeling network functions by means of an imperative language. The parser then automatically generates abstraction models from these descriptions. Basically, the parser takes as an input the definition written using our library and produces an abstract formal model describing the behavior of the network function. This gives the possibility of automatically translating the definition that is written in a well-known language into a more high-level, domain-specific constraint language, that would be difficult to deal with manually. The rest of this section presents the Java library and the parser we developed, which sum up our modeling technique.

3.2. Overview and problem statement

The framework provides a library that allows users to easily write models of virtual network functions. By means of this library, the user can simply describe the functionality of the network function by instantiating objects of the library classes and by calling certain methods that correspond to typical operations performed inside network functions and using the basic syntax of Java and the methods offered. We define library classes based on the following characteristics which represent the generic behavior of a network function:

- A network function may behave as an end host or a forwarding host.
- An end host represents a terminal node. It can receive packets, but it can also send packets in response to a received one (e.g. a response to the request) or new packets to initiate a communication (e.g. a request). Examples of end host nodes are a mail client, a web client, a mail server, etc. A forwarding host represents an intermediate node that processes traffic according to its internal logic to accomplish a specific mission, e.g. performing NAT translation, filtering packets, etc.
- A forwarding host can drop or determine which exit interface to use to send the packet to its next hop.
- A host, either an end or a forwarding one, can have network interfaces for receiving or sending packets and can be stateful, i.e. have a state that can depend on the history of the actions performed before (related to sent and received packets).
- The packets exchanged by the hosts are abstracted in order to capture only those characteristics that are relevant for verification. For example, source address, destination address, source port, destination port, protocol used, target URL, mail source, mail destination, body, etc.

Class definition: The definition of a virtual network function must be written in a unique file that must extend one of the following two abstract classes: ForwardingHost, EndHost (which in turn extends the Host class). These are the two main Java classes of the library, which are supplemented with other complementary classes depending on the type of the network function being described, either end host or forwarding host. For example, the Packet class describes a packet and objects of this type and the RoutingResult class represents the routing decision of a VNF might be included in EndHost and ForwardingHost objects. While the HostTable class represents an internal memory of the intermediate network function and hence is related to the ForwardingHost class only. Additionally, the internalNodes object of the Host class is used to differentiate the internal nodes of the corresponding network function from the external ones and the hostTableList object stores the list of available tables.

Class methods: The behavior of the network function must be described using a number of public methods provided by the library classes. The content of the methods is under the control of the user, which has to specify the network function behavior by means of the methods available in complementary classes of the library.

defineState(): this method defines the components from which the state of the network function and configuration settings are extracted. It is present either if an EndHost or a ForwardingHost is defined. This method provides instructions to create a table that stores packet fields and a flag to indicate whether the network function is able to store the packets received (sent) or not.

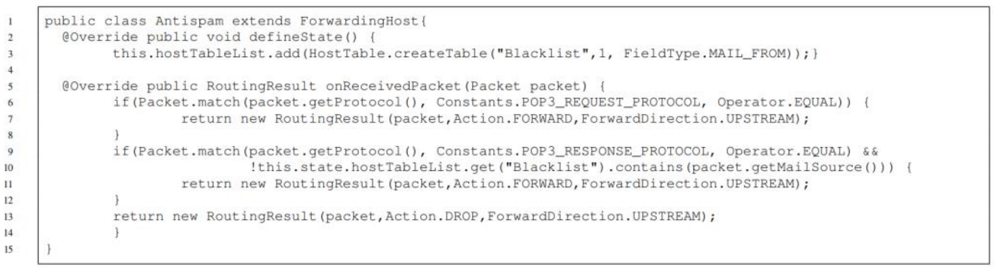
defineSendingPacket(): this method defines the characteristics of the packets sent by the current node (not the responses to the received packets but the request packets sent by the host). It may be present only if an EndHost is defined. This method must return a Packet object, which can be built inside the method by instantiating the Packet class. The Packet class offers a match() method to compare the fields of the packet against the other field or constants. There are "mutator" methods defined in this class to control changes to a packet fields.

onReceivedPacket(): this method defines the behavior of the network function in response to a received packet. The parameter of the onReceivedPacket() method is the packet that the network function receives and the return value is a RoutingResult object. As discussed above, RoutingResult is a complementary class that represents the routing decision of the VNF, after processing of the incoming packet. Its constructor receives three parameters:

- A packet object that the network function produces.
- The action to perform on this packet (forward or drop).
- The forwarding direction (i.e. the interface the packet is forwarded to in case of forward action; this can be the upstream interface or one of the other interfaces of the network function).

The actions that can be inserted inside this method are divided into the following categories: instructions to check the contents of a packet field, instructions to check the state of the network function, instructions to store a value into a table defined by the user, instructions to define the action to be performed on a packet (through a RoutingResult), and setting a value into a field of a packet.

An example of the Java description of an Antispam network function is shown in Figure 1. This is an intermediary VNF designed to handle the mail traffic between end hosts on the basis of a blacklist table. The name of the table, number of columns assigned for this table and the type of the table entries are passed as an argument to the constructor of the HostTable class in the defineState() method (row 2, Figure 1). In this scenario, the table named "Blacklist" containing a single column to store the blacklisted senders of e-mail messages with the type enum FieldType.MAIL FROM is created. Whereas the forwarding behavior of the Antispam VNF is described using the onReceivedPacket() method. In particular, lines 6 and 7 describe the forwarding action if the protocol of the received packet is equal to a POP3 request. If this is not the case and the protocol of the packet is equal to a POP3 response, then the mail source field of the packet must be checked against blacklist entries as shown in lines 9 and 10. If the mail source field of the POP3 request packet does not match any table entry, then the new RoutingResult object must be returned specifying the forwarding action (line 11). Finally, line 13 corresponds to a drop action that must be enforced by the network function, in case of no other conditions are met. It is important to mention that there is not any restriction on the order of packet processing actions performed inside the onReceivedPacket() method, as this is handled accordingly by the parser.



```
1  public class Antispam extends ForwardingHost{
2    @Override public void defineState() {
3          this.hostTableList.add(HostTable.createTable("Blacklist",1, FieldType.MAIL_FROM));}
4
5    @Override public RoutingResult onReceivedPacket(Packet packet) {
6          if(Packet.match(packet.getProtocol(), Constants.POP3_REQUEST_PROTOCOL, Operator.EQUAL)) {
7                return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);
8          }
9          if(Packet.match(packet.getProtocol(), Constants.POP3_RESPONSE_PROTOCOL, Operator.EQUAL) &&
10                     !this.state.hostTableList.get("Blacklist").contains(packet.getMailSource())) {
11               return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);
12         }
13         return new RoutingResult(packet,Action.DROP,ForwardDirection.UPSTREAM);
14         }
15 }
```

Figure 1. Java description of the behavior of the Antispam network function in response to a received packet

3.3. Parser

One of the most important aspects of our approach is the parser that analyses the user class describing the virtual network function. The parser operating principles clearly depend on the adopted verification tool. As said above, this paper considers logic-based tools. The rest of this section is then focused on this approach, but similar solutions might be applied to define parsing processes that are suitable for other formal verification techniques. This is part of our ongoing work and we plan to have a wider tool coverage in a near future.

The main functionalities of the parser in this framework are the following:
- the identification of the instructions in the Java code that lead to a packet being sent through an interface;
- the identification of the conditions (IF statements) that are traversed to reach the above mentioned send instructions.

In other words, we need to identify all the conditions that trigger a packet sending operation, which is actually what defines the behavior of a network function. We parse the source code to convert these conditions into a specific data structure, considering both the fields of the packets that traverse the function and the function status, if any.

In order to deliver the aforementioned functionalities, we take advantage of Eclipse AST API [15] in the extraction process from source code to a data structure. It is a tool that can generate an Abstract Syntax Tree (AST) [16] representation of existing Java source code. AST is a representation of a graph in the form of a tree from abstract syntactic structure of code. Using this library, our parser is able to represent every Java file as a tree of AST nodes. This step helps to perform a semantic analysis using the information in each node, where all these nodes are specialized for the symbolic events of the Java programming language. For example, there are nodes for method declaration, variable declaration, assignment, and others, while the edges describe the relationships between AST nodes. In other words, the parser helps to perform a semantic analysis using the information in each node. The parser recursively visits the AST of the code and stores in local variables all the characteristics such as: method declarations, variables, conditions, return predicates, and statements. Alternatively, the parser takes a "snapshot" of the current definition of the network function and proceeds in generating the final model in terms of high-level logical expressions. The final model is motivated by the vision of Open Flow [17] forwarding abstraction of the form <*match, action*>. This abstraction model has been borrowed from the existing modeling techniques [11, 18] and most of the verification tools of forwarding behavior [5, 6, 19] rely on the models adhering this abstraction.

To store the information obtained from the nodes, we defined a NFdefinition class. It is a sort of "container" where the characteristics of the VNF (e.g., available tables, type of the network function, protocols used for the packets etc.) obtained from the methods defineState() and defineSendingPacket() are stored in local variables. According to the structure of the onReceivedPacket() method, we store if-then statement blocks in a list of Implication objects. It is a basic form of implication and simply states that "if statement A is true, then statement B is also true" and separated with an implication (⇒) sign.

The Implication class contains the following set of condition objects and is created only in the presence of a forwarding action:

- *ifConditions* is a list of conditions that are related to a received packet, a packet received or sent in the past or an internal state of the node.
- thenConditions is a list of conditions that are related to a sending packet and storing instructions.
- result is a condition that contains an action to be performed.

In the next step, the list of Implication objects are partitioned into a list of implications before and after the implication (⇒) sign. The following rules serve as a guideline to this partitioning phase:

- Before the implication sign, there must be a send condition and all the conditions that regard only the sending packet may be present. The send condition is considered at time t_0.
- For every packet previously sent or received, a send or receive condition must be present after the implication sign. These conditions are considered at time t_1 < t_0
- For every store condition, another implication must be created. It has the store instruction before the implication sign and all the other conditions after the sign.

This is the way in which our framework extracts the required information from the user defined classes and using them creates an internal abstract representation of the network function. The further step, i.e., the final creation of the input file for the specific verification tool, consists of a simple translation from the abstract model to the specific input language. Our framework currently includes a translator for the VeriGraph tool and discussed in the following.

4. USE CASE AND RESULTS

4.1. Translation pattern. VeriGraph (Z3)

Among the existing logic-based verification tools, we selected VeriGraph [6] as a use case to show how the model generated by the parse can be exploited, after proper translation, by a real verification tool. VeriGraph is a formal verification tool that can automatically verify networks by checking certain properties before the real service deployment. VeriGraph exploits VNF models expressed as formulas in first order logic. These formulas are difficult to write. Hence, it can greatly benefit from the automatic generation of models. In this context, the term network is used to indicate a sequence of several network functions (NAT, web cache, firewall, IDS and so on) that starts from a source node and ends into a different destination node. In response to a verification request, a model of the network and the involved network functions, consisting of First Order Logic (FOL) formulas [20], is checked against the provided policies, for example reachability properties between two nodes in the network.

In order to achieve high performance, the verification engine exploits an off-the-shelf SAT solver (Z3), which determines whether the considered policies are satisfied or not, thanks to the translation of these problems into SAT problems.

VeriGraph requires VNF models written in a FOL-based formalism. Hence, we included in our framework a translator that takes elements stored in the NFdefinition object and converts them into FOL formulas, namely, into boolean constraints in the form of if-then rule-based conditional statements. For example, the onReceivedPacket() method is modeled according to this template:

```
SEND(p) -> CONDITIONS
```

This is a special language form that can be interpreted through a satisfaction relation. In particular, this kind of rule represents the operation of sending a packet by means of this recurring pattern:

```
send(VNF, destination, packet, t0) ->
recv(source, VNF, packet, t1) && t1 < t0
```

Send and recv are two specific methods defined in the VeriGraph framework that receive as an input two nodes representing the source and the destination of a packet, a packet and a time. The above formula means that VNF can send a packet to a given destination if it has previously received the packet. This is the starting point of the final rule that will be enriched during the analysis depending on the conditions needed to forward the packet. Table 1 represents these statements for the Antispam VNF, as an example. In essence, it is the translation of the Java code depicted in Figure 1. If the Antispam (row 1) can send a packet p to the node n0 at time t0 then the protocol type of the packet should be either POP3 request or response. In (2) packet p should have the address of the Antispam in its src field. In (3) if the Antispam can send a packet p to the node n0 at time t0 and if the packet protocol type is a POP3 response, then these two conditions imply that there exists another node n1 at another time t1, such that, Antispam received same packet p from another node n1 and emailFrom field is not in black list. Formula (4) can be described in a similar way.

Table 1. Translator output format for the Antispam VNF

1	send(Antispam, n_0, p, t_0) → p.proto = = POP_REQ ‖ p.proto = = POP_RESP
2	send(Antispam, n_0, p, t_0) → nodeHasAddr(Antispam, p.src)
3	send(Antispam, n_0, p, t_0) && p.proto(POP3_RESP) → (∃ n_1, t_1 : (recv(n_1, Antispam, p, t_1) && t_1 < t_0)) && !isInBlackList(p.emailFrom)
4	send(Antispam, n_0, p, t_0) && p.proto(POP3_REQ) → (∃ n_1, t_1 : (recv(n_1, Antispam, p, t_1) && t_1 < t_0))
5	isInBlackList(p.emailFrom) = = false
6	if(isInBlackList(p.emailFrom) = = or (for bl in list (p.emailFrom= =bl)) ? true : false

As it is done for each table object created in the Java code definition of the network function, the translator generates the interpretation of the isInBlackList() method as shown in formulas (5,6). By default, the method is assigned to a false value, that is equivalent to an empty table without any blacklist entries. Then, the clauses indicating the comparison between the emailFrom field of the packet for each table entries in the blacklist is added as a disjunction of new equalities to isInBlackList() in every loop iteration.

4.2. VNF catalog

The framework comes with a set of generic VNF models, written by means of our Java library during the development phase and used to evaluate the performance and the effectiveness of our method. All evaluations are executed on a workstation with 8GB RAM and an Intel i5-4210M CPU and, as described in the previous section. The available VNFs are listed in Table 2, together with the parsing time to generate each VeriGraph input model. It is worth noticing how these times are a satisfying result, also considering that the parsing process is not a real-time task and is executed only once during the data plane verification phase. In this subsection we describe some of the network functions included in the catalog.

Table 2. Time spent to parse VNF models

VFN model	Time to parse (ms)
Web Client	849
Mail Client	857
Mail Server	862
End Host	864
IDS	869
NAT	880
Web Server	920
Web Cache	952
Firewall	957
Antispam	963

IDS (intrusion detection system) VNF acts similar to the firewall but IDS performs application layer packet filtering. It is a reactive IDS that not only detects suspicious or malicious traffic and alerts the administrator but will take pre-defined proactive actions to respond to the threat. The IDS includes a table of type FieldType.BODY with a single column. This type of the table allows to store the data of Layer 5,6,7 in OSI network model. The IDS in this example, stores the data corresponding to the BODY field of the packet. If the protocol of the received packet is equal to HTTP REQUEST or HTTP RESPONSE the IDS performs a table lookup based on the BODY field of the packet. The presence of an entry corresponding to that data in the table results in a drop action of the packet. In addition, this is the structure of the code of IDS VNF share in common with Antispam VNF.

Web Server VNF checks if the protocol of the packet is HTTP REQUEST, creates a new packet by copying all the fields of the received packet. The source/destination IP and port addresses of the cloned packet are swapped and PROTOCOL field is set to HTTP RESPONSE. Analogously, MailServer VNF definition follows the same structure of the code.

Web Cache is a stateful VNF containing a table of two columns. On Received Packet () method of the class comprises four forwarding actions. The first two actions correspond to a if branch where the interface of the arrived packet is internal. If the PROTOCOL of the packet is equal to HTTP REQUEST and the table contains an entry matching the requested URL, the forwarding action is taken. This forwarding action is performed on internal interface with a new packet containing the requested web page. If the requested web page is not available in the table of the VNF, the original packet is forwarded through the external interface. On the other hand, the next two forwarding actions follow the else branch and without altering the packet. However, if the PROTOCOL of the packet is equal to HTTP RESPONSE, web content of the packet stored in the table of the VNF.

Firewall ACL (access control list) enabled stateless network function, contains table object aclTable of two columns, characterized by type FieldType.IP SRC and FieldType.IP DEST. The table acts as a “blacklist” and if the packet received matches the tuple in the table, a drop action is performed.

NAT (Network Address Translator)- enabled network function divides the network into two areas (an internal area and an external one) and applies different rules on the incoming packets if they are received from the internal or the external interface. In this case, packets received from the internal Nodes are always forwarded by replacing the source address field of the packet with the NAT IP address. Whereas, incoming packets from the external interface are forwarded to their destination only if a connection already exists.

4.3. Experimental results

In order also to check the correctness of the generated final models, we constructed a set of network topologies containing the available network functions and we demonstrate a number of custom tests on the selected verification tool. VeriGraph can perform different kinds of verification tests: reachability, which consists in checking if at least one packet can arrive at the destination from the source node, and isolation, which consists in verifying that no packet flowing from source to destination passes through a certain network function. In this section, we consider reachability properties. The overall network’s behavior (e.g., routing tables, middlebox configurations, host metadata, etc.) and the network topology information are represented as a set of additional FOL formulas and completed with other formulas that express the properties to be verified (e.g., a reachability property between two nodes in the network), in such a way that the satisfiability of the formulas implies the truth of the specified properties.

Figure 2 illustrates the set of topologies adopted for our tests. For example, topology (1) involves two firewalls and three end hosts. Firewalls are configured according to the following rules:

- Firewall 1 denies traffic between host A and host C.
- Firewall 2 denies traffic between host B and host C.

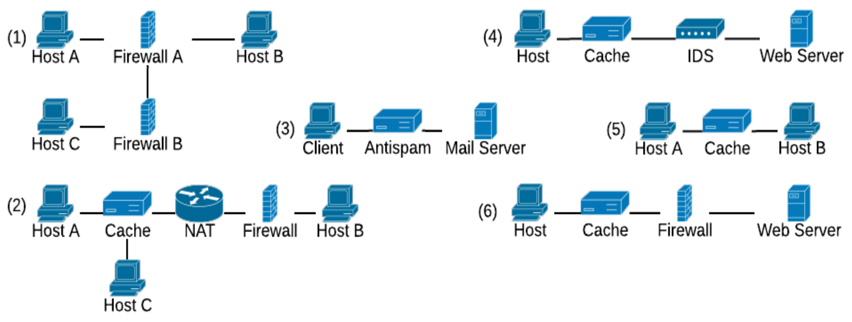


Figure 2. Set of network topologies created for the verification process

The test includes two reachability properties to be checked. In particular, we consider two packets, one flowing from node A to node C, and another flowing from node A to node B. Taking into account the above firewall policies, we expect the reachability property is not satisfied in case of A-to-C, indicating that no such packet can exist, while we expect it is satisfied in the case of A-to-B. The other test cases are set up as follows (the number in brackets refers to the corresponding topology in Figure 2):

1. Configurations: firewall denies traffic between host C and host B; the cache is set to serve any request (any URL of the resource requested is present in the cache); host A sends an HTTP REQUEST towards B; host A and C are set as internal nodes of NAT and cache. Property: reachability between host A and host B. Result: not reachable. Reason: cache never forwards packets towards host B.
2. Configurations: antispam blocks a traffic flow originating from (mail) client, thus there is a table entry with the address of the (mail) client in the blacklist of the network function. Property: reachability between mail server and (mail) client. Action: send a packet from mail client to mail server. Result: not reachable. Reason: antispam drops packets sent towards mail server.
3. Configurations: IDS drops a packet containing a specific string in the body of the packet; host sends a packet containing the specific string in the body. Property: reachability between host and web server. Result: not reachable. Reason: IDS drops packets containing the specific string in the body of the packet.
4. Configurations: requests cannot be served by the cache (the URL of the resource requested is not present in the cache). Property: reachability between host A and host B. Result: reachable. Reason: cache forwards packets, since requested packets are not present in cache table.

5. Configurations: firewall denies traffic between host and web server; cache is set to serve any request. Property: reachability between host and web server. Result: not reachable. Reason: firewall blocks the traffic originating from host and addressed to web server.

Table 3 delivers the results we obtained implementing these categories of tests and consider the two different approaches in defining the VNF models for VeriGraph. In particular, the table includes a first column referred to handcoded VNF models, and a second one referred to VNF models autogenerated by means of our framework. The string "SAT" means that the property stated in the test class is satisfied, while the string "UNSAT" refers to the case where the property is not satisfied. Comparing the test results between the hand-coded models and the automatically generated ones (starting from the Java description and then generated using the parser), we can notice how the obtained results are identical. This confirms the correctness of our modeling approach and also shows the efficiency of the developed framework. Column N represents the number of the corresponding topology illustrated in Figure 2

Table 3. Comparison of the verification results

N	Tests	Verification results using hand-coded VNF model	Verification results using autogenerated VNF model	Time to verify autogenerated VNF model (ms)
(1)	DoubleFwTest 1	UNSAT	UNSAT	214
	DoubleFwTest 2	SAT	SAT	
(2)	CacheNatFwTest	UNSAT	UNSAT	318
(3)	AntispamTest	UNSAT	UNSAT	275
(4)	IDSTest	UNSAT	UNSAT	192
(5)	CacheTest	SAT	SAT	260
(6)	CacheFwTest	UNSAT	UNSAT	200

5. CONCLUSION

This paper presents a "user-friendly" approach to VNF modeling for formal verification of VNF-based networks. We focus on breaking the barrier between the two ways of representing a VNF: the imperative-centric function definition (proper of VNF developers) and the more higher-level declarative representation (used by formal verification experts in order to instruct logic-based verification tools). The proposed approach consists of translating from the former one to the latter one automatically. Considering the ease of use, even for non-technical users and the reliability in terms of creation of the classes that describe the behavior of the VNF, it is possible to use the outcome of this project in other future, wider works that will allow transformation of the current structure of the network into a more flexible, simpler to manage and cheaper one. Considering what are the current requests of the market and looking at the possible future developments, this framework presents a further step towards the real implementation of these new concepts inside the networks. In fact, the framework and the available verification tools may be a basic structure to define Virtual Network Functions and test the overall network functionality before deployment.

REFERENCES

[1] E. G. N., V1.1.1, "Network Functions Virtualisation (NFV); Terminology," *IEEE Network*, vol/issue: 1(5), pp. 1-50, 2013. Available: http://ieeexplore.ieee.org/xpls/abs all.jsp?arnumber=4626228

[2] N. P. Lopes, et al., "Checking beliefs in dynamic networks," *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). Oakland, CA: USENIX Association*, pp. 499-512, 2015.

[3] S. Owre, et al., "Pvs: A prototype verification system," *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction, ser. CADE-11. London, UK, UK: Springer-Verlag*, pp. 748-752, 1992.

[4] H. Mai, et al., "Debugging the data plane with anteater," *Proceedings of the ACM SIGCOMM 2011 Conference, ser. SIGCOMM '11. New York, NY, USA: ACM*, pp. 290-301, 2011.

[5] R. Stoenescu, et al., "Scalable Symbolic Execution for Modern Networks," *Proceedings of the ACM SIGCOMM 2016 Conference*, pp. 314-327, 2016.

[6] S. Spinoso, et al., "Formal Verification of Virtual Network Function Graphs in an SP-DevOps Context," *Service Oriented and Cloud Computing - 4th European Conference, ESOCC 2015, Taormina, Italy*, pp. 253-262, 2015.

[7] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag*, pp. 337-340, 2008.

[8] J. C. Corbett, et al., "Bandera: ˘ Extracting Finite-state Models from Java Source Code," *Proceedings of the 22Nd International Conference on Software Engineering, ser. ICSE '00. New York, NY, USA: ACM*, pp. 439-448, 2000.

[9] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol/issue: 2(4), pp. 366-381, 2000.

[10] S. K. Fayaz, et al., "BUZZ: Testing Context-Dependent Policies in Stateful Networks," *13th USENIX Symposium on Networked Systems Design and Implementation. Santa Clara, CA: USENIX Association*, pp. 275-289, 2016.
[11] W. Wu, et al., "Automatic synthesis of nf models by program analysis," *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, ser. HotNets '16. New York, NY, USA: ACM*, pp. 29-35, 2016.
[12] J. Khalid, et al., "Paving the way for nfv: Simplifying middlebox modifications using statealyzr," *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation, ser. NSDI'16. Berkeley, CA, USA: USENIX Association*, pp. 239-253, 2016.
[13] M. Weiser, "Program slicing," *Proceedings of the 5th International Conference on Software Engineering, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press*, pp. 439-449, 1981.
[14] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, ser. PLDI '90. New York, USA: ACM*, pp. 246-256, 1990.
[15] "Eclipse JDT API Specification," Available: http://help.eclipse.org/helios/index.jsp
[16] E. Fauzi, et al., "Reverse engineering of source code to sequence diagram using abstract syntax tree," *2016 International Conference on Data and Software Engineering (ICoDSE)*, pp. 1-6, 2016.
[17] N. McKeown, et al., "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol/issue: 38(2), pp. 69-74, 2008.
[18] B. Tschaen, et al., "Sfc-checker: Checking the correct forwarding behavior of service function chaining," *NFV-SDN. IEEE*, pp. 134-140, 2016.
[19] A. Panda, et al., "Verifying isolation properties in the presence of middleboxes," *CoRR*, vol. abs/1409.7687, 2014.
[20] H. B. Enderton, "A mathematical introduction to logic," vol/issue: 40(2), pp. 317, 2001.

BIOGRAPHIES OF AUTHORS

Guido Marchetto is an assistant professor at the Department of Control and Computer Engineering of Politecnico di Torino. He got his Ph.D. in Computer Engineering in April 2008 from Politecnico di Torino. His research topics cover distributed systems and formal verification of systems and protocols. His interests also include network protocols and network architectures.

Riccardo Sisto received the Ph.D. degree in Computer Engineering in 1992, from Politecnico di Torino, Italy. Since 2004, he is Full Professor of Computer Engineering at Politecnico di Torino. His main research interests are in the area of formal methods, applied to distributed software and communication protocol engineering, distributed systems, and computer security. He has authored and co-authored more than 100 scientific papers. He is a Senior Member of the ACM.

Matteo Virgilio received the M.S. degree in Computer Engineering from Politecnico di Torino, Italy, in 2012. He got his Ph.D. in Control and Computer Engineering at Politecnico di Torino. His research interests include innovative network protocols and architectures, Content Centric Networking and formal verification techniques applied in the context SDN/NFV.

Jalolliddin Yusupov received the M.S. degree in Computer Engineering from Politecnico di Torino, Italy, in 2016. Currently, He is a Ph.D. student in Control and Computer Engineering at Politecnico di Torino. His primary research interests include formal verification of security policies in automated network orchestration. His other research interests include modeling, cyber physical systems, and cloud computing systems.