POLITECNICO DI TORINO

Facoltà di ingegneria

Corso di laurea in ingegneria informatica

Tesi di Laurea Magistrale

A framework for Virtual Network Functions (VNF) modeling and Service Graph verification in SDN/Cloud context

Relatore:

Prof. Riccardo Sisto

Prof. Guido Marchetto

Candidato:

Matteo Marchetti

Ottobre 2016

Index

Chapter 1

General presentation and objective of the work

The Telecommunications' world is becoming more and more important and new studies in this sector are coming out every day and change the entire structure of the system. But the increased necessity of flexibility leads creating something in software that before was created in hardware. In this way, network functions like antispam, firewall, NAT become a specific application that run in a virtual machine or in a non-custom hardware.
This is the context where new paradigms like Network Function Virtualization (NFV) and Software Defined Network (SDN) are born. These concepts are firstly explained in order to understand the meaning of this thesis.

NFV is a concept that converts network node functions into software blocks that may be connected to create network services. In this way instead of using different custom hardware, a Virtual Network Function could use a series of virtual machines that run different software (linked to the network function they simulate).
The benefits for the service provider are quite evident. A software network can be managed easily and the reduced amount of required hardware could reduce the cost.

SDN is a way to manage network services through abstraction of their functionality. This concept was born from the gap between the nowadays dynamic network's needs and the static architecture of traditional systems.
Using SDN, the network control became directly programmable and the infrastructure can be abstracted from the network function application.

From this scenario it is evident that a revolution in the telecommunication world is not so far. But to convert the present system into a software one, something is still missing.
At present there is nothing that allows the easily creation of the network function that you need and to verify that specific network properties are always guaranteed before deploying. It is a tricky

aspect especially in the networks that need an automatic reconfiguration in response to traffic or user events.

This is the reason of this thesis' project: to create a framework that allows Virtual Network Function modelling in a easy way and offer a tool for the verification of the service graph created.

The goal of the first part consist of using a well-known programing language that allows us to restrict the instructions available for the user.

For this reason a specific library in Java (one of the most famous object oriented programming language) is developed.

With this library the user can simply describe the functionality of the network function through the implementation of the defined methods and using the basic instructions of Java and the instructions offered.

After just running a parser an other Java file will be automatically created. It contains the same information of the first one, but written in a different way.

The goal of this part is obvious: try to translate the idea and the constraints created by the program user into an other form.

This second file represents the data source for the verification system (that will be illustrated in the following chapters).

It contains a class that can be implemented and it receives the specific configuration (if needed).

Connecting a series of these classes the user can create the network that he wants to test, and using specific instructions he can verify some basic network properties.

This is an extreme a brief outline of the work done for this thesis, the scenario where it came out and its objectives.

In the following chapters the basis of this work, the theories and the previous studies and the ideas that help to reach the objectives will be clarified.

Chapter 2

Previous studies

The work of this thesis is based on two previous studies:

1) Formal verification of Virtual Network Function graphs in an SP-DevOps context
2) Verifying isolation properties in the presence of Middleboxes

These two studies present the general ideas of the problems that would occur by developing the system object of this thesis and present some basic solutions.
A brief presentation of the studies follows, which includes their objectives and their results.

Formal verification of Virtual Network Function graphs in an SP-DevOps context [1]
This study arose from the observation of the current trend in the Telecommunications sector called "Softwarization". This term indicates the evolution of this world towards a new one in which network and service functions will be virtualized.
Following a previous UNIFY study, the paper of the study defines the meaning of Network Function Forwarding Graph (NF-FG) as an interface that combines compute, storage and network abstractions, and it defines the mapping of VNFs and their forwarding set of rules.
Moreover, it describes the paradigm of Service Provider-DevOps (SP-DevOps) as a method to simplify and automate management processes in a telecommunications infrastructure.
The focus of the study regards the UNIFY verification process, an essential task when service configurations change quickly in response to user requests or management events, and try to reduce some limitations in its model extending its concept and representation of a network.
In order to achieve high performances, the verification process is performed using Z3, a state of the art SMT solver. The rules of the network and its nodes are translated into sets of First Order Logic (FOL) formulas that are analysed by Z3.
The choice of the tool is motivated by its overall performance and scalability.
The core of the work is to create some model of VNFs and create some NF-FGs using them. Then, the verification process check the satisfiability of a given property (essential reachability problem, it leaves the verification of the other network property as a future work).

$$
\begin{aligned}
&(send(cache, n_0, p_0, t_0) \wedge \neg isInternal(n_0)) \implies \neg isInCache(p_0.url, t_0) \\
&\quad \wedge p_0.proto = HTTP\_REQ \wedge \exists (t_1, n_1) \mid (t_1 < t_0 \wedge isInternalNode(n_1) \\
&\quad \wedge recv(n_1, cache, p_0, t_1)), \forall n_0, p_0, t_0
\end{aligned} \tag{1a}
$$

$$
\begin{aligned}
&(send(cache, n_0, p_0, t_0) \wedge isInternal(n_0)) \implies isInCache(p_0.url, t_0) \\
&\quad \wedge p_0.proto = HTTP\_RESP \wedge p_0.ip\_src = p_1.ip\_dest \wedge p_0.ip\_dest = p_1.ip\_src \wedge \\
&\quad \wedge \exists (p_1, t_1) \mid (t_1 < t_0 \wedge p_1.protocol = HTTP\_REQ \wedge p_1.url = p_0.url \\
&\quad \wedge recv(n_0, cache, p_1, t_1)), \forall n_0, p_0, t_0
\end{aligned} \tag{1b}
$$

$$
\begin{aligned}
&isInCache(u_0, t_0) \implies \exists (t_1, t_2, p_1, p_2, n_1, n_1) \mid (t_1 < t_2 \wedge t_1 < t_0 \wedge t_2 < t_0 \\
&\quad \wedge recv(n_1, cache, p_1, t_1) \wedge recv(n_2, cache, p_2, t_2) \wedge p_1.proto = HTTP\_REQ \\
&\quad \wedge p_1.url = u_0 \wedge p_2.proto = HTTP\_RESP \wedge p_2.url = u_0 \wedge isInternal(n_2)) \\
&\quad \forall u_0, t_0
\end{aligned} \tag{1c}
$$

Web cache model

$$
\begin{aligned}
&(send(nat, n_0, p_0, t_0) \wedge \neg isPrivateAddress(p_0.ip\_dest)) \implies p_0.ip\_src = ip\_nat \\
&\quad \wedge \exists (n_1, p_1, t_1) \mid (t_1 < t_0 \wedge recv(n_1, nat, p_1, t_1) \wedge isPrivateAddress(p_1.ip\_src) \\
&\quad \wedge p_1.origin = p_0.origin \wedge p_1.ip\_dest = p_0.ip\_dest \wedge p_1.seq\_no = p_0.seq\_no \\
&\quad \wedge p_1.proto = p_0.proto \wedge p_1.email\_from = p_0.email\_from \wedge p_1.url = p_0.url) \\
&\quad \forall n_0, p_0, t_0
\end{aligned} \tag{2a}
$$

$$
\begin{aligned}
&(send(nat, n_0, p_0, t_0) \wedge isPrivateAddress(p_0.ip_dest)) \implies \neg isPrivateAddress(p_0.ip\_src) \\
&\quad \wedge \exists (n_1, p_1, t_1) \mid (t_1 < t_0 \wedge recv(n_1, nat, p_1, t_1) \wedge \neg isPrivateAddress(p_1.ip\_src) \\
&\quad \wedge p_1.ip\_dest = ip\_nat \wedge p_1.ip\_src = p_0.ip\_src \wedge p_1.origin = p_0.origin \\
&\quad \wedge p_1.seq\_no = p_0.seq\_no \wedge p_1.proto = p_0.proto \wedge p_1.email\_from = p_0.email\_from \\
&\quad \wedge p_1.url = p_0.url) \wedge \exists (n_2, p_2, t_2) \mid (t_2 < t_1 \wedge recv(n_2, nat, p_2, t_2) \\
&\quad \wedge isPrivateAddress(p_2.ip\_src) \wedge p_2.ip\_dest = p_1.ip\_src \wedge p_2.ip\_dest = p_0.ip\_src \\
&\quad \wedge p_2.ip\_src = p_0.ip\_dest), \forall n_0, p_0, t_0
\end{aligned} \tag{2b}
$$

NAT model

The previous two images (extracted from the paper) are examples of VNFs modelled with FOL logic formulas. They are two examples of VNFs that can modify the packet during forwarding. The respect of reachability properties between two nodes linked by one of these types of node is an important aspect that is analysed and tested in this paper (and not studied before).

In order to understand easily how these kinds of formulas can describe the behaviour of a network function (these models are one of the base of this thesis), the meaning of Web Cache formulas are clarified in the description that follows.

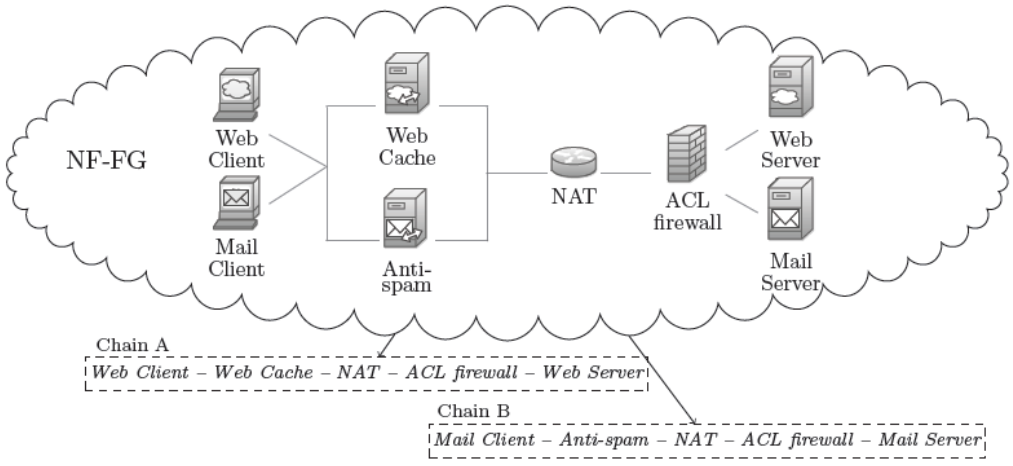
Web Cache has two interfaces connected to a private and an external network. In the private one there are clients that issue HTTP requests.

The first formula (1a) states that when a packet is sent to the external network, there must be a previous packet containing an HTTP request, received from an internal node and the request cannot be served by the cache (the url of the resource requested is not present in the Cache).

The second formula (1b) states that when a packet is sent to the internal network, its protocol must be a HTTP RESPONSE, the url of its resource must be present in the cache when the request has been received, and there must be a previous packet coming from the internal network, containing an HTTP request for the same url.

The third formula (1c) defines the isInCache function. It states that an url is present in the cache at time t0 if there is a previous packet containing an HTTP request for the same url and there is an other packet that follows received after that contains an HTTP response for the same url.

In order to carry out tests the network below was considered.



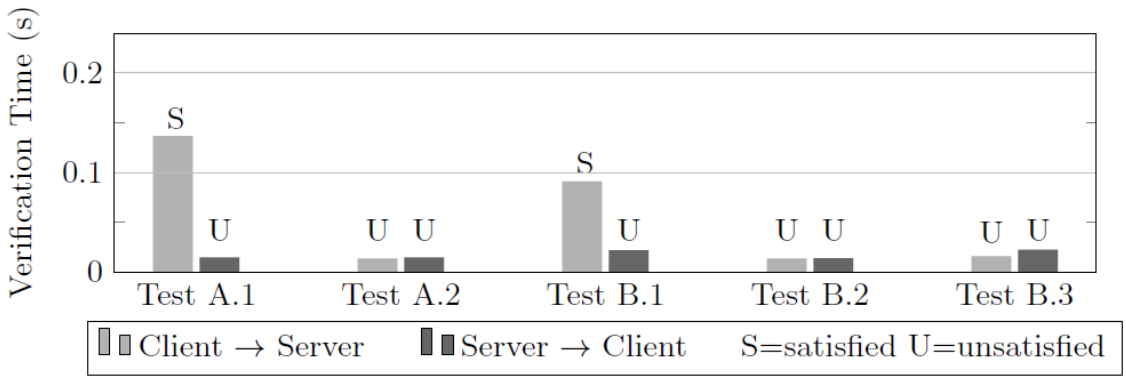
Test A.1: Chain A with firewall configured to allow all the traffic

Test A.2: Chain A with firewall configured to drop all packets exchanged between web client and server

Test B.1: Chain B, the functions allow all the traffic

Test B.2: Chain B with firewall configured to drop all packets exchanged between web client and server

Test B.3: Chain B with antispam configured to drop all the emails sent by the mail client



The previous table shows the results of the different tests in the two directions (client to server and server to client).

In the worst case the reachability problem is solved in less than 200ms and less than 50ms in most cases.

The results achieved show the validity of the work and it provides the basic structure of this thesis work.

This is only a short review of the paper that can explain some basic concepts used for the thesis. The original paper is available at: http://porto.polito.it/2624679/

Verifying isolation properties in the presence of Middleboxes [2]

This study is focused on the verification of isolation properties in very large networks that include some dynamic datapath elements (nodes that change their forwarding behaviour according to the traffic they have seen) using model checking and an SMT solver.

The growing number of middleboxes in the enterprise network justify this study, and this number will grow more with the use of Network Function Virtualization (a middlebox can be deployed without changing the hardware).

In order to make the tests feasible, the middleboxes models are simplified. They capture only the packet header and it can be derived from a general description of the middlebox's behaviour.

A theory has been developed to describe the property of a middlebox (not included here) and then they implement a system to verify invariants in a given network using Z3.

For verification they create middlebox models written in Python and a model of the network including topology information, routing tables, middlebox configurations and end host metadata.

The system converts these two inputs, adds additional assertions that describe the physical behaviour of a network and produces the input for Z3.

After the computation, Z3 returns:

1) UNSAT: indicating that the input assertions can never be satisfied
2) SAT: indicating that a satisfiable assignment was found
3) UNDEFINED: indicating the time out

To have a problem satisfiable by Z3, they introduce some restrictions to the middlebox models:

1 – Models are loop-free and all received packets are processed in a fixed number of steps

2 – Models only have access to their local state and use the network to share states

3 – Models can perform a limited set of actions: receive and send packets, check conditions, update states (not modify packet)

4 – Models must be deterministic (e.g. NAT is not deterministic in assignment ports to the data flow)

The paper also includes a possible definition of the network function written using a subset of Python.

```
1  def LearningFw(node, acl, flows):
2    p = recv(node)
3    if acl[p.src(), p.dest()]:
4      send(node, p)
5      flows.set((p.src(), p.dest(),
           p.src_port(), p.dest_port()),
           True)
6    elif flows[(p.dest(), p.src(),
         p.dest_port(), p.src_port())]:
7      send(node, p)
8
9  acl = ConfigMap((Address, Address), Bool)
10 flows = Map((Address, Address, Int, Int),
       Bool)
11 LearningFw(f, acl, flows)
```

Model for a learning firewall

The previous image is an example of a firewall written using their set of instructions, which an user can:

- Read and set values from instances of Map objects
- Read values from ConfigMap objects
- Call uninterpreted functions with finite codomains
- Use conditional constructions if, else, elif
- Call the recv function to receive a packet
- Call the send function to send a packet

From this kind of definition the creators of this project assert that an equivalent set of formulas can be derived in temporal logic and then supplied to Z3.

In particular they say that following the abstract syntax tree of the written code they can extract a path of condition that must hold in order to reach the send action, and from this path they derive the formulas. The process is not specified and it will be taken as starting point for this thesis.

After they create a set of variables that permit testing some properties of the built network.

These properties are:

- Node isolation: To check node isolation between nodes a and b. If the solver returns UNSAT the nodes are isolated.
- Flow isolation: Like node isolation, but adding the condition that b has never before sent a packet to a.
- Data isolation: It states that a never accesses data originating from b
- Node traversal: To check that all the traffic from a to b passes through some middlebox m

At the end they make some considerations about First Order Logic formulas. In general FOL is undecidable, but they show that using the restricted models and topology that they have defined, the formulas that they obtain lie in a decidable fragment of FOL (some general concepts of FOL are explained in the next paragraph).

This is only a short review of the paper that can explain some basic concept used for the thesis. The original paper is available at: http://arxiv.org/pdf/1409.7687v1.pdf

From these two articles I take same general or more specific concepts that I used for my thesis expanding or using them as a starting point of the work.

Chapter 3

Tips of First Order Logic's theory

In this chapter I would like to insert same basic concepts of First Order Logic's theory (FOL's theory) in order to better understand the work of the thesis.

First Order Logic is a collection of formal systems. It is used in a wide range of fields like mathematics and computer science, but also in philosophy and linguistic.
It uses quantified variables over non-logical objects and using them and predicates it creates sentences (these are the main differences between FOL and propositional logic, that doesn't use quantifiers and predicates).

The quantifiers are two:

1) Universal quantifier, also known as "for every" (∀). The sentence $\forall a\ p(a)$ means that p(a) is true for every choices of a.
2) Existential quantifier, also known as "there exists" (∃). The sentence $\exists a\ p(a)$ means there exists a particular choice of a that makes p(a) true.

The alphabet also includes logical and non-logical symbols.

The logical symbols are:

1) The quantifier symbols ∀ and ∃
2) The logical connectives: ∧ for conjunction, ∨ for disjunction, → for implication, ↔ for biconditional, ¬ for negation
3) Punctuation symbols like parentheses, brackets, …
4) The equality symbol =
5) An infinite set of variables indicated with lowercase letters of the end of the alphabet like x,y,z

In order to avoid the use of parentheses in some case, there is a convention for the evaluation order of the operand: first is evaluated the negation, after conjunction and disjunction, after quantifiers and at the end the implication.

The non-logical symbols are predicate symbols (denoted with uppercase letters like P,Q,R) and functional symbols (denoted with lowercase letters like f,g,h).

To understand the meaning of the created FOL formulas of the network functions, I think it may be useful insert also the truth table of conjunction, disjunction, negation and implication.

Truth table of conjunction

p	q	p ∧ q
true	true	true
true	false	false
false	false	false
false	true	false

Truth table of disjunction

p	q	p ∨ q
true	true	false
true	false	true
false	false	false
false	true	true

Truth table of negation

p	$\neg p$
true	false
true	false
false	true
false	true

Truth table of implication

p	q	$p \Rightarrow q$
true	true	true
true	false	false
false	false	true
false	true	true

Giving an assignment to each variable of the formula, and interpretation is created.
With the interpretation can be evaluated the truth values of the formula (true or false), valuating firstly the truth values of the atomic formulas and then the logical connectives, the existential quantifiers and the universal quantifiers in this order.
A formula is satisfiable if there is an interpretation that makes the formula true.
A formula is logically valid or tautology if it is true with every interpretation.

First-order logic is undecidable (or semidecidable). This means that there is not a procedure that certain can determine if a formula is logically valid. But if it possible, there are some algorithm that can prove the logical consequence relation of two formulas. [3]

As written in the previous paragraph, the formulas used in this work are a restricted part of the FOL formulas, so the SMT solver can determine in a finite period of time if the model (the set of formulas) is satisfiable or not.

Chapter 4

Software tools used

The development of the current work thesis has required the utilization and the integration of some different tools and software programming languages.

The main programming language used is **Java** version 8.
Java is a general purpose, concurrent, class-based, object-oriented language. It is thought to be simple for programmers, have as few dependencies as possible and it is intended to be a production language, not a research one.
It is based on the concept WORA (Write Once, Run Anywhere), that means that compiling Java code on a machine (obtaining Java bytecode), you can run it on any other machine regardless its architecture without recompilation. This feature is obtained using a Java Virtual Machine (JVM) that must be installed on the computer. This characteristic is one the reason of its wide use, particularly for client-server web applications.
Java is a high-level programming language, so the details of the machine representation are not available through the language.
An other interesting feature of the language is the garbage collector, an automatically system for storage management that avoids safety problems derived from explicit deallocation (like in C or C++).

Java lacks any formal standardization so the Oracle implementation is its de facto standard.
This implementation has two different distributions: Java Runtime Environment (JRE) that allows end users to run Java programs and Java Development Kit (JDK) for software developers.

The syntax of Java is similar to C++ one, but it is exclusively an object-oriented language, so all Java code is written inside classes and every data item is an object (excluding primitive types like integer and characters).
Java supports inheritance, so a class can be derived from an other one and it can access some of its fields and methods. Every class derives from superclass Object.

Java also supports interfaces. A class can implement one or more interfaces and it inherits all their abstract methods (that must be implemented).
Java class can contain local, instance or class variables and can have some methods.
Constructor is a particular method that must be present (also if it is not explained there is a default one) and it is invoked every time an object is created from a class. [4] [5]

Java also uses packages, a way to group classes, interfaces, enumerations that refer a particular aspect and which it provides access protection and namespace management. A particular package used for this thesis is `com.microsoft.z3` that allows the integration of Z3 with Java.

All the project is developed using **Eclipse**, one of the integrated developed environment (IDE) most used for programming in Java. It offers a base workspace and an extensible plug-in system for customizing the environment.
It is written mostly in Java, and it is also used to program with a wide range of languages like PHP, C, C++, JavaScript, Python and others.
Except for a small run-time kernel, everything in Eclipse is a plug-in. Using them, it offers a wide range of features (also created from third parties). [3]

Using Eclipse, starting from a Java source code, an **Abstract Syntax Tree** (AST) could be created. This represents one of the key instruments of this thesis work.
AST is the way that use Eclipse to look the source code: every Java file is represented as a tree of AST nodes. All these nodes are subclasses of ASTNode and are specialized for the different elements of Java programming language. For example, there are nodes for method declaration, variable declaration, assignment and others.
One of the core of the project is a parser that analyses the user classes of virtual network functions. This parser use an ASTParser. In order to explore the entire AST of the class, ASTVisitor are used.
The ASTVisitor is passed to any nodes of the AST, and for every ASTNode are invocated (in order) the methods: preVisit, visit, endVisit, postVisit.
If visit method returns true, the process continues recursively in the subtree of the current node.
Every subclass of ASTNode contains specific information that depends from the Java element it represent. For example, MethodDeclaration will contain information about name, return type,

parameters, and others. The information is referred as structural properties. Their values can be extracted using a specific method. [6]
This is the way which I can extract the information that I need from the user defined classes and using them create an internal representation of the network functions (the details will be explained in 7th paragraph).

An other tool used is **Z3**, a Satisfiability Modulo Theories (SMT) solver. It's a Microsoft software and it is used in several analysis, verification, testcase generation projects.
It can determine if a set of formulas are satisfiables. If this is true, the solver returns SAT, otherwise, if the set of formulas do not have a model, it returns UNSAT.
Generally speaking, in boolean case, a model is a truth assignment to the boolean variables.
Instead in the first-order case (as in this work), a model assigns values from a domain to variables and interpretations over the domain to the function and predicate symbols.
In this work it is used as oracle to prove if a property of a network can be satisfied or not. [7][8]

For the final part of the work some scripts with **Ant** have been created (their use will be explained in the 8th paragraph).
Ant is a Java library developed from Apache that allows the automation of the development process of a Java application. Using it, you can create a project that compiles, creates the documentation and the jar files and other things only running it.
Ant reads the commands from an XML file often called build.xml, that contains the available operations called target and inside them the commands called task.
Every target could have some previous ones: when this target is called, the previous ones are executed first. [9]

These are only short summary of the languages and tool used that may be useful in order to better understand their functions into the project.

Chapter 5

Starting point of the project

The starting point of this thesis work has been a previous project for porting in Java a set of Python classes developed at UC Berkeley for performing reachability verifications on chains of Virtual Network Functions (that is the study explained in the 2nd paragraph).

Inside this project there were some classes that represent the VNFs and some others that represent the network features and the tests.

They are split in 4 packages:

1) mcnet.components: it contains the classes for the generic model of a network and the way to perform the reachability verification. Some important classes contained are *Checher.java* that implements various checks for specific properties in the network, *Network.java* that defines the model of a network, including routing and wiring, *NetworkObject.java* that is the abstract class of a generic VNF which must be inherited by every specific VNF.
2) mcnet.netobjs: it contains the classes that implement the specific VNFs like *AclFirewall.java, PolitoAntispam.java, PolitoMailClient.java*, …
3) tests: it contains the classes to run the tests and in which you can choose the properties to test. The name of the tests are quite self-explicative and contain the names of the VNF under test (like *Polito_CacheNatFwTest.java*).
4) tests.example: it contains the classes that define the structure of the network for the various tests. In these classes the ones that implement the VNFs are instantiated, their properties are set and they are linked together following the topology of the network.

These classes supply the input formulas for Z3.

Running a test it is possible to verify if the property indicated in the test for the related network could be satisfied or not. If it is true, the result is SAT, otherwise the result is UNSAT.

There is a third option that is UNKNOW and it indicates that the solver could not determine if the model is satisfiable or not.

The following two tables show in a very user friendly way the inputs supply to Z3 put beside the related classes. [10]

Virtual Network Function' classes

Class - method	Constraints explaination
AclFirewall	send(fw, n_0, p, t_0) -> (∃ n_1,t_1 : (recv(n_1, fw, p, t_1) && t_1 < t_0 && !acl_func(p.src,p.dest))
	acl_func(a_0,a_1) == or(foreach ip1,ip2 in acl_map ((a_0 == ip1 && a_1 == ip2)||(a_0 == ip2 && a_1 == ip1)))
EndHost	send(node, n_0, p, t_0) -> nodeHasAddr(node,p.src)
	send(node, n_0, p, t_0) -> p.origin == node
	send(node, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, node, p, t_0) -> nodeHasAddr(node,p.dest)
PolitoAntispam	if(isInBlackList(ef_0) == or(for bl in blacklist (ef_0==bl)) ? true : false
	isInblackList(ef_0) == false
	send(politoAntispam, n_0, p, t_0) && p.proto(POP3_RESP) -> (∃ n_1,t_1 : (recv(n_1, politoAntispam, p, t_1) && t_1 < t_0)) && !isInBlackList(p.emailFrom)
	send(politoAntispam, n_0, p, t_0) && p.proto(POP3_REQ) -> (∃ n_1,t_1 : (recv(n_1, politoAntispam, p, t_1) && t_1 < t_0))
	send(politoAntispam, politoErrFunction, p, t_0) -> (∃ n_1,t_1 : (recv(n_1, politoAntispam, p, t_1) && t_1 < t_0 && p.emailFrom ==1))
	send(politoAntispam, n_0, p, t_0) -> p.proto == POP_REQ || p.protpo == POP_RESP
	send(politoAntispam, n_0, p, t_0) -> nodeHasAddr(politoAntispam,p.src)
PolitoCache	if(isInternalNode(a_0) == or(listadeinodiinterni) ? True : false
	isInCache(u_0,t_0), ∃ t_1, t_2, p_1, p_2, n_1,n_2 : (t_1< t_2 < t_0 && recv(n_1, politoCache, p_1, t_1) && recv(n_2, politoCache, p_2, t_2))) && p_1.proto == HTTP_REQ && p_2.proto == HTTP_RESP && isInternalNode(n_1) && !isInternalNode(n_2) && p_1.url == u_0 && p_2.url == u_0)
	send(politoCache, n_0, p, t_0) && !isInternalNode(n_0) -> (∃ t_1,n_1 : (t_1 < t_0 && recv(n_1, politoCache, p, t_1) && p.proto == HTTP_REQ && !isInCache(p.url,t_0))
	send(politoCache, n_0, p, t_0) && isInternalNode(n_0) -> (∃ p_1,t_1 : (t_1 < t_0 && recv(n_0, politoCache, p_1, t_1) && p_1.proto == HTTP_REQ && p.proto == HTTP_RESP && p_1.url == p.url && p.src == p_1.dest && p.dest==p_1.src && isInCache(p.url,t_0))
PolitoMailClient	send(politoMailClient, n_0, p, t_0) -> nodeHasAddr(politoMailClient,p.src)
	send(politoMailClient, n_0, p, t_0) -> p.origin == politoMailClient
	send(politoMailClient, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, politoMailClient, p, t_0) -> nodeHasAddr(politoMailClient,p.dest)
	send(politoMailClient, n_0, p, t_0) -> p.proto == POP3_REQ
	send(politoMailClient, n_0, p, t_0) -> p.dest == ip_mailServer
PolitoMailServer	send(politoMailServer, n_0, p, t_0) -> nodeHasAddr(politoMailServer,p.src)
	send(politoMailServer, n_0, p, t_0) -> p.origin == politoMailServer
	send(politoMailServer, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, politoMailServer, p, t_0) -> nodeHasAddr(politoMailServer,p.dest)
	send(politoMailServer, n_0, p, t_0) -> p.proto == POP3_RESP && p.emailFrom == 1
	send(politoMailServer, n_0, p, t_0) -> (∃ p_1,t_1 : (t_1 < t_0 && recv(n_0, politoMailServer, p_1, t_1) && p_0.proto == POP3_RESP && p_1.proto == POP3_REQ && p_0.dest == p_1.src)
PolitoErrFunction	send(politoErrFunction, n_0, p, t_0) -> 1 == 2
PolitoNat	send(nat, x, p_0, t_0) && !private_addr_func(p_0.dest) -> p_0.src == ip_politoNat && (∃ y, p_1 e t_1 | (recv(y, nat, p_1, t_1) && t_1 < t_0 && private_addr_func(p1.src) && p_1.origin == p_0.origin && <...stesso per dest,orig_body,body,seq,proto,emailFrom,url,options...>)
	send(nat, x, p_0, t_0) && private_addr_func(p_0.dest) -> !private_addr_func(p_0.src) && (∃ y, p_1 e t_1 | (recv(y, nat, p_1, t_1) && t_1 < t_0 && !private_addr_func(p1.src) && p_1.dest == ip_politoNat && p_1.origin == p_0.origin && <...stesso per src,orig_body,body,seq,proto,emailFrom,url,options...>
PolitoNat -setInternalAddress	private_addr_func(n_0) == or(n_0==n foreach internal address)
PolitoNF	myFunction(a_0,a_1) == ((a_0 == ipA && a_1 == ipB) || (a_0 == ipB && a_1 == ipA))
	send(politoNF, n_0, p, t_0) -> (∃ n_1,t_1 : (t_1 < t_0 && recv(n_1, politoNF, p, t_1) && myFunction(p.src,p.dest))
PolitoWebClient	send(politoWebClient, n_0, p, t_0) -> nodeHasAddr(politoWebClient,p.src)
	send(politoWebClient, n_0, p, t_0) -> p.origin == politoWebClient
	send(politoWebClient, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, politoWebClient, p, t_0) -> nodeHasAddr(politoWebClient,p.dest)
	send(politoWebClient, n_0, p, t_0) -> p.proto == HTTP_REQ
	send(politoWebClient, n_0, p, t_0) -> p.dest == ipServer
PolitoWebServer	send(politoWebServer, n_0, p, t_0) -> nodeHasAddr(politoWebServer,p.src)
	send(politoWebServer, n_0, p, t_0) -> p.origin == politoWebServer
	send(politoWebServer, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, politoWebServer, p, t_0) -> nodeHasAddr(politoWebServer,p.dest)
	send(politoWebServer, n_0, p, t_0) -> (∃ p_1,t_1 : (t_1 < t_0 && recv(n_0, politoWebServer, p_1, t_1) && p_0.proto == HTTP_RESP && p_1.proto == HTTP_REQ && p_0.dest == p_1.src && p_0.src == p_1.dest && p_0.url == p_1.url)

Core functions' classes

Class - method	Constraints explaination
NetContext	send(n_0, n_1, p_0, t_0) -> n_0 != n_1
	recv(n_0, n_1, p_0, t_0) -> n_0 != n_1
	send(n_0, n_1, p, t_0) -> p.src != p.dest
	recv(n_0, n_1, p, t_0) -> p.src != p.dest
	recv(n_0, n_1, p, t_0) -> send(n_0, n_1, p, t_1) && t_1 < t_0
	send(n_0, n_1, p, t_0) -> p.src_port > 0 && p.dest_port < MAX_PORT
	recv(n_0, n_1, p, t_0) -> p.src_port > 0 && p.dest_port < MAX_PORT
	recv(n_0, n_1, p, t_0) -> t_0 > 0
	send(n_0, n_1, p, t_0) -> t_0 > 0
Checker - IsolationProperty	recv(n_0,destNode,p0,t_0)
	send(srcNode,n_1,p1,t_1)
	nodeHasAddr(srcNode,p1.srcAddr)
	p1.origin == srcNode
	nodeHasAddr(destNode,p1.destAddr)
	p1.origin == p0.origin
	nodeHasAddr(destNode, p0.destAddr)
Checker - IsolationFlowProperty	recv(n_0,destNode,p,t_0)
	send(srcNode,n_1,p1,t_1)
	nodeHasAddr(srcNode,p.srcAddr)
	p.origin == srcNode
	(∄ p_2, n_2, t_2 : (send(destNode,n_2,p_2,t_2) && p_2.srcAddr == p. destAddr && p_2.srcPort == p.destPort && p_2.destPort == p.srcPort && p_2.destt == p.src && t_2 < t_0)
Checker - NodeTraversalProperty	recv(n_0,destNode,p,t_0)
	send(srcNode,n_1,p1,t_1)
	nodeHasAddr(srcNode,p.srcAddr)
	p.origin == srcNode
	(∄ n_2, t_2 : (recv(n_2,node,p,t_2) && t_2 < t_0)
	(∄ n_2, t_2 : (send(node,n_2,p,t_2) && t_2 < t_0)
Checker - LinkTraversalProperty	recv(n_0,destNode,p,t_0)
	send(srcNode,n_1,p1,t_1)
	nodeHasAddr(srcNode,p.srcAddr)
	p.origin == srcNode
	∃ t_1, t_2 : send(linkNode0,linkNode1,p,t_1) && recv(linkNode0,linkNode1,p,t_2) && t_1 < t_0 && t_2 < t_0
Checker - DataIsolationPropertyCore	recv(n_0,destNode,p,t)
	p.origin == srcNode
	for each constraint(n -> constraint)
Checker - DataIsolationProperty	recv(n_0,destNode,p,t)
	p.origin == srcNode
Network -setAddrMapping	addrToNode(addr) = node
	nodeHasAddr(node, a_0) == (a_0 = addr)
Network -setAddrMappingMulti	addrToNode(foreach ad in addr) = node
	nodeHasAddr(node, a_0) == Or(foreach ad in addr (a_0 == ad))
Network - saneSend	send(node, n_0, p, t_0) -> not(nodeHasAddr(node, p.dest))
Network - setGateway	send(node, n_0, p_0, t_0) -> n_0 = gateway
Network - compositionPolicy	foreach rtAddr,rtNode in rt(send(node, n_0, p_0, t_0) && Or(foreach rtAddr in rt destAddrPredicate(p_0,rtAddr)) -> n_0 == rtNode)
Network - compositionPolicyShunt	foreach rtAddr,rtNode in rt(send(node, n_0, p_0, t_0) && Or(foreach rtAddr in rt destAddrPredicate(p_0,rtAddr)) -> n_0 == rtNode)

Chapter 6

Project development steps

In this chapter I would like to explain the phases that I have followed from the start to the end of the work, leaving some technical details to the next paragraph.

The first step is obviously a study phase. I have studied the previous paper of Berkeley and the subsequent study of Politecnico that represent the basis of this work (the two article explained in the 2nd paragraph), in order to understand the meaning and the purposes of this thesis project.
Then I have analysed the previous project (explained in the 5th paragraph) trying to understand its structure and how the classes and the tests work.
After this, I have studied the theory of First Order Logic and I have tried to understand the meaning of the formulas and how they can describe the related VNF.

After this first studying step, my practical work began.
One of the main core of the project is the **definition of a library** that allows a generic user to write its version of a virtual network function.
Firstly I have rewritten some classes that implement the VNF and some other in the mcnet.components package in order to take most advantages from the inheritance and leaving in the classes only the distinguishing features.
Then, looking the definition of the VNF in terms of formulas, I have searched some generic features that allow the description of a generic behaviour of a network function.
I have found this characteristics:

1) A network function may be an end host or a forwarding host.

   An end host represents a terminal node and the packet that it receives have its address as destination. It could send packet in response to a received one (e.g. a response for a request) or firstly a packet (e.g. a request).

   Example of end host node are: mail client, web client, mail server, ….

   A forwarding host mostly represents not a terminal node but a linking one. The packet are not intended to this host but they transit into it for being analysed or modified. The

sending packet are a response for the received ones and they correspond to an elaboration or a simple forwarding.

Example of forwarding hosts are: firewall, antispam, intrusion detection system, NAT, web cache, …

2) A generic host have its own address and could have an internal state and some specific properties.

   The internal state can store something has happened before (related to sending and the received packets) and could modify the forwarding behaviour. This state can contain some static or dynamic data.

   A property could be a comparison element or a data for modifying the forwarding packet.
3) An host could be linked to an internal network, which nodes could have public or private addresses.
4) The action that an host could take on a packet could be drop or forward, and the forwarding direction could be the next hop or towards the source of the packet.
5) The characteristics of a packet that could be analysed are: source address, destination address, source port, destination port, protocol used, url, mail source, mail destination, body, original body and origin of the packet.

After the definition of these features for a generic network function, I have created some Java classes that represent the library that the programmer have to use in order to define his own network function.

The library is contained into the project *nf-modelling-master* inside the package *definitions*.

The main class is *Host.java*, an abstract class that defines the basis features of a network function.

It has three fields: *state*, *address* and *hostPropertyList*.

The state contains four list: *receivedPackets* (it store all the packet received from the network function), *sentPackets* (that store all the packet sent), *internalNodes* (it contains a list of the address of the nodes that belong to the internal network linked to the network function) and *hostTableList*.

The last one is the most interesting, it contains a list of *HostTable*.

An *HostTable* is a particular object offered to the programmer to store into the network function what he wants. It is a table with a variable number of columns (set by the programmer) that can

store one or more features of a packet. The user can store data inside it and use it in order to modify or take decision about a packet.

The field *address* represents the own address of the network function.

Lastly the field *hostPropertyList* represents a way to configure the VNF: it contains hostProperty, particular data to modify the behaviour of the network function or to set some parameters (like a public IP).

The class *Host.java* also contains two methods: *defineState* and *onReceivedPacket*.

The first method is used by the programmer to define the internal state: using it the programmer could define which characteristics of the state the network function uses, or if it needs some configuration properties.

The 2nd method define the behaviour of the VNF in response to a received packet.

```
1  package definitions;
2
3
4  public abstract class Host {
5      public State state;
6      public Address address;
7      public HostPropertyList hostPropertyList;
8
9      public abstract void defineState();
10
11     public abstract RoutingResult onReceivedPacket(Packet packet);
12 }
```

Host.java

From *Host.java* two other abstract classes derive: *EndHost.java* and *ForwardingHost.java*.

These are the two classes that the user must implements in order to define his own network function.

The class *ForwardingHost.java* has exactly the same fields and methods of *Host.java*, while *EndHost.java* adds the method *defineSendingPacket*: it is used to define the characteristics of a packet sent by the network function.

An other important class of the library is *Packet.java*, that obviously defines the structure of a packet.

The fields correspond to the features that can be analysed for a packet: *ip_src, ip_dest, port_source, port_dest, protocol, url, mail_from, mail_to, body, orig_body, origin*.

There are some methods to set and to get all this fields and other two static methods *match* that allow the comparison a field with an other field or with a constant value.

The library has been developed following three objectives:

i. To allow a simple definition of the network function for the user

ii. To leave some general concepts and freedom for the programmer in such a way that he could define the behaviour that he wants

iii. To can find an automatic translation from the Java definition into FOL formulas

The library also contains some other classes but here only the most significant are reported.
Its usage is well explained in Guide for programmers, that is also included in this thesis report in the 9th chapter.

```
1  package test.definition;
2
3  import definitions.Action;
12
13 public class WebCache extends ForwardingHost{
14
15     @Override
16     public void defineState() {
17         this.state.hostTableList.add(HostTable.createTable("ResorceCache",1, FieldType.URL));
18         this.state.internalNodes.setWithoutPrivateAddresses();
19         this.state.receivedPackets.set();
20     }
21
22     @Override
23     public RoutingResult onReceivedPacket(Packet packet) {
24         if(Packet.match(packet.getProtocol(), Constants.HTTP_REQUEST_PROTOCOL, Operator.EQUAL) &&
25                 this.state.hostTableList.get("ResorceCache").contains(packet.getUrl()) &&
26                 this.state.internalNodes.contains(packet.getSourceAddress())
27                 ) {
28             Packet p1 = new Packet();
29             p1.setSourceAddress(packet.getDestinationAddress());
30             p1.setDestinationAddress(packet.getSourceAddress());
31             p1.setUrl(packet.getUrl());
32             p1.setProtocol(Constants.HTTP_RESPONSE_PROTOCOL);
33             return new RoutingResult(p1,Action.FORWARD,ForwardDirection.SAME_INTERFACE);
34         }
35         if(Packet.match(packet.getProtocol(), Constants.HTTP_REQUEST_PROTOCOL, Operator.EQUAL) &&
36                 this.state.internalNodes.contains(packet.getSourceAddress()) &&
37                 !this.state.hostTableList.get("ResorceCache").contains(packet.getUrl()) &&
38                 !this.state.internalNodes.contains(packet.getDestinationAddress())
39                 ) {
40             return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);
41         }
42         Packet p2 = new Packet();
43         if(Packet.match(packet.getProtocol(), Constants.HTTP_RESPONSE_PROTOCOL, Operator.EQUAL) &&
44                 !this.state.internalNodes.contains(packet.getSourceAddress()) &&
45                 this.state.receivedPackets.contains(p2) &&
46                 Packet.match(p2.getProtocol(), Constants.HTTP_REQUEST_PROTOCOL, Operator.EQUAL) &&
47                 this.state.internalNodes.contains(p2.getSourceAddress()) &&
48                 Packet.match(p2.getUrl(), packet.getUrl(), Operator.EQUAL)
49                 ) {
50             this.state.hostTableList.get("ResorceCache").store(p2.getUrl());
51             return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);
52         }
53         return new RoutingResult(packet,Action.DROP,ForwardDirection.UPSTREAM);
54     }
55
56 }
```

```
1  package test.definition;
2
3  import definitions.Action;
12
13 public class Antispam extends ForwardingHost{
14
15     @Override
16     public void defineState() {
17         this.state.hostTableList.add(HostTable.createTable("Blacklist",1, FieldType.MAIL_FROM));
18
19     }
20
21     @Override
22     public RoutingResult onReceivedPacket(Packet packet) {
23         if(Packet.match(packet.getProtocol(), Constants.POP3_REQUEST_PROTOCOL, Operator.EQUAL)) {
24             return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);
25         }
26         if(Packet.match(packet.getProtocol(), Constants.POP3_RESPONSE_PROTOCOL, Operator.EQUAL) &&
27                 !this.state.hostTableList.get("Blacklist").contains(packet.getMailSource())
28                 ) {
29             return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);
30         }
31         return new RoutingResult(packet,Action.DROP,ForwardDirection.UPSTREAM);
32     }
33
34 }
35
```

The previous two images are examples of network functions written with the defined library.

After the rewrite of all the VNFs defined in the previous project (the classes that contain the new VNF definitions are inside the package *test.definition*), the next step has been the **research of a translating pattern** that allows the automatic translation from the Java classes to FOL formulas definition.

The concepts and the procedure used for this translation are explained in the next chapter.

Using these concepts throw a parser I can extract all the necessary information from the definition of the VNF (**parsing step**).

All the classes used for the parsing are inside the package *it.polito.parser*.

The main class is obviously *Parser.java*, and it use other classes to explore the definition files (the *…Visitor.java* ending classes) and some classes that represent particular information inside the FOL formulas (the *…Condition.java* ending classes).

The visitor classes are:

1) *ClassVisitor.java*, it is called directly from the parser. It starts the analysis of the methods defined in the VNF file
2) *DefineStateVisitor.java*, it is called from the ClassVisitor and it inspects a statement of the method *definestate*

3) *PacketDefinitionVisitor.java*, it is called from the ClassVisitor and it inspects a statement of the method *defineSendingPacket*
4) *ReceivedPacketVisitor.java*, it is called from the ClassVisitor and it starts the inspection of the statement inside the method *onReceivedPacket*
5) *IfConditionVisitor.java*, it is called from the ReceivedPacketVisitor and it inspects the if part of the if-blocks inside the method *onReceivedPacket*
6) *ThenConditionVisitor.java*, it is called from the ReceivedPacketVisitor and it inspects the then part of the if-blocks inside the method *onReceivedPacket*
7) *ConditionVisitor.java*, it is called from the IfConditionVisitor and it inspects a condition inside the if part of an if-block
8) *ThenStatementVisitor.java*, it is called from the ThenConditionVisitor and it inspects a statement inside the then part of an if-block (except for return statement)
9) *ReturnStatementVisitor.java*, it is called from the ThenConditionVisitor and it inspects the return statement inside the then part of an if-block

In order to store all the information about the VNF, I have created a class called *NFdefinition.java*. This class has a lot of fields, every one of them used by a different visitor class.
In this paragraph I explain only what is the use of these fields, the procedures to store the data into them will are explained in the next paragraph.

Fields of NFdefinition class:

1) name, it is a string and it contains the name of the network function
2) isEndHost, it is a boolean variable and if it is true indicates that the VNF is an end host
3) internalNodesPresence, it is a boolean variable and if it is true indicates that the node is linked to a private network
4) propertyList, it contains all the properties defined by the programmer
5) hostTableList, it is an hash table that has as key the name of a table defined by the programmer and as value the list of the data type that the table stores
6) hostTableStatic, it is an hash table that has as key the name of a table defined by the programmer and as value a boolean that indicates if the table is static or not (if it is not static there must be a store instruction for it)

7) implication, it is an arrayList of implication elements that are created from the *onReceivedPacket* method
8) packetDefinition, it is an arrayList of PacketDefinition elements that are created from the *defineSendingPacket* method
9) variableList, it is an hashmap that contains all the variable defined in the VNF file
10) rightFormImplication, it is an arrayList and it derives from implication and packetDefinition. The data inside them are manipulated in order to create the FOL formulas stored inside a RightFormImplication object
11) protocolHM, it contains all the protocol used by the VNF for the sending packets
12) privateAddresses, it is a boolean variable that indicates if the nodes inside the private network have private addresses or not

RightFormImplication.java is an other important class in the parsing process. It contains two arrayList of condition objects that represent the form of a FOL implication: the first arrayList contains all the conditions that are before the sign of implication, while the second all the implications after the sign.

To store the information of the differents conditions that may be inside an implication of a VNF, some specific java classes are created. They are: *BodyCondition.java, DestPortCondition.java, EmailFromCondition.java, EmailToCondition.java, InternalNodeCondition.java* (the address stored inside it belongs to the internal network), *IsInTableCondition.java* (the field stored inside is present in the table), *NodeAddressCondition.java* (it represents a condition on the address of the current node), *OriginCondition.java, ProtocolCondition.java, RecvCondition.java* (it represents a receiving packet action), *ReturnCondition.java* (it represents the action that derives from a receiving packet action), *SendCondition.java* (it represents a sending packet action), *SourceCondition.java, SourcePortCondition.java, StoreTableCondition.java* (it represents a storing action inside the table), *UrlCondition.java*.

All these classes are subclasses of *Condition.java*. Almost all of them store a particular feature of a packet and the name is quite self-explicative.

After the extraction of all the information from the VNF source file and the creation of a mine own representation, the next step has been the **creation of the VNF java class that supply the input to Z3**.

It is done through the class *CreateNFClass.java* (it is inside the package *it.polito.parser*).
This class is called by the Parser and takes all the information inside the NFdefinition object and creates a new file called *Name_of_the_VNF.java* inside the package *mcnet.netobjs.NF* of the project *j-verigraph-master*.

This file represents the final result of the process and its creation is the main objective of this thesis work.

Using the parser, all the network functions defined in the previous projects are recreated and some **tests** have been done in order to check that they represent the same information and have the same behaviour of the original classes.

In order to check some functionalities of the network functions created, some original classes have particular constraints that pilot the test results.
For example inside the test *PolitoAntispamTest* (that obviously tests the antispam behaviour):

1) the antispam blocks the packet with the field eMailFrom == 1 (this constraint is inserted through the implementation of the antispam class)
2) the mail server send all the packets with eMailFrom == 1 (this constraint is inserted inside the definition file of the mail server “MailServer.java”)

In such a way, testing the reachability from the mail server to the mail client (linked through the antispam), the result must be UNSAT.
But the condition n. 2 must not be included in a general representation of a mail server (it is inserted only to make the test), and in the class generated with the parser it is not present.
So, in order to run the test, an other class named *MailServerForTest* is created.

```
1  package mcnet.netobjs.NF;
2
3  import com.microsoft.z3.BoolExpr;
7
8  public class MailServerForTest extends MailServer{
9
10     public MailServerForTest(Context ctx, Object[] objects) {
11         super(ctx, objects);
12     }
13
14     public void addConstraintForTest() {
15         IntExpr t_2 = ctx.mkIntConst(node+"_MailServer_t_2");
16         Expr p1 = ctx.mkConst(node+"_MailServer_p1",nctx.packet);
17         Expr n_0 = ctx.mkConst(node+"_MailServer_n_0",nctx.node);
18         constraints.add(ctx.mkForall(new Expr[]{t_2,p1,n_0},ctx.mkImplies(
19                 (BoolExpr)nctx.send.apply(node, n_0, p1, t_2),
20                         ctx.mkEq(nctx.pf.get("emailFrom").apply(p1), ctx.mkInt(1))),1,null,null,null,null));
21     }
22
23
24 }
```

MailServerForTest.java

This new class implements the *MailServer* class and it adds the constraint defined in the condition n. 2.

In such a way, *MailServer.java* file represents the general behaviour of the network function. It could be recreated from the source definition file (written with the library) using the parser without any changes in the test results.

All the most interesting tests and the obtained results will be explained in the 9th paragraph.

At the end, in order to automatically repeat the creation process and the tests, four ant files have been created:

1. *createNFClass.xml*, inside the *nf-modelling-master* project, it takes as input the name of the VNF to convert and create the converted file
2. *createAllNFClasses.xml*, inside the *nf-modelling-master* project, it converts all the VNF defined in the package *test.definition*
3. *runSingleTest.xml,* inside the *j-verigraph-master* project, it takes as input the name of the test to run
4. *runAllTests.xml,* it runs all the tests inside the packages *myTests* and *tests*

Chapter 7

The main ideas

The main problem after the definition of the library has been the research of a way to correctly translate the information inside the java file into a series of FOL formulas.

Analysing the formulas written to define the VNF in the previous projects, they mainly concern 5 different aspects:

1) definition of rules that regulates the sending of a packet in response to a received one
2) definition of rules that define the features of a sending packet (not linked to a received one, only for end host nodes)
3) definition of protocols that the VNF uses to send a packet
4) definition of rules that verify the presence of an element inside a list
5) definition of rules that store information

The library has been written after the study of these 5 different aspects, in such a way that a soft link is present since the writing phase of the network function behaviour.
But the way which these concepts are written in the Java file is totally different, so I had to define some patterns to do the translation.
After the extraction, all the information are stored in the *NFdefinition* object.

Firstly the parser extracts the name of the VNF (equal to the name of the class) and the type (end host or forwarding host, it depends from the class that is extended).
Then the method *definestate* is analysed. Here I could extract information about presence of private network (and if it is present, if the internal nodes have private addresses), the properties defined and the tables created by the user (and initialised as static tables).

Then the method *defineSendingPacket* is analysed (it is present only for end host node).
Firstly the name of the packet that must send is extracted, then all the instruction that define its features are converted into an internal data representation (it is a list of Condition elements).

Using them the FOL formulas that regulate the features of the sending packet could be created (this packet is not linked to an input one).

The way which it is done is quite simple. Two implications are necessary:

1) the first is directly connected with the sending packet and has the form
   send(packet to n_0) -> condition_on_packet && condition_on_packet && …
2) the second is implicit for end host and regulates the fact that if an end host receives a packet, it is the destination of that packet. So the implication has the form
   recv(packet from n_0) -> packet_destination == this_node

These two implications here are expressed in a very user friendly way (in order to understand them), inside the *NFdefinition* object they are two *PacketDefinition* elements.

Then the parser analyses the *onReceivePacket* method. It represents the real challenge in the parsing phase.

The information inside this method is represented as a collection of if-then statements.

The structure of the data inside an if-then statement is:

```
if (
    condition about: received packet, other packets previous sent or received,
                     the state of the node &&
    condition about: received packet, other packets previous sent or received,
                     the state of the node &&
     ……
)
{
      condition about the sending packet or storing instruction &&
      condition about the sending packet or storing instruction &&
      ……..
      eventually others if-then statements
      …….
      condition about the sending packet or storing instruction &&
      condition about the sending packet or storing instruction &&
      ……
      return statement
}
```

A condition about the received packet defines a property of one of its features: it may be a comparison with a fixed value (like a protocol), with a value inside the propertyList or a value of

previous packet received or sent (and the same rules are valid for a condition about previous packet received or sent).

A condition about the state of the node may be:

1) the presence of a packet into the list of the received or sent packets
2) the presence of a value inside an hostTable (table defined by the user)
3) the presence of a source or destination address of a packet inside the internal network

A condition about the sending packet sets the value of one of its features. The source value may be a constant one (like a protocol), a value extracted from a previous packet received or sent, a value inside the propertyList or a value extracted from an hostTable.
The storing instruction represents an insertion of a value that derives from a packet inside an hostTable.
The return statement represents the action (forward or drop, and in case of forward, the forwarding direction).

The general meaning of an if-then statement is: if the condition inside the if part are satisfied, the features of a sending packet are set, the storing instructions are executed and the action inside the return is performed.
But if there are some inning if- then block, to perform the action, all the condition of the previous if parts must be satisfied, all the condition on the sending packet are set and all the storing information are performed.
The following example may clarify the behaviour:

```
If (condition_A && condition_B) {
      Set_1_on_packet_A
      Set_2_on_packet_A
      If(condition_C) {
            Set_3_on_packet_A
            Storing_instruction_1
            Action_forward_packet_A
      }
      Set_4_on_packet_A
      Action_forward_packet_A
}
```

In the previous case:

- if the conditions A,B and C are all satisfied, the packet is set with the values 1,2 and 3, the storing instruction 1 is performed and the packet is sent.
- if the conditions A and B are satisfied but not C, the packet is set with the values 1,2 and 4 and the packet is sent
- if the conditions A or B are not satisfied, the packet is not sent

From every if-then statement an *IfBlockConditions* element is created and stored into the *NFdefinition* object.

Using the *PacketDefinition* and *IfBlockConditions* elements, the implications are created and stored into *RightFormImplication* objects.

To create *RightFormImplication* objects from *PacketDefinition* elements, the procedure explained previously is used. So two implication are defined:

1) The first has a SendCondition object before the implication sign, and all the conditions defined on the sending packet placed after the implication sign. The time which refer the send is t_0
2) The second has a RecvCondition object before the implication sign and a NodeAddressCondition object after the sign (it is standard for all the end host network functions)

To create *RightFormImplication* objects from *IfBlockConditions* element the pattern is more complicated and the correct translation represents one of the key challenge of this thesis work.
From every IfBlockConditions an other middle object is created, an Implication object that has three field (derived from the structure of the if statements):

1. ifConditions -> It is an arrayList of conditions that regard the received packet, other packet received or sent previously or the internal state of the node
2. thenConditions -> It is an arrayList of conditions that regard the sending packet and the storing instructions
3. result -> It is a condition that contains the action to perform

To create an implication, the following rules are used:

a. only the forward action could create an implication
b. before the implication sign must be a send condition for the sending packet and all the conditions that regard only the sending packet may be present (but an other criteria must be added). The send condition is considered at time t_2
c. after the implication sign must be a recv condition for the received packet and it is considered at time t_1 < t_2. The source of the received packet is equal to the destination of the sending packet if the forwarding direction is *SAME_INTERFACE*.
d. after the implication sign all the conditions on the received packet and other packets previously received or sent are placed
e. for every packet previously sent or received a send or recv condition must be present after the implication sign. These conditions are considered at time t_0 < t_1
f. after the implication sign all the conditions that regard the state of the node are placed. They are considered at time t_1
g. for every store condition, an other implication must be created. It has the store instruction before the implication sign and all the other conditions after the sign

These rules give the basic structure of the implications, but other actions must be performed in order to obtain a correct definition of the network function.

Firstly all the conditions for the sending packet that could be derived from condition on other packets are extracted and inserted into the implication.

For example if there are in the same implication the following conditions:

private_address(packet_1_source)

packet_2_destination == packet_1_source

send(packet_2)

The condition that could be extracted is:

private_address(packet_2_destination)

Then two tricky aspects are considered.

The first one regards the number of conditions to include before the implication sign. Only the send condition does not permit to distinguish all the sending cases, so other conditions are necessary and could be chosen from the conditions that regards only the sending packet.

On the other hand, not all these conditions could be considered in order to not leave degrees of freedom that are not necessary to the solver (they may alter the test result).
So the number of conditions (except for send condition) to insert before the implication sign must be $log_2 n$, where n is the number of the implication for sending packet (not for received one or for store rules).

The second one is the possibility that two derived implication have the same conditions before the implication sign, but one of the conditions after the implication sign is in contrast.
In such a way the two implications, if the part before the sign are all satisfied, could not be true at the same time and the result is always UNSAT.
To solve this problem, all the created implications are analysed in order to find this situation.
The checker compares all the implications only for the part before the implication sign searching two of them that are equals.
If there are two implications with this characteristic, it compares the part after the implication sign.
If all the conditions are equal except for one that is opposite, the checker puts this condition from the second implication to the first one and adds a mark with the condition which it is in OR (putting these two conditions in OR, the implication could be true if one of them is satisfied).
After the process the second implication is deleted.

During the whole process of the conversion of the *IfBlockCondition*, other features of the VNF are extracted in order to complete its definition:

1) the protocols used for the sending packets
2) the behaviour of the hostTable defined (they are initialized as static, but in presence of a store condition on them, they become dynamic)

The whole conversion process is quite difficult and it has requested a lot of time to be perfected.
At the end its result is the filling of all the fields inside the *NFdefinition* object created for the network function.

In the next two pages there is an example of VNF converted into this kind of representation.

```
Nome classe: WebCache
End Host: false
Presente internal nodes: true
Presenti indirizzi privati: true
Property list: []
Host table list:
{ResorceCache=[FieldType.URL]}
Host table list static:
{ResorceCache=false}



Before implication condition
---------------------------------
send
SourceNode: WebCache
DestNode: n_0
Packet: p1
Time: t_2
Validity: true
---------------------------------
internal_node
Node: n_0
Validity: true
---------------------------------
After implication condition
---------------------------------
recv
SourceNode: n_0
DestNode: WebCache
Time: t_1
Packet: packet
Validity: true
---------------------------------
packet_protocol
Packet: packet
Protocol: Constants.HTTP_REQUEST_PROTOCOL
Validity: true
---------------------------------
is_in_table
TableName: "ResorceCache"
Fields: [packet.getUrl]
Validity: true
---------------------------------
internal_node
Node: n_0
Validity: true
---------------------------------
packet_source
Packet: p1
Source: packet.getDestinationAddress
Validity: true
---------------------------------
packet_destination
Packet: p1
Destination: packet.getSourceAddress
Validity: true
---------------------------------
packet_url
Packet: p1
Url: packet.getUrl
Validity: true
---------------------------------
packet_protocol
Packet: p1
Protocol:
Constants.HTTP_RESPONSE_PROTOCOL
Validity: true
---------------------------------
packet_protocol
Packet: p1
Protocol:
Constants.HTTP_RESPONSE_PROTOCOL
Validity: true
---------------------------------


Before implication condition
---------------------------------
send
SourceNode: WebCache
DestNode: n_0
Packet: packet
Time: t_2
Validity: true
---------------------------------
internal_node
Node: n_source
Validity: true
---------------------------------
internal_node
Node: n_0
Validity: false
---------------------------------
After implication condition
---------------------------------
recv
SourceNode: n_source
DestNode: WebCache
Time: t_1
Packet: packet
Validity: true
---------------------------------
packet_protocol
Packet: packet
Protocol: Constants.HTTP_REQUEST_PROTOCOL
Validity: true
---------------------------------
internal_node
Node: n_source
Validity: true
---------------------------------
is_in_table
TableName: "ResorceCache"
Fields: [packet.getUrl]
Validity: false
---------------------------------
internal_node
Node: n_0
Validity: false
---------------------------------
packet_protocol
Packet: packet
Protocol: Constants.HTTP_REQUEST_PROTOCOL
Validity: true
---------------------------------


Before implication condition
---------------------------------
store_in_table
TableName: "ResorceCache"
Fields: [p2.getUrl]
Validity: true
---------------------------------
After implication condition
---------------------------------
recv
SourceNode: n_source
```

```
DestNode: WebCache
Time: t_1
Packet: packet
Validity: true
---------------------------------
packet_protocol
Packet: packet
Protocol:
Constants.HTTP_RESPONSE_PROTOCOL
Validity: true
---------------------------------
internal_node
Node: n_source
Validity: false
---------------------------------
recv
SourceNode: n_1
DestNode: WebCache
Time: t_0
Packet: p2
Validity: true
---------------------------------
packet_protocol
Packet: p2
Protocol: Constants.HTTP_REQUEST_PROTOCOL
Validity: true
---------------------------------
internal_node
Node: n_1
Validity: true
---------------------------------
packet_url
Packet: p2
Url: packet.getUrl
Validity: true
---------------------------------
```

As defined in the previous chapter, the following step is the definition of a Java file that represents the same data into a form that could be supplied to Z3.

Even if the conversion process is not so easy and however it represents a key part of the thesis, it results quite mechanic and so here it is not explained.

WebCache.java

```
1 package mcnet.netobjs.NF;
2 import java.util.*;
3 import com.microsoft.z3.*;
4 import mcnet.components.*;
5 public class WebCache extends NetworkObject{
6     List<DatatypeExpr> private_addresses;
7     FuncDecl private_addr_func;
8     public WebCache(Context ctx, Object[]... args) {
9         super(ctx, args);
10    }
11    @Override
12    protected void init(Context ctx, Object[]... args) {
13        super.init(ctx, args);
14        isEndHost=false;
15        private_addresses = new ArrayList<DatatypeExpr>();
16    }
17    private void addPrivateAdd(List<DatatypeExpr> address){
18        private_addresses.addAll(address);
19    }
20    public List<DatatypeExpr> getPrivateAddress(){
21        return private_addresses;
22    }
23    public void setInternalAddress(ArrayList<NetworkObject> internalAddress){
24        List<BoolExpr> constr = new ArrayList<BoolExpr>();
25        Expr n_0 = ctx.mkConst("WebCache_node", nctx.node);
26        for(NetworkObject n : internalAddress){
27            constr.add(ctx.mkEq(n_0,n.getZ3Node()));
28        }
29        BoolExpr[] constrs = new BoolExpr[constr.size()];
30        constraints.add(ctx.mkForall(new Expr[]{n_0},
  ctx.mkEq(private_addr_func.apply(n_0),ctx.mkOr(constr.toArray(constrs))),1,null,null,
  null,null));
31    }
32    public void WebCache_install(){
33        private_addr_func = ctx.mkFuncDecl("private_addr_func", nctx.node,
  ctx.mkBoolSort());
34        FuncDecl ResorceCache_func = ctx.mkFuncDecl(node+"_ResorceCache_func",new
  Sort[]{ctx.mkIntSort(),ctx.mkIntSort()},ctx.mkBoolSort());
35        IntExpr t_0 = ctx.mkIntConst(node+"_WebCache_t_0");
36        IntExpr t_1 = ctx.mkIntConst(node+"_WebCache_t_1");
37        IntExpr t_2 = ctx.mkIntConst(node+"_WebCache_t_2");
38        Expr p1 = ctx.mkConst(node+"_WebCache_p1",nctx.packet);
39        Expr n_1 = ctx.mkConst(node+"_WebCache_n_1",nctx.node);
40        Expr p2 = ctx.mkConst(node+"_WebCache_p2",nctx.packet);
41        IntExpr f_0 = ctx.mkIntConst(node+"_WebCache_f_0");
42        Expr packet = ctx.mkConst(node+"_WebCache_packet",nctx.packet);
43        Expr n_source = ctx.mkConst(node+"_WebCache_n_source",nctx.node);
44        Expr n_0 = ctx.mkConst(node+"_WebCache_n_0",nctx.node);
45        constraints.add(ctx.mkForall(new Expr[]{t_2,p1,n_0},ctx.mkImplies(ctx.mkAnd(
46                (BoolExpr)nctx.send.apply(node, n_0, p1, t_2),
47                ((BoolExpr)private_addr_func.apply(n_0))),
48                ctx.mkExists(new Expr[]{t_1,packet},ctx.mkAnd(
49                        ctx.mkLt(t_1, t_2),
50                        (BoolExpr)nctx.recv.apply(n_0, node, packet, t_1),
51                        ctx.mkEq(nctx.pf.get("proto").apply(packet),
  ctx.mkInt(nctx.HTTP_REQUEST)),
52                        (BoolExpr)ResorceCache_func.apply(nctx.pf.get("url").apply(p
  acket),t_1),
53                        ((BoolExpr)private_addr_func.apply(n_0)),
54                        ctx.mkEq(nctx.pf.get("src").apply(p1),nctx.pf.get("dest").ap
```

WebCache.java

```
ply(packet)),
55                          ctx.mkEq(nctx.pf.get("dest").apply(p1),nctx.pf.get("src").ap
ply(packet)),
56                          ctx.mkEq(nctx.pf.get("url").apply(p1),nctx.pf.get("url").app
ly(packet)),
57                          ctx.mkEq(nctx.pf.get("proto").apply(p1),
ctx.mkInt(nctx.HTTP_RESPONSE)),
58                          ctx.mkEq(nctx.pf.get("proto").apply(p1),
ctx.mkInt(nctx.HTTP_RESPONSE))),1,null,null,null,null)),1,null,null,null,null));
59          constraints.add(ctx.mkForall(new
Expr[]{t_2,packet,n_source,n_0},ctx.mkImplies(ctx.mkAnd(
60                  (BoolExpr)nctx.send.apply(node, n_0, packet, t_2),
61                  ((BoolExpr)private_addr_func.apply(n_source)),
62                  ctx.mkNot(((BoolExpr)private_addr_func.apply(n_0)))),
63                  ctx.mkExists(new Expr[]{t_1},ctx.mkAnd(
64                          ctx.mkLt(t_1, t_2),
65                          (BoolExpr)nctx.recv.apply(n_source, node, packet, t_1),
66                          ctx.mkEq(nctx.pf.get("proto").apply(packet),
ctx.mkInt(nctx.HTTP_REQUEST)),
67                          ((BoolExpr)private_addr_func.apply(n_source)),
68                          ctx.mkNot((BoolExpr)ResorceCache_func.apply(nctx.pf.get("url"
).apply(packet),t_1)),
69                          ctx.mkNot(((BoolExpr)private_addr_func.apply(n_0))),
70                          ctx.mkEq(nctx.pf.get("proto").apply(packet),
ctx.mkInt(nctx.HTTP_REQUEST))),1,null,null,null,null)),1,null,null,null,null));
71          constraints.add(ctx.mkForall(new Expr[]{t_2,f_0},ctx.mkImplies(
72                  (BoolExpr)ResorceCache_func.apply(f_0, t_2),
73                  ctx.mkExists(new Expr[]{t_1,t_0,n_1,p2,packet,n_source},ctx.mkAnd(
74                          ctx.mkLt(t_1, t_2),
75                          ctx.mkLt(t_0, t_1),
76                          ctx.mkEq(nctx.pf.get("url").apply(p2), f_0),
77                          (BoolExpr)nctx.recv.apply(n_source, node, packet, t_1),
78                          ctx.mkEq(nctx.pf.get("proto").apply(packet),
ctx.mkInt(nctx.HTTP_RESPONSE)),
79                          ctx.mkNot(((BoolExpr)private_addr_func.apply(n_source))),
80                          (BoolExpr)nctx.recv.apply(n_1, node, p2, t_0),
81                          ctx.mkEq(nctx.pf.get("proto").apply(p2),
ctx.mkInt(nctx.HTTP_REQUEST)),
82                          ((BoolExpr)private_addr_func.apply(n_1)),
83                          ctx.mkEq(nctx.pf.get("url").apply(p2),nctx.pf.get("url").app
ly(packet))),1,null,null,null,null)),1,null,null,null,null));
84          constraints.add(ctx.mkForall(new Expr[]{n_0, packet, t_2},ctx.mkImplies(
85                  (BoolExpr)nctx.send.apply(node, n_0, packet, t_2),ctx.mkOr(
86                          ctx.mkEq(nctx.pf.get("proto").apply(packet),
ctx.mkInt(nctx.HTTP_REQUEST))
87                          ,ctx.mkEq(nctx.pf.get("proto").apply(packet),
ctx.mkInt(nctx.HTTP_RESPONSE))
88                          )),1,null,null,null,null));
89      }
90 }
91
```

The previous image represents an example of the final result of the whole conversion process and its creation from the source Java file represents one of the main goal of this thesis work.

All the concepts given in this paragraph could only help to understand the main ideas that have allowed the development of this thesis work, but they are not necessary to use the software created.

All the necessary concepts are explained in a very more easily way into the Guide for programmers (included in the 9th chapter).

Chapter 8

Objectives reached and possible future developments

As explained before, the main goal of this thesis work is the creation of a framework that allows the user to define his own virtual network functions in a easy way and offer a tool for the verification of the service graph created.

The way that has been chosen to reach this goal, after the definition of the basic structure of the network, is essentially the creation of a library in Java that is used to define the VNF behaviour, the definition of a parser that creates the input file for Z3 from the user definition file and a set of tests that allow the verification of the reachability property.

The ways which the different functionalities have been developed are explained in the previous paragraphs. In this one firstly I would like to analyse the result of the work, analysing two main points: the easiness to create the VNF definition file and the comparison of the tests results between the old Z3 source classes (written ad hoc) and the automatically generated ones (created with the parser).

Regarding the first point, simply looking at the VNF definition files created, it is quite evident that the goal is reached: the classes looks so simples and the number of instructions necessary to define the behaviour is almost low.
Furthermore the generality of the instructions allows the user to define the behaviour of the VNF that he wants.
All the VNF previously defined in the other projects have been written in a easy way, and the creation of a Guide for programmers helps this creation process.

Regarding the second point, to prove the efficiency of the project, all the previous tests have been rewritten and their results have been compared with the previous ones obtained with the old classes (written ad hoc).

To make the comparison an Ant file has been created. Running it, the output shows firstly the result with the previous classes and after the result with the new ones for the same tests (and it is repeated for all the tests cases).

The image that follow shows its output.

```
run:
     [echo] Class: Polito_AntispamTest
     [echo] Result with the previous class
     [java] UNSAT
     [echo] Result with the new class
     [java] UNSAT
     [echo] Class: Polito_CacheFwTest
     [echo] Result with the previous class
     [java] UNSAT
     [echo] Result with the new class
     [java] UNSAT
     [echo] Class: Polito_CacheNatFwTest
     [echo] Result with the previous class
     [java] UNSAT
     [java] UNSAT
     [echo] Result with the new class
     [java] UNSAT
     [java] UNSAT
     [echo] Class: Polito_CacheTest
     [echo] Result with the previous class
     [java] SAT
     [echo] Result with the new class
     [java] SAT
     [echo] Class: Polito_DoubleFwTest
     [echo] Result with the previous class
     [java] UNSAT
     [java] UNSAT
     [echo] Result with the new class
     [java] UNSAT
     [java] SAT

     [echo] Class: Polito_IDSTest
     [echo] Result with the previous class
     [java] UNSAT
     [echo] Result with the new class
     [java] UNSAT
     [echo] Class: Polito_SimpleTest
     [echo] Result with the previous class
     [java] SAT
     [echo] Result with the new class
     [java] SAT
```

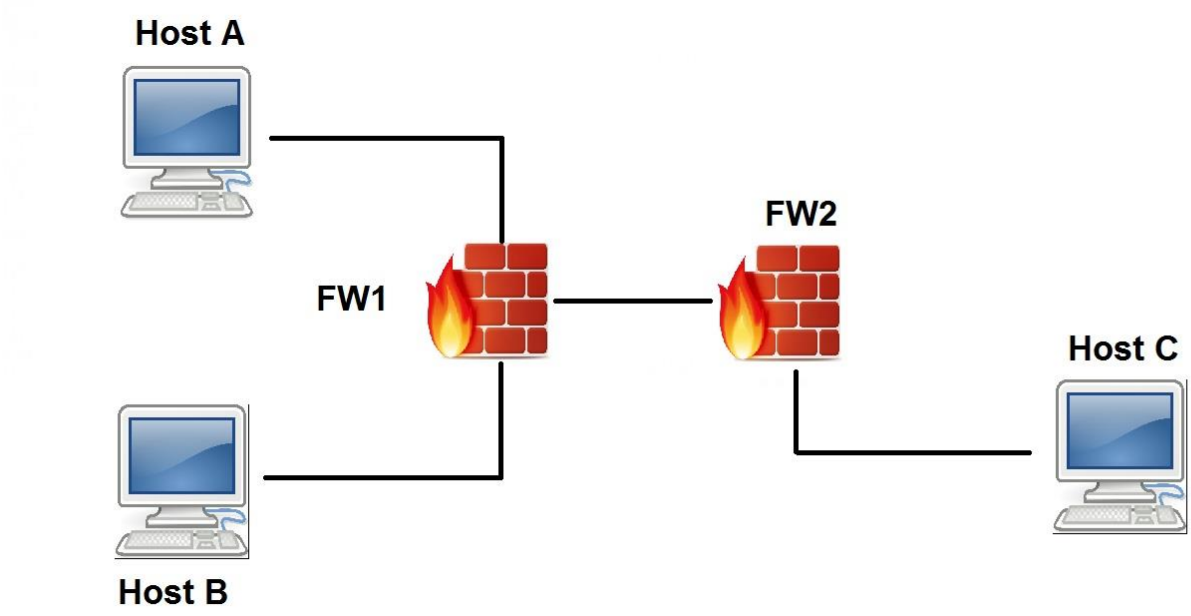
As it is evident looking the output, the new classes have the same behaviour of the old ones.

Only the results of Polito_DoubleFwTest are different.

To understand the reason of this difference, firstly the structure of its test must be explained.

The network of the Polito_DoubleFwTest contains 5 nodes:

- The nodes named A,B and C are EndHost
- The nodes named FW1 and FW2 are Firewall



Each one of the two firewall has a rule to deny traffic:

1. FW1 denies traffic between host A and host C
2. FW2 denies traffic between host B and host C

Inside the test, two properties are verified:

1. Reachability property from node A to node C
2. Reachability property from node A to node B

Considering the structure of the network, the rules inside the firewalls and the properties tested, the first result must be UNSAT, while the second result must be SAT.
Looking at the output of the test with the generated classes, there are exactly the same results (so the old classes do not give the correct results).

In conclusion, considering the results in the two key points that I have defined, I could say that the objectives of this thesis work have been reached and it has also allowed the identification of problems in the previous projects.

As regard the possible future developments, the sets of the virtual network functions and the test cases could be extended and it may represent the base of wider projects that really allow a transformation of the current networks.

Chapter 9

Guide for programmers

This chapter contains the guide realised in order to allow a programmer to use the library defined in a easy way and to write his own test cases.
It's a practical guide that shows the basic instructions and clarifies their meaning. It contains all the necessary information to understand the usage of this thesis project also for not technical users (obviously the knowledge of Java is requested).

Guide for programmers

The definition of the virtual network function must be written in a unique Java file that must include the package *definition*. So the first row after the definition of the package must be:

```
import definitions.*;
```

This guide is composed by 5 sections. Three sections explain 3 different aspects linked to the VNF creation, the 4th section contains some examples and the 5th one contains an explanation of the way to use the generated classes (with the related example).

1) **Class definition**

The class must not have a specific name but it must respect the java rules. It must extend one of the following two abstract classes:

1- **ForwardingHost**: extending this abstract class the defining VNF does not represent a terminal node of the network but normally a forwarding host. The packets must not be addressed to this host but they may transit inside it to be analysed or modified.
The sent packets represent a response to the packet received (and normally correspond to a their elaboration or a simply forwarding).
Example of network functions that are included in this category are: Firewall, Antispam, Intrusion Detection System, NAT, Web Cache, ...
So the class definition must be for example:

```
public class Antispam extends ForwardingHost
```

2- **EndHost**: in this case the network function represents a terminal node. The packets that arrive to this node have its address as destination.
It could send a packet in response to a received one (for example a response to a request) or it can send packet firstly (like a request).
Example of network functions that are included in this category are: Mail Client, Mail Server, Web Client, Web Server, …
So the class definition must be for example:

```
public class WebClient extends EndHost
```

2) <u>Class methods</u>

The class does not need a constructor and the behaviour of the network function must be described using specific methods.
Doing the automatic filling the methods to implement will be shown and they are all the ones necessary to define the class (do not define other methods that will be ignored).

According to the abstract class implemented, two or three of the following methods will be visualised:

- **defineState:** it is present either if an EndHost or a ForwardingHost is defined. The definition is:

```
@Override
public void defineState() {
```

This method defines the components from which the state of the network function and the setting parameters are extracted.

- **defineSendingPacket:** it is present only if an EndHost is defined. Its definition is:

```
@Override
public Packet defineSendingPacket() {
```

This method defines the characteristics of a packet sent by the current node (it does not represent a response to a received packet, usually it is a request packet).

- **onReceivedPacket:** it is present either if an EndHost or a ForwardingHost is defined. The definition is:

```
@Override
public RoutingResult onReceivedPacket(Packet packet) {
```

This method defines the behaviour of the network function in response to a received packet. The VNF could forward the same packet, send a new one (defining the characteristic, that could be linked to the ones of the received packet) or discard the packet.

3) Method filling

The instructions to insert into the methods depend from the method itself. Only the allowed instructions must be inserted (others instructions will not be considered).

- ***defineState method***

  As previously explained only the instructions that define the state components and the setting parameters could be inserted.

  To define the state, the instructions allowed are:

  - `this.state.internalNodes.setWithoutPrivateAddresses();`

    This instruction means that the network function will be linked to an own internal network which nodes will not have private addresses (the internal node list must be supplied at the moment in which the network will be used).

    If this instruction is used, after it will be possible to check an address to control if it belongs to the internal network (the specific instruction to perform this control will be shown inside the method in which it could be used).

  - `this.state.internalNodes.setWithPrivateAddresses();`

    This instruction means that the network function will be linked to an own internal network which nodes will have private addresses (the internal addresses list must be supplied at the moment in which the network will be used).

    If this instruction is used, after it will be possible to check an address to control if it belongs to the internal network (the specific instruction to perform this control will be shown inside the method which it could be used).

  - `this.state.sentPackets.set();`

    This instruction must be inserted when the network function must store the sending packets (into an internal memory).

    If this instruction is used, after it will be possible to check if a packet was previously sent or not (the specific instruction to perform this control will be shown inside the method which it could be used).

  - `this.state.receivedPackets.set();`

    This instruction must be inserted when the network function must store the received packets (into an internal memory).

If this instruction is used, after it will be possible to check if a packet was previously received or not (the specific instruction to perform this control will be shown inside the method in which it could be used).

- ```
  this.state.hostTableList.add(HostTable.createTable("Blacklist",1,
  FieldType.ORIGIN));
  ```
  This is an example of using the instruction necessary to create a table (that corresponds to an internal memory of the network function) that could store some packet fields.
  At the table definition moment the name must be defined (using a string or a variable that contains a string, in this case “Blacklist”), the column number and the fields that it will contain (choosing them into the enum FieldType).
  There is not a maximum number of tables that can be inserted, but everyone must have a different name.
  If this instruction is used, after it will be possible to check if a table contains or not one or more data (the specific instruction to perform this control will be shown inside the method in which it could be used).

In order to define the setting parameters of the network function the following instruction must be used:

```
this.hostPropertyList.add("ip_mailServer");
```

In this case the parameter is identified by the string “ip_mailServer” (a string or a variable containing a string must be inserted).
When the function will be implemented, it will need a value for this parameter that then will be used to perform controls or set some values (the specific instruction to extract the current value of the parameter will be shown inside the method in which it could be used).
The parameter (or property) could contain a field or a network node.
Many parameters could be inserted, but everyone must have a unique identifier.
```

- ***defineSendingPacket method***

  It contains the instructions that are necessary to define the sending packet.

  The first instruction that must be inserted is the one that creates a new packet:

```
Packet packet = new Packet();
```

  The name could be chosen by the user , but inside this method only one packet could be defined.

  The return value of this function must be the packet itself. In this case:

```
return packet;
```

  The packet definition instructions that could be inserted in this method are:

  - `packet.setSourceAddress(this.address);`
    It sets the source address of the packet equal to the address of the current node.
  - `packet.setOrigin(this.address);`
    It sets the origin of the packet equal to the address of the current node.
  - `packet.setBody(packet.getOrigBody());`
    It sets the body of the packet equal to the origin body of the packet itself.
  - `packet.setProtocol(Constants.POP3_REQUEST_PROTOCOL);`
    It sets the protocol of the packet equal to POP3_REQUEST. The protocols that could be inserted are inside the enum Constants.

  Furthermore, there is a list of setting instructions that involves the defined property that could be inserted in this method. The list includes:

  - packet.setOrigin((Address) (this.hostPropertyList.get("property_name")));
  - packet.setBody((Body) (this.hostPropertyList.get("property_name")));
  - packet.setDestinationAddress((Address) (this.hostPropertyList.get("property_name ")));
  - packet.setDestinationPort((Port) (this.hostPropertyList.get("property_name")));
  - packet.setMailDestination((Mail_Host) (this.hostPropertyList.get("property_name")));
  - packet.setMailSource((Mail_Host) (this.hostPropertyList.get("property_name")));

- packet.setOrigBody((Body) (this.hostPropertyList.get("property_name")));
- packet.setOrigin((Address) (this.hostPropertyList.get("property_name")));
- packet.setSourceAddress((Address) (this.hostPropertyList.get("property_name")));
- packet.setUrl((Url) (this.hostPropertyList.get("property_name")));

Each one of these instructions must be supported by the definition of the correct property and the meaning is equal to setting the current value of the property to the field.

- ***onReceivePacket method***

The parameter of this method is the packet that the network function receives (packet) and the return value is a RoutingResult, which constructor receives three parameters:

- The name of the packet that the network function sends as response to a received one.
- The action to perform on this packet (send or drop, one of the value inside the enum Action could be inserted)
- The forwarding direction (it determines if the packet will be send to the next hop or back to the previous node, one the value inside the enum ForwardDirection could be inserted)

An example of return instruction could be:

```
return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);
```

The actions that could be inserted inside this method could be divided in the following categories:

1. Instructions that check if a packet contains a value inside a field
2. Setting a value into a field of a packet
3. Instructions that check the state of the network function
4. Instructions that insert a value inside a table defined by the user inside the state
5. Instructions that define the action to perform on a packet (through RoutingResult)

Inside this method new packets (with the previous instruction) and some variables that contain string to insert inside the called method could be defined.

The structure that must be used is the conditional instruction if-then (without else).
Inside the if block the instruction 1 and 3 could be inserted.
Inside the then block the instruction 2, 4 and 5 could be inserted.
Every if-then instruction must contains a return instruction and the instructions inserted inside the if block must be put in AND (eventually OR must generate an other if-then block).

The following instructions represent the available ones and they are divided into different categories:

1 - Instructions that check if a packet contains a value inside a field

The basic instruction is Packet.match. It has two implementations:
- Packet.match(Field f1, Field f2, Operator op)
- Packet.match(Field f1, Constants c, Operator op)

The first parameter must be the field of the packet to analyse.
For example considering that a field of the packet "packet" has to be analyse, with the first implementation the instructions could contain:

- packet.getBody()
- packet.getDestinationAddress()
- packet.getDestinationPort()
- packet.getMailDestination()
- packet.getMailSource()
- packet.getOrigBody()
- packet.getOrigin()
- packet.getProtocol()
- packet.getSourceAddress()
- packet.getSourcePort()
- packet.getUrl()

Using the second implementation the instruction cuold contain:

- packet.getProtocol()

The second parameter for the second implementation must be a value of the enum Constants, while for the first implementation must contain a field for a comparison, that it could be a field of an other packet (the type must be equal), the value of a property (doing the casting to the right type) or the address of the current node (if the filed to analyse are origin, source address or destination address).

The value of the property could be obtained through the instruction:

```
this.hostPropertyList.get("property_name")
```

The third parameter must be Operator.EQUAL

Example of the instruction that could be obtained are:

```
Packet.match(packet.getProtocol(), Constants.HTTP_RESPONSE_PROTOCOL,
Operator.EQUAL)

Packet.match(p2.getUrl(), packet.getUrl(), Operator.EQUAL)

Packet.match(packet.getDestinationAddress(), this.address, Operator.EQUAL)

Packet.match(packet.getMailDestination(),
(Mail_Host)this.hostPropertyList.get("ip_mailServer"), Operator.EQUAL);
```

2 - Setting a value into a field of a packet

Considering that a value for a field of the packet “packet” has to be set, the available instructions are:

- `packet.setBody(body);`
- `packet.setDestinationAddress(adress);`
- `packet.setDestinationPort(port);`
- `packet.setMailDestination(mail_host);`

- `packet.setMailSource(mail_host);`
- `packet.setOrigBody(orig_body);`
- `packet.setOrigin(origin);`
- `packet.setProtocol(c);`
- `packet.setProtocol(protocol);`
- `packet.setSourceAddress(address);`
- `packet.setSourcePort(port);`
- `packet.setUrl(url);`

Except for the setProtocol instruction that accepts also a Constants, each one of these functions has a field of a packet as parameter.
The field that could be inserted could be a field of an other packet (using a get instruction shown before), this.address (only for the functions that accept an Address) or an element of the hostPropertyList.

Example of assignment instructions are:

```
packet.setSourceAddress(this.address);

p1.setProtocol(Constants.POP3_RESPONSE_PROTOCOL);

p1.setBody(p1.getOrigBody());

p1.setUrl(packet.getUrl());

p1.setUrl((Url)this.hostPropertyList.get("property_name"));
```

3 - Instructions that check the state of the network function

They are instructions that perform some controls about the internal network of the node, the tables containing the sent and received packets and the tables defined by the user.

The available instructions are:

- `this.state.internalNodes.contains(address)`

It checks if an address is present into the list of the internal nodes.

Usually it receives the source or destination address of a packet as parameter (extracted through the operations getSourceAddress or getDestinationAddress).

- `this.state.sentPackets.contains(packet)`

It checks if a packet is present into the list of the previous sent packets.

- `this.state.receivedPackets.contains(packet)`

It checks if a packet is present into the list of the previous received packets.

- `this.state.hostTableList.get("tableName").contains(field)`

It checks if the field indicated with "field" is present into the hostTable named tableName. The input parameter (field) is the field of a packet (it is extracted through the get instruction) and must have the type of the field indicated at the definition time of the table.

4 - Instructions that insert a value inside a table defined by the user inside the state

```
this.state.hostTableList.get("tableName").store(field);
```

It indicates that if the conditions inside the if block are verified, the table will store the value that is passed as parameter.

The parameter corresponds to a field of a packet that would be stored and it must have the same type of the field indicated at the table definition time.

Only one store instruction could be inserted inside an if-then block.

5 - Instructions that define the action to perform on a packet

They must be placed as last instruction of the then block and they define the action that will be performed on the packet indicated as first parameter.

There are three types of instructions (that correspond to the type of the different actions):

- `return new RoutingResult(packet,Action.FORWARD,ForwardDirection.SAME_INTERFACE);`

It sends the packet "packet" back to the previous host.

- `return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);`

It forwards the packet "packet".

- `return new RoutingResult(packet,Action.DROP,null);`

It drops the packet.

4) Example

In order to make as clear as possible this guide, two examples of two network functions described with the defined library are shown in the next pages.

```
package test.definition;
import definitions.*;

public class WebCache extends ForwardingHost{

	@Override
	public void defineState() {
		this.state.hostTableList.add(HostTable.createTable("ResorceCache",1, FieldType.URL));
		this.state.internalNodes.set();
		this.state.receivedPackets.set();
	}

	@Override
	public RoutingResult onReceivedPacket(Packet packet) {
		if(Packet.match(packet.getProtocol(), Constants.HTTP_REQUEST_PROTOCOL, Operator.EQUAL) &&
				this.state.hostTableList.get("ResorceCache").contains(packet.getUrl()) &&
				this.state.internalNodes.contains(packet.getSourceAddress())) {
			Packet p1 = new Packet();
			p1.setSourceAddress(packet.getDestinationAddress());
			p1.setDestinationAddress(packet.getSourceAddress());
			p1.setUrl(packet.getUrl());
			p1.setProtocol(Constants.HTTP_RESPONSE_PROTOCOL);
			return new RoutingResult(p1,Action.FORWARD,ForwardDirection.SAME_INTERFACE );
		}
		if(Packet.match(packet.getProtocol(), Constants.HTTP_REQUEST_PROTOCOL, Operator.EQUAL)
				&& !this.state.hostTableList.get("ResorceCache").contains(packet.getUrl()) &&
				!this.state.internalNodes.contains(packet.getDestinationAddress())) {
			return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM);
		}
		Packet p2 = new Packet();
		if(Packet.match(packet.getProtocol(), Constants.HTTP_RESPONSE_PROTOCOL, Operator.EQUAL)
				&& !this.state.internalNodes.contains(packet.getSourceAddress()) &&
				this.state.receivedPackets.contains(p2) &&
				Packet.match(p2.getProtocol(), Constants.HTTP_REQUEST_PROTOCOL,
				Operator.EQUAL)  &&
				this.state.internalNodes.contains(p2.getSourceAddress()) &&
				Packet.match(p2.getUrl(), packet.getUrl(), Operator.EQUAL)) {
			this.state.hostTableList.get("ResorceCache").store(p2.getUrl());
			return new RoutingResult(packet,Action.FORWARD,ForwardDirection.UPSTREAM );
		}
		return new RoutingResult(packet,Action.DROP,ForwardDirection.UPSTREAM );
	}

}
```

```
package test.definition;

import definitions.*;

public class MailClient extends EndHost{

	@Override
	public void defineState() {
		this.hostPropertyList.add("ip_mailServer");
	}

	@Override
	public RoutingResult onReceivedPacket(Packet packet) {
		return null;
	}

	@Override
	public Packet defineSendingPacket() {
		Packet packet = new Packet();
		packet.setSourceAddress(this.address);
		packet.setOrigin(this.address);
		packet.setBody(packet.getOrigBody());
		packet.setProtocol(Constants.POP3_REQUEST_PROTOCOL );
		packet.setDestinationAddress((Address) (this.hostPropertyList.get("ip_mailServer")));
		return packet;
	}

}
```

5) Usage of the generated class and creation of the tests

Running the Parser (which receives as input the path of the class generated with the library) an other class will be produced (inside the package mcnet.netobjs.NF). This new class will supply the input to the verifying system Z3.

The new class exposes some methods that depend on the characteristics of the network function. These methods are:

1- A constructor: its name is equal to the name of the class.

2- An *install* method: its name is equal to *ClassName_install*. If the network function has some parameters, they must be passed to this function.
The type of the object passed must correspond to type declared. So, if the property is an address, a DatatypeExpr must be passed; if it is a node a NetworkObject must be passed; in any other cases an int must be passed.

3- A *setInternalAddress* method: it is present only if the network function has its own private network. An ArrayList of NetworkObject must be passed if the nodes do not have private addresses, otherwise an ArrayList of DatatypeExpr.

4- One or more *addTable_list* methods: one for each static table defined. *Table* corresponds to the name of the table and the function takes as parameters the fields that have been previously defined (the type of the field follow the same rules of the parameters).

The methods must be used following this order:

a) Constructor
b) Install method
c) *setInternalAddress* method (if it is present)
d) *addTable_list* methods (if they are present)

In order to set up a test, a class must be created. This class must have a constructor that receives a *Context* object and the fields must be all the network functions that the network contains and a *Checker* object.

Inside the constructor the following instructions must be used:

1. An instruction that creates the lists of nodes and addresses. For example:

```
NetContext nctx = new NetContext (
      ctx,
      new String[]{"politoMailClient", "politoAntispam",},
      new String[]{"ip_client", "ip_antispam"});
```

2. An instruction that creates the network object:

```
Network net = new Network (ctx,new Object[]{nctx});
```

3. All the instructions that create the different nodes (calling the constructor). For example:

```
politoMailClient = new MailClient(
      ctx,
      new Object[]{nctx.nm.get("politoMailClient"),
      net,
      nctx});
```

4. All the instructions that create the couple node-address. For example:

```
ArrayList<Tuple<NetworkObject,ArrayList<DatatypeExpr>>> adm = new
ArrayList<Tuple<NetworkObject,ArrayList<DatatypeExpr>>>();

ArrayList<DatatypeExpr> al1 = new ArrayList<DatatypeExpr>();
ArrayList<DatatypeExpr> al2 = new ArrayList<DatatypeExpr>();


al1.add(nctx.am.get("ip_client"));
al2.add(nctx.am.get("ip_antispam"));


adm.add(new Tuple<>(politoMailClient, al1));
adm.add(new Tuple<>(politoAntispam, al2));


net.setAddressMappings(adm);
```

5. All the instructions that creates the routing tables. For example:

```
ArrayList<Tuple<DatatypeExpr,NetworkObject>> rtClient = new
ArrayList<Tuple<DatatypeExpr,NetworkObject>>();

rtClient.add(new Tuple<>(nctx.am.get("ip_mailServer"), politoAntispam));
```

```
ArrayList<Tuple<DatatypeExpr,NetworkObject>> rtAnti = new
ArrayList<Tuple<DatatypeExpr,NetworkObject>>();

rtAnti.add(new Tuple<>(nctx.am.get("ip_mailServer"), politoMailServer));
rtAnti.add(new Tuple<>(nctx.am.get("ip_client"), politoMailClient));


net.routingTable(politoMailClient, rtClient);
net.routingTable(politoAntispam, rtAnti);
```

6. An instruction that adds the nodes to the network. For example:

```
net.attach(politoMailClient, politoAntispam);
```

7. All the instruction that use the methods: *Install, setInternalAddress, addTable_list*

8. The instruction that creates the checker:

```
check = new Checker(ctx,nctx,net);
```

Then, in order to use the test class, an other class must be created.

This class must have as field a *Context* object and two methods: *resetZ3* and *main*.

The definition of the *resetZ3* method is:

```
public void resetZ3() throws Z3Exception{
      HashMap<String, String> cfg = new HashMap<String, String>();
      cfg.put("model", "true");
      ctx = new Context(cfg);
}
```

The *main* method must throw *Z3Exception*. It must implement the class itself and call the *resetZ3* method.

After it must implement the test class passing the *Context* object. For example:

```
PolitoDoubleFwTest model = new PolitoDoubleFwTest(p.ctx);
```

Then it must define an *IsolationResult* object and call the following method in order to test the isolation property (a and c are two nodes of the network under test):

```
IsolationResult ret =model.check.checkIsolationProperty(model.a,model.c);
```

The result of the test is a Status value and it could be obtained through `ret.result`.

In order to clarify as more as possible, an example of tests follows:

```
import java.util.ArrayList;

import com.microsoft.z3.Context;
import com.microsoft.z3.DatatypeExpr;

import mcnet.components.Checker;
import mcnet.components.NetContext;
import mcnet.components.Network;
import mcnet.components.NetworkObject;
import mcnet.components.Tuple;
import mcnet.netobjs.NF.*;
/**
* <p/>
 * Cache - Nat - Fw     test                                          <p/>
 *
 *    | HOST_A | ----| CACHE |----- | NAT | ----| FW |----- | HOST_B |   <p/>
 *   ……..…….....|                                                   <p/>
 *    | HOST_C | --------                                            <p/>
 *
 */
public class PolitoCacheNatFwTest {

        public Checker check;
        public EndHost a;
        public WebCache politoCache;
        public WebClient hostA;
        public WebServer hostB,hostC;
        public WebServer server;
        public NAT politoNat;
        public Firewall politoFw;

        public      PolitoCacheNatFwTest(Context ctx){

                NetContext nctx = new NetContext (ctx,new String[]{"hostA", "hostC", "politoCache","politoNat","politoFw","hostB"},
                                new String[]{"ip_hostA", "ip_hostC", "ip_politoCache","ip_politoNat","ip_politoFw","ip_hostB"});
                Network net = new Network (ctx,new Object[]{nctx});

                hostA = new WebClient(ctx, new Object[]{nctx.nm.get("hostA"), net, nctx});
                hostB = new WebServer(ctx, new Object[]{nctx.nm.get("hostB"), net, nctx});
                hostC = new WebServer(ctx, new Object[]{nctx.nm.get("hostC"), net, nctx});
                politoCache = new WebCache(ctx, new Object[]{nctx.nm.get("politoCache"), net, nctx});
                politoNat = new NAT(ctx, new Object[]{nctx.nm.get("politoNat"), net, nctx});
                politoFw = new Firewall(ctx, new Object[]{nctx.nm.get("politoFw"), net, nctx});

                ArrayList<Tuple<NetworkObject,ArrayList<DatatypeExpr>>> adm = new
ArrayList<Tuple<NetworkObject,ArrayList<DatatypeExpr>>>();
```

```
ArrayList<DatatypeExpr> al1 = new ArrayList<DatatypeExpr>();
ArrayList<DatatypeExpr> al2 = new ArrayList<DatatypeExpr>();
ArrayList<DatatypeExpr> al3 = new ArrayList<DatatypeExpr>();
ArrayList<DatatypeExpr> al4 = new ArrayList<DatatypeExpr>();
ArrayList<DatatypeExpr> al5 = new ArrayList<DatatypeExpr>();
ArrayList<DatatypeExpr> al6 = new ArrayList<DatatypeExpr>();

al1.add(nctx.am.get("ip_hostA"));
al2.add(nctx.am.get("ip_hostB"));
al3.add(nctx.am.get("ip_hostC"));
al4.add(nctx.am.get("ip_politoCache"));
al5.add(nctx.am.get("ip_politoNat"));
al6.add(nctx.am.get("ip_politoFw"));

adm.add(new Tuple<>(hostA, al1));
adm.add(new Tuple<>(hostB, al2));
adm.add(new Tuple<>(hostC, al3));
adm.add(new Tuple<>(politoCache, al4));
adm.add(new Tuple<>(politoNat, al5));
adm.add(new Tuple<>(politoFw,al6));

net.setAddressMappings(adm);

 ArrayList<Tuple<DatatypeExpr,NetworkObject>> rtA = new ArrayList<Tuple<DatatypeExpr,NetworkObject>>();
rtA.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostB"), politoCache));
rtA.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostC"), politoCache));
rtA.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoCache"), politoCache));
rtA.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoNat"), politoCache));
 rtA.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoFw"), politoCache));

ArrayList<Tuple<DatatypeExpr,NetworkObject>> rtB = new ArrayList<Tuple<DatatypeExpr,NetworkObject>>();
rtB.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostA"), politoFw));
rtB.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostC"), politoFw));
rtB.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoCache"), politoFw));
rtB.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoNat"), politoFw));
rtB.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoFw"), politoFw));

ArrayList<Tuple<DatatypeExpr,NetworkObject>> rtC = new ArrayList<Tuple<DatatypeExpr,NetworkObject>>();
rtC.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostB"), politoCache));
rtC.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostA"), politoCache));
rtC.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoCache"), politoCache));
rtC.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoNat"), politoCache));
rtC.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoFw"), politoCache));

ArrayList<Tuple<DatatypeExpr,NetworkObject>> rtCache = new ArrayList<Tuple<DatatypeExpr,NetworkObject>>();
rtCache.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostA"), hostA));
rtCache.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostB"), politoNat));
rtCache.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostC"), hostC));
rtCache.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoNat"), politoNat));
```

```
                rtCache.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoFw"), politoNat));

                ArrayList<Tuple<DatatypeExpr,NetworkObject>> rtNat = new ArrayList<Tuple<DatatypeExpr,NetworkObject>>();
                rtNat.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostA"), politoCache));
                rtNat.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostB"), politoFw));
                rtNat.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostC"), politoCache));
            rtNat.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoCache"), politoCache));
            rtNat.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoFw"), politoFw));

            ArrayList<Tuple<DatatypeExpr,NetworkObject>> rtFw = new ArrayList<Tuple<DatatypeExpr,NetworkObject>>();
            rtFw.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostB"), hostB));
            rtFw.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostA"), politoNat));
            rtFw.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_hostC"), politoNat));
            rtFw.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoNat"), politoNat));
            rtFw.add(new Tuple<DatatypeExpr,NetworkObject>(nctx.am.get("ip_politoCache"), politoNat));

            //Configuring routing tables of middleboxes
            net.routingTable(politoFw, rtFw);
            net.routingTable(politoNat, rtNat);
            net.routingTable(politoCache, rtCache);
            net.routingTable(hostA, rtA);
            net.routingTable(hostC, rtC);
            net.routingTable(hostB, rtB);

            //Attaching nodes to network
            net.attach(hostA, hostB, hostC, politoCache, politoNat, politoFw);

            ArrayList<DatatypeExpr> ia = new ArrayList<DatatypeExpr>();
            ia.add(nctx.am.get("ip_hostA"));
            ia.add(nctx.am.get("ip_hostC"));
            ArrayList<Tuple<DatatypeExpr,DatatypeExpr>> acl = new ArrayList<Tuple<DatatypeExpr,DatatypeExpr>>();
            acl.add(new Tuple<DatatypeExpr,DatatypeExpr>(nctx.am.get("ip_politoNat"),nctx.am.get("ip_hostB")));

            //Configuring middleboxes
            hostA.WebClient_install(nctx.am.get("ip_hostB"));
            hostB.WebServer_install();
            hostC.WebServer_install();
            politoCache.WebCache_install();
            ArrayList<NetworkObject> list = new ArrayList<NetworkObject>();
            list.add(hostA);
            politoCache.setInternalAddress(list);
            politoNat.NAT_install(nctx.am.get("ip_politoNat"));
            politoNat.setInternalAddress(ia);
            politoFw.addAcl_list(acl);
            politoFw.Firewall_install();

            check = new Checker(ctx,nctx,net);

        }
}
```

Chapter 10

Conclusion

As explained at the beginning of this thesis report, the objectives of this work is to follow the current trend of the evolution of the networks.
Considering what are the current requests of the market and looking at the possible next future developments, this thesis would be a step toward the real implementation of new concepts inside the networks. In fact, the framework and the tool produced may be a basic structure to define Virtual Network Functions and to test the wanted networks before the deployment.

Considering the results obtained in terms of easiness of project's usage also for non technical users (thanks also to the guide developed) and the reliability in terms of creation of the classes that describe the behaviour of the Virtual Network Functions, I think it is possible to say that this project could be used in other future wider works that really will allow a transformation of the current structure of the network into an other one more flexible, cheaper and simple to manage.
I conclude with the mine personal hope that these next generation networks could improve the efficiency of the telecommunication technologies and lead us to an always more interconnected world.

Special thanks to Ph.D. Matteo Virgilio and Professors Riccardo Sisto and Guido Marchetto that have followed me in the development of this thesis project.

Bibliography

[1] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, Scott Shenker UC Berkeley, MPI SWS, EPFL, TAU, ICSI. *Verifying Isolation Properties in the Presence of Middleboxes*. 2014

[2] Spinoso, Serena; Virgilio, Matteo; John, Wolfgang; Manzalini, Antonio; Marchetto, Guido; Sisto, Riccardo. *Formal verification of Virtual Network Function graphs in an SP-DevOps context*. 2015

[3] Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki

[4] Oracle documentation. http://docs.oracle.com/javase/specs/jls/se8/html/jls-1.html

[5] Tutorialpoint website. https://www.tutorialspoint.com/java

[6] Eclipse website. http://www.eclipse.org/

[7] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, TACAS, Budapest, Hungary, 2008 (Powerpoint Slides)

[8] Leonardo de Moura. *SMT Solvers: Theory and Implementation*

[9] HTML.it website. http://www.html.it/

[10] Giacomo Costantini. *Porting to Java of Python code for reachability verification in VNF chains*