POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Translation from Java Virtual Network Function (VNF) models to specific models for Service Graph verification tools

Relatore
prof. Riccardo Sisto
Correlatore
prof. Guido Marchetto

Candidato
Alessandra Orrù

A.A. 2017-2018

Ai miei nipoti
A Tappi

Contents

Chapter 1

Introduction

1.1 Context

The IT world has made a huge leap forward in recent years. It has developed a plethora of "smart" services, starting a revolution in digital services. For example, systems that automate vehicle driving have been developed and smartphones are now able to recognize people's faces. Faced with this technological gap, networking must adapt these recent innovations. Actually, the network aims to connect and make available applications and services.

Nowadays, the network is still structured as in the early days, its language is limited by the set of bits that make up a packet. The "Session" level-5, defined by the ISO/OSI[1] standard, is used by the networks to interact with the applications. This level allows bidirectional exchange of packets and provides a limited number of attributes for control purposes (e.g. start time, bandwidth rate, etc.). To date, the wired network architecture does not allow implementing and managing the needs of new services, such as achieving low costs, increasing flexibility and speed, or performing more complex controls.

Therefore, the telecommunications world is evolving rapidly. Networks are changing their structure by replacing hardware devices in favour of software. Several network devices, such as firewalls and NATs, become applications executed on virtual machines [1]. At the current state of the art there is a growing diffusion and a continuous evolution of the paradigms of Network Function Virtualization (NFV) and Software-Defined Network (SDN). We will describe them briefly because this thesis work relates to this context. Firstly, the basic functionality of the paradigm known as SDN is the ability to program the network. This is a remarkable advantage because it allows a more flexible and reliable network management. Secondly, SDN provides a solution to add new applications in real time and without interruptions or reconfigurations of the whole network. With the SDN, network services are managed through the abstraction of their functionality. Also, the NFV paradigm was born with the aim of increasing the flexibility of networks. In fact, this network model allows IT professionals to modernize their networks with modular software to run on standard server platforms [2].

1 The Open Systems Interconnection (OSI) from Cisco DocWiki: *http://docwiki.cisco.com/wiki/Internetworking_Basics*

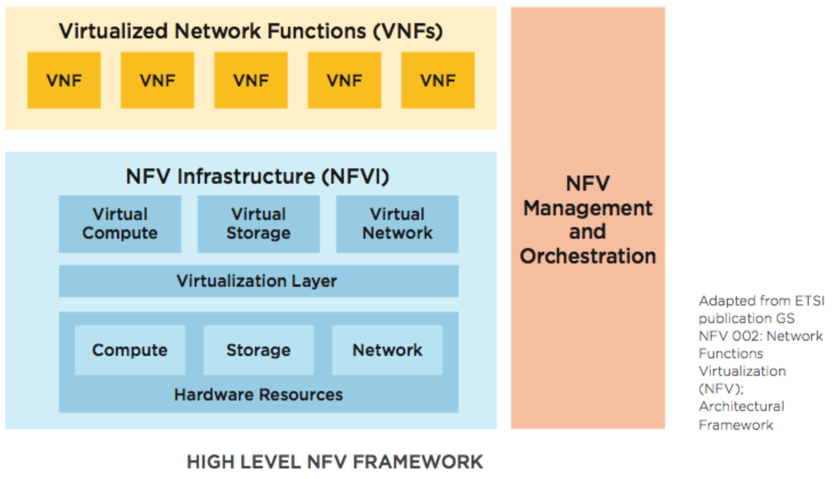

Virtualized Network Functions (VNFs)
VNF
VNF
VNF
VNF
VNF
NFV Infrastructure (NFVI)
Virtual Compute
Virtual Storage
Virtual Network
Virtualization Layer
Compute
Storage
Network
Hardware Resources
NFV Management and Orchestration
Adapted from ETSI publication GS NFV 002: Network Functions Virtualization (NFV); Architectural Framework
HIGH LEVEL NFV FRAMEWORK

Figure 1.1

The high-level NFV framework is shown in Figure 1.1 [3]. We can see that a NFV architecture has 3 different components:

- The virtualization infrastructure of the network functions such as the hardware and software infrastructure necessary to run the applications.
- The Virtual Network Functions (VNFs) which are software applications that provide specific network functions (e.g. firewall, NAT, anti-spammer, etc.).
- The management, automation and orchestration part of the network which is the real framework for the administration of the infrastructure and its functions (VNFs).

The main advantages of this architecture are the flexibility (elastic scale up and scale down of capacity), very fast deploy, more effective in power and space, reduction of costs in purchasing network equipment.

On the other hand, NFV turned out to be complex and hard to use. Network virtualization requires watchful verification before proceeding with the deployment, since it must safeguard and guarantee the consistency of the network (such as the absence of loops, preservation of traffic safety, etc.). The concepts of dynamism and security are becoming essential in modern networks. In fact, corporate networks are increasingly complex and are made up of thousands of integrated devices. Whether all these devices are not configured correctly, they can cause numerous types of errors on the network. For example, they can cause disconnections, loss of route, black holes and other security vulnerabilities. For that reason, there is a growing interest in the development of rigorous verification tools that can guard the correctness of the network configuration. There are sophisticated verification tools called SAT solver[2], which deal with a formal verification of the properties of virtual networks. These tools are specific and complex and are not always easily usable for those unfamiliar with formal methods.

2 Boolean satisfiability problem, from Wikipedia: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

There are no systems that allow you to easily create network functions and verify them before deployment. This feature is difficult to achieve especially in networks that need an automatic reconfiguration (in response to the user events or traffic flood).

Finally, we arrived at the framework object of the thesis. It is a tool that allows us to automatically extract a model of the network functions to generate the input files related to different verification instruments. More in detail, in the thesis work we want to demonstrate that the framework allows to model the network functions in a format that is generic enough to support the verification through different tools like VeriGraph[4], SymNet[5] and SFC-Checker[6].

1.2 Tools and Language

1.2.1 The eXtensible Markup Language

The eXtensible Markup Language (XML) is a meta-language developed by W3C and derived from SGML[3]. One of the main features about XML is that the data is standalone. This means that data incorporate the type information. XML language has many advantages. For example it is directly usable on the internet. Furthermore, it is widely open and compatible.

An XML document has a tree structure where each node is named *element.* An element can have *attributes.* And both the element and the attributes can carry data. The whole XML document must follow the SGML syntax in order to be well formed. For example, the attribute values are always enclosed in quotes. We can also check if the XML application is valid, that is when the XML document respects the given structure. This structure can be specified with a Document Type Definition (DTD[4]) that is a sequence of element declarations and attribute declarations called rules. Furthermore, there is a more powerful language to describe the rules of the XML document and it is the XML Schema[7]. Our framework uses an XML Schema [7] to describe the XML applications, so now we can proceed with the description of this language. The XML Schema[7] is itself an XML document and its basic structure has a root element called *schema*. This element has some important attributes that are used for define *namespace* and *targetnamespace*. The first one is the default namespace, it indicates that the elements and data types used in the schema come from the address associated with the attribute. The second one indicates that the elements defined by this schema come from the *targetnamespace*. When a *targetnamespace* occurs, the *elementFormDefault* attribute must be set as *qualified* to remember that all types must be qualified in the namespace. Valid nested element for the root are annotations, global element and attribute declaration, type and group declaration.

3 Standard Generalized Markup Language (SGML): https://www.w3.org/MarkUp/SGML/

4 We can found more information on DTD in the W3C tutorial available at https://www.w3schools.com/xml/xml_dtd_intro.asp

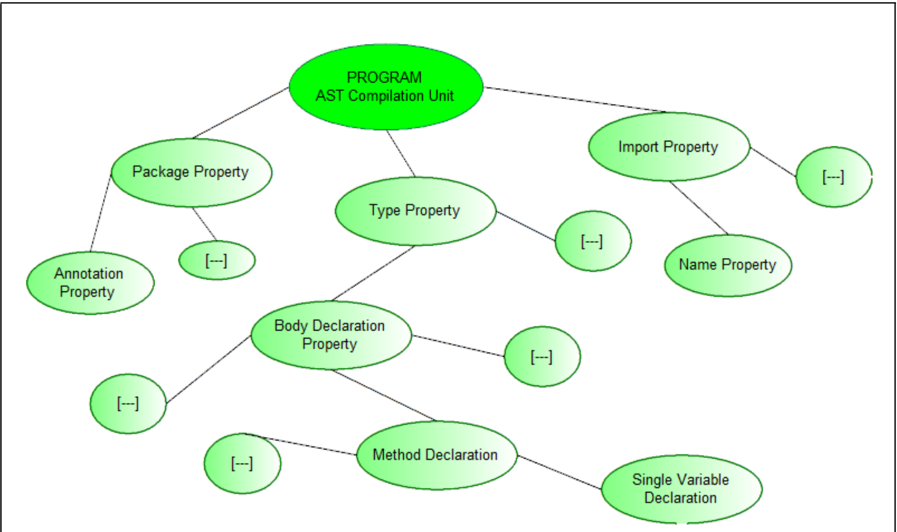


Figure 1.2

There are several techniques to write an XML schema:

- All elements are declared globally, and they can be used by other elements with the attribute *ref.* This structure is similar to the DTD style.
- Nested style is a simple structure that does not use types or references, all elements are anonymous and are inside the root element.
- Global style is a structure that declares all types globally with a concrete definition. Each element is nested in the root and is defined by a specific type. This style is used by our framework.

1.2.2 The Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a way to represent the source code of a program as a tree in which its elements are described with a specific programming language[14]. The Java DOM/AST[5] package is the set of classes that models the source code of a Java program such as a structured document. Figure 1.2 shows an example of AST of a general java program. In this thesis work the DOM/AST package is used in the translation phase of the network model from the Java language to a more generic representation in XML and vice versa.

1.2.3 The JAXB

The JAXB[6] is a Java API called the Java Architecture for XML Binding [13], a software framework that provides methods for writing Java content trees in XML documents and vice versa for reading XML documents in Java content trees. The writing phase is named marshalling and the reading phase is named unmarshalling.

5 Eclipse IDE documentation of Java DOM/AST: https://help.eclipse.org/oxygen/index.jsp
6 JAXB: https://docs.oracle.com/javase/tutorial/jaxb/intro/arch.html

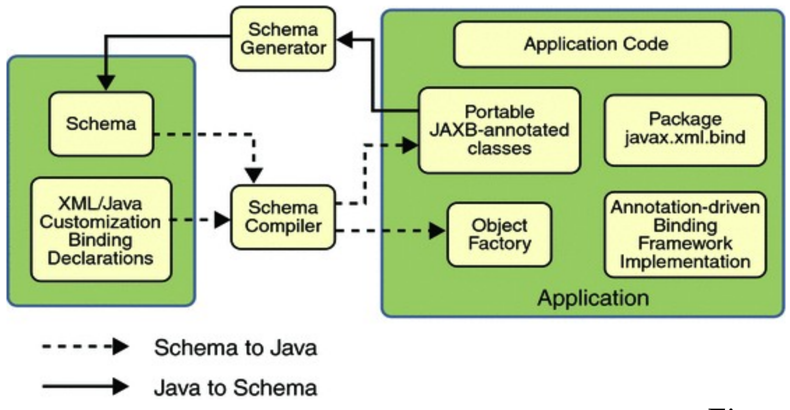


Figure 1.3

In Figure 1.3 is represented an architectural overview of JAXB. We can see a *Schema compiler* that has the role of associating a source schema with a set of program elements derived from the schema. The inverse operation of the *Schema compiler* is performed by the *Schema generator.* It maps a set of existing program elements with a derived schema. The mapping is described by program annotations. Another important element is the *Binding runtime framework* that provides unmarshalling and marshalling operations for accessing, manipulating, and validating XML content using existing program elements.

Marshalling and unmarshalling are two phases of the JAXB data binding process. An XML documents written according to the constraints in the source schema is unmarshalled by the JAXB binding framework. Unmarshalling provides a client application the ability to convert XML data to Java objects derived from JAXB. In contrast, the marshalling phase provides to a client application the ability to convert a Java object tree derived from JAXB into XML data.

1.3 Outline

The objective of this thesis is to increase the functionalities of the framework, for verification oriented VNF Modeling, already existing and in development. In the first part we tried to enrich the set of functions available in the library of the framework. We created a virtual network function that simulates the behaviour of a traffic classifier. This first part also aims to explain and understand the operation of the tool. In the second part of the thesis work, we wanted to show that the framework allows to generate network functions generic enough to be tested with different tools. Specifically for SymNet[5], SFC-Checker[6] and VeriGraph[4]. A semantic translation to SFC-Checker was performed and a parser was created to translate the VNFs into the format supported by the SymNet. In the last part of the thesis, some tests were developed to verify the correct operation of the carried out work. More specifically, several tests have been created to control the behaviour of the new Traffic Classifier function. It has been inserted in a specific VeriGraph service graph. Then, some tests were carried out to show that the output generated by the framework produces functions that are compatible with SymNet. Even in this case the generated network modules were used in a network graph that tests the reachability property.

Chapter 2

Proposed Framework

The tool object of this thesis is the framework for user-friendly verification-oriented VNF modeling. Its goal is to describe virtual network functions easily through Java classes and to translate these functions into a model in compliance with the formal verification made by several instruments (e.g. VeriGraph[4]).

The framework is available at: *https://github.com/netgroup-polito/vnf-modeling-verigraph*.

Before modifying the tool, we will go on describing how each module works. There are three main components:

- The Library is the module used by users to define their virtual network functions.
- The Parser analyses the classes written by users and produces an abstract formal model of the forwarding behaviour of these input functions. The function model is an XML document that should be complied to the XML Schema supplied by the tool.
- The Translator performs a conversion from the abstract format model of the network functions to the verification models. The first implementation of the tool generates a VNF model for VeriGraph[4]. One goal of the thesis is the development of a translator for the SymNet[5] and SFC-Checker[6] verification tools.

2.1 Library

The framework library[9] provides the resources to implement functions similar to existing network systems. The Java classes of the library are in the `it.polito.nfdev.lib` package.

Packet class represents an IP packet and illustrates the data exchanged between several network components that we want analyse and check. The class contains all the header fields of an IP packet (e.g. IP SOURCE, IP DESTINATION, PROTOCOL, etc.) as elements of a Java Map. In addition, there are some public constants that define the possible values for the header's fields. For example, `POP3_REQUEST` is an admissible value for the `PROTOCOL` field. However, this list of constants can be modified to cover a wider range of values. The methods of the class are the basic ones: constructors, getters, setters and *equalsField*. The last one checks if a header field is equal to a value and it is used to define the behaviour of the VNFs.

NetworkFunction is the core class of the library. A user must extend this abstract class to implement its own virtual network function. Each VNF can be associated with a set of logical interfaces through which it is possible to receive and send packets. Those interfaces can be tagged as internal or external to the network of the middle-box.

The interfaces are described by the *Interface* class. Each instance of this class has an identifier, the type of interface (internal or external) and an IP address. The *isInternal* method of the *Interface* class is useful for defining the behaviour of network functions. For example, NAT uses this function to differentiate packets coming from inside the network from those coming from outside the network. The operation of the virtual network function must be defined in the *onReceivedPacket* method of the main *NetworkFunction* class. The method has two parameters: the incoming packet and input interface. This method is the most important, because its result will be examined by the Parser to extract the information needed to build the middle-box policies. The *onReceivedPacket* returns an element of the *RoutingResult* class. It carries all the information about the process of network function. More specifically there are:

- The action, it is an enumeration that can be either `FORWARD` or `DROP`. It reports if the packet will be forwarded or discarded.
- The packet, eventually modified.
- The output interface.

Sometimes middle-boxes need to store information. For example, a firewall must be able to store the denied addresses. So, the library makes a container for these information through the *Table* and *TableEntry* classes.

2.2 Parser

The parsing of a virtual network function consists of 3 phases: it reads the input Java classes and extracts the information necessary to describe its behaviour, it creates the model and finally it writes the result of the transformation. The output model must respect the structure described by an XML Schema[7]. The XML Schema used by the parser is called *LogicalExpression.xsd* and is located in the `NFDev\xsd` folder. The schema has a *targetNamespace* declaration. Therefore, each type, element and attribute defined in the schema, at global level, will belong to this namespace. And we must use the target namespace label (`xmlns:tns`) for each type definition. In the definition of the XML schema, a global[1] style was used and the elements can be divided into 4 categories. The first category contains the basic element of the XML document, which is the root element (type: *ExpressionResult*). All the rules of the network function must be put in the root as elements of the type *ExpressionObject*. The second category contains all logical expressions. The possible expressions are defined through a series of complex types such as *LO_AND*, *LO_Equals*, *LO_Less_Than* and so on. The third category contains logical units. They model the elements used in the definition of a rule, that is the nodes of the network (type: *LU_Node*) and the packets that contain the data (type: *LU_Packet*). Finally, the last category of the model contains the elements related to the basic functions of a network device. For example, *LF_Send*, *LF_FieldOf*.

Firstly, the parser reads the Java source code of the network function and models it with an AST[14] representation. The main class of the parser is in the `it.polito.parser` package and is called *Parser*. This class has the instructions to create and initialize a new AST[14], the source file path is given as an argument to *Parser*'s main method.

1 Different types of XML Schema's structure at Chapter 1.2.1

The parser begins to visit the code in the root node of the AST. The root is the *CompilationUnit* created by the AST constructor. Moreover, the framework uses various methods of the Context classes to analyse the compilation unit. (e.g. *Context*, *MethodContext*, *StatementContext*, etc.). The second phase of parsing reads the gather data and produces the rules in the XML[2] format. The core class of this phase is *RuleGenerator* of the `it.polito.rule.generator` package. Its methods scan each *ReturnSnapshot* nodes and build the condition necessary to generate the final forwarding rule of the network function. The last phase stores the rules in the XML file with JAXB[13].

2.3 Translator

The parser generates a model of the forwarding rules, which can be used to verify the basic invariants of the network such as reachability and isolation. The first implementation of the framework executes a translation for the use case VeriGraph[4]. It is a formal verification tool for network properties such as reachability and isolation. VeriGraph sees the network as a sequence of network functions, and models each function with a set of first order logic formulas (FOL)[11] placed in a Java class. The role of the translator is to translate from the XML model to the Java model for VeriGraph.

ClassGenerator and *RuleUnmarshaller* are the Java classes that executed the translation and are located in `it.polito.rule.unmarshaller`. In order to perform the translation are used the methods implemented by the JAXB[13] framework.

The translation phase starts when the *Parser* creates a new instance of *ClsassGenerator*. This class generates the AST[14] nodes that represent the basic elements of the Java VNF class for the VeriGraph tool. The constructor initializes all the header fields of the Java output file. For example, it imports all the libraries used to process the network function (e.g. `import com.microsoft.z3.Solver`). Moreover, it assigns some information used to generate the output class, such as the network function name (*nfName*) variable. The main method of *Classgenerator* is *startGeneration*. It contains the Package Declaration and Type Declaration[3] for the new Java class. Furthermore, the method calls all the other methods of the *ClassGenrator* in order to create the AST[14] nodes of the features required by Verigrapgh[4] such as *initDeclaration*, *getZ3NodeDeclaration* and so on.

RuleUnmarshaller is the class that generates the policies of the virtual network function using the JAXB[13] framework. It reads the input XML document and create the VeriGraph rules. There is a method for each type of XML Schema. For example the type *LF_Send* is checked by the method *generateSend* that models the method for VeriGraph with the AST[14] elements such as *MethodInvocations* node and *Expressions*. The transformation starts at the AST[14] root node of the XML document and recursively scans all the nested nodes. The *getType* method identifies the node type and calls the appropriate method to generate the corresponding rule. In this thesis we want to adapt the translator to several verification tools. Therefore, the methods of the *ruleUnmarshaller* class change so that they translate the types into the format required by the verification tool.

2 General introduction to eXtensible Markup Language is provided in Chapter 1.2.1
3 Type Declaration and Package declaration are classes of org.eclipse.jdt.core.dom library.

Chapter 3

Library Extension

3.1 Traffic Classifier

Packet classifiers classify packets flowing through them according to policy and either select them for special treatment or mark them, in particular for differentiated services [Clark95, RFC 2475]. They may alter the sequence of packet flow through subsequent hops, since they control the behaviour of traffic conditioners[1]. In this thesis work, we define the abstract model of the Traffic Classifier network function, which will be described further.

3.1.1 Description

The Traffic Classifier is non-data driven VNF because it is supposed that its table is not updated by incoming traffic but it is written by the user. Data driven is an adjective used to refer a process that is determined by incoming data. More specifically, in our case, a middle-box is based on data if its forwarding rules table is updating entries dynamically with incoming traffic. On the other hand, it's a middle-box called non-data driven when the rules are static and independent of the traffic.

This middle-box can have many interfaces like in the following example. In Figure 3.1 there is a possible context where the Traffic Classifier has a behaviour like this: the `HOST_IN` send one packet with field application = `POP_REQUEST.` The traffic classifier has a policy table written by the user that contains some rules. For example, if the incoming packet has a POP_REQUEST protocol, it sends the packet via the `e4` interface. The Traffic Classifier checks its policies and decides whether to drop the incoming packet or send it through a specific interface.

We model the Classifier function in Java, a user friendly language. The class *Classifier* has a table named *ClassifierTable* where the user can write all the policies. This table has three columns:

- Priority, it is an integer type for defining the priority classification (e.g. 1=sensitive traffic, 2=best effort traffic, etc.).
- Application Protocol, it is a string type that represent the application protocol specified in the packet.
- Next Hop, it is an interface type of the middle-box. It is used to decide in which interface forward the packet.

1 Source: *http://www.rfc-editor.org/info/rfc3234*

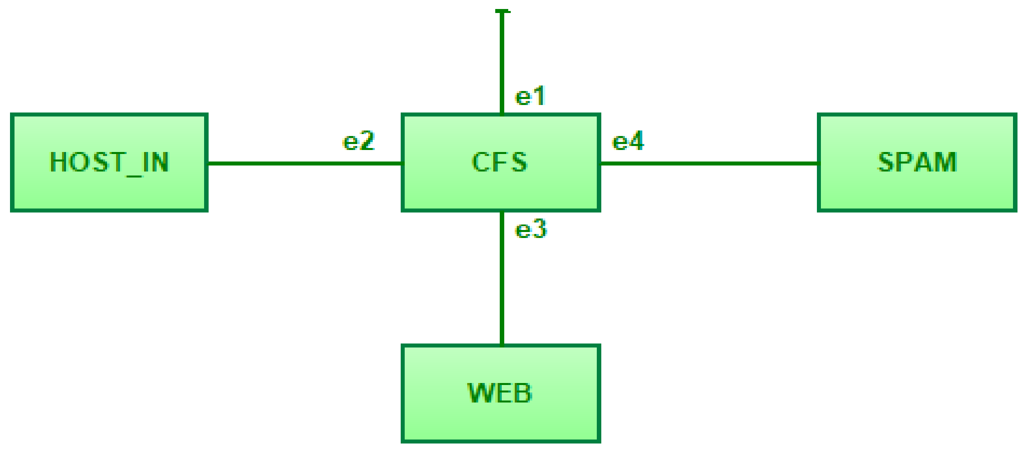


Figure 3.1

CFS = VNF Traffic Classifier; **SPAM** = VNF antispam; **WEB** = VNF Web Server;
HOST_IN = general host send/receive packet; **e1,e2,e3,e4** = Interfaces of VNF

To manage this Table there are the methods described below:

- addClassifierRule = Write one entry on the table
- removeRule = Remove one entry from the table
- clearTable = Delete the whole table
- getClassifierTable = Return the whole table
- setClassifierTable = Associate an already existing table to the middle-box

The main method of this class is *onReceivedPacket*. It implements the behaviour of the network function. First of all, it takes the *applicationProtocol* field of the incoming packet as an input. Then it check if there is a line corresponding to this protocol in the classifier table. When an entry exists, the middle-box can send the packet through the interface specified inside this entry. But if the output interface is the same as the input interface the middle-box drop the packet to avoid a loop. Finally, if the entry does not exist the packet is dropped, because it is unclassified.

```
public RoutingResult onReceivedPacket(Packet packet, Interface iface)
{
    TableEntry entry =classifierTable.matchEntry
                  (packet.getField(PacketField.APPLICATION_PROTOCOL));
    if(entry!=null){
            Interface ifSend = (Interface) entry.getValue(2);
            if(ifSend!=null && ifSend!=iface)
                    return new RoutingResult(Action.FORWARD,packet,ifSend);    }
    return new RoutingResult(Action.DROP,null,null); }
```

3.1.2 Future possible changes

- Use the port destination to classify the traffic.
- Modify the behaviour, for example one packet with unknown protocol should be send through one default interface.
- Implement a data driven classifier. It can scan the application layer (of the ISO model) of the packet, and create a new entry into the Classifier Table.

3.1.3 Updated Framework

The parser[2] analyses the classes that describe the virtual network functions. It generates an AST[14] representation of existing Java source code to perform a semantic analysis. The parser recursively visits the AST[14] of the code and stores in local variables all the characteristics such as: method declarations, variables, conditions, return predicates and statements. The output of the parser is an XML format model. It is a generic format that can be easily imported and unmarshalled[3] by any application.

The main behaviour of the Traffic Classifier network function is to select the output interface. The parser does not have an element or an attribute for this function. So it is necessary to introduce a new component. The first improvement to the previous parser is the introduction of the concept of output interface in the XML Schema[7]. Within the Core Element we need to update the complex type *LU_Node* that represents a logical node from which it is possible to send and receive packets. The basic structure has only a name attribute (*Value*). We add a new attribute to represent the interface, which is called *IF_OUT* [Listing 3.1]. The second step is to change the parser so that is possible to generate a *LU_Node* unit with an associated output interface. The parser adds one *IF_OUT* attribute in the XML rules only if the network function sends the packet through an interface different from the default one. The rule generator explores all nodes of the AST[14] tree and when it finds a variable declaration, it calls the method *generateRuleForVariable* of the *RuleContext* class. It has one switch case on the type name of the analysed variable. We need to add a case for the interface type in order to generate the corresponding rule in the XML document. The code in Listing 3.2 shows how to write a new rule about the interface selection. There is a flag variable called *ifSend*. It is declared as a global variable and its role is to memorize whether the output interface is already set up. The method check if the flag has already been set. Afterwards there are some instructions to set negation, just in case the variable is in a negative declaration line. The negation is set at lines 8 to 10 where a new expression is set to the value *not*. The rule is generated at lines 15 to 30 with the construction of right and left expressions. The right-hand expression corresponds to the *IF_OUT* attribute of the XML Schema and the left-hand expression is the name of the selected interface. The method *setLastExpression* at line 31 adds the rule to the model. Moreover, to complete the model, we have introduced the switch case to identify the interface type in the *checkVariabl* and *getVariable* methods of the *RuleContext* class.

2 See Chapter 2.2 Parser
3 See Chapter 1.2.3 The JAXB

Listing of Chapter 3

```
<complexType name="LU_Node">
        <complexContent>
              <extension base="tns:LogicalUnit">
                    <attribute name="Value" type="string"></attribute>
                    <attribute name="IF_OUT" type="string"></attribute>
              </extension>
        </complexContent>
</complexType>
```

Listing 3.1

```
case Constants.INTERFACE_TYPE:
 if (!ifSend) {
 if (operator.equals(Operator.EQUALS))
        negated = true;

        ExpressionObject exp = factory.createExpressionObject();

        if (negated) {
              LONot not = factory.createLONot();
              not.setExpression(factory.createExpressionObject());
              exp.setNot(not);
              exp = not.getExpression();
        }

        if (containsLogicalUnit(netFunction)) {
              LOEquals equals = factory.createLOEquals();
              equals.setLeftExpression(
                                 factory.createExpressionObject());
              equals.setRightExpression(
                                 factory.createExpressionObject());
              if (checkVariable(Constants.INTERFACE_ID)
                                  .compareTo(Constants.NONE) != 0) {
                    equals.getLeftExpression()
                                 .setFieldOf(fieldOf(
                                 netFunction, Constants.IF_OUT));
                    equals.getRightExpression()
                                 .setParam(Constants.INTERFACE_ID);
                    exp.setEqual(equals);

                    setLastExpression(exp);
                    this.ifSend = true;
              }
        }
 }
 // .…
```

Listing 3.2

Chapter 4

Translator Extension

4.1 Network Verification Tools

In this chapter we will look at how some verification tools work and how they can interact with the VNF Modeling framework[16]. The aim of the thesis is proved that the VNFs of our framework can be easily places, after a suitable translation, as modules on different verification tools. A formal verification system provides a mechanism to prove or disprove the correctness of a system's algorithms mathematically. It must verify that the system respects the properties specified for it and usually returns an abstract mathematical model of the system. The most popular Satisfiability Module Theories (SMT) solver is Z3[10]. It is released by Microsoft Research and its role is to decide if a set of formulas is satisfiable or not. In the case of networks, we want to create a graph that connects different hosts, which can be simple end-nodes or specific network functions that modify the traffic. Given a particular configuration of the network, formal verification must prove the properties of reachability and isolation. The reachability property controls that a host can send packets to another host. In contrast, the isolation property controls that a source host can not send data to a specific destination. VeriGraph[4] is one of the verification tools available. It models the network as a set of First Order Logic (FOL) formulas that are analysed by Z3[10]. Another verification tool, based on Z3[10], is SymNet[5]. It is a network analysis system based on symbolic execution. We also analyse another tool that uses its own verification algorithm, it is called SFC-Checker[6].

4.2 SFC-Checker

4.2.1 Description

The main objective of this tool is checking the correct forwarding behaviour of Service Function Chain (SFC). The SFC-Checker[6] can verify stateful service chain, by analysing the middle-boxes' stateful forwarding behaviour. It examines whether packets' flows are forwarded correctly according to the service chaining policies. More specifically, the SFC-Checker[6] can control the sequence of network functions (NFs) any flow should traverse and the NFs configurations. It is interesting that the service chain can be altered dynamically. SFC-Checker[6] only handle the forwarding behaviour of an SFC and this is a similitude with our VNF Modeling framework[16].

This verification tools is focuses on checking stateful reachability invariants.

The aim of this thesis work is demonstrated that the VNFs models of our framework are general enough to work in several network verification tools. Therefore, now we will see how the SFC-Checker[6] models the network functions to highlight the similarities with our models.

The NF abstraction illustrated on SFC-Checker[6] consist of two parts: a match action table and the state machine. The match-action table contains the NF's rules, which can:

- check the packet header and the internal states
- perform the action on packets
- change the internal state

The SFC-Checker[6] rules are modeled with a set of primitives. We list these primitives and their description in Table 4.1.

Type	Name	Description
Units	`f`	`Flow f`
	`f.p`	`Packet p in flow f`
	`f.p.field`	`Header field in packet p in flow f`
State Operations	`set(f,val)`	`Set f's state to val`
	`get(f)`	`Get f's state`
	`timeout(f,val)`	`f's state is removed after val ms`
Pre-Conditions	**`IF(f.p,P)`**	`Match f.p on pattern P`
	`IF(f,P)`	`Match f on P`
	`IF(protoAnalyzer(f),P)`	`Match event from protoAnalyzer on P`
Actions	**`Modify(f.p.field,val)`**	`Modify the header field in packet p in flow f with value val`
	`forward(f.p,port)`	`Send the packet p in flow f through the port`
	`drop(f)`	`Drop the flow f`
	`encap(`f_{in}`,`f_{out}`)`	`Operation of encapsulation and decapsulation`
	`decap((`f_{in}`,`f_{out}`)`	
	`rate_limit(f,val)`	

Table 4.1

4.2.2 Semantics Translation

The behaviour of our VNFs is modeled as match-action tuples such as in SFC-Checker[6]. Now it is possible to generate a mapping between the rules generated by our parser and those used by SFC-Checker[6]. We list the mapping for the following virtual network functions: AclFirewall, Antispam, IDS and Mail Server.

Each VNF in the list contains the description of forwarding behaviour, the match-action rules of our framework and the SFC-Checker[6] match-action rules.

AclFirewall

This is a non-data driven middle-box. In its Table it stores a pair of IP addresses that represent the source and destination of the packet. A packet in ingress on this middle-box is dropped if the source address of the packet and the destination address of the packet are equal from one pair in the Table of the AclFirewall.

In Table 4.2 we can find the match-action rules of the firewall for both VNF Modeling and SFC-Checker[6]. We can easily map the *LF_Send* VNF Modeling type in a *forward* SFC-Checker[6] primitive and the *LF_MatchEntry* VNF Modeling type in an *IF* SFC-Checker[6] primitive. Our AclFirewall function model can be translated in SFC-Checker[6] primitive rules.

- AclFirewall - *Match-action in VNF Modeling*			- AclFirewall - *Match-action in SFC-Checker*
Match	Flow	**recv**(p_0,INTERNAL)	**IF**(f.p.src,IP_SRC), **IF**(f.p.dst,IP_DST)
		!(**matchEnty**(p_1.IP_SRC, p_1.IP_DST))	
Action	Flow	**send**(p_0,EXTERNAL)	**forward**(f.p,out)

Table 4.2

Antispam

This is a non-data driven middle-box. In its Table it stores one list of words that represent a black-list. It performs application layer packet filtering. The Antispam controls two fields of the incoming packet. Firstly, it checks if the transport protocol field is equal to `POP3_REQUEST` or `POP3_RESPONSE`. Secondly, it checks if the field *EMAIL_FROM* produces a lookup match within the Antispam table. Whether the two tests are true, the packet it is dropped. Otherwise, it is forwarded.

In Table 4.3 we can find the match-action rules of the antispam for VNF Modeling and in Table 4.4 we can find the match-action rules of the antispam for SFC-Checker[6]. The VNF Modeling types and the SFC-Checker[6] primitives used in this network function are the same used in the AclFirewall, so it is possible translate the model.

- Antispam - *Match-action in VNF Modeling*		
Match	Flow	**recv**(p_0,INTERNAL)
		p_1.PROTO == POP3_REQUEST||p_1.PROTO == POP3_RESPONSE
		!**matchEnty**(p_1.EMAIL_FROM)
Action	Flow	**send**(p_0,EXTERNAL)

Table 4.3

- Antispam - *Match-action in SFC-Checker*	
Match(f,s)	**Action**
`IF(f.p.proto,POP3_REQUEST), !IF(f.p.emailFrom,BLACK-LIST)`	`forward(f.p)`
`IF(f.p.proto,POP3_RESPONSE), !IF(f.p.emailFrom,BLACK-LIST)`	`forward(f.p)`

Table 4.4

Intrusion Detection System

The Intrusion Detection System (IDS) is similar to the previous Antispam network function. But on the contrary, it stores a black-list of URLs in the Table. This middle-box controls two fields of the incoming packet. Firstly, it checks if the transport protocol field is equal to `HTTP_REQUEST` or `HTTP_RESPONSE`. Secondly, it checks if the field *URL* produces a lookup match within the IDS table. Whether the two tests are true, the packet it is dropped. Otherwise, it is forwarded. The VNF Modeling types and the SFC-Checker[6] primitives used in this network function are the same used in the Antispam, so it is possible translate the model.

Traffic Classifier

This is a non-data driven middle-box and the Java model of the NF was described in Section 3.1.1. In its Table it saves pairs made up of a transport protocol and an interface. The incoming packet on this box is deleted if there are no pairs in the table corresponding to the *TRANSPORT_PROTOCOL* field of the packet. Otherwise, the packet is forwarded through the interface specified in the pair. In Table 4.5 we can find the match-action rules of this middle-box for both VNF Modeling and SFC-Checker[6]. The VNF Modeling types and the SFC-Checker[6] primitives used in this network function are the same used in the Antispam, so it is possible translate the model.

- Traffic Classifier - *Match-action in VNF Modeling*			- Traffic Classifier - *Match-action in SFC-Checker*
Match	Flow	`recv(p_0,INTERNAL)`	`IF(f.p.proto,PROTO)`
		`!matchEnty(p_0.PROTO)`	
Action	Flow	`send(p_0,EXTERNAL)`	`forward(f.p,out)`

Table 4.5

Mail Server

This is a non-data driven middle-box. The network function Mail Server, compared to the previous functions, does not have a rules Table. It has the following behaviour. First of all there is a check on the *TRANSPORT_PROTOCOL* field of the incoming packet. If it is the same as `POP_REQUEST`, it makes some modifications to the packet.

More in detail: it swaps the *ip-address-source* with the *ip-address-destination*, assigns the value `POP_RESPONSE` to the *TRANSPORT_PROTOCOL* field, assigns the value `RESPONSE` to the *EMAIL_FROM* field. Finally, it forwards the packet. Otherwise, if the transport protocol is not `POP_REQUEST`, the packet is dropped.

In Table 4.6 we can find the match-action rules of this middle-box for both VNF Modeling and SFC-Checker[6]. We can easily map the *LF_Equal* VNF Modeling type in a M*odify* SFC-Checker[6] primitive. The others mapping of the types and primitives are like in the previous middle-boxes. Therefore, the Mail Server can be translated from our model to the SFC-Checker[6] model.

- Mail Server - ***Match-action in VNF Modeling***			**- Mail Server -** ***Match-action in SFC-Checker***
Match	Flow	**recv**(p_0,INTERNAL) p_0.PROTO == POP3_REQUEST	**IF**(f.p.proto,POP3_REQUEST)

Action	Flow	p_0.SRC = p_0_DST
		p_0.DST = p_0_SRC
		p_0.PROTO = POP3.RESPONSE
		send(p_0,EXTERNAL)
			Modify(f.p.src,f.p.dst),
			Modify(f.p.dst,f.p.src),
			Modify(f.p.proto,POP3_RESPONSE),
			forward(f.p,out)

Table 4.6

Overall, we list the mapping from our model to SFC-Checker[6] model in order to summarise all possible rule's elements and highlight the possibility to build a translation (Table 4.7).

VNF Modeling' types	SFC-Checker' primitives
LU_Packet	Unit: f.p
Field	Unit: f.p.field
TableFields	Patterns: p
Param	Patterns: p
LF_IsInternal	Pre-Conditions: IF(f.p.src,INTERNAL)
LF_Send	Action: forward(f.p,out)
LF_MatchEntry	Pre-Condition: IF(f.p.field,P)
→ LO_Equals(p.1==x) → LO_Equals(p.0==x)	Pre-Condition: IF(f.p.field,P) Action: Modify(f.p.field)
LO_And	Pre-Condition
LO_Not	Pre-Condition: !IF(someting)
LO_Or	Pre-Condition
LF_FieldOf	Action: IF(f.p.field,p)

Table 4.7

4.3 SymNet

In this thesis work we want to demonstrate that the VNF Modeling framework[16] is able to serialise the abstraction of the network function model across different languages and is able to target different verification programs. SymNet[5] has several similarities with our framework, so in this chapter we will evidence these similarities, and we make a translation from VNF Modeling to SymNet.

4.3.1 Description

SymNet[5] is a network static analysis tool based on symbolic execution. It is written in Scala language[1]. The network behaviour is described through SEFL[2] instructions. Static analysis of network data planes is interesting for different reasons. For example, it allows economic, fast and very thorough verification of packet reachability. It guarantees the absence of loop and the bidirectional forwarding.

Symbolic execution explores all possible values for each variable at every point. In the context of network the symbolic analysis can detect the incoming packet headers to check if they are allowed or whether need to be modified. Moreover, it is interesting to know that in SymNet each path must be connected to an active packet passing through the network's nodes.

The only disadvantage of symbolic execution is the scalability. This is because the symbolic execution explores all possible paths through the program, providing possible values for each variable at each point. To reduce this problem SymNet uses a *model* (a simplified representation of the forwarding behaviour of network functions through a limited set of instructions) of the code. One way used by SymNet, to reduce complexity, is unfolding loops and executing both branches of an "if" conditional instructions. This means repeating a set of instructions for each iteration of the loop and executing both the true and false branches of the if statements.

The SEFL language simplifies programming and modelling network boxes. To analyse a network configuration, SymNet requires as input the descriptions of all network elements and their unidirectional connections. Values in SymNet can be concrete or symbolic. Each value has a unique identifier and a list of constraints associated with it.

The network verifications performed by SymNet are reachability, loop detection, header visibility (analysis of value stack of header field) and header memory safety (control the *tags* setting). Finally, SymNet uses the Microsoft Z3[10] solver to check the network model constraints described with SEFL instructions. We have chosen this tool to demonstrate that our network modelling allows for a simple translation to different verification tools. The user definition of a friendly Java model can easily be translated into the SymNet format.

The SymNet tool is easy to install and use. It is available to the repository *https://github.com/nets-cs-pub-ro/Symnet*. There is more information about the installation of SymNet in Chapter 5.1.1.

1 Scala is a new object-oriented and functional programming language. *https://www.scala-lang.org/*
2 See Chapter 4.3.2 SEFL Set Instructions

4.3.2 SEFL Set Instructions

The Symbolic Execution Friendly Language (SEFL) is a language designed specifically to SymNet[5] and it models the network's behaviour. It generates safe code. In fact, its memory usage is bounded and its termination is guaranteed.

Below there are all the instructions provided by SEFL:

- **Allocate(v[,s,m]).** Allocates new stack for variable *v*, of size *s*. If *v* is a string, the allocation is handled as metadata and the optional *m* parameter controls its visibility; it can be global (default) or local to the current module. If *v* is an integer it is allocated in the packet header at the given address; size is mandatory.
- **Deallocate(v[,s]).** Destroys the stack of variable *v*. If provided, the size *s* is checked against the allocated size of *v*. The execution path fails when the sizes differ or there is no stack allocated for variable *v*.

These instructions create header fields and metadata. To simplify access to header fields and to enable layering, any number of *tags* can be defined.

- **Assign(v,e).** Symbolically evaluates expression *e* and assigns the result to variable *v*. All constraints applying to variable *v* in the current execution path are cleared.
- **CreateTag(t,e).** Creates tag *t* and sets its value *e*, where *e* must evaluate to a concrete integer value.
- **DestroyTag(t).** Destroys tag *t*.
- **Constrain(v,cond).** Ensures that variable *v* always satisfies expression *cond*. The execution path fails if it doesn't. Allows programmers to model filtering behaviour without branching.
- **Fail(msg).** Stops the current path and prints message *msg* to the console.
- **If (cond,i1,i2).** Two execution paths are created; the first one executes *i1* as long as *cond* holds. The second path executes *i2* as long as the negation of *cond* holds.
- **For(v in regex,instr).** Binds *v* to all map keys that match *regex* and executes instruction *instr* for each match. The loop is unfolded before execution.
- **Forward(i).** Forwards this packet to output port *i*.
- **Fork(i1,i2,i3,…).** Duplicates the packet and forwards a copy to each output port *i1*, *i2*, etc.
- **InstructionBlock(i,…).** Groups a number of instructions that are executed in order. Should be used that groups more instructions into a single compound instruction.
- **NoOp.** Does nothing. If any branch is empty, *NoOp* can be used instead.

SEFL only supports simple expression such as referencing, subtraction, addition and negation.

We can find the implementation of the SEFL instruction described above in the folder of the SymNet[5] project called:

```
.\Symnet\src\main\scala\org\change\v2\analysis\processingmodels\instructions
```

4.3.3 Mapping

SymNet[5], like our framework[16], models the network forwarding behaviour in the form of "match-entry" formalism. We can easily translate the rules from our network's models to the SymNet models. The SEFL model for network uses a packet layout that mimics real implementations. Packet headers are variables, but each header has an absolute offset at which it is allocated. All SEFL headers must be allocated individually including the size. Both SymNet and VNF Modeling[16] use the canonical network fields in the packet definition, they are list below:

- VNF Models: IP_SRC, IP_DST, PORT_SRC, PORT_DST, PROTO, ORIGIN, ORIG_BODY, BODY, INNER_SRC, INNER_DEST, SEQUENCE, EMAIL_FROM, URL, OPTIONS, ENCRYPTED,
- SymNet[5]: ("IP-Src", "IP-Dst", "IP-Proto", "IP-TTL", "IP-Version", "IP-IHL", "IP-Length", "IP-ID", "IP-Checksum","Port-Src", "Port-Dst")

As we can see, the network fields are clearly similar and it is easy to modify them according to the requirements. VNF Modeling represents a packet with a complex type *LU-Packet* that contains a different attribute for each field. Also in SymNet, the *packet* class contains a different SEFL *Tag* for each field. For example, we can see the statements used to allocate the IP source address in the first row of the Table 4.8.

	VNF Modeling	SEFL
2	`<FieldOf> <Unit>p_0</Unit> <Field>IP_SRC</Field> </FieldOf>`	`CreateTag("IPSrc"), Tag("L3HeaderStart") + 96), Allocate(Tag("IPSrc"), 32), Assign(Tag("IPSrc"),SymbolicValue())`
3	`<Send> <Source>n_AAA</Source> <Destination>n_0</Destination> <Packet_out>p_0</Packet_out> </Send>`	`Forward("default"))`
4	`<Equal> <LeftExpression> <FieldOf> <Unit>p_1</Unit> <Field>PROTO</Field> </FieldOf> </LeftExpression> <RightExpression> <Param>HTTP_RESPONSE</Param> </RightExpression> </Equal>`	`Constrain(Proto, :==:(ConstantValue( HTTPREQUEST.value)) )`

Table 4.8

The network's policies are described by several complex types in the VNF Modeling such as the *LO_Equals*, the *LF_MatchEntry*, the *LF-Send,* etc. In contrast, SymNet uses a combination of SEFL instructions such as *Constrain*, *If*, *Fail,* etc. Below we going to list the main rules implemented by the two tools.

- **Forwarding rule**

An important behaviour of network functions is the ability to send packets through a specific port or interface. The VNF Modeling uses the *LF-Send* type to model the logical forwarding functions. It has three attributes with which indicates the source, the destination and the outgoing packet. In comparison, SymNet[5] uses a SEFL *Forward*. These instructions can be found in the second row of the Table 4.8.

- **Equality rule**

The VNF Modeling uses the *LO_Equals* type in order to check whether a header field of the incoming packet has a specific value. In SymNet[5] the equivalent function is the SEFL *Constrain*. For instance, in Table 4.8 (line 3) we can see how the two tools represented the equality rule for the *Protocol* field.

- **Lookup rule**

The *LF_MatchEntry* type checks whether the incoming packet has some fields that are the same as those in the VNF's table. This function has a variable number of parameters. These are the header fields used in the lookup test. The VNF's table can be a black-list or a white-list. When the *LF_MatchEntry* is positive, the VNF's table is a white-list because the packets that do not match are dropped. In contrast, when *LF_MatchEntry* is in a negation statement, the VNF's table is a black-list because packets that have a match are dropped. There is no corresponding function in SymNet, so we have to remake it with the basic functions of the SEFL language. The algorithm of the positive *LF_MatchEntry* function scans the entire VNF's table to see if it contains the values of the incoming packet. SEFL statements do not have loops, so we need to create a function with the scala language that runs the loop and builds a SEFL statement for each iteration. At the end, we will have a SEFL instruction for each row and column in the table. The new scala method is called *addrule* [Listing 4.1]. This method receives a list of parameters that are the fields in the VNF table. In line 6 of the Listing 4.1 there is the loop that iterates all table's entries. In line 7 there is the SEFL *InstructionBlock* generated for the current iteration. This *InstructionBlock* contains a variable numbers of SEFL *Constraint*, one for each field in the entry. Whether all the entry match with the fields in the packet, the *addrule* method generates a SEFL *Fail*. (line 13 of the Listring 4.1). In contrast, whether the packet's fields do not match, the *addrule* generates a SEFL *NoOp* instruction which means to proceed (line 14, 15 of the Listing 4.1). The *addrule* method returns a list of all the SEFL *InstructionBlock.* The algorithm of the negative *LF_MatchEntry* function returns a SEFL *Fail* only if there is no match at the end of the table scan [Listing 4.2]. To do this we need a flag. When all the entry match the packet's fields, the SEFL *Assign* sets the flag to true (line 10 of Listing 4.2). There is an extra SEFL *Constrain* (line 18 of Listing 4.2) to check the flag's value at the end of the lookup in the table. Whether the flag is false, the SEFL *Fail* statement is build, otherwise the SEFL *NoOp* is build.

- addrule method for black-list VNF's table -

```
def addrule(p:List[ConfigParameter]): Array[InstructionBlock] = {
    var rule: Array[InstructionBlock] = null
    var rules = Array(InstructionBlock(Nil))
    val limit = p.length / 2
    var i = 0
    while (i <= limit) {
       rule = Array(InstructionBlock(
       If(Constrain(
            IPSrc, :==:(ConstantValue(ipToNumber(p(i + 0).value)))),
          InstructionBlock(
             If(Constrain(
                IPDst, :==:(ConstantValue(ipToNumber(p(i + 1).value)))),
                Fail("Match-in-blacklist"),
                NoOp)),
          NoOp)))
       rules = Array.concat(rules, rule)
       i = i + 2
    }
    rules
  }
```

Listing 4.1

- addrule method for white-list VNF's table -

```
def addrule(p:List[ConfigParameter]): Array[InstructionBlock] = {
   [...]
    while (i < limit) {
      rule = Array(InstructionBlock(
      If(
         Constrain("flag", :==:(ConstantValue(0))),
         InstructionBlock(If(
              Constrain(Proto,:==:(ConstantValue(p(i + 0).value.toInt))),
              InstructionBlock(
                Assign("flag", ConstantValue(1)),
                Assign("idIfSend", ConstantValue(p(i + 1).value.toInt))),
              NoOp)),
         NoOp)))
      rules = Array.concat(rules, rule)
      i = i + 2
    }
    rule = Array(InstructionBlock(
    If(Constrain("flag", :==:(ConstantValue(0))),
       Fail("No-Match"),
       NoOp)))
    rules = Array.concat(rules, rule)
    rules
  }
```

Listing 4.2

4.3.4 AST Translation

In this chapter, we are going to describe how translate the virtual network function from our framework[16] to SymNet[5]. The translation of a network model takes place in three distinct phases:

1. In the first phase, the network model written by the user in Java is translated in the most generic XML format.
2. In the second phase, the generic network model is translated into a Java class that contains the SEFL statements. We call this intermediate model *SEFLinJava*.
3. In the third phase, the 'SEFLinJava' model is translated in Scala language. This last phase consists of two steps. The first step translates the 'SEFLinJava' class from the Java language to the Scala language. The second step simply have to reassign the name of some SEFL functions that are not usable in Java.

We skip the first phase because it is described in the previous work[16]. In the second phase of the translation we use the JAXB framework [13] to access the XML document of the VNF's rules. Moreover, we use the Java package org.eclipse.jdt.core.dom[14] to generate the new SEFLinJava class. Now we briefly describe the main AST[14] classes. We can find all variables and methods of these classes in the UML diagram in Figure 4.1 and Figure 4.2.
The SEFLinJava class of all virtual network functions translated has the same structure. It contains:

- **A Compilation Unit node type**

This type is the root of AST.

```
this.cu = ast.newCompilationUnit();
```

- **A Package Declaration AST node type**

It indicates the name of the SymNet package in which all the translated VNFs will be inserted in order to be verified by the tool. The translator uses only the method *setName*.

```
PackageDeclaration pd = ast.newPackageDeclaration();
pd.setName(ast.newName(new String[]
      { "org", "change", "v2", "abstractnet", "click","sefl" }));
cu.setPackage(pd);
```

- **Import Declarations**

The translator uses the Import Declaration AST type node to add all import statements needed to run the new network function in the SymNet environment. In more detail, there are imported the package of SEFL instructions description, the package of State used for the verifications, the package of conversion and definitions. The method used is *setName* where the *name* is the package to be imported.

- **Method Declaration**

The translator builds the public *generate_rules* method which receives the configuration of the network functions as parameter and returns the VNF's SEFL rules.

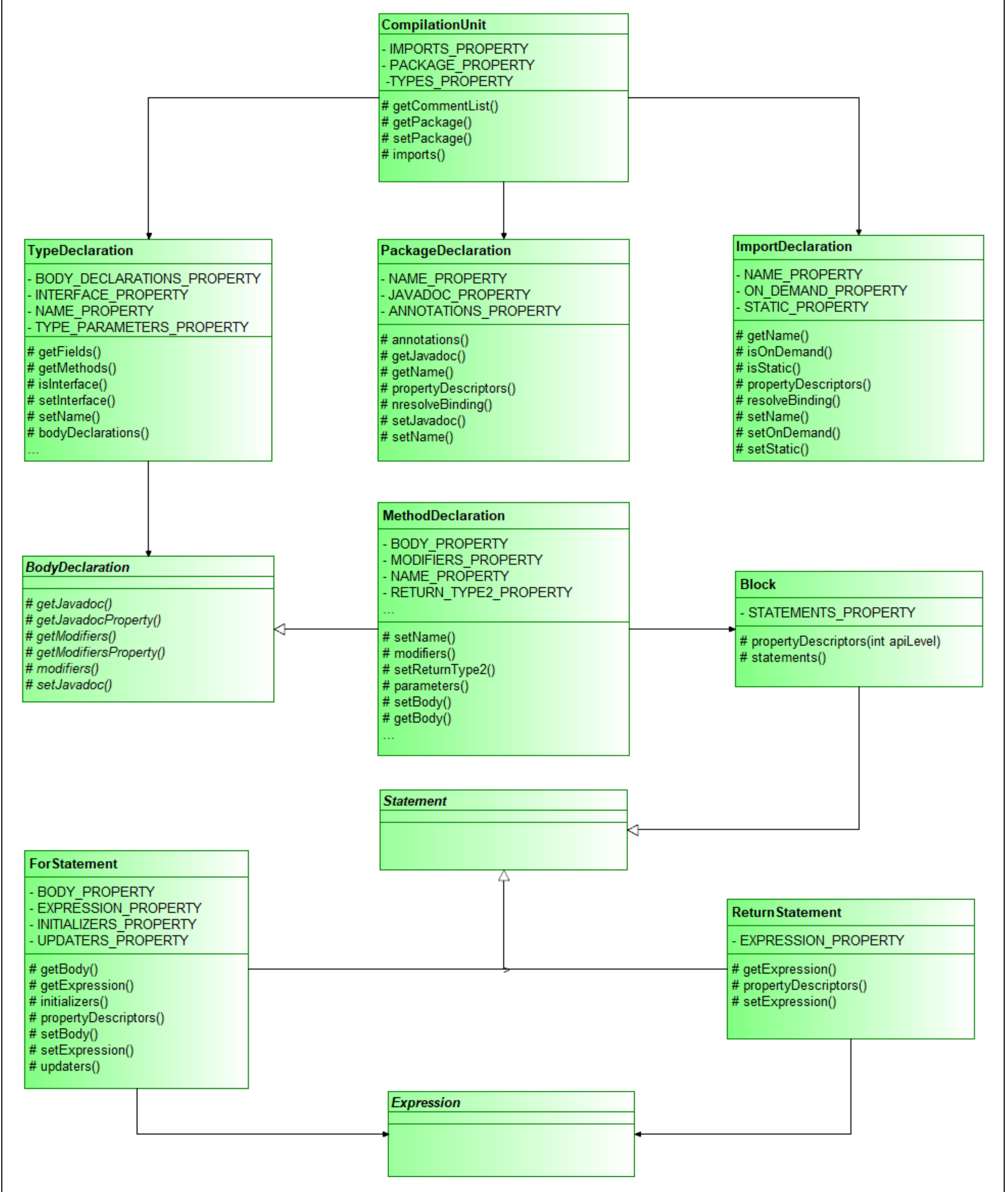
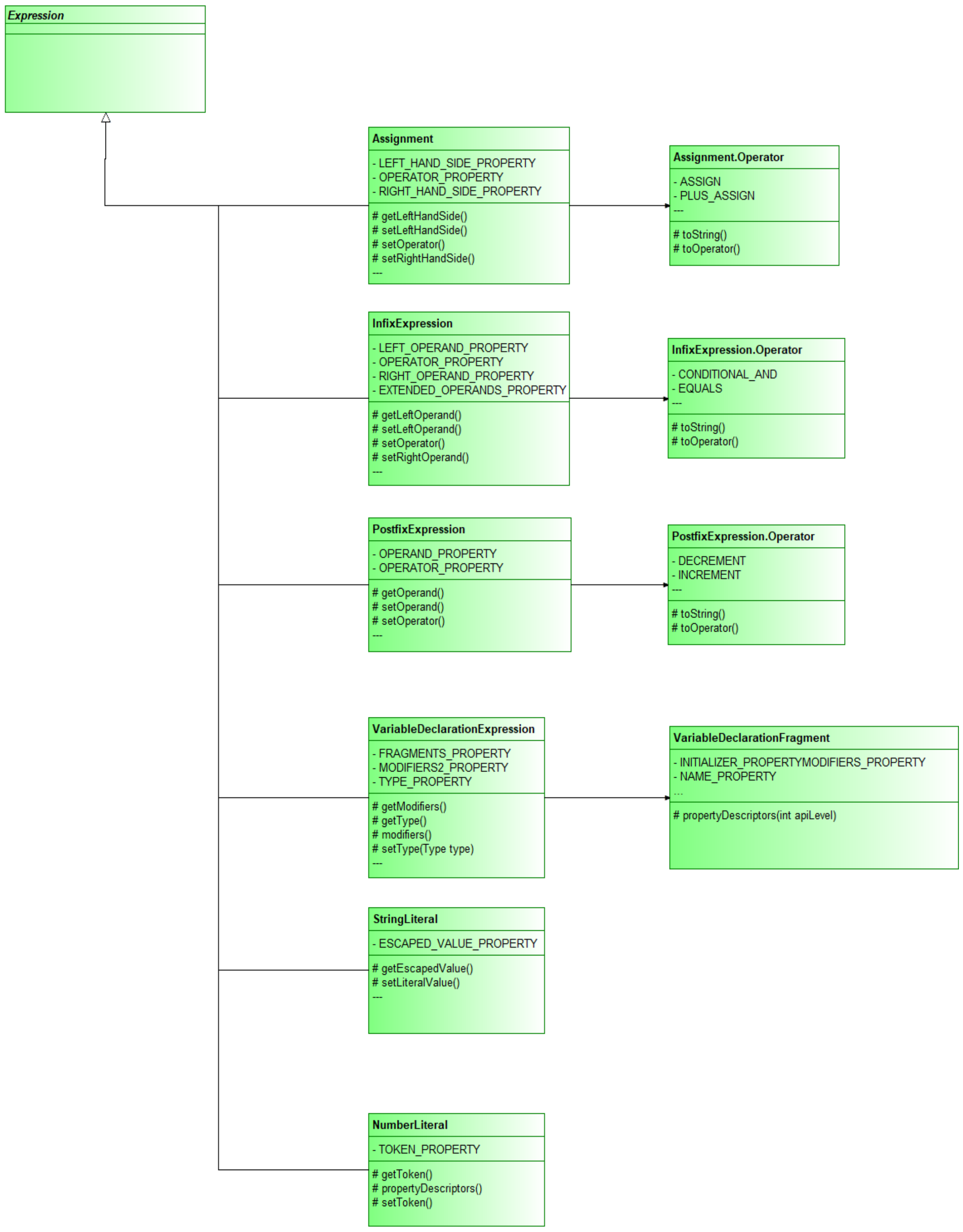

CompilationUnit
- IMPORTS_PROPERTY
- PACKAGE_PROPERTY
-TYPES_PROPERTY
# getCommentList()
# getPackage()
# setPackage()
# imports()
TypeDeclaration
- BODY_DECLARATIONS_PROPERTY
- INTERFACE_PROPERTY
- NAME_PROPERTY
- TYPE_PARAMETERS_PROPERTY
# getFields()
# getMethods()
# isInterface()
# setInterface()
# setName()
# bodyDeclarations()
...
PackageDeclaration
- NAME_PROPERTY
- JAVADOC_PROPERTY
- ANNOTATIONS_PROPERTY
# annotations()
# getJavadoc()
# getName()
# propertyDescriptors()
# nresolveBinding()
# setJavadoc()
# setName()
ImportDeclaration
- NAME_PROPERTY
- ON_DEMAND_PROPERTY
- STATIC_PROPERTY
# getName()
# isOnDemand()
# isStatic()
# propertyDescriptors()
# resolveBinding()
# setName()
# setOnDemand()
# setStatic()
BodyDeclaration
# getJavadoc()
# getJavadocProperty()
# getModifiers()
# getModifiersProperty()
# modifiers()
# setJavadoc()
MethodDeclaration
- BODY_PROPERTY
- MODIFIERS_PROPERTY
- NAME_PROPERTY
- RETURN_TYPE2_PROPERTY
...
# setName()
# modifiers()
# setReturnType2()
# parameters()
# setBody()
# getBody()
...
Block
- STATEMENTS_PROPERTY
# propertyDescriptors(int apiLevel)
# statements()
Statement
ForStatement
- BODY_PROPERTY
- EXPRESSION_PROPERTY
- INITIALIZERS_PROPERTY
- UPDATERS_PROPERTY
# getBody()
# getExpression()
# initializers()
# propertyDescriptors()
# setBody()
# setExpression()
# updaters()
ReturnStatement
- EXPRESSION_PROPERTY
# getExpression()
# propertyDescriptors()
# setExpression()
Expression

Figure 4.1


Expression
Assignment
- LEFT_HAND_SIDE_PROPERTY
- OPERATOR_PROPERTY
- RIGHT_HAND_SIDE_PROPERTY
# getLeftHandSide()
# setLeftHandSide()
# setOperator()
# setRightHandSide()

Assignment.Operator
- ASSIGN
- PLUS_ASSIGN

# toString()
# toOperator()
InfixExpression
- LEFT_OPERAND_PROPERTY
- OPERATOR_PROPERTY
- RIGHT_OPERAND_PROPERTY
- EXTENDED_OPERANDS_PROPERTY
# getLeftOperand()
# setLeftOperand()
# setOperator()
# setRightOperand()

InfixExpression.Operator
- CONDITIONAL_AND
- EQUALS

# toString()
# toOperator()
PostfixExpression
- OPERAND_PROPERTY
- OPERATOR_PROPERTY
# getOperand()
# setOperand()
# setOperator()

PostfixExpression.Operator
- DECREMENT
- INCREMENT

# toString()
# toOperator()
VariableDeclarationExpression
- FRAGMENTS_PROPERTY
- MODIFIERS2_PROPERTY
- TYPE_PROPERTY
# getModifiers()
# getType()
# modifiers()
# setType(Type type)

VariableDeclarationFragment
- INITIALIZER_PROPERTYMODIFIERS_PROPERTY
- NAME_PROPERTY
...
# propertyDescriptors(int apiLevel)
StringLiteral
- ESCAPED_VALUE_PROPERTY
# getEscapedValue()
# setLiteralValue()

NumberLiteral
- TOKEN_PROPERTY
# getToken()
# propertyDescriptors()
# setToken()

Figure 4.2

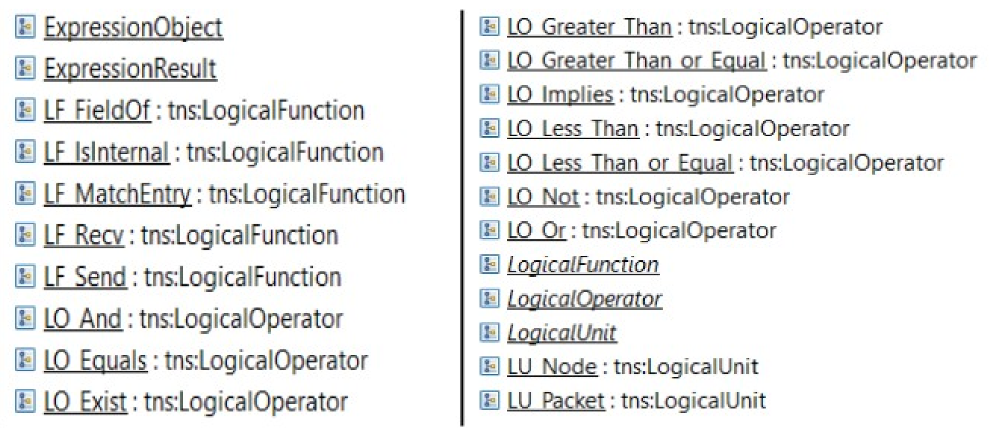

ExpressionObject
ExpressionResult
LF_FieldOf : tns:LogicalFunction
LF_IsInternal : tns:LogicalFunction
LF_MatchEntry : tns:LogicalFunction
LF_Recv : tns:LogicalFunction
LF_Send : tns:LogicalFunction
LO_And : tns:LogicalOperator
LO_Equals : tns:LogicalOperator
LO_Exist : tns:LogicalOperator
LO_Greater_Than : tns:LogicalOperator
LO_Greater_Than_or_Equal : tns:LogicalOperator
LO_Implies : tns:LogicalOperator
LO_Less_Than : tns:LogicalOperator
LO_Less_Than_or_Equal : tns:LogicalOperator
LO_Not : tns:LogicalOperator
LO_Or : tns:LogicalOperator
LogicalFunction
LogicalOperator
LogicalUnit
LU_Node : tns:LogicalUnit
LU_Packet : tns:LogicalUnit

Figure 4.3

The classes that implement the translation can be found in the package:

```
        it.polito.translator.symnet
```

The second phase of the translation begins when the *Parser* creates an instance of the *ClassGeneratorS* class and calls the *startGeneration* method. It builds the new 'SEFLinJava' class. First of all, it generates the basic structure of the class. Afterward it analysis the XML file. The unmarshalling starts when the *ClassGeneratorS* instantiate the *RuleUnmarshallerS* and initialise the JAXBContext. The *generateRule* method of the *RuleUnmarshallerS* scans all JAXBelement nodes to map the virtual network functions rules. The type associated with each XML element is the core information for the transformation of the rules. The possible types are those listed in the XML Schema and shown in the Figure 4.3. The type of each node is captured by the *getType* method which in turn will call the methods for the SEFL mapping. Below we are going to list and describe these methods.

- **generateNot**

It maps the *LO_Not* type. It is placed in front of the expressions of which we want to have the negation. In SymNet we can not put a 'Not' in front of the statements. We need to generate different constraints. For example, if the type *LO_Not* it is put in front of *LF_MatchEntry* type the translator have to generate a different set of SEFL instructions. The *generateNot* method sets the *flagnot*, then calls the *getType* of the nested type of the *LO_Not* expression, and finally resets the *flagnot*. In Listing 4.3 we can find the *generateNot* code.

- **generateMatchEntry**

It maps the *LF_MatchEntry* type. This type depends on the configuration of the middle-box. We must generate a Lookup rule described in the Chapter 4.3.3. The translator builds the Lookup rule only at the end of the unmarshal phase because it must create a new Method Declaration AST node type. For this reason, the *generateMatchEntry* sets two flags (*match, blacklist)* and goes on to the next node. In Listing 4.3 we can find the *generateMatchEntry* code.

```
    - generateNot method of RuleUnmarshallerS -
private Expression generateNot(LONot not) {
       flagnot = true;
       Expression exp = getType(not.getExpression());
       if(exp!=null) {
       startblock.arguments().add(exp);}
       flagnot = false;
       return null;
}
    - generateMatchEntry method of RuleUnmarshallerS -
private Expression generateMatchEntry(LFMatchEntry matchEntry) {
       if (flagnot) {
          blacklist = true;
       }
       params = new ArrayList<String>();
       for (LFFieldOf temp : matchEntry.getValue()) {
            params.add(temp.getField());
       }
       match = true;
       return null;
}
```

Listing 4.3

- **generateImplies, generateAnd**

They map respectively the *LO_Implies* type and the *LO_And* type. These types do not generate a specific SEFL instruction. They call the *getType* method on the nested expressions of the node.

- **generateOr**

It maps the *LO_Or* type. The logical operator or puts together a series of expressions among which there must be at least one true. The basic instructions of the SEFL do not have the logical operator OR. The function *generateOr* generates a series of 'if-then-else' statements that repeat the same behaviour of the *LO_Or*. For example, the `LO_Or(A,B,C)` expression is mapped as: `If(Constraint(A), NoOp, If(Constrain(B), NoOP, Constrain(C)))`.

Once the unmarshalling phase is over, *ClassGeneratorS* calls the method *generateMatch* method. It checks the value of the match flag, whether it is false, the translator has not found a LF_MatchEntry and will do not generate a new method. Otherwise, the method will generate a Lookup rule method. Now the ClassGeneratorS can write the 'SEFLinJava' output file.

The last phase of the translator performs a language switch from Java to Scala. To do this, we use an existing Java to Scala conversion tool called Scalagen[15]. It uses a Java based parser for Java sources and provides modular transformation of the AST to match Scala idioms. The resulting transformed AST is serialized into Scala format. We use Scalagen with a Maven plugin provided by the tool. It can be used directly via the command line like this:

```
mvn com.mysema.scalagen:scalagen-maven-plugin:0.2.2:main -DtargetFolder=scala
```

The explicit configuration of Scalagen Maven plugin in a POM is:

```
<plugin>
  <groupId>com.mysema.scalagen</groupId>
  <artifactId>scalagen-maven-plugin</artifactId>
  <version>0.2.2</version>
</plugin>
```

The last step of the translation is a post processing operation that changes the name of some functions. In Java we used symbolic names for some SEFL functions that could not be represented. For example, we change the SEFL *Constraint* as follows:

```
Constrain("flag",postParsef(ConstantValue(0))) →
      Constrain("flag",:==:(ConstantValue(0)))
```

The post processing is built by the *PostProcess* Java class of the 'it.polito.translator.symnet' package of our framework.

4.3.5 Translated Network Functions

The general rules for the network functions of the VNF Modeling[16] are translated according to the mapping. We have the rules of Access Control List(Acl) Firewall, Antispam, IDS, Traffic Classifier, Mail Server and Web Server. We can find the rules of the AclFirewall in [Listing 4.4] and the rules of Traffic Classifier [Listing 4.5] as examples.

Since the SymNet[5] verification tools can inject only one packet per execution, we can not translate the behaviour of our data driven virtual network functions such as the Network address translation (NAT) and Web Cache. In addition, these functions generate different rules for packets that come from the internal or external network. There is no such network division in SymNet. Anyhow, we have translated the Web Cache into a non-data driven implementation without distinguishing the internal/external network. As for the NAT's rules we have many dependencies on previous packets in transit on the network and these dependencies can not be translated into SymNet. Our translator generates several SEFL statements describing the behaviour of the given function independently of the location in which it will be installed. The NAT provided by SymNet combines basic elements such as *FromDevice*, *ToDevice*, *IPClassifier* and *IPRewriter*[3]. It performs a different implementation of the NAT for each network graph that aims to verify. For this reason it is not possible to achieve this result only with the SEFL instructions regardless of the configuration of the network graph to be verified. However, this aspect does not go to undermine the goodness of our model. The framework correctly extracts a generic model of the behaviour of the network function. In the case of the NAT there is not a problem related to the model generated but to the behaviour of the same that as it was implemented is not compatible with SymNet.

3 The behaviour of some of these basic SymNet[5] elements are described in the Chapter 5.

- Rule_AclFirevall.sccala -

```
package org.change.v2.abstractnet.click.sefl

import org.change.v2.analysis.expression.concrete._
import org.change.v2.analysis.memory.State
import org.change.v2.analysis.memory.TagExp._
import org.change.v2.analysis.memory.Tag
import org.change.v2.analysis.processingmodels.instructions._
import org.change.v2.util.conversion.RepresentationConversion._
import org.change.v2.util.canonicalnames._
import org.change.v2.analysis.memory.Value
import org.change.v2.abstractnet.generic._

class Rule_AclFirewall {

  def generate_rules(params:List[ConfigParameter]): InstructionBlock = {
    val code = InstructionBlock(
        Assign("flag", ConstantValue(0)),
        InstructionBlock(),
        InstructionBlock(addrule(params)))
    code
  }

  def addrule(p:List[ConfigParameter]): Array[InstructionBlock] = {
    var rule: Array[InstructionBlock] = null
    var rules = Array(InstructionBlock(Nil))
    val limit = p.length / 2
    var i = 0
    while (i <= limit) {
      rule = Array(InstructionBlock(
      If(Constrain(IPSrc, :==:(ConstantValue(ipToNumber(p(i + 0).value)))),
         InstructionBlock(
         If(Constrain(IPDst,:==:(ConstantValue(ipToNumber(p(i + 1).value)))),
            Fail("Match-in-blacklist"),
            NoOp)
         ), NoOp)))
      rules = Array.concat(rules, rule)
      i = i + 2
    }
    rules
  }
}
```

Listing 4.4

- Rule_Classifier.sccala -

```
package org.change.v2.abstractnet.click.sefl

import org.change.v2.analysis.expression.concrete._
import org.change.v2.analysis.memory.State
import org.change.v2.analysis.memory.TagExp._
import org.change.v2.analysis.memory.Tag
import org.change.v2.analysis.processingmodels.instructions._
import org.change.v2.util.conversion.RepresentationConversion._
import org.change.v2.util.canonicalnames._
import org.change.v2.analysis.memory.Value
import org.change.v2.abstractnet.generic._

class Rule_Classifier {

  def generate_rules(params:List[ConfigParameter]): InstructionBlock = {
    val code = InstructionBlock(
        Assign("flag", ConstantValue(0)),
        InstructionBlock(),
        InstructionBlock(addrule(params)))
    code
  }

  def addrule(p:List[ConfigParameter]): Array[InstructionBlock] = {
    var rule: Array[InstructionBlock] = null
    var rules = Array(InstructionBlock(Nil))
    val limit = p.length / 2
    var i = 0
    while (i < limit) {
      rule = Array(InstructionBlock(
           If(Constrain("flag", :==:(ConstantValue(0))),
              InstructionBlock(
              If(Constrain(Proto,:==:(ConstantValue(p(i + 0).value.toInt))),
                 InstructionBlock(
                   Assign("flag", ConstantValue(1)),
                   Assign("idIfSend", ConstantValue(p(i + 1).value.toInt))),
                 NoOp)),
              NoOp)))
      rules = Array.concat(rules, rule)
      i = i + 2
    }
    rule = Array(InstructionBlock(
       If(Constrain("flag", :==:(ConstantValue(0))),
          Fail("No-Match"),
          NoOp)))
    rules = Array.concat(rules, rule)
    rules
  }
}
```

Listing 4.5

Chapter 5

Tests

5.1 VeriGraph

VeriGraph[4] is a tool for the formal verification on SDN/NFV networks. It can verify network properties (such as reachability, isolation, traversal), and it is also able to model the stateful middle-boxes (such as functions that may dynamically change the forwarding path of a traffic flow according to their local algorithms and states)[12]. The network scenario (named *service graph*) is modeled as a set of First Order Logic formulas [11] that express the network policies, and then these formulas are analysed by the Z3[10] solver. In this context, a graph is a set of virtual network functions (nodes) connected by directed arcs[12].

Virtual network functions are represented in VeriGraph through Java classes. The core method of these classes is *installVNF*. Inside this method all the rules that were stored in the XML are unmarshalled and saved in the appropriate Z3 format by our parser. This method installs and adds all the constraints of the middle-box to the solver, this is necessary in order to make the model of the middle-box interact with the other elements of the VeriGraph network[9].

Our framework performs a translation from the VNFs defined by the user to the Java classes supported by VeriGraph. This process of transformation uses the JAXB framework[13] and the Java package *org.eclipse.jdt.core.dom*[14]. We have added a new element in the XML Schema of our framework to model the new VNF traffic classifier (described in Section 3.1.1), but this add-on is not yet supported in VeriGraph. Therefore, it is necessary to modify the translation phase towards VeriGraph in order to skip the new rule linked to the selection of the forwarding interface.

In this chapter we will perform some tests in VeriGraph to understand how this verification tool works and to check if the VNF models behaves in the same way in different test environments.

5.1.1 Traffic Classifier

The following examples report the test cases generate in VeriGraph[4] in order to check whether the Traffic Classifier generated by our parser works as expected. All the test check the isolation property between two hosts.

Test chain n.1

In this first example we will use the *service graph* which is shown in Figure 5.1. There are two hosts (Host_A, Host_B) and a network function (Traffic Classifier). These three nodes are connected by two bidirectional links (l1, l2).

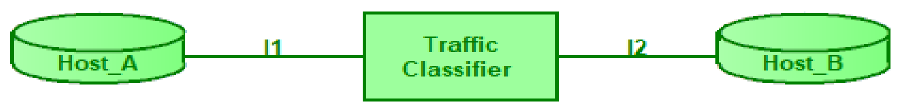


Figure 5.1

Host_A and the Host_B are modeled as two *PolitoEndHost* VNF of VeriGraph. They can be configured as a simple end host that generates and receives packets or as Web Client or Mail Client. The Traffic Classifier node is the VNF generated by our parser and described in Section 3.1.1. It is called *Rule_Classifier* in VeriGraph. Briefly, his behaviour is the following: delete all incoming packets that have a transport protocol different from those in the Table of the node, which are specified during configuration.

To write a new test in VeriGraph, the first thing to do is to list the nodes of the *service graph* and specify the routing table for each one. Next, we have to give the configuration of the nodes and install them. We can also give some indication on how the packet have to be when it is crossing the graph. Finally we can write the main class that call the SAT solver.

Test of satisfiability A

In this test we will check if the Host_A can reach the Host_B based on the following configurations. The packet crossing the graph must have an IP destination address equal to the IP of the Host_B and the transport protocol equal to `HTTP_REQUEST`. The table of the Traffic Classifier has two items: `HTTP_REQUEST` and `HTTP_RESPONSE` protocols. These two protocols are represented in VeriGraph with an integer value. Respectively 1 and 2. The expected result is the satisfiability of the policy, because the packet protocol constraint is known to the Traffic Classifier and the routing tables link the Host_A to the Host_B via the Traffic Classifier. Moreover, we expect to see two packets. The first one in transit on the link *l1*, it is sent by the Host_A to the Traffic Classifier, the second one is sent by the Traffic Classifier to the Host_B on link *l2*. The Listing 5.1 shows the output of this test. Here we can find the result of the test. It is SAT, which means satisfiability. We can also find the packet that crosses the *service graph.* For each packet there is the sender node, the receiver node and all header fields.

```
      - Output of the Test of satisfiability A in VeriGraph-
SAT
(define-fun send!85 ((x!0 Node) (x!1 Node) (x!2 packet)) Bool
  (ite (and (= x!0 cf)   //Sender node: Traffic Classifier(cf)
            (= x!1 b)    //Receiver node: Host_B(b)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 1 16 17 18 false)))
  (ite (and (= x!0 a)    //Sender node: Host_A(a)
            (= x!1 cf)   //Receiver node: Traffic Classifier(cf)
           (= x!2 (packet ip_a ip_b null null a 14 14 15 1 16 17 18 false)))
```

Listing 5.1

The packet's fields given in the output of the test are:

```
ip_a = Source field, IP address of the node a (Host_A)
ip_b = Destination field, IP address of the node b (Host_B)
null = Inner source field
null = Inner destination field
a  = Source field, Host_A
14 = Origin body field
14 = Body field
15 = Sequence field
1 = Protocol field, HTTP_REQUEST
16 = Email From field
17 = URL field
18 = Options field
false = Encrypted field
```

Test of UnSatisfiability

In this test we will check if the Host_A can not reach the Host_B based on previous configurations with only one change. The packet crossing the graph must have the transport protocol equal to `POP3_REQUEST`. The expected result is the one of UnSatisfiability because the Traffic Classifier is configured to delete packet with `POP3_REQUEST` protocol. The output of the test has only one line with the message "*UNSAT*".

These tests are available in the VNF Modeling project under the folder "`.\src\it\polito\verigraph\usecase\Test_Classifier_Base.java`".

Test chain n.2

In the second example we will use the *service graph* which is shown in Figure 5.2. There are two hosts (Host_A, Host_B) and two network functions (Traffic Classifier_A, Traffic Classifier_B). These elements are connected by three bidirectional links (e1, e2, e3).

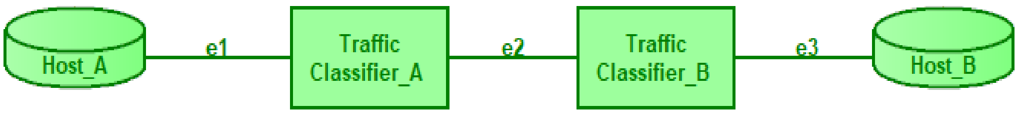


Figure 5.2

Test of satisfiability B

In this test we will check if the Host_A can reach the Host_B given the follow configurations:

- The Traffic Classifier_A drop all the incoming packets with transport protocol different to `HTTP_REQUEST` or `POP3_REQUEST`.
- The Traffic_Classifier_B drop all the incoming packets with transport protocol different to `HTTP_REQUEST` or `HTTP_RESPONSE`.
- The packet crossing the graph must have the IP destination address equal to the IP of the Host_B and the transport protocol equal to `HTTP_REQUEST`.

The expected result is that of Satisfiability. Moreover, we expect to see three packets. The first one in transit on the link e1, it is sent from the Host_A(a) to the Traffic Classifier_A. The second one is sent from the Traffic Classifier_A(cf1) to the Traffic Classifier_B(cf2) on link e2. The last one is sent from the Traffic Classifier_B to Host_B(b) on link e3. The Listing 5.2 reports the output of this test.

This test is available in the VNF Modeling project under the folder “`.\src\it\polito\verigraph\usecase\Test_Classifier_Chain.java`”.

```
    - Output of the Test of satisfiability B in VeriGraph -

SAT
(define-fun send!286 ((x!0 Node) (x!1 Node) (x!2 packet)) Bool
  (ite (and (= x!0 cf2)
            (= x!1 b)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 1 16 17 18 false)))
    true
  (ite (and (= x!0 cf1)
            (= x!1 cf2)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 1 16 17 18 false)))
    true
  (ite (and (= x!0 a)
            (= x!1 cf1)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 1 16 17 18 false)))
    true
    false))))
```

Listing 5.2

Test chain n.3

In the third example we will use the *service graph* which is shown in Figure 5.3. There are two hosts (Host_A, Host_B) and two virtual network functions (Traffic Classifier, Firewall). These elements are connected by three bidirectional links (l1, l2, l3).

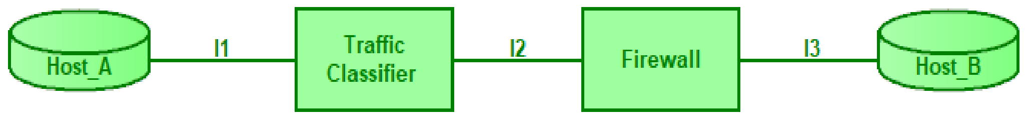


Figure 5.3

Test Satisfiability C

In this test we will check if the Host_A(*a*) can reach the Host_B(*b*) given the follow configurations:

- The Traffic Classifier(*cf*) is configured to drop all the incoming packets with transport protocol other than `HTTP_REQUEST` or `POP3_REQUEST`.
- The Firewall(*fw*) drops an incoming packet when it has the fields *IP address source* and *IP address destination* like at one pair present inside the Firewall's Table. In the example the Firewall's Table contain only one pair: `<ip_a, ip_fw>`.
- The packet crossing the graph must have the IP destination address equal to the IP of the Host_B and the transport protocol equal to `HTTP_REQUEST`.

The expected result is that of Satisfiability. Moreover, we expect to see three packets. The first one in transit on the link l1 (from *a* to *cf*). The second one is sent from Traffic Classifier to Firewall on link l2. The last one is sent from Firewall to Host_B on link l3. The Listing 5.3 reports the output of this test.

This test is available in the VNF Modeling project under the folder "`.\src\it\polito\verigraph\usecase\Test_Classifier_Firewall.java`".

```
    - Output of the Test of satisfiability C -
SAT
(define-fun send!387 ((x!0 Node) (x!1 Node) (x!2 packet)) Bool
  (ite (and (= x!0 fw)
            (= x!1 b)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 1 16 17 18 false)))
    true
  (ite (and (= x!0 cf)
            (= x!1 fw)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 1 16 17 18 false)))
    true
  (ite (and (= x!0 a)
            (= x!1 cf)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 1 16 17 18 false)))
    true
    false))))
```

Listing 5.3

5.2 SymNet

We have performed several tests with the SymNet[5] verification tool. These tests demonstrate that the VNFs generated with the VNF Modeling framework[16] are compatible with the SymNet framework. More specifically, we have generated tests on the following middle-boxes: Firewall, Antispam, Intrusion Detection System (IDS), Traffic Classifier, Mail Server, Web Cache and Web Server.

We have done some tests to verify the reachability policy and others to verify the isolation policy. SymNet shows the path that a packet makes from the source to the destination. The values of packet header can be analysed, at each port traversed by the packet, to find out whether it is allowed to pass or not. In this chapter we will explain the main components of the tests.

5.2.1 Basic elements

The core elements used to perform tests in SymNet[5] are as follows:

- **The graph of the network.**

It is described with Click[1] syntax. We have listed the VNFs of the network and the connections between them in the graph. We can now list the middle-boxes that make up our network in the following way:

```
label :: NomeNF(configuration)
```

label is a simple tag that we will use to describe the links, *NameNf* is the name of the class that describes the behaviour of our virtual network function, *configuration* represents the data necessary to initialize the middle-box. In order to list the links between the elements of the network we use the following structure:

```
label1 -> label2[0] -> [0]label3
          label2[1] -> label4
```

The symbol '->' represents the one-way connection between a network function and the other. If nothing is specified, the packet transits on the default input and output ports. Otherwise, we can specify a different entry or exit port in *square brackets*. The brackets that precede the *label* indicates the input port and the brackets that follow the *label* indicates the output port. All the graphs used in the tests are in the folder:

```
"./Symnet/src/main/resources/click_test_files/vnf_model"
```

- **The main class.**

It performs the following activities to execute the test. The main takes the network configuration and analyses it to extract the SEFL models. After, it runs the verification process and prints the result. The listing 5.4 shows the core instructions of the main. All the main classes written to run the tests are called *<nf_name> Runner* and are located in the folder:

```
"./Symnet/src/main/scala/org.change.v2.runners.models"
```

1 Click is a new software for building routers.

```
   - The main Scala class of the verification -

//clickConfig is the path of the network graph:
val clickConfig =
"src/main/resources/click_test_files/vnf_model/GraphMailServer.click"
//absNet builds the VNFs:
val absNet = ClickToAbstractNetwork.buildConfig(clickConfig)
//executor runs the solver:
val executor = ClickExecutionContext.fromSingle(absNet).setLogger(JsonLogger)
....
output.println(
    successful.map(_.jsonString).mkString("Successful: {\n", "\n", "}\n") +
         failed.map(_.jsonString).mkString("Failed: {\n", "\n", "}\n")
)
```

Listing 5.4

- **The elements of the network (VNFs)**.

SymNet[5] provides a number of network elements, and we create new ones for our VNFs. We are going to see better how these elements are made in the description of the tests. All network elements are in the folder:

```
"./Symnet/src/main/scala/org.change.v2.abstractnet.click.sefl"
```

- **The test results**.

They are saved in a *file.output*. We have to analyse the files in output in order to be able to know the outcome of the test. Below there is a draft of the elements present in the *file.output*.

```
Successful: {{"status":"OK",
"port_trace":[ .... ]
"instruction_trace":[ ... ]
"memory":{ .... }
}
Failed: {
{"status": "message"
"port_trace":[ .... ]
"instruction_trace":[ ... ]
"memory":{ .... }
}
```

Rechability tests are successful if there is a *Successful* element inside the *file.output*. To better understand what happened, we can look the internal elements of *Successful*. The *status* element reports an 'Ok' message. In the *port_trace* element there is a list of the ports that the packet has traversed. For example, if the graph has three nodes (`A → B → C`), the *port_trace* element will have the following list of ports: `A-in, A-out, B-in, B-out, C-in, C-out`. The *instructions_trace* element reports a list of all the SEFL instructions executed on each network nodes port. For example, for the graph (`A → B → C`), in the *instruction_trace* element we will find `'A-in: Forward (A-out)'`. The data of the packet are in the *memory* element.

```
 - Set of valid values for the source IP address -
Allocate(IPSrc, 32),
Assign(IPSrc, SymbolicValue()),
Constrain(IPSrc,
     :&:(:>=:(ConstantValue(0)), :<=:(ConstantValue(4294967296L)))),
```

Listing 5.5

For instance, in memory we can find the following information: the Protocol field of the packet is at position ‘72’ and has value ‘30’. Each field which is in the packet is placed in a specific memory location. The memory mappings are into the *packet.scala* file. It can be found in the folder:

```
"./Symnet/src/main/scala/org.change.v2.util".
```

Below we list some memory mapping values:

```
val IPSrcOffset = 96,      val IPSrc = L3Tag + IPSrcOffset
val IPDstOffset = 128,     val IPDst = L3Tag + IPDstOffset
val URLOffset = 16,        val URL = L7Tag + URLOffset
val EmailFromOffset = 16, val EmailFrom = L7Tag + EmailFromOffset
```

The value we can assign to each field in the packet is limited by the number of bits used to represent it. We can look for the *State.scala* file in the folder `"./Symnet/src/main/scala/org.change.v2.analysis.memory"` in order to know the range of valid values for a field. For example, the source IP field range is between 0 and 4294967296L [Listing 5.5].

Even if the test was successful, we can find more than one *Fail* element in the *file.output*. These elements are in there because the symbolic verification tool explores all possible values. For example, if a packet has the protocol field equal to 10. And there is an instruction that checks if the value of this field is ‘10’. There are two possible results. One is *Successful*, the protocol is equal to 10. Another is *Failed*, the protocol can not be different from 10. Both results are true. The *Failed* element contains a *state*, it is a message that explains why the execution was not successful. Moreover, it reports the path crossed by the packet from the beginning to the point where a fail has occurred.

Isolation policy can not be executed directly, but we can check the Rechability policy by an inverse interpretation of the *file.output.* We check whether the reachability is denied. For example, given the graph A → B → C, we want to verify that A can not reach B. The isolation property is true if the reachability returns a *Failed* statement and no one *Successful*.

5.2.2 Network graph with a Firewall

In this reachability test, we check if the VNF of the firewall generated by the VNF Modeling framework[16] works properly in the SymNet[5] environment. To do this we insert the VNF module in a specific network scenario. In Figure 5.4 there is the network topology of this test and in Listing 5.6 we can find the equivalent network graph in the Click format.

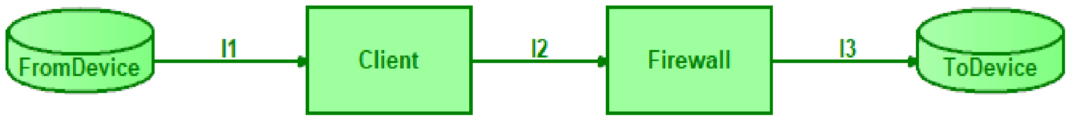


Figure 5.4

The graph is composed by 4 different elements:

- **FromDevice** is a basic element of SymNet. This host generates and sends a standard packet that initialises all header fields with symbolic values.

The packet's fields created by *FromDevice* is empty and the user can not set specific values using the Click basic elements. In this way, we have generated some elements to set packet's fields specifically for our tests.

- **Client** is a new element that sets the IP source address and the IP destination address of the packet. It receives a packet through the default input port, sets the *IPSrc* and *IPDst* fields with the two parameters received as configuration, and finally sends the packet through the default output port. The class written with Scala language for the Client is called *ClientIPsrcIPdst* (label *cli*). In the test, we use *IPSrc* equal to 0.0.0.24 and *IPDst* equal to 0.0.0.64 (line 1 of Listing 5.6).
- **Firewall** is a new element that contains the behaviour of the firewall generated by VNF Modeling. The firewall is configured with a black-list as a list of <*IPSrc, IPDst*>. Every packet with *IPSrc* and *IPDst* contained into the black-list will be blocked by the firewall. The class written with Scala language for the firewall is called *ZmodelFw*. In the test, we initialise the *ZmodelFw* (label *mfw*) with two pair:

  < 0.0.0.8, 0.0.0.16 >, < 0.0.0.24, 0.0.0.32 > (line 2 of Listing 5.6)
- **ToDevice** is a basic element of SymNet. It is the terminal point of the packet.

```
     - GraphFw.click -
1.   Cli :: ClientIPsrcIPdst(0.0.0.24,0.0.0.64)
2.   myf :: ZmodelFw(0.0.0.8,0.0.0.16,0.0.0.24,0.0.0.32)
3.   FromDevice -> cli -> myf -> ToDevice
```

Listing 5.6

```
      - TestResultFirewall.output -
Successful: { {"status":"OK",
"port_trace":[
{"0":"fromDevice-0-in"}, {"1":"fromDevice-0-out"}, {"2":"cli-in"},
{"3":"cli-0-out"}, {"4":"myf-in"}, {"5":"myf-0-out"},
{"6":"toDevice-0-in"}, {"7":"toDevice-0-out"}],
"instruction_trace":[ [...]
{"63":"Forward(fromDevice-0-out)"},{"65":"CreateTag(L3HeaderStart,+0)"},
{"66":"CreateTag(IPSrc,L3HeaderStart+96)"}, {"67":"AllocateRaw(IPSrc,32)"},
{"68":"CreateTag(IPDst,L3HeaderStart+128)"}, {"69":"AllocateRaw(IPDst,32)"},
{"70":"AssignRaw(L3+96,[Const(24)],GenericNumeric)"},
{"71":"AssignRaw(L3+128,[Const(64)],GenericNumeric)"},
{"72":"AssignNamedSymbol(flag,[Const(0)],GenericNumeric)"},
{"73":"Forward(cli-0-out)"},
{"75":"AssignNamedSymbol(flag,[Const(0)],GenericNumeric)"},
{"76":"ConstrainRaw(L3+96,:~:(:==:([Const(8)])),Some(~(==([Const(8)]))))"},
{"77":"org.change.v2.analysis.processingmodels.instructions.NoOp$@6166e06f"},
{"78":"ConstrainRaw(L3+96,:==:([Const(24)]),Some(==([Const(24)])))"},
{"79":"ConstrainRaw(L3+128,:~:(:==:([Const(32)])),Some(~(==([Const(32)]))))"},
{"80":"org.change.v2.analysis.processingmodels.instructions.NoOp$@6166e06f"},
{"81":"Forward(myf-0-out)"},
[...]
```

Listing 5.7

The reachability test in SymNet starts with the parsing of the network's graph, which is then translated it into SEFL instructions to be sent to the SAT solver[10].

The expected result in this test is *Successful,* since we insert in the graph a packet with *IPSrc* and *IPDst* different from the pairs blocked by the firewall. In the *file.output* [Listing 5.7] we can see all the ports traversed by the packet and the SEFL instructions executed by the solver. The instructions n. 70 and n.71 are about the assignment of *IPSrc* and *IPDst*. Instructions n. 76 and n. 78 check if the *IPSrc* of the incoming packet is different from the *IPSrc* addresses denied by the firewall. Since the *IPSrc* of the packet is equal to the *IPSrc* of the second pair denied, it follows the instruction n. 79 that controls the *IPDst* field of that pair. It is different from the *IPDst* of the packet, so the firewall can forward it (instruction n.81).

5.2.3 Network graph with an Antispam

In this reachability test, we check if the VNF of the antispam generated by the VNF Modeling framework[16] works properly in the SymNet[5] environment. To do this we insert the VNF module in a specific network scenario. In Figure 5.5 there is the network topology of this test and in Listing 5.8 we can find the equivalent network graph in the Click format.

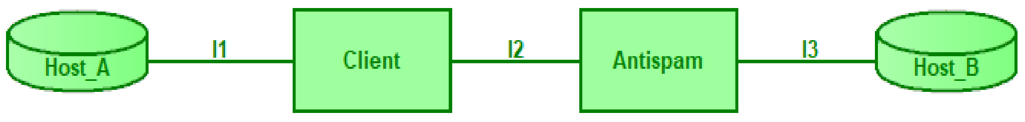


Figure 5.5

The graph is composed by 4 different elements:

- **Host_A** is a *FromDevice* element of SymNet. The behaviour is explained in the section of the firewall test 5.2.2.
- **Client** is a new element that sets the protocol (*Proto*) field and the *EmailFrom* field of the packet. It receives a packet through the default input port, sets the *Proto* and *EmailFrom* fields with the two parameters received as configuration, and finally it sends the packet through the default output port. The class written with Scala language for the Client is called *ClientProtocol* (label *cli*). In this test, we use *Proto* equal to 20 (POP3_REQUEST) and *EmailFrom* equal to 7 (line 1 of Listing 5.8).
- **Antispam** is a new element that contains the behaviour of the antispam generated by VNF Modeling. The antispam is configured with a black-list as a list of <*EmailFrom*>. Every packet with *Proto* equal to 20 (POP3_REQUEST) or equal to 30 (POP3_RESPONSE) and with *EmailFrom* contained in the black-list will be blocked by the antispam. The class written with Scala language for the antispam is called *ZmodelAs* (label *as*). In this test, we initialise the antispam with three values: 3, 4, 8 (line 2 of Listing 5.8).
- **Host_B** is the *ToDevice* element of SymNet. It is the terminal point of the packet.

```
     - GraphAntispam.click -
1.   cli :: ClientProtocol(20,7)
2.   as :: ZmodelAs(3,4,8)
3.   FromDevice -> cli -> as -> ToDevice
```

Listing 5.8

```
      - TestResultAntispam.output -
Successful: {
{"status":"OK",
"port_trace":[
{"0":"fromDevice-0-in"}, {"1":"fromDevice-0-out"},
{"2":"cli-in"}, {"3":"cli-0-out"},
{"4":"as-in"},  {"5":"as-0-out"},
{"6":"toDevice-0-in"}, {"7":"toDevice-0-out"}],
"instruction_trace":[
[...]
{"63":"Forward(fromDevice-0-out)"},
{"64":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3ffcd140"},
{"65":"AssignRaw(L3+72,[Const(20)],GenericNumeric)"},
{"66":"AssignRaw(L7+16,[Const(7)],GenericNumeric)"},
{"67":"Forward(cli-0-out)"},
{"68":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3ffcd140"},
{"69":"AssignNamedSymbol(flag,[Const(0)],GenericNumeric)"},
{"70":"ConstrainRaw(L3+72,:==:([Const(20)]),Some(==([Const(20)])))"},
{"71":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3ffcd140"},
{"72":"ConstrainRaw(L7+16,:~:(:==:([Const(3)])),Some(~(==([Const(3)]))))"},
{"73":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3ffcd140"},
{"74":"ConstrainRaw(L7+16,:~:(:==:([Const(4)])),Some(~(==([Const(4)]))))"},
{"75":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3ffcd140"},
{"76":"ConstrainRaw(L7+16,:~:(:==:([Const(8)])),Some(~(==([Const(8)]))))"},
{"77":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3ffcd140"},
{"78":"Forward(as-0-out)"},
{"79":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3ffcd140"},
{"80":"Forward(toDevice-0-out)"},
[...]
```

Listing 5.9

The expected result for this test is *Successful,* since we put into the graph a packet with *EmailFrom* different from the values blocked by the antispam. In the *file.output* [Listing 5.9] we can see all the ports traversed by the packet and the SEFL instructions executed by the solver. The instructions n. 65 is about the assignment of the *Proto* and the instruction n. 66 is about the assignment of the *EmailFrom*. Instructions n. 70 checks if the *Proto* of the incoming packet is different from the value 20 (POP_REQUEST). Since it is equal, there is the instructions n.72, 74, 76 that control the *EmailFrom* field of the incoming packet. It is different from the black-list's values, so the antispam can forward it (statement n. 80).

5.2.4 Network graph with an IDS

In this isolation test, we check if the VNF of the Intrusion Detection System(IDS) generated by the VNF Modeling framework[16] works properly in the SymNet[5] environment. To do this we put the VNF module in a specific network scenario. In Figure 5.6 there is the network topology of this test and in Listing 5.10 we can find the equivalent network graph in the Click format.

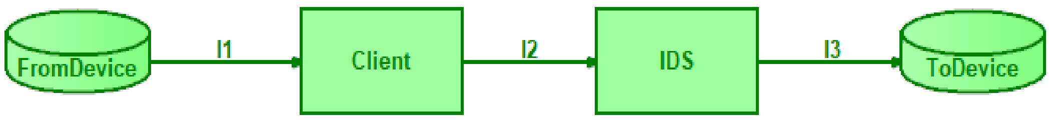


Figure 5.6

The graph is composed by 4 different elements:

- **FromDevice** is a basic element of SymNet. The behaviour is explained in the section of the firewall test 5.2.2.
- **Client** is a new element that sets the protocol (*Proto*) field and the *URL* field of the packet. It receives a packet through the default input port, sets the *Proto* and *URL* fields with the two parameters received as configuration, and finally sends the packet through the default output port. The class written with Scala language for the Client is called *ClientIDS* (label *cli*). In the test, we use *Proto* equal to 80 (HTTP_RESPONSE) and *URL* equal to 7 (line 1 of Listing 5.15).
- **IDS** is a new element that contains the behaviour of the IDS generated by VNF Modeling. The IDS is configured with a black-list as a list of <*URL*>. Every packet with *Proto* equal to 70 (HTTP_REQUEST) or equal to 80 (HTTP_RESPONSE) and with the *URL* contained in the black-list will be blocked by the IDS. The class written with Scala language for IDS is called *ZmodelIDS* (label *ids*). In the test, we initialise this middle-box with three values: 10, 7, 30 (line 2 of Listing 5.15).
- **ToDevice** is a basic element of SymNet. It is the terminal point of the packet.

- GraphIDS.click -

```
1.   cli :: ClientIDS(80,7)
2.   ids :: ZmodelIDS(10,7,30)
3.   FromDevice -> cli -> ids -> ToDevice
```

Listing 5.10

```
      - TestResultIDS.output -
Failed: {
[...]{
"status":"Match-in-blacklist",
"port_trace":[
{"0":"fromDevice-0-in"},
{"1":"fromDevice-0-out"},
{"2":"cli-in"},
{"3":"cli-0-out"},
{"4":"ids-in"}],
"instruction_trace":[
[...]
{"63":"Forward(fromDevice-0-out)"},
{"64":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3444d69d"},
{"65":"AssignRaw(L3+72,[Const(80)],GenericNumeric)"},
{"66":"AssignRaw(L7+48,[Const(7)],GenericNumeric)"},
{"67":"Forward(cli-0-out)"},
{"68":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3444d69d"},
{"69":"AssignNamedSymbol(flag,[Const(0)],GenericNumeric)"},
{"70":"ConstrainRaw(L3+72,:~:(:==:([Const(70)])),Some(~(==([Const(70)]))))"},
{"71":"ConstrainRaw(L3+72,:==:([Const(80)]),None)"},
{"72":"ConstrainRaw(L7+48,:~:(:==:([Const(10)])),Some(~(==([Const(10)]))))"},
{"73":"org.change.v2.analysis.processingmodels.instructions.NoOp$@3444d69d"},
{"74":"ConstrainRaw(L7+48,:==:([Const(7)]),Some(==([Const(7)])))"},
{"75":"Fail(Match-in-blacklist)"}],
"memory":{"tags":[{"L4":160},{"START":0},
[...]
```

Listing 5.11

The expected result for this test is *Failed,* since we insert in the graph a packet with *URL* equal from the value blocked by the IDS. In the *file.output* [Listing 5.11] we can see all the ports traversed by the packet (before the failure) and the SEFL instructions executed by the solver. *Failed* is the first element of the *file.output.* The failure message reports the correspondence in the black-list of the IDS. The instruction n. 65 is about the assignment of *Proto* and the instruction n. 66 is about the assignment of *URL*. Instruction n. 70 checks if the *Proto* of the incoming packet is different from the value 70 (HTTP_REQUEST). Instruction n. 71 checks if the *Proto* of the incoming packet is different from the value 80 (HTTP_RESPONSE). Since it is the same, it follows instructions n.72, 74 that control the *URL* field of the incoming packet. It is equal to a value in the black-list, so the IDS drops the packet with a *Failed* (statement n. 75).

5.2.5 Network graph with a Traffic Classifier

In this reachability test, we check if the VNF of the Traffic Classifier generated by the VNF Modeling framework[16] works properly in the SymNet[5] environment. To do this we insert the VNF module into a specific network scenario. In Figure 5.7 there is the network topology of this test and in Listing 5.12 you can find the equivalent network graph in the Click format.

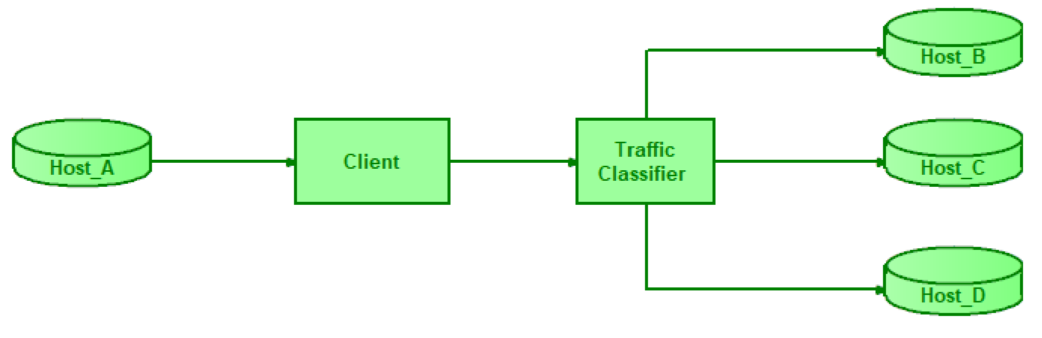


Figure 5.7

The graph is composed by 6 different elements:

- **Host_A** is a *FromDevice* element of SymNet. The behaviour is explained in the section of the firewall test 5.2.2.
- **Client** is the same as *ClientProtocol* (label *hosta*) used in the antispam test in section 5.2.3. In this test, we set the *Proto* field with the value 20 (POP3_REQUEST). The *EmailFrom* field is not important for this test, so we set the value to 0 (line 1 of Listing 5.12).
- **Traffic Classifier** is a new element that contains the behaviour of the Traffic Classifier generated by VNF Modeling. This VNF is configured with a whit-list as a list of <*Proto, Interface*>. Every packet with *Proto* contained into the white-list will be send through the *Interface* selected in the pair. Otherwise, the packet is blocked. The class written with Scala language for the Traffic Classifier is called *ZmodelCf* (label *cf*). In the test, we initialise this middle-box with three pairs (line 2 and 7 of Listing 5.12):
  1. <20,1> If the *Proto* is equal to 20 (POP3_REQUEST), it sends the packet through the *Interface* number 1 (Host_C).
  2. <10,0> If the *Proto* is equal to 10, it sends the packet through the *Interface* number 0 (Host_B).
  3. <80,2> If the *Proto* is equal to 80 (HTTP_RESPONSE), it sends the packet through the *Interface* number 2 (Host_D).
- **Host_B, Host_C and Host_D** are modeled with the *ToDevice* element of SymNet. It is the terminal point of the packet.

```
    - GraphClassifier.click -
hosta :: ClientProtocol(20,0)
cf :: ZmodelCf(20,1,10,0,80,2)
hostb :: ToDevice
hostc :: ToDevice
hostd :: ToDevice
FromDevice -> hosta -> cf[0] -> hostb
                       cf[1] -> hostc
                       cf[2] -> hostd
```

Listing 5.12

The expected result for this test is *Successful,* since we insert in the graph a packet with *Proto* present in the white-list of the Traffic Classifier. In the *file.output* [Listing 5.13] we can see all the ports traversed by the packet and the SEFL instructions executed by the solver. The instruction n. 65 is about the assignation of *Proto* and the instruction n. 66 is about the assignation of *URL*. Instructions n. 71 checks if the *Proto* of the incoming packet is equal from the value 20 (POP3_REQUEST). It is the same so the statement n.73 sets the output *interface*. After, there are some SEFL instructions to check if the *interface* exist and the packet is forwarded through the interface 1 to Host_C (label *hostc*).

```
    - TestResultClassifier.output -
Successful: { {"status":"OK",
"port_trace":[
{"0":"fromDevice-0-in"},{"1":"fromDevice-0-out"},
{"2":"hosta-in"},{"3":"hosta-0-out"},
{"4":"cf-in"},{"5":"cf-1-out"}, {"6":"hostc-in"},{"7":"hostc-out"}],
"instruction_trace":[ [...]
{"63":"Forward(fromDevice-0-out)"},
{"65":"AssignRaw(L3+72,[Const(20)],GenericNumeric)"},
{"66":"AssignRaw(L7+16,[Const(0)],GenericNumeric)"},
{"67":"Forward(hosta-0-out)"},
{"69":"AssignNamedSymbol(flag,[Const(0)],GenericNumeric)"},
{"70":"ConstrainNamedSymbol(flag,:==:([Const(0)]),Some(==([Const(0)])))"},
{"71":"ConstrainRaw(L3+72,:==:([Const(20)]),Some(==([Const(20)])))"},
{"72":"AssignNamedSymbol(flag,[Const(1)],GenericNumeric)"},
{"73":"AssignNamedSymbol(idIfSend,[Const(1)],GenericNumeric)"},
[...]
{"80":"ConstrainNamedSymbol(idIfSend,:==:([Const(1)]),Some(==([Const(1)])))"},
{"81":"Forward(cf-1-out)"},
{"85":"Forward(hostc-out)"},
[...]
```

Listing 5.13

5.2.6 Network graph with a Mail Server

In this reachability test, we check if the VNF of the Mail Server generated by the VNF Modeling framework[16] works properly in the SymNet[5] environment. To do this we insert the VNF module in a specific network scenario. In Figure 5.8 there is the network topology of this test and in Listing 5.14 you can find the equivalent network graph in the Click format.

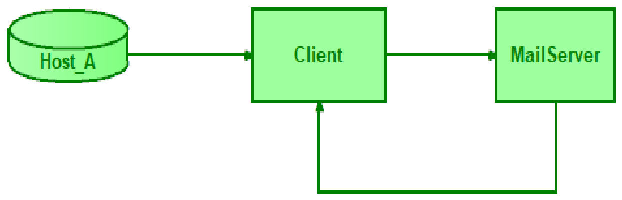


Figure 5.8

The graph is composed by 3 different elements:

- **Host_A** is a *FromDevice* element of SymNet[5]. The behaviour is explained in the section of the firewall test 5.2.2.
- **Client** is a new element that sets the *IPSrc, IPDst, Protocol and EmailFrom* fields of the packet. It receives a packet through the input port 0, sets the *Proto* to 20 (POP3_REQUEST) and sets *IPSrc, IPDst* and *EmailFrom* fields with the parameters received as configuration. Finally, it sends the packet through the default output port. The class written with Scala language for the Client is called *ClientMail* (label *mc*). In the test, we use *IPSrc* equal to 0.0.0.22, *IPDst* equal to 0.0.0.28 and *EmailFrom* equal to 6 (line 1 of Listing 5.14).
- **Mail Server** is a new element that contains the behaviour of the Mail Server generated by VNF Modeling. This VNF does not have input parameters. Packet with *Proto* equal to 20 (POP3_REQUEST) will be send after some modification. Otherwise the packet is blocked. The outgoing packet has:
  1. switches the *IPSrc* and the *IPDst* fields;
  2. the *Proto* field equal to 30 (POP3_RESPONSE)
  3. the *EmailFrom* field equal to 1 (RESPONSE)

  The class written with Scala language for the Traffic Classifier is called *ZmodelMS* (label *ms*).

```
    - GraphMailServer.click -
1.  mc :: ClientMail(0.0.0.16,0.0.0.32,6)
2.  ms :: ZmodelMS()
3.  FromDevice -> [0]mc -> ms -> [1]mc
```

Listing 5.14

```
      - TestResultMailServer.output -
Successful: {{
"status":"OK",
"port_trace":[
{"0":"fromDevice-0-in"},{"1":"fromDevice-0-out"},
{"2":"mc-fd-in"}, {"3":"mc-out"},
{"4":"ms-in"}, {"5":"ms-0-out"},
{"6":"mc-ms-in"}],
"instruction_trace":[
[...]
{"63":"Forward(fromDevice-0-out)"},
{"64":"org.change.v2.analysis.processingmodels.instructions.NoOp$@12028586"},
{"65":"AssignRaw(L3+72,[Const(20)],GenericNumeric)"},
{"66":"AssignRaw(L3+96,[Const(22)],GenericNumeric)"},
{"67":"AssignRaw(L3+128,[Const(28)],GenericNumeric)"},
{"68":"AssignRaw(L7+16,[Const(6)],GenericNumeric)"},
{"69":"Forward(mc-out)"},
{"70":"org.change.v2.analysis.processingmodels.instructions.NoOp$@12028586"},
{"71":"AssignNamedSymbol(flag,[Const(0)],GenericNumeric)"},
{"72":"ConstrainRaw(L3+72,:==:([Const(20)]),None)"},
{"73":"AssignRaw(L3+72,[Const(30)],GenericNumeric)"},
{"74":"AllocateSymbol(tmp)"},
{"75":"AssignNamedSymbol(tmp,Address(L3+96),GenericNumeric)"},
{"76":"AssignRaw(L3+96,Address(L3+128),GenericNumeric)"},
{"77":"AssignRaw(L3+128,Symbol(tmp),GenericNumeric)"},
{"78":"DeallocateNamedSymbol(tmp)"},
{"79":"AssignRaw(L7+16,[Const(1)],GenericNumeric)"},
{"80":"Forward(ms-0-out)"},
{"81":"org.change.v2.analysis.processingmodels.instructions.NoOp$@12028586"},
{"82":"org.change.v2.analysis.processingmodels.instructions.NoOp$@12028586"}],
"memory":{"tags":[{"L4":160},{"START":0},{"L3":0},{"L7":320},{"END":12320}],
"meta_symbols": [...]
Failed: {
}
```

Listing 5.15

The expected result for this test is *Successful,* since we insert in the graph a packet with *Proto* equal to 20 (POP3_REQUEST). In the *file.output* [Listing 5.15] we can see all the ports traversed by the packet and the SEFL instructions executed by the solver. The instructions from n. 65 to n. 68 are about the assignment of packet's fields. Instructions n. 72 checks if the *Proto* of the incoming packet is equal from the value 20 (POP3_REQUEST). It is equal so the statements n.73,76,77,79 modifying the outgoing packet according to the Mail Server rules. Finally, the instruction n. 80 forwards the packet.

5.2.7 Network graph with Web VNFs

In this reachability test, we check if the VNFs of the Web Server and Web Cache generated by the VNF Modeling framework[16] work properly in the SymNet[5] environment. To do this we insert the VNFs modules in a specific network scenario. In Figure 5.9 there is the network topology of this test.

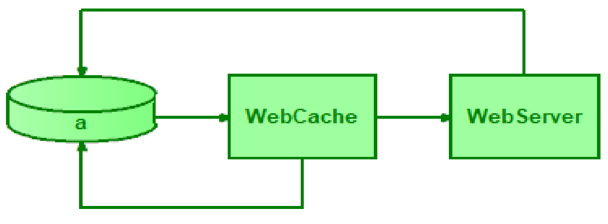


Figure 5.9

The graph is composed by 3 elements:

- **a** is a new element that sets the *IPSrc, IPDst, Protocol, EmailFrom, URL* fields of the packet. It receives a packet through the input port 0, sets the packet's fields with the parameters received as configuration. Finally, it sends the packet through the default output port. The class written with Scala language for the Client is called *ClientG*. In the test, we use *IPSrc* equal to 0.0.0.16, *IPDst* equal to 0.0.0.48, *Proto* equal to 70 (HTTP_REQUEST), *EmailFrom* equal to 100, *URL* equal to 31.
- **Web Cache** is a new element that contains the behaviour of the Web Cache generated by VNF Modeling. This VNF receives as parameters one list of known URLs. Packet with *Proto* equal to 70 (HTTP_REQUEST) and unknown *URL* field will be send to the default next hop (Web Server). If the *URL* packet's field is known, the Web Cache modifies some packet's fields and forward it. The new packet has:
  - switches the *IPSrc* and the *IPDst* fields;
  - the *Proto* field equal to 80 (HTTP_RESPONSE)

  Finally, the packet with *Proto* different from 70 is blocked.

  The class written with Scala language for the Web Cache is called *ZmodelWC*.
- **Web Server** is a new element that contains the behaviour of the Web Server generated by VNF Modeling. This VNF does not have input parameters. Packet with *Proto* equal to 70 (HTTP_REQUEST) will be send after some modification. Otherwise the packet is blocked. The outgoing packet has:
  - switches the *IPSrc* and the *IPDst* fields;
  - the *Proto* field equal to 80 (HTTP_RESPONSE).

  The class written with Scala language for the Web Server is called *ZmodelWS*.

The expected result for this test is *Successful,* since we insert in the graph a packet with *Proto* equal to 70 (HTTP_REQUEST). Furthermore, we expect to see that the packet does not reach the Web Server when the *URL* is known by the Web Cache. The VNF behaves as expected. In fact, the Web Cache responds without calling the Server. We repeated the test by entering *URL* equal to 100. In this case we observed that the packet is correctly sent to the Web Server as it is not known by the Web Cache. It follow the command line to launch the test in SymNet project: `‘sbt tweb’.`
The network topology is in the folder `.\Symnet\src\main\resources\click_test_files\vnf_model\GraphWebCache.click.`
And the *file.output* is in the folder `.\Symnet\output\TestResultWebCache.output`

5.2.8 Network graph with multiple VNFs

In this last reachability test, we check if the VNFs generated by the VNF Modeling framework[16] works properly in the SymNet[5] environment. To do this we insert the middle-boxes in a specific network scenario. In Figure 5.10 there is the network topology of this test.

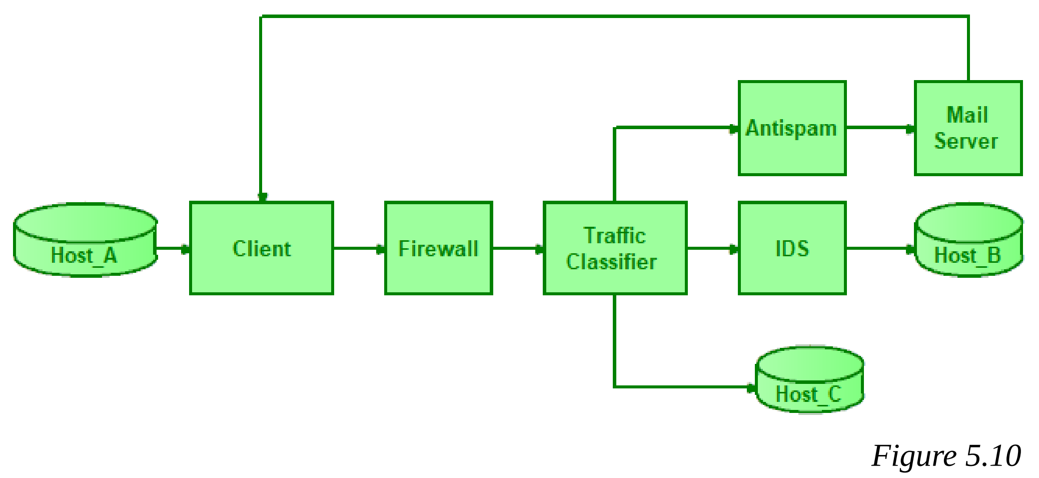


Figure 5.10

The graph is composed by 9 different elements:

- **Host_A** is a *FromDevice* element of SymNet.
- **Client** is a new element that sets the *IPSrc, IPDst, Protocol, EmailFrom, URL* fields of the packet. It receives a packet through the input port 0, sets the packet's fields with the parameters received as configuration. Finally, it sends the packet through the default output port. The class written with Scala language for the Client is called *ClientG*. In the test, we use *IPSrc* equal to 0.0.0.24, *IPDst* equal to 0.0.0.48, *Proto* equal to 80, *EmailFrom* equal to 5, *URL* equal to 2.
- **Firewall** is the new element such as that in the firewall test. His behaviour is explained in the section 5.2.2. In this test, we initialise the middle-box with two pairs: < 0.0.0.8, 0.0.0.16 >, < 0.0.0.24, 0.0.0.32 >.
- **Traffic Classifier** is the new element such as that in the Traffic Classifier Test. His behaviour is explained in the section 5.2.5. In this test, we initialise this middle-box with five pairs:
  - <20,0> If the *Proto* is equal to 20 (POP3_REQUEST), it sends the packet through the *Interface* number 0 (the next hop is the Antispam).
  - <30,0> If the *Proto* is equal to 30 (POP_RESPONSE), it sends the packet through the *Interface* number 0.
  - <10,2> If the *Proto* is equal to 10, it sends the packet through the *Interface* number 2 (next hop is the Host_C).

    - <70,1> If the *Proto* is equal to 70 (HTTP_REQUEST), it sends the packet through the *Interface* number 1 (next hop is the IDS).
    - <80,1> If the *Proto* is equal to 80 (HTTP_RESPONSE), it sends the packet through the *Interface* number 1.
- **Antispam** is the new element such as that in the Antispam test (section 5.1.3). In this test, we initialise the antispam with three values: 3, 4, 8.
- **IDS** is the new element such as that in the IDS test (section 5.1.4). In this test, we initialise this middle-box with three values: 10, 7, 30.
- **Mail Server** is the new element such as that in the Mail Server test (section 5.1.6).
- **Host_B and Host_C** are modeled with the *ToDevice* element of SymNet.

In this test we will check if the Host_A can reach the Host_B based on previous configurations. The expected result is *Successful.* Because, we insert in the graph a packet with *IPSrc* and *IPDst* different from the pairs blocked by the firewall. So the packet can transit to the Traffic Classifier. The packet has *Proto* field present in the white-list of the Traffic Classifier. And the packet can transit to the next hop. The *Proto* field is equal to 80 (HTTP_RESPONSE), so the next hop selected by the Traffic Classifier is the IDS node. The packet has *URL* field different from those present in the IDS black-list so the packet is send to the next hop (Host_B). This test can be performed in the SymNet project with the following command line: `‘sbt tallvnf’`.

This last test brings together all the VNFs. So we have a more realistic network scenario that brings together more functionality. If we change the input values to the VNFs we can 'play' with this model of the network to study all the possible combinations and analyse its behaviour. In the next part we will set up different configurations for the VNFs in the graph (Figure 5.10) to analyse different behaviours of the middel-boxes. In order to run these tests we have to modify the configurations in the graph (`.\Symnet\src\main\resources\click_test_files\vnf_model\graphAll.click`) and execute the following command: `‘sbt tallvnf’`.

- Configuration_A: Packet blocked by the Firewall.
    - The *ClientG* node sets the follow packet’s field

      IPSrc = 172.21.81.8; *IPDst* = 195.24.65.215;
      Proto = 80 (HTTP_RESPONSE);
      EmailFrom = 5;
      URL = 7;
    - The Firewall’s configuration has three pairs <*IPSrc*, *IPDst*> in the black-list:

      <0.0.0.8 – 0.0.0.16>; <0.0.0.24 – 0.0.0.16>; <172.21.81.8 – 192*.24.65.215>*
    - The others elements’ configuration does not change compared to the main test.
    - The result is *Failed.* Since*,* we insert in the graph a packet with *IPSrc* and *IPDst* equal from the pairs blocked by the firewall. In the *file.output* we can find a *Failed* element with the message “`Match-in-blacklist`”.

- Configuration_B: Packet blocked by the Traffic Classifier.
  - The *ClientG* node sets the follow packet's field
    IPSrc = 172.21.81.8; *IPDst* = 195.24.65.215;
    Proto = 12 (Unknown);
    EmailFrom = 5; *URL* = 7.
  - The others elements' configuration does not change compared to the main test.
  - The result is *Failed.* Since*,* we insert in the graph a Packet which has *Proto* field different from the white-list of the Traffic Classifier. In the *file.output* we can find a *Failed* element with message "`No-Match`".
- Configuration_C: Packet blocked by the IDS
  - The *ClientG* node sets the follow packet's field
    IPSrc = 172.21.81.8; *IPDst* = 195.24.65.215;
    Proto = 70 (HTTP_REQUEST);
    EmailFrom = 5; *URL* = 7.
  - The others elements' configuration does not change compared to the main test.
  - The result is *Failed.* We insert in the graph a Packet which has *Proto* field equal to a pair in the Traffic Classifier (<70,1>). Then the Packet is forwarded to the IDS node. Since the *URL* packet's field is equal to a value in the IDS's black-list, the packet is deleted. In the *file.output* we can find a *Failed* element with message "`Match-in-black-list`".
- Configuration_D: Packet blocked by the Antispam
  - The *ClientG* node sets the follow packet's field
    IPSrc = 172.21.81.8; *IPDst* = 195.24.65.215;
    Proto = 20 (POP3_REQUEST);
    EmailFrom = 800;   *URL* = 7.
  - The Antispam's black-list is initialised with seven values: 3, 4, 8, 300, 400, 800, 6.
  - The others elements' configuration does not change compared to the main test.
  - The result is *Failed.* We insert in the graph a Packet which has *Proto* field equal to a pair in the Traffic Classifier (<20,0>). Then the Packet is forwarded to the Antispam node. Since the *EmailFrom* packet's field is equal from a value in the Antispam's black-list, the packet is deleted. In the *file.output* we can find a *Failed* element with message "`Match-in-black-list`".

- Configuration_E: Packet send from Host_A to Host_C
  - The *ClientG* node sets the follow packet's field
    IPSrc = 172.21.81.48; *IPDst* = 195.24.65.128;
    Proto = 16;
    EmailFrom = 33; *URL* = 1000.
  - The Firewall's configuration has six pairs <*IPSrc*, *IPDst*> in the black-list:
    <0.0.0.8 – 0.0.0.16>; <0.0.0.24 – 0.0.0.16>; <172.21.81.8 – 192*.24.65.215>*
    <0.0.0.16 – 0.0.0.8>; <0.0.0.16 – 0.0.0.24>; <172.21.81.215 – 192.24.65.8>
  - The Traffic Classifier's configuration has eight pairs <*Proto*, *Interface*>:
    <20,0>; <30,0>; <10,2>; <70,1>; <80,1>; <16,2>;<200,2>; <300,2>;
  - The others elements' configuration does not change compared to the main test.
  - The result is *Successful.* We insert in the graph a packet with *IPSrc* and *IPDst* different from the pairs blocked by the firewall. So the packet can transit to the Traffic Classifier. The packet has *Proto* field present in the white-list of the Traffic Classifier. And the packet can transit to the next hop. The *Proto* field is equal from 16, so the next hop selected by the Traffic Classifier is the Host_C node.

- Configuration_F: Packet send from Host_A to Mail Server
  - The *ClientG* node sets the follow packet's field
    IPSrc = 172.21.81.48; *IPDst* = 195.24.65.128;
    Proto = 30 (POP3_RESPONSE);
    EmailFrom = 33; *URL* = 1000.
  - The others elements' configuration does not change compared to the main test.
  - The result is *Failed.* We insert in the graph a packet with *IPSrc* and *IPDst* different from the pairs blocked by the firewall. So the packet can transit to the Traffic Classifier. The packet has *Proto* field present in the white-list of the Traffic Classifier. And the packet can transit to the next hop. The *Proto* field is equal from 30, so the next hop selected by the Traffic Classifier is the Antispam node. It follow a check on the *Proto* and *EmailFrom* before transfer the packet to the Mail Server. The Mail Server checks the *Proto,* and since it is different from 20 (POP_REQUEST) the packet is dropped.

5.3 SymNet and VeriGraph compared

In this last chapter we will compare two verification tools to demonstrate that the virtual network functions generated by our framework can be tested both in SymNet[5] and VeriGraph[4] with the same result. On the other hand, we will also highlight the different behaviour of these instruments.

5.3.1 Test Scenario A

Given the network topology illustrated in Figure 5.11, we want to test the reachability policy from the node ‘a’ to the node ‘b’. We will generate the same scenario both in VeriGraph and in SymNet.

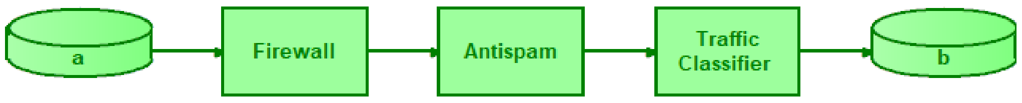


Figure 5.11

The first step for generate the reachability test is generating the graph and configuring the node of the network.

- **Host a**

We use a *PolitoEndHost* element in VeriGraph. And, we need two different elements in SymNet: the *FromDevice* to generate the packet and the *ClientG* to set the packet’s fields. As can be seen here there is the first difference between the two tools: the packet which traverse the network. As for SymNet, the node 'a' deals with the creation of a new packet and the assignment of data to it. On the contrary, in VeriGraph the packet is a separate element. This difference seems subtle but in reality hides a different behaviour in the verification phase. Both tools perform a reachability check. But in the case of VeriGraph, the question that is asked to the solver is: “is there AT LEAST A package that from node 'a' reaches node 'b', given the specific network configuration?” For SymNet the reachability test makes a slightly different question: “Is THE package generated on node ‘a’ able to cross the network graph and reach node ‘b’? In the case of VeriGraph the answer can be yes or no. Whether ‘a packet’ exists, it tells us the path that the packet does. Otherwise, if ‘a packet’ does not exist, it can not give us any other information. As far as SymNet is concerned, the answer can be yes, the package crosses these nodes and gets to the node ‘b’. Or not, the packet runs through the network up to node ‘x’ and then is interrupted. In this case, SymNet is better because in the case of fail, more information is returned to understand what has happened. Moreover, an other difference of the tools concern some fields of the packet. We can insert a packet with any value in SymNet. On the contrary, we can not specify all the packet's fields in VeriGraph. For example, the *IPSrc* field is decided by the solver when it looks for a possible packet to be sent from node 'a' to node 'b'.

- **Firewall (fw)**

It is the VNF of our model translated in the two verifications tools. We want configure this middle-box by setting the pairs of IP source address and IP destination address denied. Here too, we find a difference between the two instruments. In SymNet we can insert any pair of addresses in the black-list. VeriGraph, on the other hand, is more restrictive. It allows inserting in the black-list only the addresses that have been inserted in the configuration of the *service graph*. The Table 5.1 shows the configuration of the firewall's black-list both for VeriGraph and SymNet.

Tools	Pair n.1	Pair n.2
VeriGraph	`ip_a, ip_fw`	`172.2.16.248(ip_a), 0.0.0.16(ip_fw)`
SymNet	`ip_fw, ip-cf`	`0.0.0.16(ip_fw), 0.0.0.48(ip_cf)`

Table 5.1

- **Antispam (as)**

It is the VNF of our model translated in the two verifications tools. In this case there are not differences. We initialise the black-list with three values: 3, 300, 42.

- **Traffic Classifier (cf)**

It is the VNF of our model translated in the two verifications tools. There are some differences in this middle-box. Because, VeriGraph can not give a rules to select the output interface. More information on how this function is implemented on the two instruments is given in chapter 5.1.1 and 5.2.4.
SymNet white-list:

- <20,0> If the *Proto* is equal to 20 (POP3_REQUEST), it sends the packet through the *Interface* number 0 (the next hop is Antispam).
- <70,0> If the *Proto* is equal to 70 (HTTP_REQUEST), it sends the packet through the *Interface* number 0.
- <80,0> If the *Proto* is equal to 80 (HTTP_RESPONSE), it sends the packet through the *Interface* number 0.

VeriGraph white-list: 1 (HTTP_RESPONSE), 2 (HTTP_REQUEST), 3 (POP3_REQUEST)

- **Packet's fields**

The Table 5.2 shows the configuration of the packet both for VeriGraph and SymNet.

Packet's fields	VeriGraph	SymNet
IP source address	-	172.2.16.248 (ip_a)
IP destination address	ip_b	172.2.32.128 (ip_b)
Protocol	3 (POP3_REQUEST)	20 (POP3_REQUEST)
EmailFrom	0	0
URL	0	0

Table 5.2

- **Host b**

We use a *PolitoEndHost* element In VeriGraph. And, we use a *ToDevice* element in SymNet.

Another difference between the two verification tools is the definition of the network graph. SymNet lists the network nodes and links. VeriGraph is more specific, for each node we must indicate an address and the routing table.

Now we can go on to examine the result of the test. The expected result is Satisfiability. And effectively we have the same result in both environments. VeriGraph return *SAT* and, SymNet return *Successful.* Moreover, both verification tools return the path that the packet follows. SymNet gives this information in the *port_trace* element [Listing 5.16] and VeriGraph gives this information in the *send* element [Listing 5.17].

```
      - TestResult_A.output (SymNet)-
Successful: {
{"status":"OK",
"port_trace":[
{"0":"fromDevice-0-in"}, {"1":"fromDevice-0-out"},
{"2":"a-fd-in"},{"3":"a-0-out"}, {"4":"fw-in"},{"5":"fw-0-out"},
{"6":"as-in"},{"7":"as-0-out"}, {"8":"cf-in"},{"9":"cf-0-out"},
{"10":"b-in"},{"11":"b-out"}],
[...]
```

Listing 5.16

```
      - TestResult_A.output (VeriGraph)-
SAT
(define-fun send!410 ((x!0 Node) (x!1 Node) (x!2 packet)) Bool
  (ite (and (= x!0 cf)
            (= x!1 b)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 3 16 17 18 false)))
  (ite (and (= x!0 aspam)
            (= x!1 cf)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 3 16 17 18 false)))
  (ite (and (= x!0 fw)
            (= x!1 aspam)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 3 16 17 18 false)))
  (ite (and (= x!0 a)
            (= x!1 fw)
            (= x!2 (packet ip_a ip_b null null a 14 14 15 3 16 17 18 false)))
    false)))))
```

Listing 5.17

This test can be performed:

1. In the SymNet project with the following command line: `‘sbt testa’`.
2. In the VNF Modeling project by running the following class: `.\vnf-modeling-veriraph-ao\src\it\polito\verigraph\usecase\Test_Graph_A.java`

5.3.2 Test Scenario B

Given the network topology illustrated in Figure 5.12, we want to generate several tests in order to check the reachability policy. We will generate the same scenario both in VeriGraph and in SymNet.

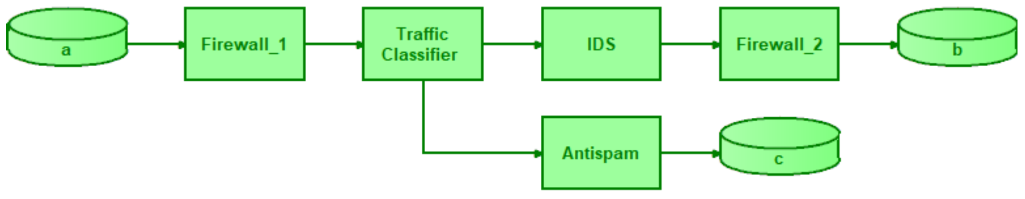


Figure 5.12

The basic configuration of this network's graph is:

- **Host_A (a)**
  - SymNet: *FromDevice* and *ClientG*.
  - VeriGraph: *PolitoEndHost*
- **Firewall_1 (fw1)**
  - Both in SymNet and VeriGraph we initialise the '*fw1*' with two pair:
    < ip_a, ip_fw2 >, < ip_a, ip_fw2 >.
- **Traffic Classifier (tc)**
  - SymNet white-list:

    <20,1> If the *Proto* is equal to 20 (POP3_REQUEST), it sends the packet through the *Interface* number 1 (the next hop is the Antispam node).

    <70,0> If the *Proto* is equal to 70 (HTTP_REQUEST), it sends the packet through the *Interface* number 0 (the next hop is the IDS node).

    <80,0> If the *Proto* is equal to 80 (HTTP_RESPONSE), it sends the packet through the *Interface* number 0.
  - VeriGraph white-list:

    1 (HTTP_RESPONSE), 2 (HTTP_REQUEST), 3 (POP3_REQUEST)
- **IDS (ids)**
  - Both in SymNet and VeriGraph we initialise the '*ids*' with the following black-list:
    6, 66, 666, 16, 61, 601
- **Antispam (as)**
  - Both in SymNet and VeriGraph we initialise the '*as*' with the following black-list:
    3, 300, 42, 33, 333
- **Firewall_2 (fw2)**
  - Both in SymNet and VeriGraph we initialise the '*fw2*' with two pair:
    < ip_b, ip_fw1 >, < ip_b, ip_as >.

- **Host_B (b) and Host_C (c)**
  - SymNet: *ToDevice.*
  - VeriGraph: *PolitoEndHost*
- **IP addresses**

Node	VeriGraph	SymNet
a	ip_a	172.2.12.61
b	ip_b	172.2.12.62
c	ip_c	172.2.12.63
fw1	ip_fw1	172.2.16.1
fw2	ip_fw2	172.2.16.2
ids	ip_ids	172.2.16.3
as	ip_as	172.2.16.4
cf	ip_cf	172.2.18.1

- **Packet's fields**

  IPSrc = ip_a
  IPDst = ip_b;
  Proto = HTTP_REQUEST;
  EmailFrom = 100;
  URL = 100.

The Table 5.3 illustrates all the use-case tests generated on this network's topology. In the first column of the table we can find the number of the test executed. In the second and third columns we indicate the source and destination nodes on which the reachability test is performed. The Configuration column indicates the packet's fields or/and the node's table that are modified to run the test. Finally, the last three columns report the result of the tests. Respectively we indicate: E for the expected result, V for the VeriGraph result and S for the SymNet result. The results can be satisfied (SAT) or unsatisfied (UNSAT).

N.	Node		Configuration	Results		
	Src	Dst		E	V	S
1	a	b	base	SAT	SAT	SAT
2	a	b	URL = 666	UNSAT	UNSAT	UNSAT
3	a	b	fw1's black-list: + <ip_a,ip_b>	UNSAT	UNSAT	UNSAT
4	a	b	fw2's black-list: + <ip_a,ip_b>	UNSAT	UNSAT	UNSAT
5	a	b	Proto = HTTP_RESPONSE	SAT	SAT	SAT
6	a	c	Proto = HTTP_REQUEST	UNSAT	UNSAT	UNSAT
7	a	c	Proto = POP3_REQUEST	SAT	SAT	SAT
8	a	c	Proto = POP3_REQUEST, EmailFrom = 333	UNSAT	UNSAT	UNSAT
9	a	c	Proto = POP3_REQUEST, fw1's black-list: + <ip_a,ip_c>	UNSAT	UNSAT	UNSAT

Table 5.3

There is a clearly defined pattern to the Table 5.3, and this can be taken to mean that the VNFs translated by our framework can be used both in VeriGraph and SymNet to achieve the same behaviour. It follow the information to reproduce the same test in the two environments:

- SymNet project:
  - Command line: `‘sbt testa’`
  - Network's graph:

    `.\Symnet\src\main\resources\click_test_files\vnf_model\Graph_Test_B.click`
  - Result: `.\Symnet\output\TestResult_B.output`
- VNF Modeling projects (VeriGraph):

  `.\vnf-modeling-verigraph-ao\src\it\polito\verigraph\usecase\Test_Graph_B.java`

5.3.3 Test Scenario Bis

It is easy represent links bidirectional in VeriGraph through the definition of the routing tables of the nodes. For that reason, the network’s graph of the previous example (Chapter 5.3.2 Test Scenario B) can be also used to check the reachability for the inverse configurations. For instance, we can verify the reachability between the ‘b’ node and the ‘a’ node, and between the ‘c’ node and the ‘a’ node. On the contrary, in SymNet the links are unidirectional and to do the specular tests we have to write new graphs.

Table 5.4 shows the use case tests generated on the topology of the specular network to the test scenario B.

N.	Node		Configuration	Results		
	Src	Dst		E	V	S
1	b	a	base	SAT	SAT	SAT
2	b	a	URL = 666	UNSAT	UNSAT	UNSAT
3	b	a	fw1’s black-list: + <ip_b,ip_a>	UNSAT	UNSAT	UNSAT
4	b	a	fw2’s black-list: + <ip_b,ip_a>	UNSAT	UNSAT	UNSAT
5	b	a	Proto = HTTP_RESPONSE	SAT	SAT	SAT
6	c	a	Proto = HTTP_REQUEST	UNSAT	USAT	UNSAT
7	c	a	Proto = POP3_REQUEST	SAT	SAT	SAT
8	c	a	Proto = POP3_REQUEST, EmailFrom = 333	UNSAT	UNSAT	UNSAT
9	c	a	Proto = POP3_REQUEST, fw1’s black-list: + <ip_c,ip_a>	UNSAT	UNSAT	UNSAT

Table 5.4

It follow the information to reproduce the same tests in the two environments:

- SymNet project:
  - Command line (Tests N. 1 to 5): `‘sbt tstbb’`

- Command line (Tests N. 6 to 9): `‘sbt tstbter’`
- Network's graph (Tests N. 1 to 5):
  `.\Symnet\src\main\resources\click_test_files\vnf_model\Graph_Test_Bbis.click`
- Result (Tests N. 1 to 5): `.\Symnet\output\TestResult_Bbis.output`
- Network's graph (Tests N. 6 to 9):
  `.\Symnet\src\main\resources\click_test_files\vnf_model\Graph_Test_Bter.click`
- Result (Tests N. 6 to 9): `.\Symnet\output\TestResult_Bter.output`

  ○ VNF Modeling projects (VeriGraph):
  `.\vnf-modeling-verigraph-ao\src\it\polito\verigraph\usecase\Test_Graph_B.java`

5.3.4 Test Scenario C

Given the network topology illustrated in Figure 5.13, we want to generate several tests in order to check the reachability policy. We will generate the same scenario both in VeriGraph and in SymNet.

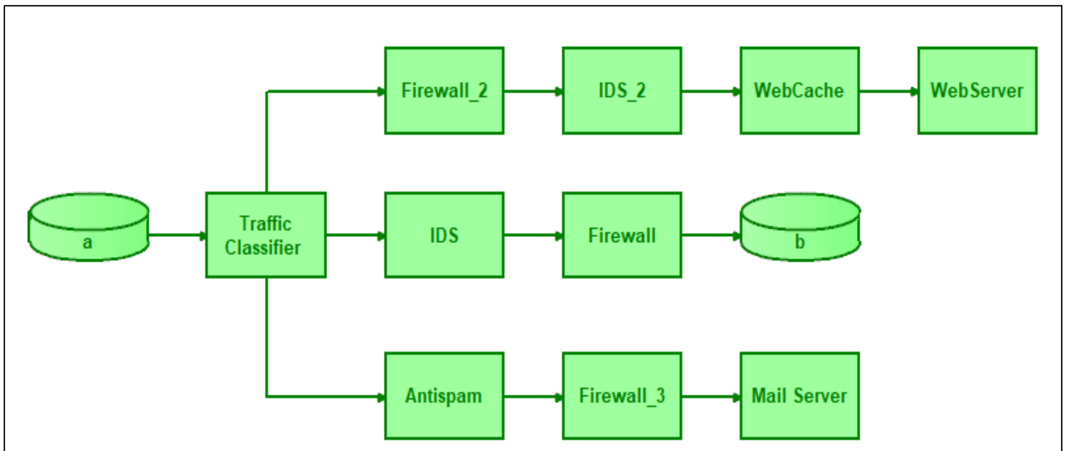


Figure 5.13

The basic configuration of this network's graph is:

- **Traffic Classifier (cf)**
  - SymNet white-list:

    <20,2> If the *Proto* is equal to 20 (POP3_REQUEST), it sends the packet through the *Interface* number 2 (Antispam).

    <70,1> If the *Proto* is equal to 70 (HTTP_REQUEST), it sends the packet through the *Interface* number 1 (IDS).

<80,0> If the *Proto* is equal to 80 (HTTP_RESPONSE), it sends the packet through the *Interface* number 0 (Firewall).

- VeriGraph white-list:

  1 (HTTP_RESPONSE), 2 (HTTP_REQUEST), 3 (POP3_REQUEST)

- **IDS (ids)**
  - Both in SymNet and VeriGraph we initialise the '*ids'* with the following black-list: 6, 66, 666, 16, 61, 601
- **Firewall (fw)**
  - Both in SymNet and VeriGraph we initialise the '*fw'* with two pair:

    < ip_a, ip_fw >, < ip_b, ip_a >.
- **Firewall_2 (fw2)**
  - Both in SymNet and VeriGraph we initialise the '*fw2'* with two pair:

    < ip_a, ip_fw >, < ip_a, ip_b >.
- **IDS_2 (ids2)**
  - Both in SymNet and VeriGraph we initialise the '*ids2'* with the following black-list: 26, 266, 2666, 216, 261, 2601
- **Antispam (as)**
  - Both in SymNet and VeriGraph we initialise the '*as'* with the following black-list:

    3, 300, 42, 33, 333
- **Firewall_3 (fw3)**
  - Both in SymNet and VeriGraph we initialise the '*fw2'* with three pair:

    < ip_a, ip_fw >, < ip_a, ip_b > <ip_a, ip_ws>.

The others VNF do not have any specific configuration different from the previous examples.

- **IP addresses**

Node	Symbol	VeriGraph	SymNet
End Host	a	ip_a	172.0.0.1
Traffic Classifier	cf	ip_cf	172.0.0.2
IDS(ids)	ids	ip_ids	172.0.0.3
Firewall	fw	ip_fw	172.0.0.4
End Host	b	ip_b	172.0.0.5
Firewall_2	fw2	ip_fw2	172.0.0.6
IDS_2	ids2	ip_ids2	172.0.0.7
Web Cache	wc	ip_wc	172.0.0.8
Web Server	ws	ip_ws	172.0.0.9
Antispam	as	ip_as	172.0.0.10
Firewall_3	fw3	ip_fw3	172.0.0.11
Mail Server	ms	ip_ms	172.0.0.12

- **Packet's fields**

  IPSrc = ip_a
  IPDst = ip_b;
  Proto = HTTP_REQUEST;
  EmailFrom = 100;
  URL = 100.

The Table 5.5 illustrates all the use-case tests generated on this network's topology. In the first column of the table we can find the number of the test executed. In the second and third columns we indicate the source and destination nodes on which the reachability test is performed. The Configuration column indicates the packet's fields or/and the node's table that are modified to run the test. Finally, the last three columns report the result of the tests. Respectively we indicate: E for the expected result, V for the VeriGraph result and S for the SymNet result. The results can be satisfied (SAT) or unsatisfied (UNSAT).

N.	Node		Configuration	Results		
	Src	Dst		E	V	S
1	a	b	Proto = HTTP_RESPONSE	SAT	SAT	SAT
2	a	b	Proto = 16	UNSAT	UNSAT	UNSAT
3	a	b	Proto = HTTP_RESPONSE, URL = 61	UNSAT	UNSAT	UNSAT
4	a	b	Proto = HTTP_RESPONSE, fw's black-list: + <ip_a,ip_b>	UNSAT	UNSAT	UNSAT
5	a	ws	base	SAT	SAT	SAT
6	a	ws	fw2's black-list: + <ip_a,ip_ws>	UNSAT	UNSAT	UNSAT
7	a	ws	URL = 2666	UNSAT	UNSAT	UNSAT
8	a	ms	Proto = POP3_REQUEST, fw3's black-list: + <ip_a,ip_ms>	UNSAT	UNSAT	UNSAT
9	a	ms	Proto = POP3_REQUEST	SAT	SAT	SAT
10	a	ms	Proto = POP3_REQUEST, EmailFrom = 300	UNSAT	UNSAT	UNSAT

Table 5.5

Chapter 6

Conclusion

The telecommunications scenario is rapidly evolving towards the new SDN/NFV paradigms. A lot of work has still to be done in the context of virtualization of network devices. One of the aspects to be expanded is the representation of the network functions, that are becoming increasingly numerous. Another important aspect is related to the verification of the systems before their deployment. This thesis work is related to these themes. In the first part of the work, the VNF Modeling tool [16] was analysed and extended, allowing a generic and user-friendly representation of the VNFs. In the second part of the thesis work, our purpose is to show that the obtained model is generic and can be moved to different verification systems.

Generated network models have been tested with both VeriGraph[4] and SymNet[5]. Despite the models very well represent the behaviour of forwarding network functions, there are problems related to their configuration within a specific network context. In VeriGraph, the part related to represent network functions capable of selecting a different network interface according to the incoming traffic is still lacking. As regards SymNet, there are some difficulties in representing stateful network functions. Nevertheless, the network function model is generic enough to describe the forwarding behaviour of a function independently of the tool that will be used to verify it. The translation phase allows to insert the rules of behaviour of forwarding the VNF on different verification systems. However, the part related to the configuration of the middle-boxes must be implemented by the verification tool and the possibility that these can be completely integrated depends on the fact that the desired behaviour can be represented by the verification tool.

Overall, an important aspect that can be appreciated during this thesis work is the possibility of obtain a modular programming for some network functions. Our tool allows to describe a network function in the widely used Java programming language and, after the translation, to verify its behaviour simply by inserting the new module in a graph of the verification system.

Bibliography

[1] Patrick MeLampy, “If applications speak in sessions, shouldn’t your router be session smart?” In NETWORKWORLD From IDG, FEB 26, 2018. *https://www.networkworld.com/article/3258645/router/if-applications-speak-in-sessions-shouldnt-your-router-be-session-smart.html*

[2] Lee Doyle, “The birth of the network function virtualization (NFV) ISV community” In NETWORKWORLD From IDG, APR 30, 2013. *https://www.networkworld.com/article/2165812/lan-wan/the-birth-of-the-network-function-virtualization--nfv--isv-community.html*

[3] Alan Weissberger, “VNF” In IEEE Communications Society-Technology Blog, December 6, 2017. *http://techblog.comsoc.org/tag/vnf/*

[4] S. Spinoso, M. Virgilio, W. John, A. Manzalini, G. Marchetto, and R. Sisto, “Formal Verification of Virtual Network Function Graphs in an SP-DevOps Context,” in Service Oriented and Cloud Computing - 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015. Proceedings, 2015, pp. 253–262. [Online]. Available: *http://dx.doi.org/10.1007/978-3-319-24072-5_18*

[5] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, “Scalable Symbolic Execution for Modern Networks,” Proceedings of the ACM SIGCOMM 2016 Conference, pp. 314–327, 2016.

[6] B. Tschaen, T. B. Ying Zhang, J. L. Sujata Banerjee, and J.-M. Kang, “SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. *https://users.cs.duke.edu/~btschaen/papers/SFC-Checker.pdf*

[7] W3C documentation for the XML Schema language is available at *https://www.w3.org/TR/xmlschema-1*

[8] M. Hashimoto, *Vagrant Up and Running,* USA, O’Reilly Media, June 2013

[9] Francesco Conti, “Development of a parser for a framework for Virtual Network Functions (VNF) modelling and Service Graph verification in SDN/Cloud context” pp.33

[10] Leonardo De Moura and Nikolaj Bjrner. \Z3: An efficient SMT solver". In: Tools and Algorithms for the Construction and Analysis of Systems (2008), pp. 337-340.

[11] H. B. Enderton, A mathematical introduction to logic, 2001, vol. 40, no. 2, p. 317

[12] Mariacristina Sinagra, “Comparison of tools for Formal Verification of Virtual Network Function Graphs” pp.19-23

[13] Kohsuke Kawaguchi, Sekhar Vajjhala, Joe Fialli, The Java™ Architecture for XML Binding (JAXB) 2.2, Sun Microsystems, December 2009

[14] Thomas Kuhn, Eye Media GmbH, Olivier Thomann, “Abstract Syntax Tree”, IBM Ottawa Lab, November 2006. *https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/*

[15] The repository of javatoscala tools is: *https://github.com/koofr/javatoscala*

[16] J. Yusupov, R. Sisto, G. Marchetto, “A Framework for Verification-oriented User-friendly VNF Modeling”