POLITECNICO DI TORINO

Master Degree Course in Computer Engineering

Master Degree Thesis

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

Supervisors
prof. Guido Marchetto
prof. Riccardo Sisto
dott. Fulvio Valenza, dott. Jaloliddin Yusupov

Candidate
Antonio VARVARA
student ID: 234531

YEAR 2017-2018

Contents

Chapter 1

Introduction

In its original idea, the Internet was designed to interconnect different computers to the same network using a series of protocols, each related to a level in the ISO/OSI protocol stack. This subdivision into levels allowed to build relatively simple devices that were able to work only on a specific level, hiding the unnecessary information of the other levels.

Nowadays networks are evolving rapidly and the need for a more dynamic and automated network management is taking a very important role in defining the direction of this evolution. The new paradigms that are drastically changing the rule of the game, are *Software Defined Networking* (SDN) and *Network Function Virtualization* (NFV) that allow to adopt a dynamic network model capable of adapting to future changes without great effort by operators.

The SDN paradigm defines a new type of architecture for the realization of a network, separating the control plane from the data plane, switching to a more centralized approach. In particular, the control plane is entrusted only to a single controller in order to achieve greater scalability and security regardless of the apparatus used for the data plane, encouraging an even more dynamic network that is no more bound to the different number of protocols used by the different brands.

The NFV paradigm, on the other hand, leads to the virtualization of services and applications that usually run on proprietary hardware appliances (e.g. firewall, DPI, etc.). This abstraction allows a more flexible network in which the network functions (that in this context are called *Virtual Network Function*, VNF) are no more linked to a single physical server and can be moved from one server to another with ease. Moreover, a series of network functions can be chained together to offer more complex services. The use of virtualized applications instead of physical hardware also allows to take network functions in and out of service, and scale them up and down easily. The NFV paradigm is based on the usage of an orchestrator that decouples the physical instances of the network functions from the requested virtual service. Even though the orchestrator provides a simple way to deliver or release a service composed by a series of VNFs, it does not provide a verification for the correctness of the whole service. For example, it does not check if two end points are indeed reachable in the service, as this would require the knowledge of how the different VNFs work. In fact, even though the NFV automation approach offers undeniable great benefits like scalability and flexibility, on the other hand it poses new issues regarding misconfigurations and security flaws since it relies upon the external input to configure the VNFs. This problem can be prominent especially in corporate networks, where the configuration of the system could reach high level of complexity and there might be some unexpected errors that cause unwanted behaviours. The synergy between SDN and NFV allows the formation of a programmable network between virtualized functions.

In this context, a software module called Verifoo, further described in section 3.2, has been developed. It can be utilized in combination with an NFV orchestrator in order to automate the choice of the optimal deployment of a network service with a formal verification of requested policies. Verifoo uses a custom XML format and acts like a verification service which ensure that the network service that is being deployed is compliant with what the user expressed in a series of high-level policies. Moreover, it also provides the optimal deployment on an infrastructure based on the available resources in the physical machines.

The aim of this thesis is to further develop Verifoo which, being in its early stage of development, presents certain limitations and model simplifications whose overcoming, in conjunction with the extension of its functionalities, will be described in the following chapters. Initially, Verifoo only supported chains of VNFs as network services, with the very restrictive limitation of using only one client and only one server. Its deployment algorithm considered only the disk storage as allocation resource that affected the deployment. In the Verifoo simulation model, the information about the TCP/UDP ports in a packet header was missing. This was mainly because no VNF needed such information. In fact, even the behavioural model of a firewall, which usually needs those fields, represented only an initial implementation which described a blacklist firewall that solely focused on the addresses to decide what to do for a specific packet.

All the aforementioned limitations were addressed and some totally new functionalities were added, as further described in chapter 6. In particular, the new changes promote the resolution of the configuration problem that currently affects an NFV orchestrator. In fact, usually the orchestrator relies on the configurations given by the administrator (or the user) and instructs other tools to inject that configuration in the correspondent VNF. Exploiting the Verifoo framework, the new features have the objective to automatically produce a configurations that satisfies the constraints introduced in the user-level policies.

This thesis work introduced the new features, improving the already present ones, with the aim to explore a solution for the current problems in an NFV framework, left as study items [1] in the standard. Even though the proposed solutions are based strictly on Verifoo, they can be easily generalized to be of inspiration for new methods to address the same problems. This topic will be studied in depth in the other parts of the thesis which is structured in the following way:

- chapter 2 provides a general description of the new technologies involved in the thesis;
- chapter 3 describes some useful concept that will often recur in the other chapters; moreover, it delineates the general functioning of the tools exploited during the work;
- chapter 4 outlines what are the objectives pursued in this thesis;
- chapter 5 gives an insight on what were the design choices present in the final solution;
- chapter 6 contains all the indications on what has been added or modified in Verifoo during the thesis;
- chapter 7 elaborates on the experimental results obtained in the testing phase;
- Chapter 8 draws conclusions on the work done and gives some suggestions on how to further improve Verifoo.

Chapter 2

NFV and SDN

In this chapter, the new frontiers of virtualization are described. In section 2.1 the NFV paradigm is illustrated with a particular focus on the Open Source MANO project which is an implementation of an NFV orchestrator that is the current reference point for Verifoo. In section 2.2 a quick overview of the SDN paradigm is given. Finally, in section 2.3, the role that Verifoo can have in a virtual architecture that exploits these new virtualization techniques is framed and what are the challenges it faces.

2.1 Network Function Virtualization

2.1.1 NFV Introduction

In recent years ISPs have faced a multitude of challenges arising from the composition of their network architecture, mainly made up of devices dedicated to a specific function (e.g. firewall, WAN accelerators, etc.). This solution carries an intrinsic rigidity that restrains the ability of the architecture to change which is however opposed to the exponential increase in customer needs. In order to maintain an acceptable level of service, an ISP usually faces this increase adding more physical devices, being forced to deal with a growing operational and maintenance costs without burden the final user to avoid hindering its competitiveness. In this regard, an NFV solution exploits the benefits deriving from the virtualization to bring radical changes in how to conceive a network architecture. It transforms all the dedicated devices in virtual instances executable on general purpose machine whose capacity and computational power make them able to host a significant number of those instances. The virtual devices usually are called Virtual Network Function (VNF) and can be freely moved in the physical infrastructure. The advantages in this approach are the reduced operational costs and the possibility to have a more rapid evolution of a certain service since the interconnection of various network functions no longer demands a new hardware addition in the infrastructure, nor a network administrator to be assigned to its manual configuration.

2.1.2 NFV ETSI Framework

Nowadays these technologies are still being heavily researched and in order to sidestep the chaos that can be associated with industry fragmentation and entry barriers, various regulations have been promulgated. In particular, the European Telecommunication Standards Institute (ETSI), an independent standardization group, formed a group (ETSI Industry Specification Group for Network Functions Virtualization, ETSI ISG NFV) charged with developing requirements and architecture specifications for hardware and software infrastructure needed to make sure virtualized functions are maintained. In its standardization [1], the NFV framework identifies three domains:

- **Virtualized Network Function**, the software counterpart of a network function which runs on the NFVI
- **NFV Infrastructure (NFVI)**, the physical infrastructure onto which the VNFs are executed
- **NFV Management and Orchestration (NFV MANO)**, the orchestrator that oversees all the management tasks for the VNFs

In particular, the NFV Management and Orchestration (MANO) is the ETSI-defined framework for the management and orchestration of all resources in the cloud data center. This includes computing, networking, storage, and virtual machine (VM) resources. NFV MANO is broken up into three functional blocks as shown in Figure 2.1:

- **NFV Orchestrator**: Responsible for on-boarding of new network services (NS) and virtual network function (VNF) packages; NS life-cycle management; global resource management; validation and authorization of network functions virtualization infrastructure (NFVI) resource requests
- **VNF Manager**: Oversees life-cycle management of VNF instances; coordination and adaptation role for configuration and event reporting
- **Virtualized Infrastructure Manager (VIM)**: Allocates the resources for a VNF, namely it controls and manages the interaction of a VNF with computing, storage, and network resources. It also provides fault information, as well as information about resources monitoring and optimization

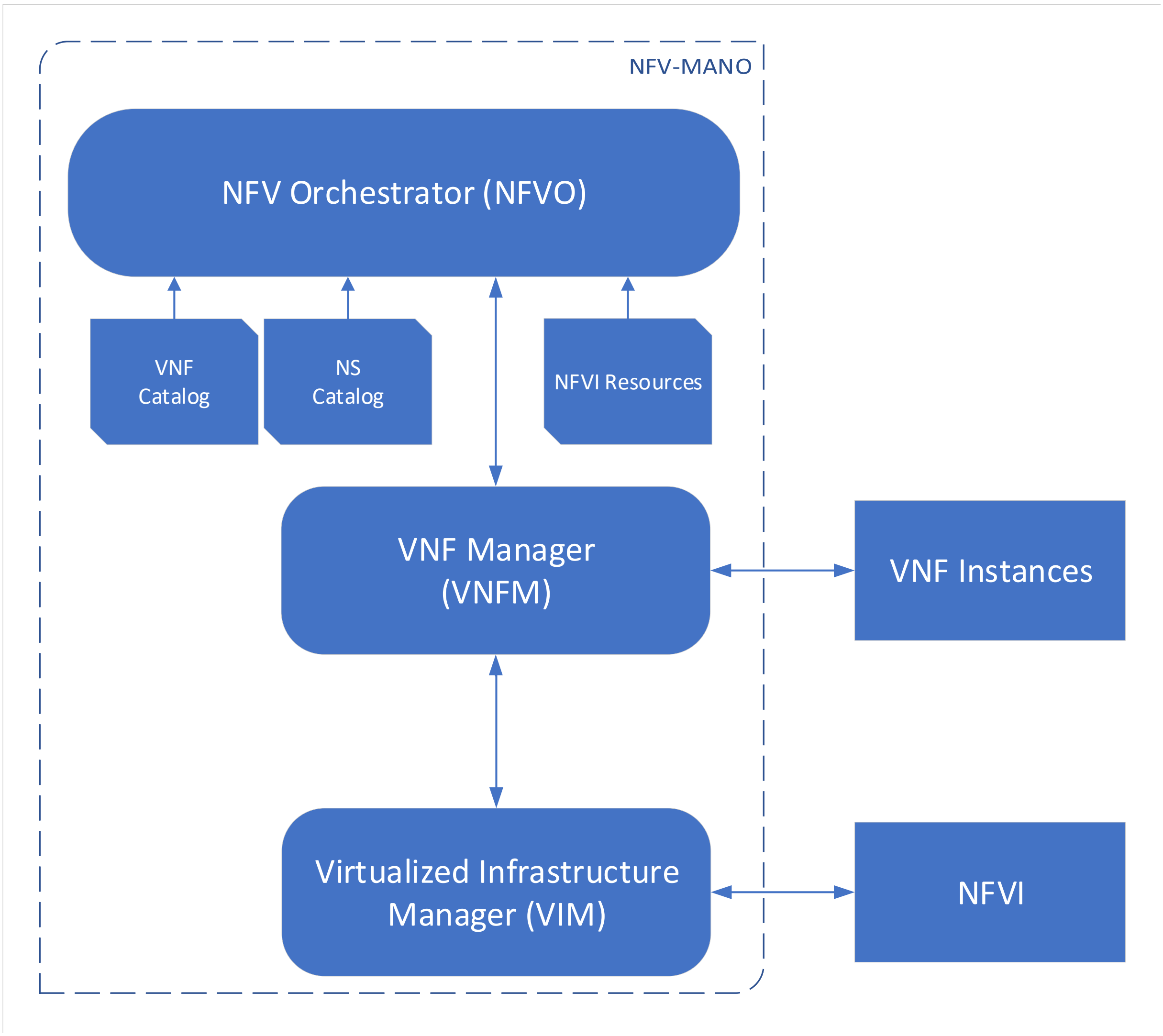


Figure 2.1. NFV MANO Architecture

For the NFV MANO architecture to work properly and effectively, it must be integrated with open application program interfaces (APIs) in the existing systems. The MANO layer works with templates for standard VNFs and gives users the power to pick and choose from existing NFVI resources to deploy their platform or element.

2.1.3 Open Source MANO

Following the NFV MANO specifications, various projects have been developed. Among them, there is *Open Source MANO* (OSM), which is an open source project that provides a practical implementation of the reference architecture for Management & Orchestration under standardization at ETSI's NFV ISG. OSM is written in python and uses tools under the Apache Public License2.0. The open source nature of the project encourages the creation of new solutions from a sure to grow community. It consists of three main software components that map the NFV MANO functional blocks [2]:

- **openvim**: reference implementation of an NFV VIM (Virtualised Infrastructure Manager). It interfaces with the compute nodes in the NFV Infrastructure and an openflow controller in order to provide computing and networking capabilities and to deploy virtual machines. It offers a northbound interface, based on REST (openvim API), where enhanced cloud services are offered including the creation, deletion and management of images, flavours, instances and networks. The implementation follows the recommendations in NFV-PER001.
- **openmano**: reference implementation of an NFV-O (Network Functions Virtualization Orchestrator). It interfaces with an NFV VIM through its API and offers a northbound interface, based on REST (openmano API), where NFV services are offered including the creation and deletion of VNF templates, VNF instances, network service templates and network service instances.
- **openmano-gui**: web GUI to interact with openmano server, through its northbound API, in a friendly way.

The openmano orchestrator is, in turn, composed by a series of software modules, whose task are different and independent but strictly interwoven in order to realize the management functionalities in an NFV architecture. It is formed by three main parts:

- *VNF Configuration & Abstraction (VCA)*, which is the module that deals with the lifecycle management of the network functions, giving the possibility to configure them at run-time.
- *Resource Orchestrator (RO)*, which has been developed to manage the resource orchestration of a NFVI through a VIM.
- *Service Orchestrator (SO)*, which copes with the coordination of the previous modules and the entire management of a network service, from the descriptors on-boarding to the online management of the network service.

The workflow of Open Source MANO, aligned with the basic functionality that an NFV orchestrator should have, envisages the modelling of a network service using a GUI or through file descriptors which will be later deployed automatically onto a physical infrastructure. Each of the VNF composing the network service can be configured following external indications provided by an administrator or a third-party tool. To handle traffic flow and VNF interconnections, Open Source MANO uses an SDN approach with a particular version of an OpenFlow controller called Floodlight [3].
The software module developed during this thesis work, Verifoo, can be integrated in the OSM orchestrator as a module that can provide the optimal deployment of a network service on a well-known network infrastructure, with a formal verification of requested policies. The orchestrator should interact with Verifoo through the available REST API sending a request to deploy a network service on a physical topology. After receiving a result, it should communicate the result to other modules that may benefit from that knowledge. Currently Verifoo is able to interact with SONATA, better described in Section 3.2, which is the currently VIM adopted by OSM.

2.2 Software Defined Networking

ISPs networks contain a variety of proprietary hardware that implicitly possess less flexibility and dynamism than the innovation cycles require. The launch of new services often demands network reconfiguration and on-site installation of new equipment which in turn requires additional floor space, power, and trained maintenance staff. Hard-wired network with single functions boxes is tedious to maintain,

slow to evolve, and prevent service providers from offering dynamic services.
A network handled with an SDN approach conceives a network directly programmable in which the control plane is separated from the data plane and centralized in an SDN controller who has the global view of the network. The main purpose of an SDN architecture is to be dynamic, manageable, economically efficient and adaptable, in order to be useful for the dynamic nature and high bandwidth usage of today applications. As a result, the network devices end up doing only a simple forwarding task based on some rules defined by the controller. This also allows the configuration of the entire network to be executed only on one node, the controller, which then deals with applying the changes to the devices that compose the network. As a direct consequence, the networks are agile and capable to respond automatically to the needs of the traffic and services running over it. However, at the moment the controller centralization also represents the most critical point for this technology, since it creates a single point of failure causing concerns about its security and scalability.
As shown in figure 2.2, in an SDN architecture the applications run on top of the controller layer and inform it about their specific needs. The controller then instructs the switches in the network to forward the packet of a specific flow, identified by different fields of the header of the packets, based on the information received from the application. The communication between the controller and the networking devices are usually carried out with OpenFlow.

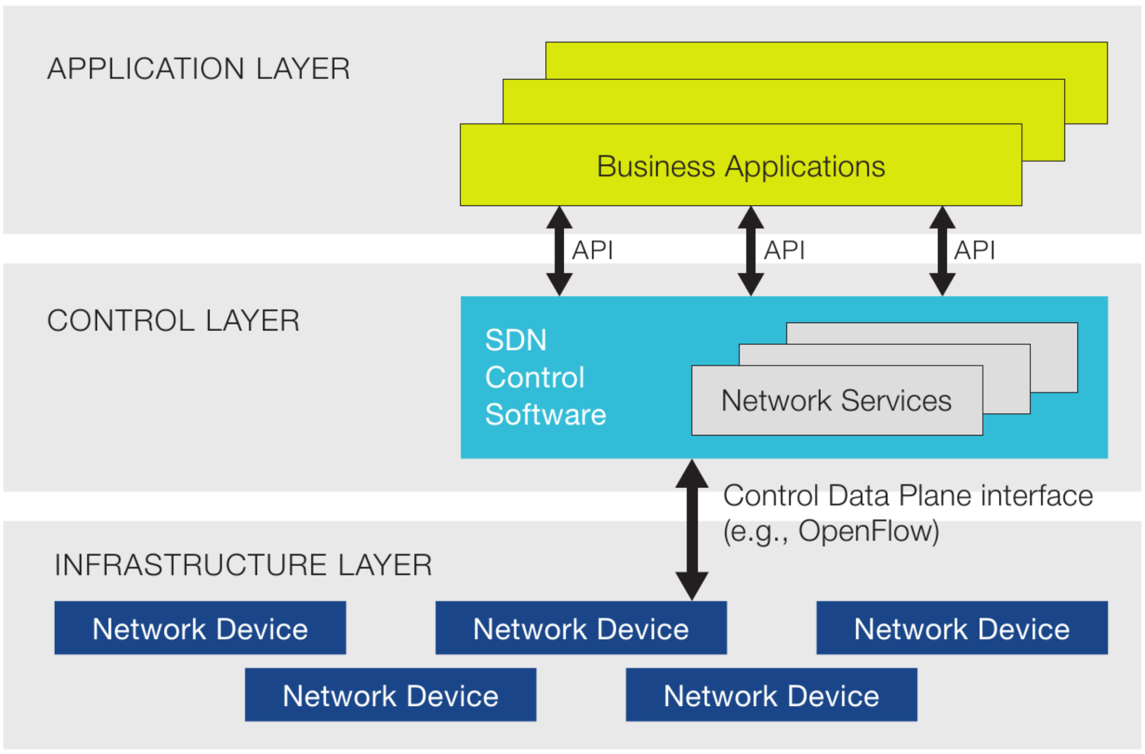


Figure 2.2. SDN Architecture

OpenFlow is an open source communication protocol and is one of the first protocol standardized to be used in an SDN environment. However, there are other standards and open-source organizations with SDN resources, so OpenFlow is not the only protocol that makes up SDN. Using open standards makes the design and the maintenance of the network easier because the same instructions sent from the SDN controller are supported by devices of different vendors that in the classic approach rarely are interoperable.

SDN and NFV are two independent technique, hence it is possible to design a network that uses one technology and not the other. However, their complementarity allows both of them to be used in the same network with great benefits. Both technologies stir towards an abstraction from the underlying hardware, promoting a higher dynamicity and a faster evolution of the services. For instance, a service could be realized a series of VNFs dynamically and automatically chained thanks to an SDN controller based on the specific requested solution.

2.3 The role of Verifoo in an NFV framework

In the NFV reference architectural framework there are still some study items [1] that are the subjects of many of the works published in the recent years [4].

The decoupling of a network function from the physical hardware it runs on, poses new challenges in instantiating VNFs at appropriate location and keeping track of it, allocating and scaling hardware resources accordingly, all in order to map an end-to-end service onto an NFV infrastructure. In the ETSI standardization the process of how to deterministically deploy VNF instances on the NFV Infrastructure is not explicitly described. Giving the optimization nature of the problem, performance-efficient methods of deployment should be encouraged. Verifoo, among its functionalities, addresses the deployment of VNFs and offers a way to efficiently place them on a given infrastructure.

Another challenge in the NFV world regards the VNFs configurations. The network functions functional behaviour is well known but an NFV orchestrator manages only their lifecycle without worrying about the effect of their configurations. This can cause problems that can be discovered only after the actual deployment of the service which can be risky in some cases (e.g. a firewall that let a dangerous flow pass). Verifoo also takes care of this problem performing a preliminary verification of some policies declared by an administrator in order to check the correctness of the whole service. This verification is achieved using internally simple models of the network functions in order to simulate the operating service. How Verifoo achieves the previously mentioned objectives will be further described in section 3.2 after the introduction of some other useful concepts that help to understand its functioning.

Verifoo has been designed to work closely with an NFV orchestrator to act as a verification and deployment manager. Its basic workflow is based on an external input which can be seen as the first touch point with an NFV framework.The input follows a custom XML schema which will be described in section 6.1. A general idea of the interaction can be seen in Figure 2.3.

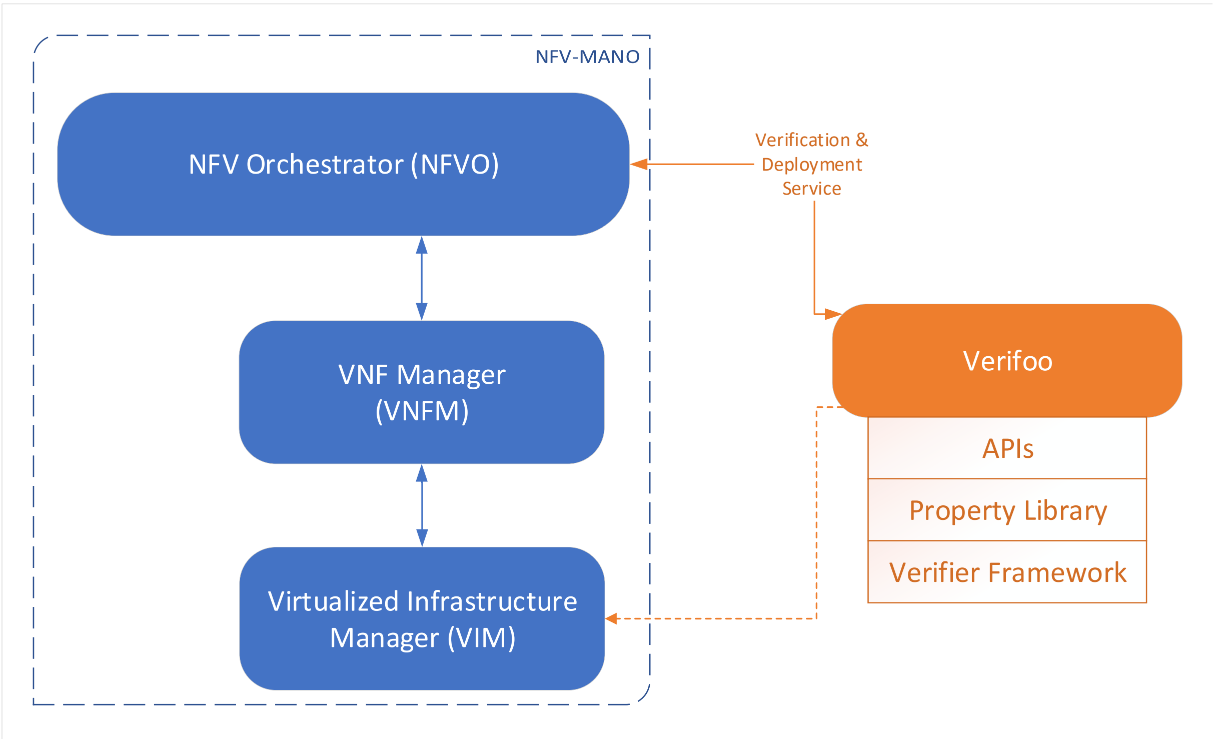


Figure 2.3. NFV Orchestrator with Verifoo

Received the XML describing a network service and a physical topology, Verifoo is able to generate the optimal deployment optimizing the network performance and the resources allocation of the physical machines. The result is returned as output and can be elaborated by the orchestrator, which can then instruct the VIM accordingly. This last step can also be done directly by Verifoo in order to act as an additional layer above the virtualized infrastructure and hide the optimization details from the orchestrator. In the current version, as a proof of concept, this is achieved thanks to the integration of SONATA, which is a VIM supported by OSM. All the interactions can happen through the RESTful APIs included in Verifoo.

All the design choices of the project elaborated during this thesis will be presented in chapter 5, while more low-level details will be described in chapter 6.

Chapter 3

Terminology and exploited tools

In order to fully understand the content of this thesis, a set of concepts and the used tools are presented in the following paragraphs.

3.1 Concepts

In this section two high-level concepts commonly referred throughout the thesis will be introduced. These concepts represent the basic information that Verifoo needs for its execution, the service graph described in section 3.1.1 and the physical topology described in 3.1.2.

3.1.1 Service Graph

A Service Graph (or Network Function Forwarding Graph, NFFG, as it is referred to in the ETSI-defined framework [5]) describes a network service requested by an end-user from a Service Provider. It is modelled as a graph, where the nodes can be network functions (e.g., firewall, NAT, etc.), clients, and servers composing the requested service, while the arcs represent packet forwarding paths. There are no limitations whatsoever for the network functions to be in a single operator network. Each of the function concurs to the high-level behaviour of the whole service which can be described through policies that state if a particular flow of packets from a source is able to reach a specific destination. The source and the destination are usually endpoints and correspond to client applications or server applications. The clients are out of the NFV area of activity therefore the operator cannot exercise its authority on it, hence the clients are not included in the deployment computation and their position is always fixed. The endpoints are connected to the physical infrastructure which provides a logical interface with the network functions. In turn, the network functions are interconnected via logical links provided by the infrastructure. The NFV paradigm emphasizes that the deployment of the VNFs on the physical infrastructure is not relevant for the end-to-end service. This allows a VNF to be instantiated on different physical resources, also geographically dispersed, as long as the boundary conditions are still met (e.g. service performance and/or policy constraints).

In Verifoo, a service graph is characterised by an id, which uniquely identifies it, and a set of nodes. Each node has a name, which is unique, and is associated to a “functional type” that represents which kind of network function will be deployed. A node is also characterised by a set of neighbours that represents the unidirectional links towards the other nodes. In order to model a more realistic scenario, each node can also be associated with constraints such as the memory or the disk storage it requires.

3.1.2 Physical Topology

A Physical Topology is the network functions virtualization infrastructure (NFVI) mentioned in Section 2.1.2. Basically, it is a collection of interconnected infrastructure hosts, each one capable of hosting Virtual Network Functions (VNFs). A VNF can be an endpoint VNF (i.e. a client or a sever), or a middlebox VNF that requires to be placed (e.g. a firewall, a NAT, etc.). A physical topology is designed to be distributed, so the hosts are spread through different location which also allows to meet locality and/or latency requirements, needed in some use cases, adding a great flexibility to the global architecture.

In Verifoo, each infrastructure host is characterised by its name, the maximum number of VNFs that can be allocated to it, the number of cores it has and their frequency, the amount of memory and of disk storage available in the host. For each pair of interconnected hosts, it is also possible to express the latency of the physical channel that connects them.
One or more graph can be allocated onto a physical topology by allocating each node of each graph onto one host of the physical topology.

3.2 Tools

In this section a series of tool will be introduced. Verigraph and z3 represent the foundation of the Verifoo project and are described respectively in sections 3.2.2 and 3.2.1. Some basic information about Verifoo are presented in section 3.2.3 but more information are available throughout the whole thesis. Finally, two tools integrated in Verifoo during the course of this work are outlined, SONATA in section 3.2.4 and Neo4j in section 3.2.5.

3.2.1 z3

z3 [6] is a theorem prover from Microsoft Research that receives as input sets of First Order Logic (FOL) formulas in a format that is an extension of the one defined by the SMT-LIB 2.0 standard. Under the hood, z3 is a Satisfiability Modulo Theories (SMT) solver. It can resolve constraint satisfaction problems and thus formalize an approach to constraint programming. Its main objective is to check the satisfiability of logical formulas, i.e. finding a solution to a set of constraints, and also to produce models for satisfiable formulas. An extension of z3 is z3Opt [7], that can solve the Maximum Satisfiability (MaxSAT) problem in those scenarios in which arbitrary models are not enough and applications want to minimize or maximize one or more values. As z3 is a low-level tool, it provides a number of APIs that can be exploited by other tools that require solving logical formulas. The version of the libraries used currently in the project is the 3.3.4.

3.2.2 Verigraph

Verigraph [8] is a verification service for policies within network graphs. The graphs can be described in JSON or TOSCA OASIS YAML/XML-based format. It can receive a description of a SG and of network function configurations and, using z3, it can analyse the given SG and related configurations in order to check if reachability policies (e.g. the possibility for packets of a certain flow generated by a certain node of the SG to reach another SG node) are satisfied or not. The interaction with the software is possible through RESTful and gRPC interface, as well as a complete CLI. Verigraph is the foundation on which Verifoo has been designed. The verification framework is very similar as it shares various portion of the code. On Verifoo though some extensions have been made in order to generalize the approach including also a physical topology.

3.2.3 Verifoo

Verifoo (Verification and optimization orchestrator) is an extension of Verigraph, and is capable of performing joint optimization and verification. It follows the recent trend for NFV management and orchestration (MANO) of many platforms, adding the integration of formal verification with the placement procedure and thus formally verifying that services work before their actual deployment. Verifoo "is able to deploy requested SGs by searching from a shared catalogue of resources. Since multiple mappings of a SG onto an infrastructure are possible, the orchestrator component also performs an optimal placement on the basis of given performance parameters and delivers a formal assurance of safety and security policies." [9]. It requires, as an input, a SG, the physical topology, and a set of network policies to be verified. Using z3, Verifoo produces, as an output, the verified optimal placement of the nodes of the SG on top of physical hosts, in an XML format. The interaction with Verifoo is possible through the REST APIs that are available with the tool. More information will be given throughout the various chapters.

3.2.4 SONATA

SONATA [10] is an emulation platform, created to support network service developers to locally prototype and test complete network service chains in realistic end-to-end multi-PoP (point of presence) scenarios. It allows the execution of real network functions, packaged as Docker, in emulated network topologies running locally on the network service developer's machine. While the simulation is running, it is possible to push to the emulator a package that describes the service chain, through a particular component called Dummy Gatekeeper. This component is a daemon process, listening on a specific port, launched with the simulation, that takes care of the instantiation and execution of all the specified dockers containing the network functions. The deployment of the dockers in the emulated network is done in a robin-round fashion without considering the actual resources available on a specific server. However, this round robin approach can be easily extended with slight modifications on the SONATA source code in order to ensure the optimal deployment, which is provided by a file dynamically generated during the execution of Verifoo. The emulation platform is based on Containernet [11] and the interactions with it are possible thanks to various CLI commands and REST APIs. Further information on this topic are available at `https://github.com/sonata-nfv/son-emu`. This platform has been recently adopted by ETSI's OSM project as part of their DevOps MDG under its new name vim-emu to work as a Virtual Infrastructure Manager.
In this thesis work, the use of SONATA is aimed at allowing Verifoo to have a place in which persistently store deployment information and test the service. The version of the libraries used currently in Verifoo is the v3.0.

3.2.5 Neo4j

Neo4j [12] is a NoSQL graph database management system written in Java. It's a transactional database, hence the ACID properties are valid. In Neo4j, everything is stored in the form of either an edge, a node, or an attribute. Each node and edge can have any number of attributes that can be used to narrow searches. Neo4j can be used either in embedded mode or in server mode. In the embedded mode, the database is included in the application and it is executed in the JVM. In the server mode, the Neo4j process is independent from the application and can be accessed through the REST API or through the dedicated driver using the Cypher Query Language, a declarative graph query language originally created by the Neo4j developers.
Neo4j's graphs handling capabilities are used in Verifoo, at the moment, as a debugging tool to visualize the SG that is given as an input. The version of the libraries used currently in Verifoo is the 3.3.4.

Chapter 4

Thesis objectives

The main objective of this thesis is to extend the functionalities of Verifoo in order to create a tool that can actively support and enhance the activity of an NFV orchestrator. In this chapter, a summarized view on how this has been achieved and why, will be given.

4.1 Thesis motivations

The NFV paradigm introduced a radical change in how the network architecture are being designed, splitting the bond between hardware and software, allowing for the development of network services as software application. It has progressively gained attention in the industry in such a way that it largely affected how networks will be built in the future years. One of the secrets of its success has been the choice to adopt a philosophy mainly open source, since it allows to regulate projects in a more flexible way, possibly introducing a 'de facto' standard.

In order to lay the basis of this evolution, the ETSI ISG listed some challenges [1] that could be seen as highly beneficial improvements for the technology. The continuous research on these topics can certainly bring fresh ideas and original resolutions methods. Verifoo sets itself in this scenario, proposing a novel solution for two of those problems. As better described in section 2.3, Verifoo mainly targets how to deterministically deploy VNF instances on the NFV Infrastructure with performance-efficient algorithms and how to check the correctness of the whole service respect to some policies declared by an administrator. The resolution of these two challenges in an NFV framework is vital for a great evolution of the technology itself. For this reason, the improvements implemented in Verifoo delivered a software that proposes a new resolution approach that can perfectly fits in an NFV framework. This can help to define a new standard or give a useful foundation for new projects that share the same vision, all aimed towards a general progression.

4.2 Design and implementation of Verifoo extensions

The expression of all the work is represented by the custom XML format that laid out the foundation of all the other major extensions. The solution, presented in section 6.1, ensures that complex network services can be expressed in a context fully compliant with NFV and SDN technologies.

The main part of the thesis focused obviously on the improvement of Verifoo. During the design phase, particular attention was paid to the modularity of the final architecture to encourage future expansions and integrations. In the implementation part, great emphasis was given to the usability of the software, removing some limitations whose overcoming was previously delayed in order to deliver a working proof of concept of the tool. For this reason, the network simulation model has been extended and the firewall model was totally revamped to allow to express more realistic scenarios. A detailed description of these improvements can be found in section 6.4.

Exploiting the already present network simulation environment, a new feature has been introduced, the auto-configuration, that is described in section 5.3. Basically it gives the opportunity to generate VNF configurations that satisfy the user level policies. This feature greatly exploits the z3 libraries to define the optimal set of rules that solve the presented problem. Automate this task is a great challenge, nevertheless the rewards that can derive from it are also significant. The designed solution to this problem will be presented in section 6.4.8

In addition to the improvements applied to the tool itself, other services have been integrated in Verifoo.

The SONATA integration has been a step towards the integration of Verifoo in the Open Source MANO project, since it is already part of OSM as the virtualized infrastructure manager. A set of RESTful APIs have been made available for this module to ease future expansions of this feature. The sections 5.1 and 6.2 describe respectively the design of this integration and its actual implementation in Verifoo.

The Neo4j introduction in the Verifoo project laid down the foundation for data persistence in order to maintain a state, although in the current version is not strictly necessary for the correct functioning of the application. In section 5.2, the motivation behind the choice of this particular non-relational database will be highlighted, while in section 6.3, what is actually implemented in Verifoo will be presented.

4.3 Tests and performance evaluation

As an indication of the quality of the new introductions a series of performance tests have been carried out on the final solution. These results are crucial to understand the real feasibility and applicability of the integration of Verifoo in an NFV framework. Regardless of the results, the methodology presented in chapter 7 is general enough to be considered as a benchmark to test also future expansions.

Chapter 5

Verifoo Extensions design

In this chapter the design choices adopted to introduce the new features will be outlined. The chapter is structured in the following way:

- In section 5.1 it will be described how Verifoo is now able to interact with SONATA, focusing on those parts of SONATA that are interesting for the interaction. A class diagram will also be presented to explain how it is possible to replace SONATA with other third-party tools that can act in the same way. Moreover, the RESTful web service associated with the simulation environment will be described.
- In section 5.2 the reasons behind the introduction of the Neo4j database in the project will be explained.
- In section 5.3, the new workflow of Verifoo will be outlined including all the new developed features.

5.1 SONATA integration design

The SONATA emulation platform has been integrated with Verifoo through the development of a software module that can translate the output of Verifoo into the right sequence of operations needed to have a running simulation of the topology with deployed VNFs. The module design will be described in section 5.1.1. The simulation can be instantiated using a specific Java class or through a set of RESTful APIs whose design choices will be described in section 5.1.2. The development of this APIs allows a more independent approach following the recent trends in micro-service architectures.

5.1.1 Verifoo interaction

As shown in Figure 5.1, Verifoo interfaces with the SONATA module in two points:

- it exploits the Topology API to load topology definitions that the Emu-Core will simulate
- through the SONATA CLI, Verifoo executes all those operations that allow the deployment of a service (since the SONATA CLI commands are available only on a Linux machine, the portability of the software is limited for now)

Even though a Resource API is available in SONATA in order to handle the resources reservation (e.g. the amount of disk storage that a VNF will occupy), it hasn't been used, as this task is already performed in Verifoo by z3, since it's strictly interwoven with the deployment operation. Hence, the host resources are stored in SONATA only as host meta-data, leaving all the logic to Verifoo.

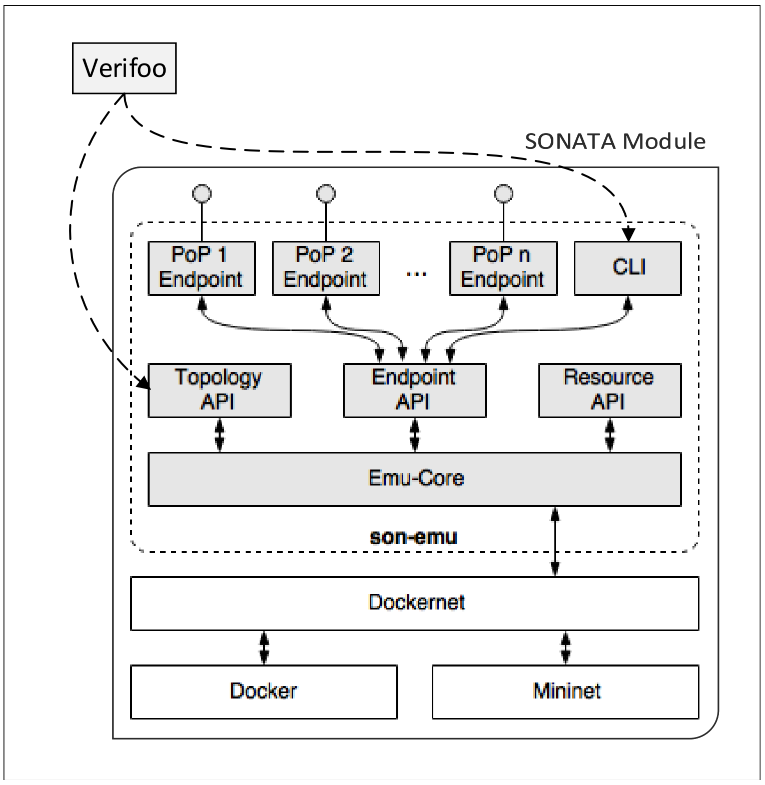


Figure 5.1. SONATA Interaction Example

A REST API that allows the interaction with the module has been developed and it will be described in a next paragraph. A simple example of its usage is depicted in Figure 5.2.

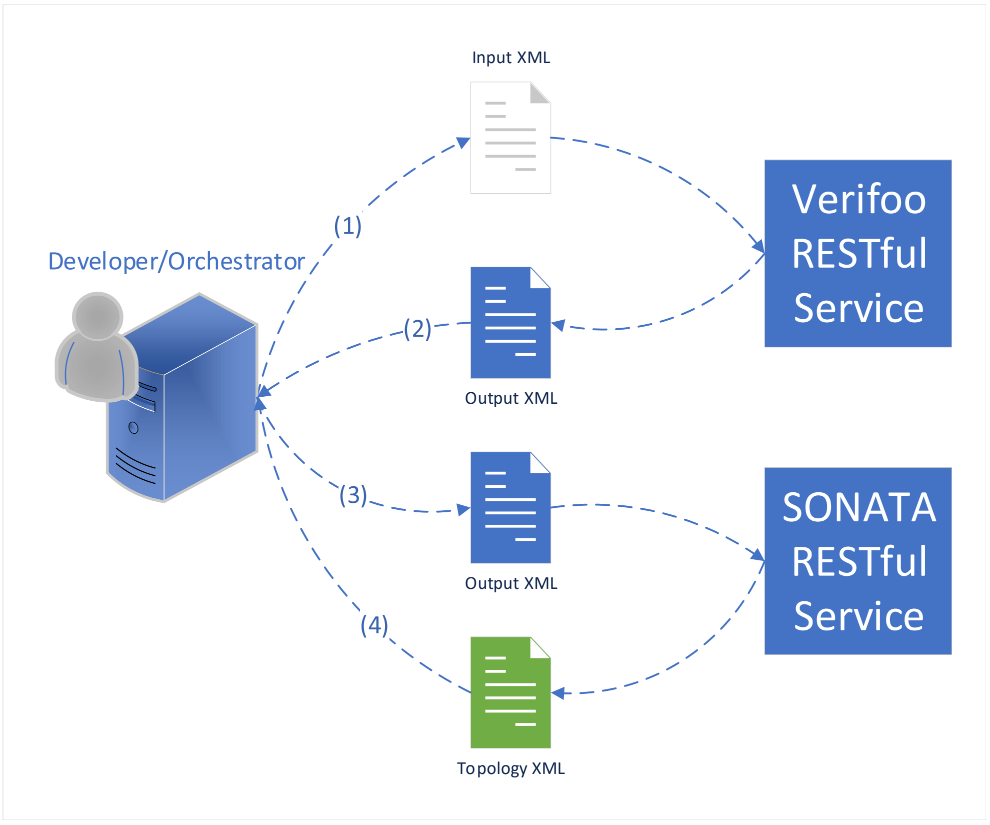


Figure 5.2. SONATA Interaction Example

The orchestrator provides an input XML to Verifoo REST web service describing the service graph, the physical topology and some policies that need to be verified (step 1). Verifoo produces the output XML in which is specified if the policies are satisfied or not, and in case they are, the XML includes also the optimal deployment (step 2). This same XML file can be forwarded to the SONATA REST web service that will instantiate the simulation of the service (step 3). The orchestrator can later use a REST API to retrieve the information on the running emulated topology (step 4) and use them to build new requests taking into account an already set up scenario.
Since the task done by the SONATA module can also be executed by other tools, the Java classes in the code are organized in such a way that it will be easy in the future to replace one with the other. The UML in Figure 5.3 represents the classes schema.

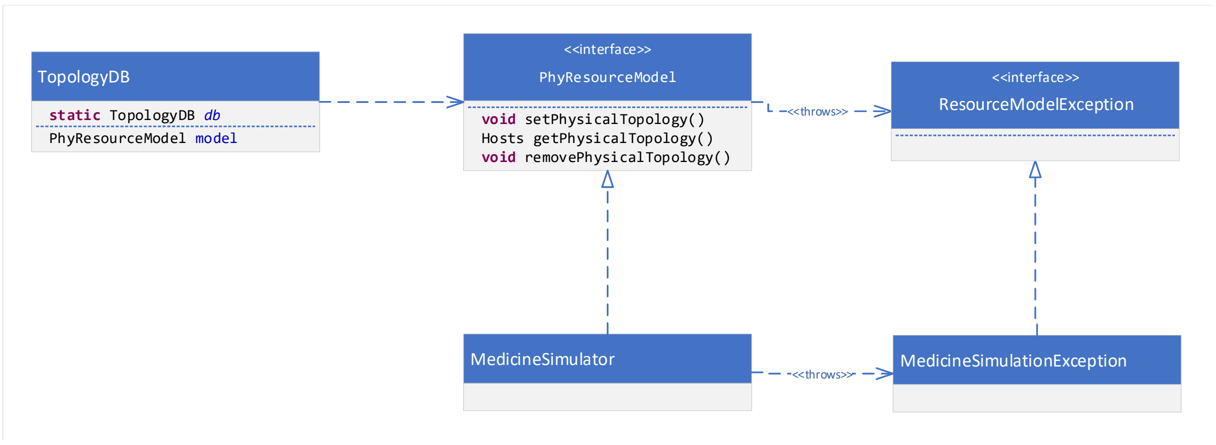


Figure 5.3. UML classes organization

The class dedicated to storing information is the TopologyDB class, implemented with the singleton pattern. It gathers information in a PhyResourceModel object, whose interface exposes the methods to store a physical topology, retrieve it or delete it. MedicineSimulator is a class that implements that interface and offers a real implementation of the abstract methods, mapping the operations in the following way:

- Storing a physical topology translates into launching the SONATA emulator with the given data
- Retrieving a physical topology translates into interacting with the SONATA emulator in order to rebuild the initial data
- Deleting a physical topology translates into stopping the SONATA emulator

(note that the MedicineSimulator class has other attributes and methods but they are not relevant in this context).
If in the future, based on other needs, another module will be chosen to store the data of the physical topology, another class, that implements the PhyResourceModel interface, can be developed.

5.1.2 RESTful API

In this paragraph the RESTful API associated with the SONATA translation module will be described. The service is composed by a single resource on which various operations are possible. The operations allow the storage of physical network information and its recovery.

Service Design

On the /simulation resource, three HTTP method are possible. Sending a POST, with the output XML of Verifoo in the body, will activate the the simulation of the topology and then it will return an HTTP message with the 200 status code and an empty body. If there is an already running simulation, a 500 status code will be returned. If the received XML has some unsatisfied properties, a 400 status code will be return together with a body in which will be present an XML with an ApplicationError element (already available in the initial Verifoo schema) that will describe the error.

Performing a GET on the same resource will return an XML with an Hosts element as root, containing the information about the physical topology previously stored with a POST.

With a DELETE it is possible to stop the simulation, making room to store another one.

Resources	Method	Req. body	Status	Resp.body	Meaning
/simulation	POST	NFV	200		The simulation is up and running
/simulation	GET		200	Hosts	XML describing the physical topology
/simulation	DELETE		200		The simulation has been stopped

5.2 Neo4j integration design

One of the most common architecture in software applications is the Layered Application Architecture which splits a system in layers distinct from each other. Every level has its own role and communicates with the others without creating dependency or interferences. This approach simplifies the code management and its evolution through time. Usually, the number of the layers in this type of architecture is three:

- **Presentation layer**, which defines the GUI, the user interaction input acquisition and data visualization
- **Application layer**, which basically represents the business logic and implements all the functionalities that can also be invoked by other services.
- **Data management layer**, which deals with data persistence and data access towards a data source (e.g. a database).

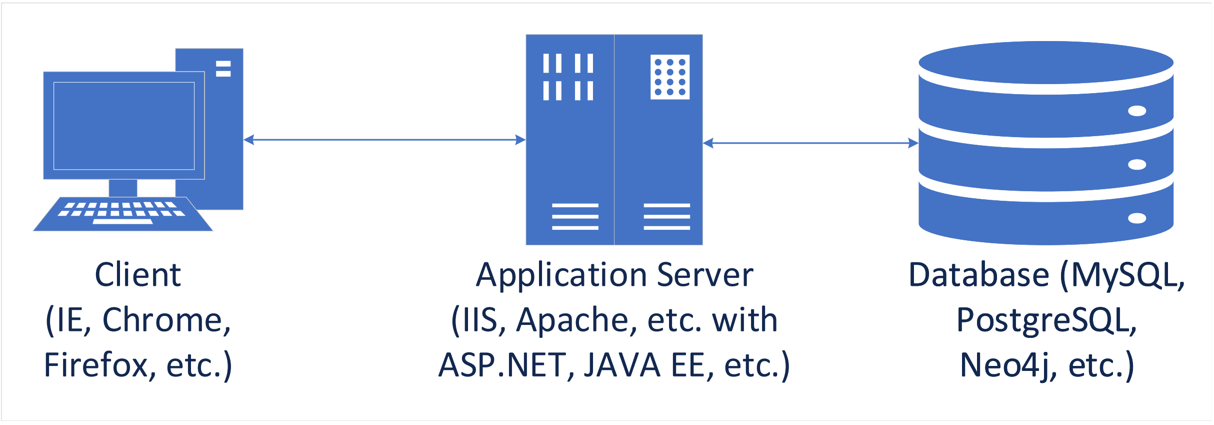


Figure 5.4. 3 Tiered Architecture

At the moment, Verifoo represents the application layer and there is no need at all for a data management layer since all the information needed for its execution is contained in the input XML. However, being able to save a state for a series of requests could open up new possibilities of interactions. Moreover, storing information can reduce the amount of data transmitted in every HTTP request, avoiding the repetition of those elements that are intrinsically disinclined to change, like a network infrastructure. The network services themselves could be saved and categorized to be retrieved afterwards. Verifoo could operate as a mere intermediary and provide a seamless and secure access to data.

The solution explored in this thesis envisages the preliminary introduction of the Neo4j database. The choice of this particular type of non-relational database comes precisely from its intrinsic nature since, as described in section 3.2.5, Neo4j is a graph database management system which fits perfectly the structure of network services and infrastructures. Moreover, it natively supports graph path finding algorithms that can be used to find the shortest path or perform other operation directly on the graphs structure [13]. Currently, given the explorative nature of the thesis task, Verifoo exploits the JDBC driver for the Neo4j environment and the queries are implemented at a low-level. If this part takes a more prominent role in the project, it is advisable to use the Neo4j-OGM library [14]. An OGM (Object Graph Mapper) provides the support to automatically persists annotated domain objects and references in nodes and relationships in a graph (it is the equivalent of an ORM for the relational databases).

Neo4j can be used either in embedded mode or in server mode. In this case, the approach that has been chosen is the server mode where the Neo4j process is independent from the Verifoo application and is accessed through the dedicated driver using the Cypher Query Language. This approach encourages a micro-service architecture favouring a higher dynamicity.

5.3 Verifoo workflow

As described in previous chapter, Verifoo fits into an NFV framework to work as a verification and deployment service, coordinated by the action of an orchestrator. In this section the workflow of this interaction will be presented. A high-level schema of the implemented functionalities can be seen in Figure 5.5.

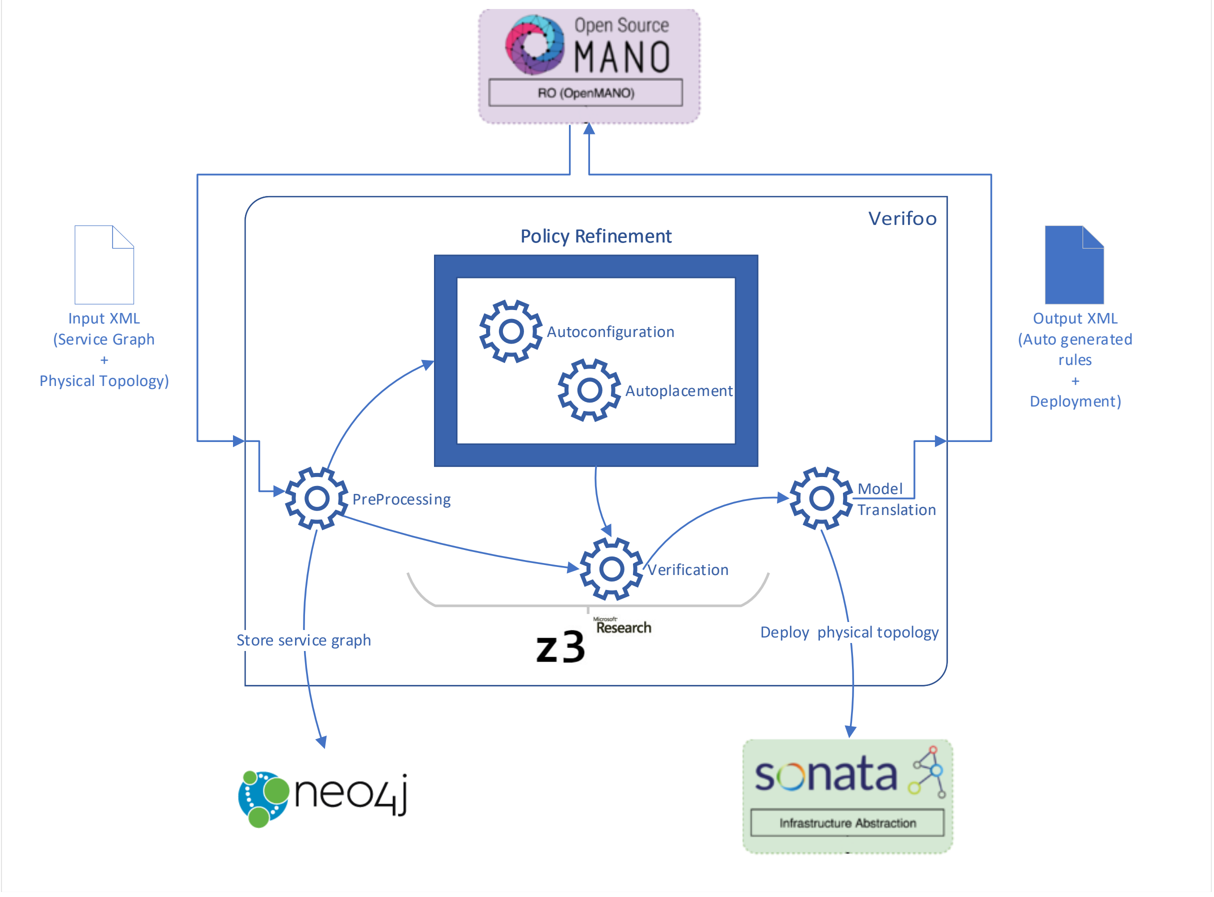


Figure 5.5. Verifoo Workflow

How each of the task depicted in the image has been implemented will be described in depth in Chapter 6. In this chapter a more abstract overview will be delineated, instead.

The input received by Verifoo from the orchestrator describes a deployment problem of interconnected VNFs. Therefore, it includes the network service graph and the physical topology which contains all the information about the available resource present on each machine and how the physical links connect them. More indications can be given regarding the requirements for the various VNFs (e.g. how much disk storage they need, how much memory, etc.). The network service can be a graph arbitrarily complex. To highlight the specific sequence of nodes that packets will traverse is also possible to specify network forwarding paths. This possibility is compliant with a service conceived for an SDN architecture in which the same graph can be used to process different kind of flows simply instructing the switch to follow different paths. The input also contains a series of high-level policies that express a reachability condition between two endpoints (e.g. if a client is able to reach or not a server). These policies will characterize the service verification whose satisfaction is the precondition before the deployment can be computed.

The input is pre-processed to transform the XML elements that represent the VNFs in the right set of FOL formulas that model their behaviour in z3. Also the policies are inserted in the z3 environment as constraints. In addition, the topology is explored to understand the various deployment possibilities. During this phase, the information can be stored in Neo4j to build a network state database.

Following the pre-processing task, the execution can follow two distinct routes. Each exploits a different module which solves a different problem:

- the *verification* module, which checks if the constraints introduced by the policies are satisfied in the described service graph
- the *policy refinement* module, which reverses the previous point, generating configurations for those VNFs that requires it, in order to have a service graph that satisfies the constraints introduced by the policies. In addition, it also removes from the service graph those VNFs that are not necessary for the final solution (i.e. their configuration results empty even after the computation). The first task is mainly executed by the auto-configuration sub-module, while the auto-placement sub-module takes care of the second one.

Both modules produce a z3 model which contains information about the optimal deployment and the generated configurations, if there were any. This information needs to be transferred into the actual Java objects and finally into the XML that will be returned. After having a solution for the deployment, a running simulation of the topology can be instantiated in the SONATA environment.

The output XML is therefore a repetition of what has been received in input with the addition of new information about the deployment and the configurations. This file can be sent back to the orchestrator that can exploit the new knowledge to instruct a VIM on how to deploy the network service and inform a VNF Manager on which configuration needs to be injected.

Chapter 6

Verifoo Extensions Implementation

As anticipated in the previous chapters, Verifoo is a tool that performs joint optimal placement computation and formal verification of network properties (policies). Its early development state gives the opportunity to further refine it, reducing the gap with a possible future implementation in an NFV MANO framework. Its integration in a framework such as Open Source MANO, can be seen as a future development path that can follow the guidelines already traced by the general operating idea behind Verifoo as presented in previous chapters.

All of the following work has to be seen as an extension of the one presented in the Verifoo published paper [9] where can be found its operating principles. Based on Verigraph, Verifoo describes the forwarding behaviour of virtual middleboxes, setting optimization objectives, using First Order Logic (FOL) formulas which are computed by z3. While the network model is described using hard clauses, that are formulas that must be satisfied, the optimization objective are formalized using soft clauses, meaning that they can also be not satisfied at the cost of a specified weight penalty. Z3 provides a solution that satisfy all the hard clauses and introduces the minimal penalty for the unsatisfied soft clauses. In this chapter all the modification made on the software are described. At first, the input XML format required by Verifoo will be explained in details in section 6.1. After that, section 6.2 explains how the integration with the SONATA module has been achieved and how it is possible to use a REST API to interact with it through Verifoo. In section 6.3 the initial Verifoo layer for data persistence using Neo4j is described. Finally, in section 6.4 all the improvements introduced in the Verifoo model are described in details.

6.1 XML Input Format

In this section, with the help of some examples, will be presented a complete description of the input XML needed by Verifoo in order to perform its evaluation. In the next paragraph, each element of the XML will be described in details. An example of a complete input file can be the following:

Listing 6.1. XML Example

```
<NFV>
  <graphs>
    <graph id="0">
     <node functional_type="WEBCLIENT" name="nodeA">
        <neighbour name="node1"/>
        <configuration description="A simple description" name="confA">
          <webclient nameWebServer="nodeB"/>
        </configuration>
      </node>
      <node functional_type="FIREWALL" name="node1">
        <neighbour name="nodeA"/>
        <neighbour name="nodeB"/>
        <configuration description="A simple description" name="conf1">
          <firewall>
            <elements>
              <source>nodeA</source>
              <destination>nodeB</destination>
              <src_port>5000</src_port>
              <dst_port>80</dst_port>
            </elements>
          </firewall>
        </configuration>
      </node>
      <node functional_type="NAT" name="node2">
        <neighbour name="node1"/>
        <neighbour name="nodeB"/>
        <configuration description="A simple description" name="conf3">
         <nat>
            <source>nodeA</source>
         </nat>
        </configuration>
      </node>
      <node functional_type="WEBSERVER" name="nodeB">
        <neighbour name="node1"/>
        <configuration description="A simple description" name="confB">
          <webserver>
            <name>nodeB</name>
          </webserver>
        </configuration>
      </node>
    </graph>
  </graphs>
  <Constraints>
      <NodeConstraints>
            <NodeMetrics node="node1" memory="2" reqStorage="38"/>
      </NodeConstraints>
      <LinkConstraints>
            <LinkMetrics src="node1" dst="node2" reqLatency="10"/>
      </LinkConstraints>
  </Constraints>
  <PropertyDefinition>
        <Property graph="0" name="IsolationProperty" src="nodeA" dst="nodeB">
            <HTTPDefinition url="polito.it" body="weapons"/>
        </Property>
  </PropertyDefinition>
  <Hosts>
    <Host name="hostA" cpu="1000" cores="2" memory="4" diskStorage="10" type="CLIENT"
        ↪ fixedEndpoint="nodeA"/>
    <Host name="host1" cpu="3000" cores="16" memory="16" diskStorage="50" maxVNF="1"
        ↪ type="MIDDLEBOX">
        <SupportedVNF functional_type="CACHE"/>
    </Host>
    <Host name="host2" cpu="4000" cores="4" memory="16" diskStorage="20" maxVNF="2"
        ↪ type="MIDDLEBOX">
        <SupportedVNF functional_type="FIREWALL"/>
    </Host>
    <Host name="host3" cpu="3000" cores="8" memory="16" diskStorage="10" maxVNF="3"
        ↪ type="MIDDLEBOX">
        <SupportedVNF functional_type="CACHE"/>
        <SupportedVNF functional_type="FIELDMODIFIER"/>
    </Host>
    <Host name="hostB" cpu="1000" cores="2" memory="2" diskStorage="10" type="SERVER"
        ↪ fixedEndpoint="nodeB"/>
  </Hosts>
  <Connections>
    <Connection sourceHost="hostA" destHost="host1" avgLatency ="2"/>
    <Connection sourceHost="host1" destHost="host2" avgLatency ="10"/>
    <Connection sourceHost="host2" destHost="hostB" avgLatency ="5"/>
  </Connections>
  <NetworkForwardingPaths>
    <Path id="0">
        <pathNode name="nodeA"/>
        <pathNode name="node1"/>
        <pathNode name="node2"/>
        <pathNode name="nodeB"/>
    </Path>
  </NetworkForwardingPaths>
  <ParsingString></ParsingString>
</NFV>
```

NFV

NFV is the root element of the XML schema, it contains:

- A list of **Graphs** that represent the network services that will be deployed
- A list of **Constraints**
- A **Property Definition** element that contains a list of **Properties** that express the policies that will be checked.
- A list of **Hosts** that form the physical topology
- A list of **Connections** between hosts
- An optional **NetworkForwardingPaths** element that allows the definition of all the paths that a packet flow will traverse.
- An optional **Parsing String** that is the raw output of Verifoo, it is necessary for a REST API.

Graph

A Graph element represents a requested service graph that will be deployed in the network. Verifoo can check and deploy multiple graphs but it's important to notice that is mandatory to have at least one client and one server in each of the graphs, otherwise, an exception will be thrown.
Graph is characterized by:

- A *unique* **ID**
- A list of **Nodes**

Listing 6.2. Graphs Example

```
<graphs>
    <graph id="0">
      <node ...>
      ...
    </graph>
    <graph id="1">
      ...
    </graph>
</graphs>
```

Node

A Node is a logical network element that corresponds to a network function. A node is characterized by:

- A *unique* **name**
- A **functional_type** attribute that represents the network function of the node
- A List of **neighbour** through which is possible to define the topology of the service graph (i.e. how the nodes are connected together)
- A **configuration** for the specific functional type

Listing 6.3. Node Example

```xml
<node functional_type="FIREWALL" name="node1">
    <neighbour name="nodeA"/>
    <neighbour name="node2"/>
    <neighbour name="node3"/>
    <configuration ...>
        ....
    </configuration>
</node>
```

Functional Type

The functional type of a node is an enumeration and can be one of the following:

- **FIREWALL**
- **ENDHOST**
- **ANTISPAM**
- **CACHE**
- **DPI**
- **MAILCLIENT**
- **MAILSERVER**
- **NAT**
- **VPNACCESS**
- **VPNEXIT**
- **WEBCLIENT**
- **WEBSERVER**
- **FIELDMODIFIER**

Configuration

In this section, different type of supported configurations are described. The configuration element must coincide with the functional type specified in the attribute of the node element, otherwise, an exception will be thrown. A configuration is characterized by an *unique* name and by an *optional* description. Below, for each type of configuration, an essential description, needed to understand how Verifoo works, will be provided. For further details, refer to the Verigraph documentation [8].

EndHost

An EndHost is a particular type of client whose configuration allows to define various fields of the packet that will be generated by the node. The settable fields are:

- the body of the packet, useful for the DPI model
- the sequence number
- the protocol (HTTP request or response, POP3 request or response)
- the email_from field, useful for the antispam model
- the url that represents the application level destination, useful for the cache model
- the options
- the destination

Listing 6.4. End Host Configuration Example

```xml
<configuration description="A simple description" name="conf2">
  <endhost body="thisisarequest"/>
</configuration>
```

Firewall

A Firewall configuration contains a list of ACLs (elements) that represents the flow of packets that will be blocked. In the ACL is mandatory to define the source node and the destination node; optionally also a source port and a destination port can be defined.

Listing 6.5. Firewall Configuration Example

```xml
<configuration description="A simple description" name="conf1">
  <firewall>
    <elements>
      <source>nodeC</source>
      <destination>nodeD</destination>
      <src_port>5000</src_port>
      <dst_port>80</dst_port>
    </elements>
  </firewall>
</configuration>
```

It is also possible to define a firewall with no configuration and in this case Verifoo will calculate one that satisfies the requested policies, if possible. This configuration will be present in the output XML provided by Verifoo.

Listing 6.6. Firewall Without Configuration Example

```xml
<configuration description="A simple description" name="conf1">
  <firewall/>
</configuration>
```

Cache

The configuration element of a cache contains a list of resources that represent the nodes that will be served by the cache (usually they are all the upstream clients in the network service topology).

Listing 6.7. Cache Configuration Example

```xml
<configuration description="A simple description" name="conf3">
  <cache>
    <resource>clientA</resource>
  </cache>
</configuration>
```

NAT

A NAT configuration contains a list of internal nodes specified in the source elements, whose packets will be translated following the normal NAT behaviour (also in this case, the source elements are usually all the upstream clients in the topology).

Listing 6.8. NAT Configuration Example

```xml
<configuration description="A simple description" name="conf4">
 <nat>
    <source>clientA</source>
 </nat>
</configuration>
```

DPI

A DPI configuration contains a list of notAllowed elements that define the strings that, if found in the body of a packet, will make the DPI drop that packet.

Listing 6.9. DPI Configuration Example

```xml
<configuration description="A simple description" name="conf2">
  <dpi>
    <notAllowed>SomeString</notAllowed>
  </dpi>
</configuration>
```

Like for the firewall, it is possible to have a DPI with no configuration which will be provided by Verifoo based on the desired policies.

Listing 6.10. DPI Without Configuration Example

```xml
<configuration description="A simple description" name="conf1">
  <dpi/>
</configuration>
```

Antispam

An Antispam configuration contains a list of source nodes that represent the blacklisted nodes. A packet will be dropped if its email_from field is equal to one of the blacklisted elements

Listing 6.11. Antispam Configuration Example

```xml
<configuration description="A simple description" name="conf5">
 <antispam>
    <source>nodeA</source>
 </antispam>
</configuration>
```

Like for the firewall and the DPI, also an antispam can be declared with no configuration and, following the same pattern, Verifoo will come up with one that satisfies the policies.

Listing 6.12. Antispam Without Configuration Example

```xml
<configuration description="A simple description" name="conf1">
  <antispam/>
</configuration>
```

MailServer

A Mail Server configuration contains the Mail Server names (more than one name is possible).

Listing 6.13. MailServer Configuration Example

```xml
<configuration description="A simple description" name="confB">
  <mailserver>
    <name>nodeB</name>
  </mailserver>
</configuration>
```

MailClient

A Mail Client configuration contains the Mail Server name. The use of a mail client, instead of a generic endhost, sets the protocol of all the packets sent from the node to be POP3 requests, requiring also that all the packets received are POP3 responses.

Listing 6.14. MailClient Configuration Example

```
<configuration description="A simple description" name="confB">
    <mailclient mailserver="nodeB"/>
</configuration>
```

WebServer

A Web Server configuration contains the Web Server names (more than one name is possible).

Listing 6.15. WebServer Configuration Example

```
<configuration description="A simple description" name="confB">
  <webserver>
    <name>nodeB</name>
  </webserver>
</configuration>
```

WebClient

A Web Client configuration contains the Web Server name. As for the mail client, the use of a web client, instead of a generic endhost, sets the protocol of all the packets sent from the node to be HTTP requests, expecting to receive only HTTP responses.

Listing 6.16. Web Client Configuration Example

```
<configuration description="A simple description" name="confB">
    <webclient webserver="nodeB"/>
</configuration>
```

VpnAccess

A VpnAccess configuration contains the VpnExit name.

Listing 6.17. VpnAccess Configuration Example

```
<configuration description="A simple description" name="conf1">
  <vpnaccess vpnexit="node2" />
</configuration>
```

VpnExit

A VpnExit configuration contains the VpnAccess name.

Listing 6.18. Vpn Exit Configuration Example

```
<configuration description="A simple description" name="conf2">
  <vpnexit vpnaccess="node2"/>
</configuration>
```

Constraints

The Constraints element encloses a **NodeConstraints** element, that contains all the node requirements, and a **LinkConstraints** element that holds the requirements for the links between the nodes.

Listing 6.19. Constraints Example

```
<Constraints>
     <NodeConstraints>
          <NodeMetrics node="node1" nrOfOperations="1000" maxNodeLatency="10" memory=
             ↪ "2" reqStorage="38" cores="1"/>
          <NodeMetrics node="node2" nrOfOperations="1000" maxNodeLatency="20" memory=
             ↪ "2" reqStorage="20" cores="2" optional="true"/>
     </NodeConstraints>
     <LinkConstraints>
          <LinkMetrics src="node1" dst="node2" reqLatency="10"/>
     </LinkConstraints>
 </Constraints>
```

NodeConstraints

The NodeConstraints element is a list of NodeMetrics that characterize a specific node, referenced through the **node** attribute, with the following information:

- The estimated **nrOfOperations**, used to calculate the latency that the node itself will have after the deployment on a host that has a known computational power
- The **maxNodeLatency** requirement; used in conjunction with the previous attribute, imposes a constraint on the maximum latency (expressed in ms) the node can introduce after the deployment on a host
- The **memory** that the node requires to operate
- The **reqStorage** that represents the disk requirement of the node
- The **cores** that represent the minimum number of cores a host can have in order to be selected as a valid deployment host for the node
- The **optional** attribute, that is used to inform Verifoo that a node can be omitted in the final deployment. This attribute is strictly related with the auto-placement feature which will be described in a later chapter.

All of the previous attributes are optional, except for the node attribute, allowing a customizable level of detail. When an attribute is not specified, the use of the default values ensures that there will be no restrictions on possible deployment scenarios (i.e. the hosts can contain as many node as possible without worrying about the resource shortage). Hence, the default value assigned to the memory, the reqStorage and cores is zero.

LinkConstraints

The LinkConstraints element is a list of LinkMetrics that characterize a specific link between two nodes. It has the following attributes:

- The **src** and **dst** attributes that refer to the names of the node elements in a graph
- The **reqLatency** attribute that imposes a constraint on the maximum latency of the physical connection that links together the hosts on which the source and destination nodes will be deployed (if the nodes are deployed on the same host, the resultant latency is 0).

Property Definition

The **PropertyDefinition** element represents a list of properties that will be checked by Verifoo for a specific graph. Currently, there are two properties that are supported by Verifoo, the isolation and the reachability property. The **Property** element is characterized by some mandatory attributes that are:

- A **Graph** attribute that represents the graph on which the property will be checked.
- The **src** and the **dst** attributes, that tell Verifoo between which nodes the property needs to be verified
- A **Name** that is an enumeration and represents the property that will be checked. With IsolationProperty, Verifoo checks if there is no packet that from the source can reach the destination. With ReachabilityProperty, Verifoo checks if at least one packet from the source can reach the destination.

If no other attributes are used, Verifoo checks if the properties are satisfied for a generic flow of packets. If there is a particular need to also specify these other fields, there are some optional sub-elements and attributes that can be used:

- A **src_port** and a **dst_port** attribute that will force the property to be checked using packets that have those specific values in the TCP ports fields
- An **HTTPDefinition** sub-element that forces the sent packets to be HTTP requests. In this sub-element can also be specified all the relevant fields of a HTTP packet, such as the body and the URL.
- A **POP3Definition** sub-element that forces the sent packets to be POP3 requests. Here can also be specified some other characteristics of a POP3 packet, such as the body and the email_from field.

The HTTPDefinition and POP3Definition are used alternatively and should match the type of the protocol specified by the source node defined in the property, if it specifies one, in order to avoid contradictions. The sub-element approach in the Property allows an easy introduction of other types of packets in case of future needs.

Listing 6.20. Property Definition Example

```xml
<PropertyDefinition>
      <Property graph="0" name="IsolationProperty" src="nodeA" dst="nodeB" src_port="
         ↪ 5000" dst_port="80">
         <HTTPDefinition url="polito.it" body="weapons"/>
      </Property>
      <Property graph="0" name="ReachabilityProperty" src="nodeC" dst="nodeD"
         ↪ src_port="3000" dst_port="110">
         <POP3Definition email_from="polito" body="weapons"/>
      </Property>
</PropertyDefinition>
```

At the end of the computation, Verifoo enriches the Property element with an additional attribute, the **isSat** attribute, that informs if the property is satisfied or not.

Host

An host is a physical machine present in the network infrastructure. An host is characterised by:

- A *unique* **Name**
- A **type** to distinguish between clients, servers and host on which there will be deployed middle-boxes
- The **cpu** attribute that represents the host processor frequency expressed in GHz

- The **cores** available on the host
- The **diskStorage** attribute that represents the available space in the mass storage of the host
- The **memory** available on the host
- The **maxVNF** that represents the maximum number of nodes that can be deployed on the host
- The **fixedEndpoint** attribute that tells Verifoo where to put a specific client or server. While is mandatory for a client to have explicitly a host on which to be deployed, for a server it is not. If a server doesn't have a fixed host on which to be deployed, it will be deployed together with the other nodes in the physical topology
- The **active** attribute, imposed by Verifoo at the end of its computation. It's a boolean attribute that is true if at least one node has been deployed on the host.

After the execution of Verifoo, the host will also contain a list of **NodeRef** elements that is a list of the nodes that Verifoo decided to deploy on that host. As a consequence of the deployment, the host resources will be updated subtracting the requirement of the nodes, as specified in the NodeMetrics element.

Listing 6.21. Hosts Example

```
<Host name="hostA" cpu="1000" cores="2" memory="4" diskStorage="10" type="CLIENT"
    ↪ fixedEndpoint="nodeA"/>
    <Host name="host1" cpu="3000" cores="16" memory="16" diskStorage="50" maxVNF="1"
        ↪ type="MIDDLEBOX" active="true">
        <SupportedVNF functional_type="FIREWALL"/>
        <SupportedVNF functional_type="CACHE"/>
        <SupportedVNF functional_type="FIELDMODIFIER"/>
        <NodeRef node="node1"/>
        <NodeRef node="node2"/>
    </Host>
    <Host name="host2" cpu="4000" cores="4" memory="16" diskStorage="20" maxVNF="2"
        ↪ type="MIDDLEBOX" active="true">
        <SupportedVNF functional_type="FIREWALL"/>
        <NodeRef node="node3"/>
    </Host>
    <Host name="host3" cpu="3000" cores="8" memory="16" diskStorage="10" maxVNF="3"
        ↪ type="MIDDLEBOX">
        <SupportedVNF functional_type="CACHE"/>
        <SupportedVNF functional_type="FIELDMODIFIER"/>
    </Host>
    <Host name="hostB" cpu="1000" cores="2" memory="2" diskStorage="10" type="SERVER"
        ↪ fixedEndpoint="nodeB" active="true">
        <NodeRef node="nodeB"/>
  </Hosts>
```

Connection

A **connection** element represents a unidirectional physical connection between two hosts. It's characterized by:

- A **source** and a **destination**
- The **avgLatency** attribute that represents the average latency on the physical link between the source and the destination.

Listing 6.22. Connections Example

```
<Connections>
   <Connection sourceHost="host1" destHost="host2" avgLatency ="1"/>
   <Connection sourceHost="host1" destHost="host3" avgLatency ="10"/>
</Connections>
```

NetworkForwardingPaths

The NetworkForwardingPaths is an optional element that consists in a list of **Paths** each one of them containing a list of nodes, from a client to a server. This list represents the exact sequence of nodes a packet must traverse during the evaluation of a property characterized by that exact combination of client and server. If more than one path has the same client and server, the corresponding properties must be satisfied in all of them.

Listing 6.23. NetworkForwardingPaths Example

```
<NetworkForwardingPaths>
    <Path id="0">
        <pathNode name="nodeA"/>
        <pathNode name="node1"/>
        <pathNode name="node2"/>
        <pathNode name="nodeB"/>
    </Path>
    <Path id="1">
        <pathNode name="nodeC"/>
        <pathNode name="node3"/>
        <pathNode name="node1"/>
        <pathNode name="node2"/>
        <pathNode name="node3"/>
        <pathNode name="nodeD"/>
    </Path>
 </NetworkForwardingPaths>
```

Parsing String

This element contains the string that is obtained as the raw output of Verifoo execution. It is used only by the converter REST API of Verifoo.

Listing 6.24. An extract of ParsingString Example

```
<ParsingString>
...
(define-fun check_isolation_n_0_nodeA_nodeB () Node
  node5)
(define-fun integer_host1 () Int
  1)
(define-fun node3@host7 () Bool
  false)
(define-fun node3@host2 () Bool
  true)
  ....
</ParsingString>
```

6.2 SONATA integration implementation

In this section a complete description on how SONATA has been integrated with Verifoo will be given, following the general ideas outlined in section 5.1.

In order to work correctly, the Verifoo module needs to know the root directory of the SONATA libraries, which can be specified by setting the system property called *it.polito.verifoo.rest.sonata.rootDirectory*, otherwise it uses the default value "/home/sonata".

6.2.1 Service Graph Simulation

In this paragraph further details will be given on how the output from Verifoo is used to run the SONATA emulator. As anticipated in chapter 2, some modifications to the original SONATA code have been made in order to achieve an optimal deployment in the simulated network. In particular, the dummygatekeeper.py file, included in the library, has been modified in such a way that, when a service package is received, before attempting to deploy the network functions in the round-robin, default way, it first tries to call a function imported from a file called CustomPlacement.py, dynamically generated by Verifoo, that will provide the right deployment. If something goes wrong during this call, it tries deploying the nodes using the default method and then it continues with the normal execution of the program.
Along with the various steps of the algorithm used to run the simulation, there will be some example files that will refer to the following simple scenario in which the only possible deployment is to have each of the nodes of the service graph installed on a different host (n1 on h1, n2 on h3 and n3 on h3):

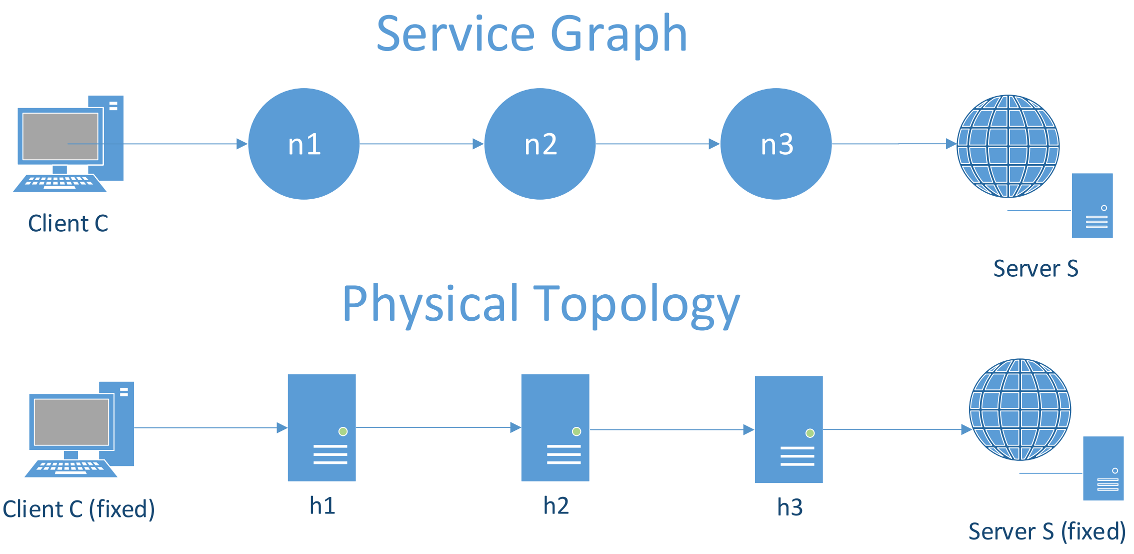


Figure 6.1. SONATA Example Topology

The first step is to start the simulation of the physical network and for this purpose, a python file is dynamically generated; following a syntax very similar to the containernet one, it describes all the hosts, with their available resources, and how they are connected, specifying also the delay of the connection. Lastly, in this file are also declared the REST API endpoint, useful to interact with the simulation via HTTP requests, and the SONATA Dummy Gatekeeper endpoint, needed to allow the deployment of the network functions through the upload of a specific package; these endpoints are then hooked to each of the host in the topology. It is important to notice that in the file the net object has its enable_learning attribute equal to false in order to prevent communication between all the hosts, as it would happen in a switched network, and enable the creation of paths based on entirely on the network service chaining, as it happens in an SDN network.

Listing 6.25. Extract of Topology file

```
def create_topology1():
    net = DCNetwork(controller=RemoteController, monitor=False, enable_learning=False)
    hostA = net.addDatacenter("hostA", metadata="{\"name\":\"hostA\", \"cores\":\"2\",
        ↪  \"cpu\":\"2\", \"memory\":\"4\", \"diskStorage\":\"10\", \"maxVNF\":\"null
        ↪ \", \"type\":\"CLIENT\", \"fixedEndpoint\":\"nodeA\", \"supportedVNF\":[]}"
        ↪ )
    host1 = net.addDatacenter("host1", metadata="{\"name\":\"host1\", \"cores\":\"4\",
        ↪  \"cpu\":\"1\", \"memory\":\"16\", \"diskStorage\":\"50\", \"maxVNF
        ↪ \":\"0\", \"type\":\"MIDDLEBOX\", \"supportedVNF\":[{ \"functionalType\":
        ↪ \"FIREWALL\"},{ \"functionalType\": \"CACHE\"},{ \"functionalType\": \"DPI
        ↪ \"}]}")
    ...
    net.addLink(hostA, host1, delay="1ms")
    net.addLink(host1, host2, delay="1ms")
    net.addLink(host2, host3, delay="1ms")
    net.addLink(host3, hostB, delay="1ms")
    rapi1 = RestApiEndpoint("0.0.0.0", 5001)
    rapi1.connectDCNetwork(net)
    rapi1.connectDatacenter(hostA)
    rapi1.connectDatacenter(host1)
    ...
    rapi1.start()
    sdkg1 = SonataDummyGatekeeperEndpoint("0.0.0.0", 5000, deploy_sap=True)
    sdkg1.connectDatacenter(hostA)
    sdkg1.connectDatacenter(host1)
    ...
    sdkg1.start()
```

The package pushed to the Dummy Gatekeeper is essentially an archive, that contains folders and files in a particular structure, that models the requested service. The archive is obtained through a specific SONATA CLI command that, in addition to the packaging, also performs the validation of the service descriptors assuring their correctness.

Listing 6.26. Package Folder Structure

```
service-folder/
        project.yml
        sources/
            nsd/
                nsd.yml
            vnf/
                node1/
                    node1-vnfd.yml
                node2/
                    node2-vnfd.yml
                ...
```

The second step consists in creating the CustomPlacement.py file. It is dynamically generated in the application classpath and then it is moved to the right SONATA library folder. It translates the information of the actual deployment computed by Verifoo into the instructions needed by the Dummy Gatekeeper in order to achieve that deployment, as it can be seen in the following example.

Listing 6.27. Placement file

```
class CustomPlacement(object):
    def place(self, nsd, vnfds, saps, dcs):
        vnfds['nodea']["dc"] = dcs['hostA']
        vnfds['node1']["dc"] = dcs['host1']
        vnfds['node2']["dc"] = dcs['host2']
        vnfds['node3']["dc"] = dcs['host3']
        vnfds['nodeb']["dc"] = dcs['hostB']
```

After this, for each node, a VNF descriptor file is generated. It models the docker that will run the chosen VNF applying all the information about the node metrics (memory, storage, etc.) if present in the Verifoo output XML, otherwise the default values of 1, which is the lowest accepted value, is used for each metric in order to prevent an exception during the SONATA validation process. This file also declares the network interfaces available in the docker that later will be used to connect the VNFs.

Listing 6.28. VNF Descriptor file (node2-vnfd.yml)

```yaml
descriptor_version: "vnfd-schema-01"
vendor: "eu.sonata-nfv"
name: "node2-vnf"
version: "0.1"
author: "Verifoo"
description: "VNF descriptor automatically generated for node2"
virtual_deployment_units:
  - id: "1"
    vm_image: "sonatanfv/sonata-empty-vnf"
    vm_image_format: "docker"
    resource_requirements:
      cpu:
        vcpus: 1
      memory:
        size: 1
        size_unit: "GB"
      storage:
        size: 1
        size_unit: "GB"
    connection_points:
      - id: "vdu01:cp00"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:cp01"
        interface: "ipv4"
        type: "internal"
virtual_links:
  - id: "input0"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:cp00"
      - "input0"
  - id: "output0"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:cp01"
      - "output0"
connection_points:
  - id: "input0"
    interface: "ipv4"
    type: "external"
  - id: "output0"
    interface: "ipv4"
    type: "external"
```

After all the VNF descriptors have been created, the service descriptor file is generated. This file contains the indications on how to connect the VNF dockers while, at the same time, a script file is populated with all the SONATA CLI commands needed to instruct the physical topology to enable the communication between the hosts that contains those dockers. The commands need only to specify the dockers that are being connected while the SONATA tool will set the appropriate rules in the physical topology.

Listing 6.29. Extract of Service Descriptor file (nsd.yml)

```
descriptor_version: "1.0"
vendor: "eu.sonata-nfv"
name: "sonata-service"
version: "0.1"
network_functions:
   - vnf_id: "nodea"
     vnf_vendor: "eu.sonata-nfv"
     vnf_name: "nodea-vnf"
     vnf_version: "0.1"
   - vnf_id: "node1"
     vnf_vendor: "eu.sonata-nfv"
     vnf_name: "node1-vnf"
     vnf_version: "0.1"
    ...
virtual_links:
  - id: "link-nodeA_to_node1"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "nodea:output0"
      - "node1:input0"
  - id: "link-node1_to_node2"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "node1:output0"
      - "node2:input0"
    ...
```

Listing 6.30. Network Script file

```
#!/bin/sh
son-emu-cli network add -b -src nodea:output0 -dst node1:input0
son-emu-cli network add -b -src node1:output0 -dst node2:input0
son-emu-cli network add -b -src node2:output0 -dst node3:input0
son-emu-cli network add -b -src node3:output0 -dst nodeb:input0
```

When all the files are ready, the emulator is lunched with the prepared topology file. The basic folder structure required by the SONATA tool, is created and filled with the descriptor files and then it is fed into the SONATA validator that creates the package that will be pushed to the Dummy Gatekeeper which will deploy the service graph following the Verifoo placement. At this point the simulation is up and running and can be interacted with through the already available REST API of the SONATA libraries.

6.2.2 Description of the simulation REST API

Following the guidelines described in section 5.1.2, a low-level usage example of the REST API will be given, emphasizing the different type of information exchanged during the interaction.

The first step of the interaction is to save the physical topology and create a running simulation of the service in the SONATA environment. Therefore, the fist operation is execute a POST request containing the result of the Verifoo execution (i.e. the physical topology with the information about the deployment of the requested service). Performing a GET without a running simulation will return a server error since there is no information to be retrieved. Consecutive POST will overwrite the simulation previously stored since for now only one topology at a time can be saved.
Example POST request

- **POST** http://localhost:8080/verifoo/rest/simulation

- Accept: **APPLICATION_XML**;
- Content: XML file with the desired service chain and the physical topology with integrated deployment information (it's the output of the Verifoo deployment REST API).

An example of the content of a POST request could be the following:

Listing 6.31. POST body Example

```xml
<NFV>
    <graphs>
        <graph id="0">
            <node name="nodeA" functional_type="MAILCLIENT">
                <neighbour name="node1"/>
                <configuration name="confA" description="A simple description">
                    <mailclient mailserver="nodeB"/>
                </configuration>
            </node>
            <node functional_type="FIREWALL" name="node1">
                <neighbour name="nodeA"/>
                <neighbour name="node2"/>
                <configuration description="A simple description" name="conf1">
                    <firewall>
                        <elements>
                            <source>nodeC</source>
                            <destination>nodeD</destination>
                        </elements>
                    </firewall>
                </configuration>
            </node>
            <node functional_type="CACHE" name="node2">
                <neighbour name="node1"/>
                <neighbour name="nodeB"/>
                <configuration description="A simple description" name="conf2">
                    <cache>
                        <resource>nodeA</resource>
                        <resource>node1</resource>
                    </cache>
                </configuration>
            </node>
            <node name="nodeB" functional_type="MAILSERVER">
                <neighbour name="node2"/>
                <configuration name="confB" description="A simple description">
                    <mailserver>
                        <name>nodeB</name>
                    </mailserver>
                </configuration>
            </node>
        </graph>
    </graphs>
    <Constraints>
        <NodeConstraints/>
        <LinkConstraints/>
    </Constraints>
    <PropertyDefinition>
        <Property name="ReachabilityProperty" graph="0" src="nodeA" dst="nodeB" isSat="
            ↪ true"/>
    </PropertyDefinition>
    <Hosts>
        <Host name="hostA" cpu="2" cores="2" diskStorage="10" memory="16" type="CLIENT"
            ↪  fixedEndpoint="nodeA"/>
        <Host name="host1" cpu="1" cores="4" diskStorage="40" memory="16" maxVNF="1"
            ↪ type="MIDDLEBOX" active="true">
            <SupportedVNF functional_type="FIREWALL"/>
            <SupportedVNF functional_type="CACHE"/>
            <NodeRef node="node1"/>
        </Host>
        <Host name="host2" cpu="3" cores="8" diskStorage="0" memory="16" maxVNF="3"
            ↪ type="MIDDLEBOX" active="true">
            <SupportedVNF functional_type="FIREWALL"/>
            <SupportedVNF functional_type="CACHE"/>
            <NodeRef node="node2"/>
        </Host>
        <Host name="host3" cpu="4" cores="4" diskStorage="50" memory="16" maxVNF="4"
            ↪ type="MIDDLEBOX">
            <SupportedVNF functional_type="FIREWALL"/>
            <SupportedVNF functional_type="CACHE"/>
        </Host>
        <Host name="hostB" cpu="2" cores="2" diskStorage="10" memory="16" type="SERVER"
            ↪  fixedEndpoint="nodeB" active="true">
            <NodeRef node="nodeB"/>
        </Host>
    </Hosts>
    <Connections>
        <Connection sourceHost="hostA" destHost="host1" avgLatency="1"/>
        <Connection sourceHost="host1" destHost="host2" avgLatency="1"/>
        <Connection sourceHost="host2" destHost="host3" avgLatency="1"/>
        <Connection sourceHost="host3" destHost="hostB" avgLatency="1"/>
    </Connections>
</NFV>
```

At this point it is possible to perform a GET to retrieve the stored information about the physical topology.
Example GET request

- **GET** http://localhost:8080/verifoo/rest/simulation

Example GET response

- 200: **OK**
- Content-Type: **APPLICATION_XML**;
- Content: An XML file containing the hosts of the physical topology previously stored with a POST request.

The GET result in the previous example would be:

Listing 6.32. POST body Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Hosts>
    <Host name="hostA" cpu="2" cores="2" diskStorage="10" memory="16" type="CLIENT"
        ↪ fixedEndpoint="nodeA"/>
    <Host name="host1" cpu="1" cores="4" diskStorage="40" memory="16" maxVNF="1" type="
        ↪ MIDDLEBOX" active="true">
        <SupportedVNF functional_type="FIREWALL"/>
        <SupportedVNF functional_type="CACHE"/>
        <NodeRef node="node1"/>
    </Host>
    <Host name="host2" cpu="3" cores="8" diskStorage="0" memory="16" maxVNF="3" type="
        ↪ MIDDLEBOX" active="true">
        <SupportedVNF functional_type="FIREWALL"/>
        <SupportedVNF functional_type="CACHE"/>
        <NodeRef node="node2"/>
    </Host>
    <Host name="host3" cpu="4" cores="4" diskStorage="50" memory="16" maxVNF="4" type="
        ↪ MIDDLEBOX">
        <SupportedVNF functional_type="FIREWALL"/>
        <SupportedVNF functional_type="CACHE"/>
    </Host>
    <Host name="hostB" cpu="2" cores="2" diskStorage="10" memory="16" type="SERVER"
        ↪ fixedEndpoint="nodeB" active="true">
        <NodeRef node="nodeB"/>
    </Host>
</Hosts>
```

At an time a DELETE request can be sent. This action stops the simulation and erases any information about the topology.
Example DELETE request

- **DELETE** http://localhost:8080/verifoo/rest/simulation

6.3 Neo4j Integration

The integration of the Neo4j database in Verifoo has been explored, however only a proof of concept has been introduced in the code. Currently, the Neo4j database is used more like a visualization tool made available for the developer. In fact, during the debugging, it could be very useful to have this kind of tool to check the consistency and the correctness of the input graphs included in the requests. The present section elaborates on this aspect. An automatic evolution of this feature could be using Neo4j as a database to implement a layer of data persistence.

As described in section 5.2, the Verifoo service and the Neo4j service are two separated process, potentially running in two separate machines and because of this Verifoo needs to know the address of the server and the credentials for the authentication. This information can be provided to Verifoo through specific system property that are the following:

- *it.polito.verifoo.rest.neo4j.neo4jURL*, following the pattern *address:port* for declaring the address
- *it.polito.verifoo.rest.neo4j.neo4jUsername* and *it.polito.verifoo.rest.neo4j.neo4jPassword* for specifying the credentials

If some of them are not configured, default values will be used (currently they are “127.0.0.1:7687” for the address, “neo4j” for the username and “password” for the password). These system properties can be set in the code or, for instance, in an ant script to avoid the recompiling, as shown in the following example.

Listing 6.33. Ant Script Example

```
<target name="main" description="Run main">
          <java classname="it.polito.verifoo.rest.main.Main" failonerror="true" fork=
              "yes">
              <sysproperty key="it.polito.verifoo.rest.neo4j.neo4jURL" value="
                  127.0.0.1:7687"/>
              <sysproperty key="it.polito.verifoo.rest.neo4j.neo4jUsername" value="
                  user"/>
              <sysproperty key="it.polito.verifoo.rest.neo4j.neo4jPassword" value="
                  pwd"/>
          </java>
    </target>
```

Using the dedicated drivers, Verifoo contacts the Neo4j database server at the specified address using the bolt protocol which is a “connection-oriented network protocol used for client-server communication in database applications that operates over a TCP connection or WebSocket” [15]. It has been initially developed precisely for Neo4j graph database. Of course, as the application expects to communicate with the bolt protocol, it’s important to specify a bolt endpoint in the URL property (usually it is automatically instantiated by the Neo4j server).
After having received an input XML composed by a certain number of graphs, using the information retrieved in the previously mentioned system properties, Verifoo tries to authenticate into the Neo4j database, If the connection is successful, Verifoo first runs a query to remove overlapping information (i.e. the graphs already stored that have the same id of the ones that are going to be committed). After cleaning, it runs the query that stores the graphs present in the input file as entities with the label *Graph*. It also stores the nodes of the graphs as entities with the label *Node* that have as properties all the metrics (memory, disk requirement, etc.) of that specific node. Each graph references the nodes contained in it with a relationship called *CONTAINS*. The links between the various nodes are represented as relationships named *CONNECTED_WITH*.
This information can be accessed afterwards using the Neo4j GUI having a result similar to the one in Figure 6.2. At the moment Neo4j is used only as a visualization tool for debugging purposes but this initial implementation is the first block that allows the developing of a series of features that, exploiting the same driver, can access all the available Neo4j APIs.

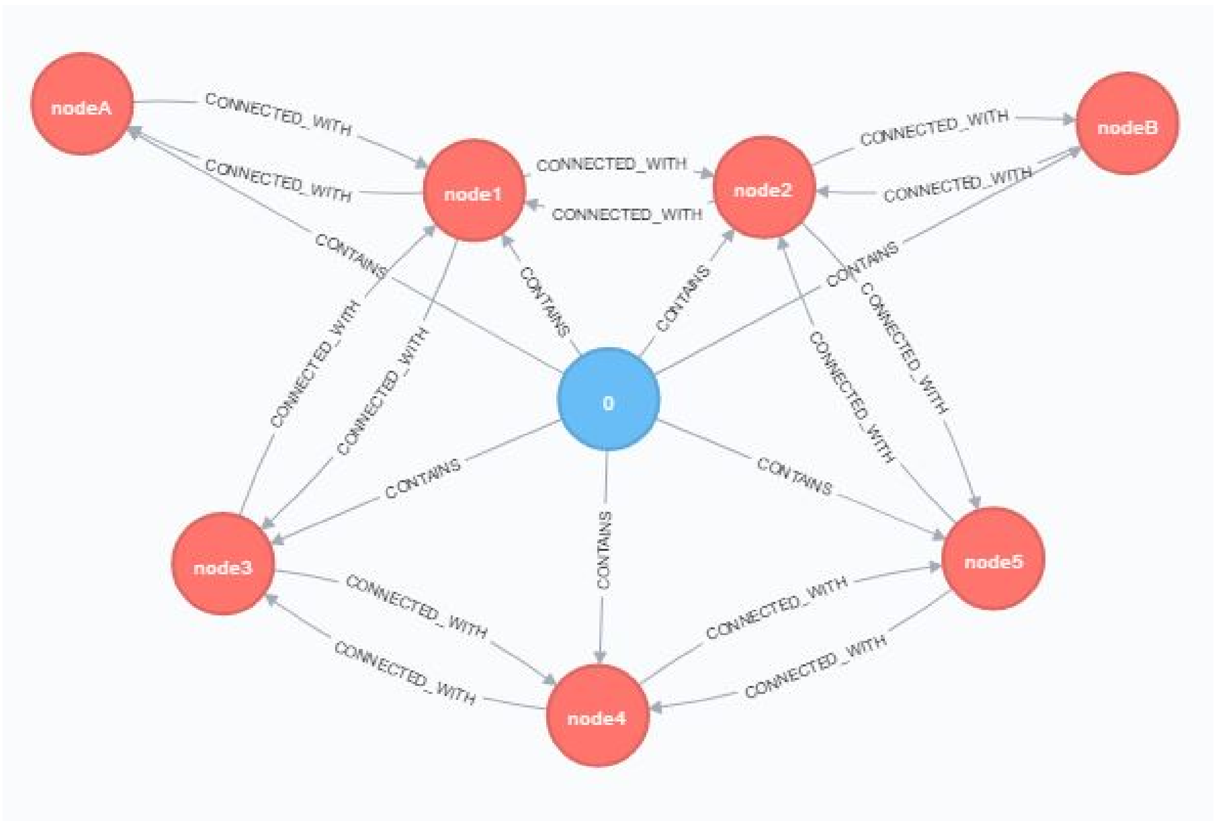


Figure 6.2. Neo4j Interaction Example

6.4 Verifoo Improvements

This section contains all the information about changes and improvements introduced in the Verifoo model and in the framework itself. Besides the basic aspects already described in previous chapters, more information will be given in following paragraphs regarding some of the concepts that are necessary to understand the reasons behind the improvements that have been made. In particular, the topics that will be discussed are:

- The use of a pre-processing task in Verifoo that allows the computation of the right deployment scenarios
- The new possibility of having more than one client and/or more than one server in a network service
- The addition of new constraints that formally model the new node and host metrics added in the XML schema
- The addition of a new policy, the isolation policy, that Verifoo can formally check
- The modifications made on the Verifoo network model in order to allow also the deployment of service graphs and not only of service chains
- The refinement of the firewall model in order to have a more realistic behaviour
- The addition of new functionalities, the auto-configuration and the auto-placement, that expand further the Verifoo capabilities, adding the possibility to perform a completely automated policy refinement (for now limited only to three VNFs)

6.4.1 Pre-Processing

One of the limitations Verifoo had, was the necessity to manually enumerate the variables that represented the possible deployments, together with the cost that those deployments would have had (in the form of the latency between each pair of sequential nodes). Obviously, this raised various issues on the scalability and the error-proneness of the approach. In this paragraph, it will be discussed the algorithm adopted in order to compute the right set of formulas using a completely automated pre-processing job. During this phase, the solution space is explored to define all the possible deployment scenarios leaving afterwards to z3, the task of choosing the optimal one taking also into account all the boundary conditions (e.g. the disk storage available on the host or the memory, etc.).
Initially, the algorithm considers the physical topology described in the input XML. It extracts from the topology all the paths that lead from all the client hosts, that are fixed in the topology, to all the server hosts, that may or may not be fixed. If no fixed server is specified, during the enumeration of the paths, in turn, each host in the topology that is not a client, is considered as a server. A client is considered to be fixed in the topology because usually, when requesting a service, its position is known a priori. On the contrary, the position of a server can be flexible since it can be both in a known location or included among the nodes that needs to be deployed. Given the extracted paths, the algorithm proceeds to evaluate the deployment of the service graph on each of them, extrapolating on which hosts a specific node can be deployed. The constraints that are considered in order to define a correct deployment are:

1. The position of the client node is fixed on the first host (which, for construction, is the client host), and the position of the server node is fixed on the last host (which, for construction, is the server host)
2. Each sequential node in the graph needs to be deployed either on the same host (in which case the latency between the two nodes is considered to be zero), or on hosts that are directly reachable (in which case the latency is considered to be equal to the latency of the physical connection between the two hosts)

About the reachability mentioned in 2, in reality all the hosts are usually directly reachable thanks to the routers that can deliver packets to any of those hosts. However, the need for an explicit direct reachability allows to include also the cases where two hosts are not connected (e.g. network failure) or should not be (e.g. for security reasons). Anyway, the direct reachability among all the hosts can be achieved with the declaration of a full mesh in the input XML (see the Chapter 3.1 to have further details on how this can be easily done). For each path, the pre-processing task uses a recursive algorithm that proceeds to deploy all the nodes in the service graph on the hosts, taking care that the order of the nodes is preserved, thus reducing significantly the solution space. In fact, a node will always be deployed on hosts that, in the path, are located after the ones on which its previous nodes are deployed, so not all the possibility are explored, as it will be better described in a next example. When it finds a configuration in which the previous two constraints are satisfied, the recursive algorithm backtracks and saves the information relative to the exact positioning of each node in the physical topology together with the latency towards the node that follows in the current configuration.
To visualize the approach better, a simple example will be presented, considering the requested service graph and the physical topology depicted in Figure 6.3:

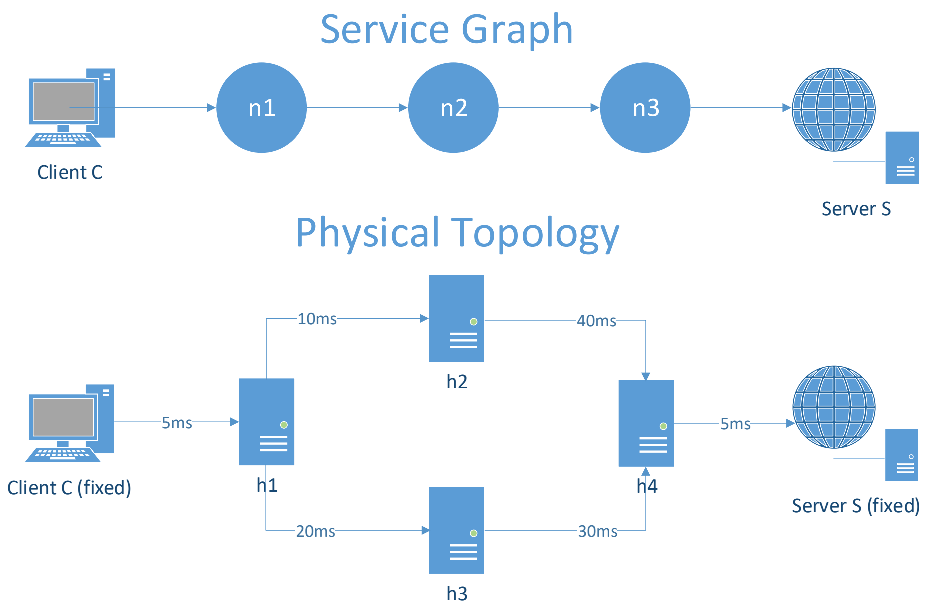


Figure 6.3. Simple Scenario

The first step is to extract from the physical topology all the paths that from a client bring to a server. In this case they are only two and are the ones listed in Figure 6.4:

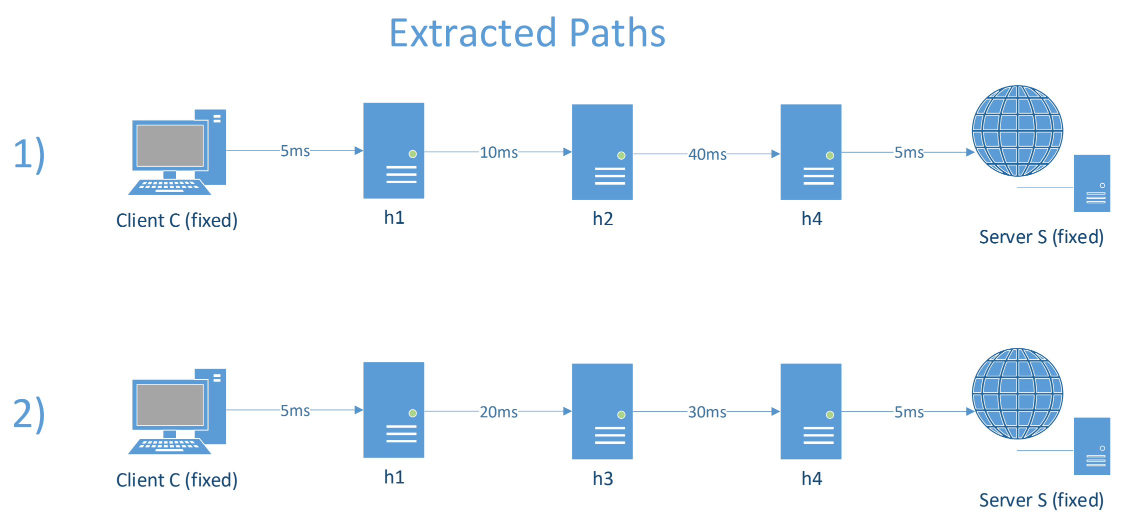


Figure 6.4. Extracted Paths

For each path, the execution of the recursive algorithm extracts the possible configurations that satisfy both constraints 1 and 2, as shown in Figure 6.5:

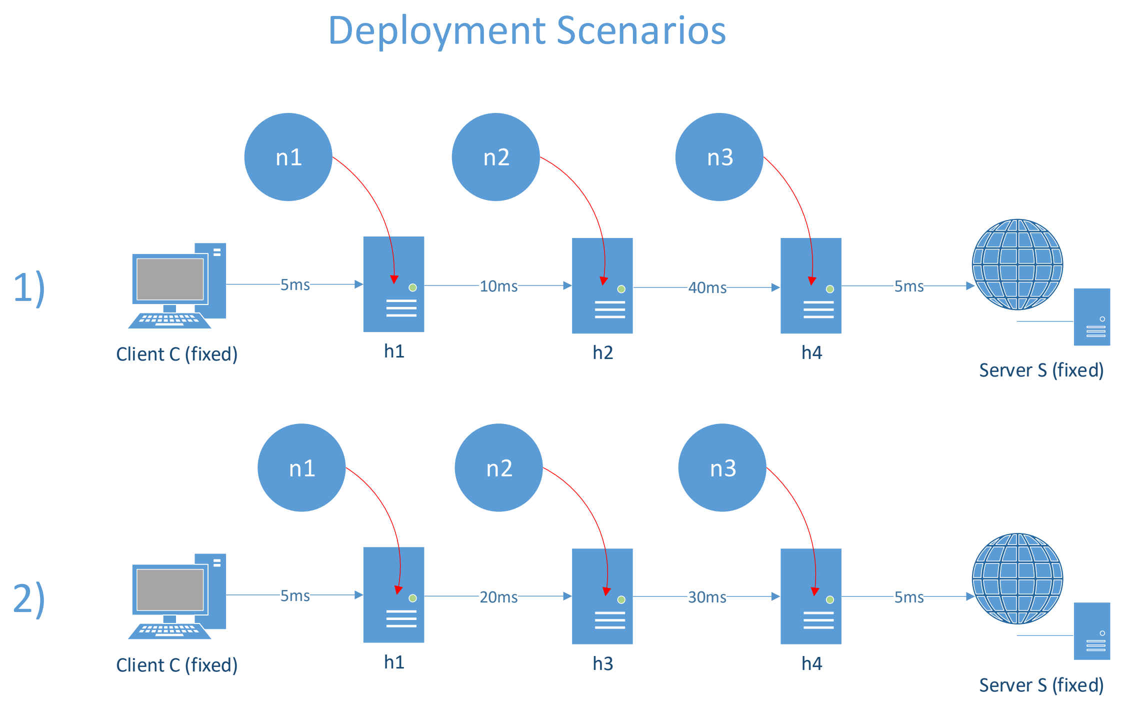


Figure 6.5. Possible configurations

At the end of the pre-processing job, the important obtained knowledge is that n1 can be deployed on h1, n2 can be deployed either on h2 or h3, and n3 can be deployed on h4. Moreover, also the latency of the links that connect the nodes after the deployment on the hosts is stored. So between C and n1, and between n3 and S, only one value is stored (5ms in both cases), while between n1 and n2, and between n2 and n3, two values for each couple (i.e. n1-n2, n2-n3) are stored because it depends on whether n2 will be deployed on h2 or h3, as will be further described in the example below. In detail, considering the scenario in which z3 chooses to deploy n2 on h2, between n1 and n2 the latency is equal to 10ms, while between n2 and n3 it is 40ms. In the configuration where z3 chooses to deploy n2 on h3, the latency is 20ms between n1 and n2 and 30 ms between n2 and n3. From here, the approach followed is the same that is described in [9]. To sum up, the approach uses two functions:

- the $Soft$ function which allows to declare a soft constraint. It takes two arguments, one is the expression that will be checked, and the other is a weight associated with the expression.
- the *route* function which models the choice of a next hop considering also the latency between the nodes

Applying the same methodology, the variables extracted during the pre-processing in the previous example are then transformed into the following formulas:
For n1:

$$Soft((route(n1,n2,l_{12}) \Rightarrow n1@h1 \wedge n2@h2),10)$$
$$Soft((route(n1,n2,l_{13}) \Rightarrow n1@h1 \wedge n2@h3),20)$$

For n2:

$$Soft((route(n2,n3,l_{24}) \Rightarrow n2@h2 \wedge n3@h4),40)$$
$$Soft((route(n2,n3,l_{34}) \Rightarrow n2@h3 \wedge n3@h4),30)$$

For n3:

$$Soft((route(n3,S,l_{4S}) \Rightarrow n3@h4),5)$$

Taking as example the two conditions derived for n1, it can be seen how the *route* function refers to the same nodes but the latency is different. In fact, the implications state different deployments for n2 which obviously affect the latency that will be present between the nodes. In general, for each next hop, a distinct soft clause is declared. A similar discussion can be made for the conditions extracted for n2. The sequentiality of the nodes, requested in the service graph, is maintained also in the deployment thanks to the AND operation between in the conditions on the right side of the implication (e.g. if, for n1, z3 verifies $n1@h1 \wedge n2@h2$, for n2 the only valid configuration is the one that implies that $n2@h2$ is true, and so on). For n3, since the positioning of the server is fixed and assumed as known, on the right side of the implication there is only the condition about n3 without specifying the server.
Using these formulas, together with all the other ones about the resources available in the hosts, z3 will then compute which configuration is the optimal one.

6.4.2 Multiple Endpoints

Among all the improvements that have been made on Verifoo, the possibility of handling multiple clients and servers is one of the most important. Initially Verifoo was able to deploy only service chain in which the number of client and server was limited to one for each. Being capable of deploying service graph in which could be present more clients and servers, has favoured an improvement in the expressivity of the tool, which is now able to accept a wider range of different types of scenarios. In fact, having more clients that access the same network is a very common situation, as well as having more than one server in the same topology that may offer different services (e.g. one handles the mails and another the web pages). The introduction of this feature contributed also to the development of the auto-configuration feature, since it allowed to depict more interesting and realistic scenario for a policy refinement job. An example of this improvement can be seen in Figure 6.6, where two endpoints are used.

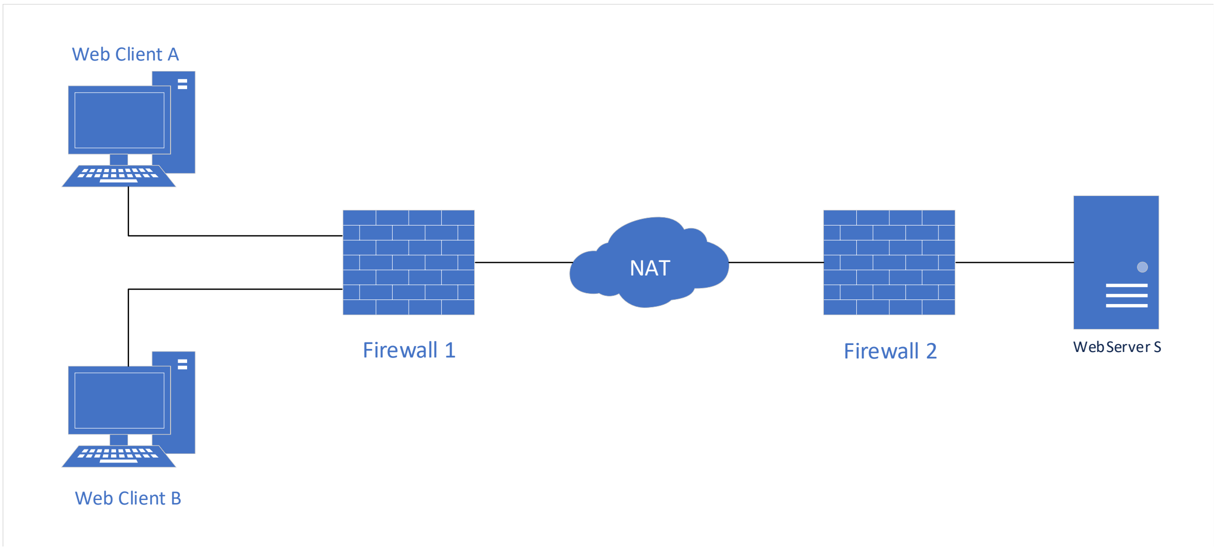


Figure 6.6. Policy Refinement Example

Allowing this type of scenario also gave the possibility to express a series of policies for each individual endpoint, increasing the complexity of the problem to a level where it can be advantageous to have a software that automates the policy refinement task.
To enforce this new functionality the pre-processing task, described in the previous chapter, has been slightly modified in order to take into account the possibility of having multiple endpoints. The issue about this thematic is in how the clients communicate with the servers (i.e. which VNFs are traversed and which servers are actually meaningful for the communication). In this regard, the NFV ETSI IGS, in one of its group specifications [5], has introduced the concept of network forwarding path as an "ordered list of connection points forming a chain of NFs, along with policies associated to the list". The use of this list gives the possibility to specify a path that connects a client to a server, thus resolving the previously mentioned issues. As described in chapter 3.1, an entity responsible to express this concept has been introduced as an XML element (the NetworkForwardingPaths element). Following the ETSI specification, it will serve as a way for the user to specify the actual communication paths. If no NetworkForwardingPaths element is defined in the input XML, Verifoo considers that every client could potentially communicate with every server and so it extracts the variables about the possible deployment considering all the cases. If there is a NetworkForwardingPaths element in the input XML, only the described paths are considered. A deployment variable (i.e. a variable to express that a specific node can be deployed on a specific host) is then passed to z3 for its computation, only if it has been encountered at least once in every client-server combination (i.e. every combination allows a specific node to be deployed on a specific host), otherwise it is discarded. Formally, the set of variables passed to z3, V, is the intersection of all the sets that are computed for each combination.

$$V = \cap_i V_i \tag{6.1}$$

(where V_i is the set of variables obtained for the i-th client-server combination).
This single addition however, is not enough to cover all the possible scenarios. In fact, if for a specific client-server combination, a node is never traversed in order to reach the server, during the pre-processing no variables are computed for that node, expressing implicitly that no deployments are possible for it. This, however, would be the exact opposite of what is the true meaning of this kind of situation: for a specific client-server combination, if a node is not on the path that leads to the server, it doesn't matter where that node will be deployed because it introduces no constraint (i.e. it can theoretically be deployed on any of the hosts). Due to the intersection operation, explicitly expressing these further variables allows to avoid the exclusion of correct possibilities.
An example, portrayed in Figure 6.7, will be used to clarify this aspect.

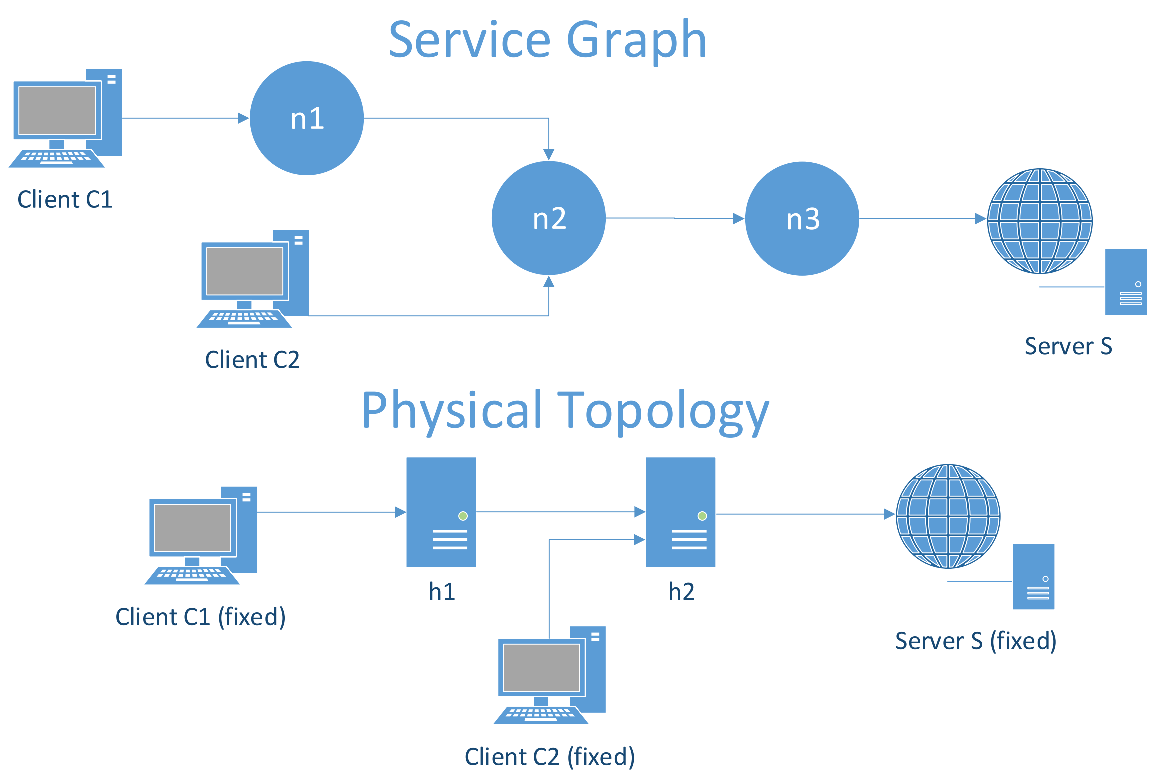


Figure 6.7. Multiple Endpoints Example

In this scenario, there are two client-server combination, C1-S and C2-S. For the first one, C1-S, the structure of the physical topology imposes that n1 can be deployed only on h1 (because it must be directly connected to C1), while n3 must be deployed on h2 (because it must be directly connected to S), leaving to n2 the possibility to be deployed either on h1 or h2. Now, considering the second combination, C2-S, n2 acquires the new constraint of being directly connected to C2 discarding the possibility of being deployed on h1. In fact, if n2 had been deployed on h1, C2 would not have been able to reach it directly, thus breaking the sequence imposed in the service graph. Lastly, n3 can still be only deployed on h2. To reach the server, C2 never goes through n1, hence it doesn't matter where it will be deployed. Normally no variable about n1 would be enumerated, however, to avoid the unwanted exclusion of variables present in the previous computed set, they are explicitly added.

V_1 (variables for C1-S)

```
n1@h1
n2@h1
n2@h2
n3@h2
```

V_2 (variables for C2-S)

```
n1@h1 (expl. added)
n1@h2 (expl. added)
n2@h2
n3@h2
```

V (variables fed to z3)

```
n1@h1
n2@h2
n3@h2
```

Finally, the set of variables V that is fed to z3, is the intersection of the previous two sets. The listed variables are the result of the pre-processing algorithm described in Chapter 3.4.1 and are used to enumerate the possible deployments of nodes that will be useful in the declaration of the soft constraints that express the different routing options.
Using multiple endpoints in specific type of scenarios can also cause behaviours that can seem unexpected at first. To explain this, the scenario in Figure 6.8 will be considered.

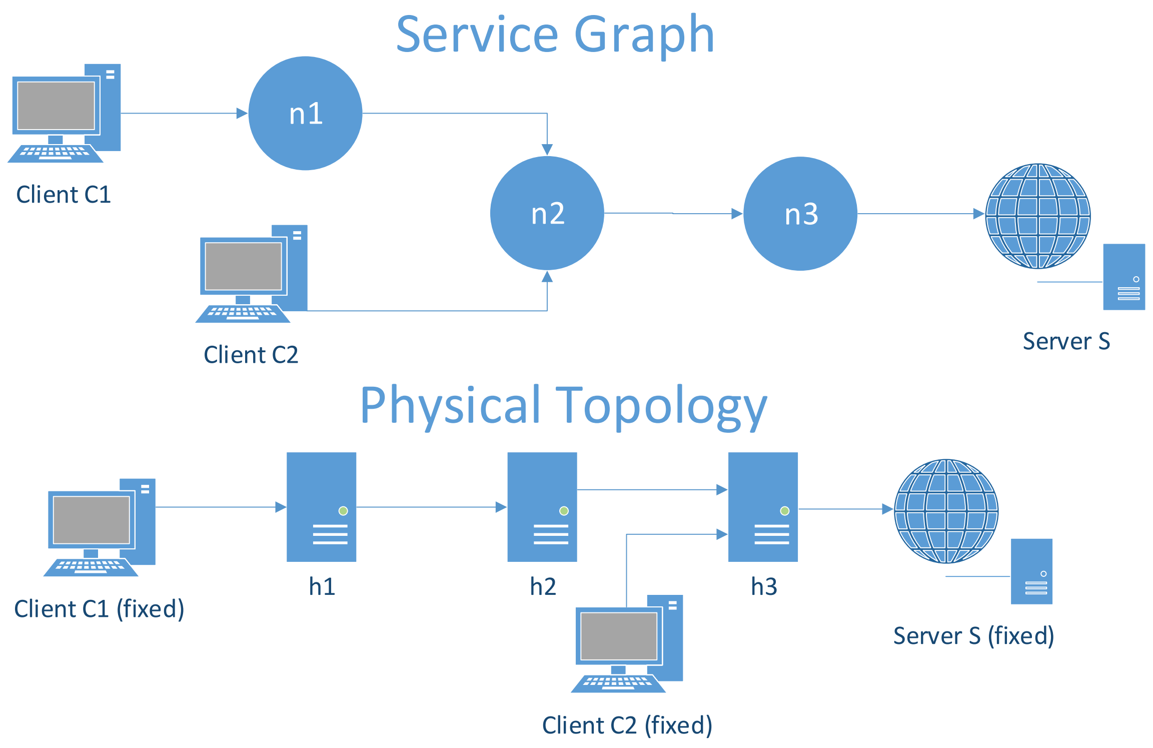


Figure 6.8. Multiple Endpoints Example 2

The network service is the same as in the previous example, while the physical topology is slightly different. A scenario like this, however, will always result in an undeployable service. The problem in this type of request is to be found in the constraints introduced by the fixed positions of the clients in the physical topology in contrast with the ones required for the service graph. In particular, the constraints for the graph are:

- n1 must be directly reachable from C1
- n2 must be directly reachable from C2
- n2 must be directly reachable from n1

Considering the physical topology however, there is no configuration that can fulfill the combination of these three requirements, that is, there is no deployment that can make n2 directly reachable from C2 and from n1, at the same time, without violating the constraint for C1 and n1, thus the always unsatisfied result. This is indeed the desired behaviour since there is no link connecting C2 to h2, if there were the scenario would become very similar to the one presented in Figure 6.7 and Verifoo would return the deployment.

6.4.3 Constraints

Initially, Verifoo supported a model of the nodes that considered only the disk space occupied by a VNF as a deployment constraint. With the addition of other node metrics in the schema, it was possible to extend the Verifoo expressivity enriching it with a number of new optional constraints for a more precise modelling of a real scenario, considering other possible requirements a node can have. These constraints are based on formulas that were already present in the initial Verifoo model. They can be summarized, as described below, with variables that assume specific values based on certain conditions. Formally, they can be expressed as follows:

- h_j represents a generic host and is equal to one if at least one node is deployed on that host; $(property)_{h_j}$ expresses that the <property> of the host h_j is considered. The set that includes all the host will be referred as H
- n_i represents a generic node, it is only used to specify a requirement for that node with the notation $(< property >)_{n_i}$ with the same meaning as before. The set that include all the nodes will be referred as N
- $d_{n_i h_j}$ is a generic deployment condition that states that the node n_i is deployed on the host h_j and it's computed during the pre-processing; this condition is true only if the node will actually be deployed on that host, false otherwise. When $d_{n_i h_j}$ is true, the respective h_j is equal to one. The set that include all the $d_{n_i h_j}$ will be referred as D
- $int(< booleancondition >)$ transforms the boolean condition into an integer, following the convention that true is translated into a one and false into a zero

Using the previous variables, the new constraints can be formally described:

- For all the deployment conditions, the computational latency a node will have after the deployment must be less or equal to a maximum latency declared as requirement; the latency of a node is calculated dividing the number of operations that a VNF performs, by the computational power of the host on which it will be deployed.
  With formulas, $\forall d_{n_i h_j} \in D$:
  $$\frac{(nr\_of\_operations)_{n_i}}{(cpu)_{h_j} * 1000000000} \times int(d_{n_i h_j}) \leq (max\_node\_latency)_{n_i} \times int(d_{n_i h_j}) \tag{6.2}$$
  (the cpu frequency is multiplied by 1000000000 because in the schema it's expressed in GHz, while the latency is expressed in ms)
- For all the deployment conditions, the cores required by each node deployed on a certain host must be less or equal than the cores that host has.
  With formulas, $\forall d_{n_i h_j} \in D$:
  $$(cores)_{n_i} \times int(d_{n_i h_j}) \leq (cores)_{h_j} \times h_j \tag{6.3}$$
- For each host, the sum of the RAM required by the nodes deployed on the host must be less or equal than the RAM present on that host.
  With formulas, $\forall h_j \in H$:
  $$\sum_{n_i \in M_{h_j}} ((memory)_{n_i} \times int(d_{n_i h_j})) \leq (memory)_{h_j} \times h_j \tag{6.4}$$
  Where $M_{h_j} \subseteq N$ and represents the set that contains all the nodes that can be deployed on the host h_j
- For each host, the number of the nodes deployed on the host must be less or equal than the maximum number of VNFs that host can have.
  With formulas, $\forall h_j \in H$:
  $$\sum_{n_i \in M_{h_j}} d_{n_i h_j} \leq (max\_vnf)_{h_j} * h_j \tag{6.5}$$
  Where $M_{h_j} \subseteq N$ and represents the set that contains all the nodes that can be deployed on the host h_j
- For all the pairs of sequential nodes, the latency between them must be less or equal than a maximum latency specified as a requirement. In this case, the nodes and the host are considered in pair in order to define a link. During the verification the latency between two nodes that are deployed on the same host is considered to be equal to zero, while if they are deployed on different hosts, the latency is considered to be equal to the one of the connection between the two hosts, which must

be directly reachable in the physical topology. A valid pair of sequential nodes is modelled with an AND between two deployment conditions, whose nodes are sequential in the service graph and whose host are directly reachable in the physical topology. The set that contains all these valid pairs will be referred as D_{pair}
With formulas, $\forall(d_{n_{i1}h_{j1}}, d_{n_{i2}h_{j2}}) \in D_{pair}$:

$$(latency)_{h_{j1}h_{j2}} \times int(d_{n_{i1}h_{j1}} \wedge d_{n_{i2}h_{j2}}) \leq (max\_latency)_{n_{i1}n_{i2}} \times int(d_{n_{i1}h_{j1}} \wedge d_{n_{i2}h_{j2}}) \tag{6.6}$$

- For all the hosts, a deployment condition that is true implies that the functional type of the node is supported by the host.
  With formulas, $\forall h_j \in H$:

$$d_{n_ih_j} \Rightarrow (functional\_type)_{n_i} \in (supported\_functional\_types)_{h_j} \tag{6.7}$$

To visualize better all the previous constraints, a simple example will be considered, using the scenario presented in Figure 6.9:

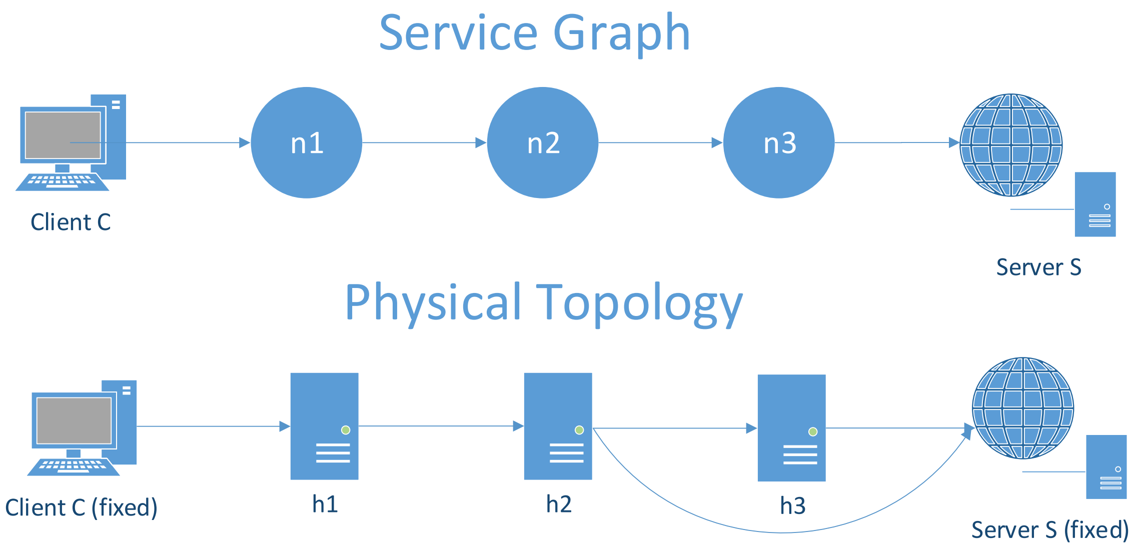


Figure 6.9. Example Scenario

In this scenario, the pre-processing will return the following sets:

$$D = \{n1@h1, n2@h1, n2@h2, n3@h2, n3@h3\}$$
$$\begin{aligned} D_{pair} = \{&(n1@h1, n2@h1),\\ &(n1@h1, n2@h2),\\ &(n2@h1, n3@h2),\\ &(n2@h2, n3@h2),\\ &(n2@h2, n3@h3)\} \end{aligned}$$

The nodes constraints and the host attributes are listed in the following extract of the input XML of Verifoo

Listing 6.34. Extract of input XML

```
  ...
  <node functional_type="FIREWALL" name="n1">
```

```
    ...
    <node functional_type="DPI" name="n2">
    ...
    <node functional_type="CACHE" name="n3">
    ...
   <Constraints>
     <NodeConstraints>
            <NodeMetrics node="n1" nrOfOperations="1000" maxNodeLatency="10" memory="2"
                ↪ cores="1"/>
            <NodeMetrics node="n2" nrOfOperations="1000" maxNodeLatency="20" memory="1"
                ↪ cores="2"/>
            <NodeMetrics node="n3" nrOfOperations="4500" maxNodeLatency="10" memory="8"
                ↪ cores="1"/>
      </NodeConstraints>
      <LinkConstraints>
            <LinkMetrics src="n1" dst="n2" reqLatency="20"/>
            <LinkMetrics src="n2" dst="n3" reqLatency="50"/>
      </LinkConstraints>
  </Constraints>
  ...
   <Hosts>
    <Host name="C" cpu="2" cores="2" memory="16" diskStorage="10" type="CLIENT"
        ↪ fixedEndpoint="nA"/>
    <Host name="h1" cpu="1" cores="4" memory="16" diskStorage="50" maxVNF="2" type="
        ↪ MIDDLEBOX">
        <SupportedVNF functional_type="FIREWALL"/>
        <SupportedVNF functional_type="CACHE"/>
    </Host>
    <Host name="h2" cpu="3" cores="8" memory="16" diskStorage="60" maxVNF="3" type="
        ↪ MIDDLEBOX">
        <SupportedVNF functional_type="FIREWALL"/>
        <SupportedVNF functional_type="DPI"/>
    </Host>
    <Host name="h3" cpu="4" cores="2" memory="8" diskStorage="80" maxVNF="4" type="
        ↪ MIDDLEBOX">
        <SupportedVNF functional_type="CACHE"/>
        <SupportedVNF functional_type="DPI"/>
    </Host>
    <Host name="S" cpu="2" cores="2" memory="32" diskStorage="10" type="SERVER"
        ↪ fixedEndpoint="nB"/>
  </Hosts>
  <Connections>
    <Connection sourceHost="C" destHost="h1" avgLatency ="1"/>
    <Connection sourceHost="h1" destHost="h2" avgLatency ="10"/>
    <Connection sourceHost="h2" destHost="h3" avgLatency ="20"/>
    <Connection sourceHost="h2" destHost="S" avgLatency ="1"/>
    <Connection sourceHost="h3" destHost="S" avgLatency ="1"/>
  </Connections>
  ...
```

The computed constraints are:

1. Computational latency

   - n1:

$$\frac{1000}{1 \times 1000000000Hz} \times int(n1@h1) \leq 10ms \times int(n1@h1)$$

- n2:

$$\frac{1000}{1 \times 1000000000Hz} \times int(n2@h1) \leq 20ms \times int(n2@h1)$$
$$\frac{1000}{3 \times 1000000000Hz} \times int(n2@h2) \leq 20ms \times int(n2@h2)$$

- n3:

$$\frac{4500}{3 \times 1000000000Hz} \times int(n3@h2) \leq 10ms \times int(n3@h2)$$
$$\frac{4500}{4 \times 1000000000Hz} \times int(n3@h3) \leq 10ms \times int(n3@h3)$$

2. Cores

- n1:

$$1 \times int(n1@h1) \leq 4 \times int(n1@h1)$$

- n2:

$$2 \times int(n2@h1) \leq 4 \times int(n2@h1)$$
$$2 \times int(n2@h2) \leq 8 \times int(n2@h2)$$

- n3:

$$1 \times int(n3@h2) \leq 8 \times int(n3@h2)$$
$$1 \times int(n3@h3) \leq 2 \times int(n3@h3)$$

3. RAM

- h1 with $M_{h_1} = n1@h1, n2@h1$:

$$2 \times int(n1@h1) + 1 \times int(n2@h1) \leq 16 \times h1$$

- h2 with $M_{h_2} = n2@h2, n3@h2$

$$1 \times int(n2@h2) + 8 \times int(n3@h2) \leq 8 \times h2$$

- h3 with $M_{h_3} = n3@h3$

$$8 \times int(n3@h3) \leq 32 \times h3$$

4. Max VNFs

- h1 with $M_{h_1} = n1@h1, n2@h1$:

$$int(n1@h1) + int(n2@h1) \leq 2 \times h1$$

- h2 with $M_{h_2} = n2@h2, n3@h2$

$$int(n2@h2) + int(n3@h2) \leq 3 \times h2$$

- h3 with $M_{h_3} = n3@h3$

$$int(n3@h3) \leq 4 \times h3$$

5. - link between n1 and n2:

$$0 \times int(n1@h1 \land n2@h1) \leq 20 \times int(n1@h1 \land n2@h1)$$
$$10 \times int(n1@h1 \land n2@h2) \leq 20 \times int(n1@h1 \land n2@h2)$$

- link between n2 and n3:

$$
\begin{aligned}
10 \times int(n2@h1 \wedge n3@h2) &\leq 50 \times int(n2@h1 \wedge n3@h2) \\
0 \times int(n2@h2 \wedge n3@h2) &\leq 50 \times int(n2@h2 \wedge n3@h2) \\
20 \times int(n2@h2 \wedge n3@h3) &\leq 50 \times int(n2@h2 \wedge n3@h3)
\end{aligned}
$$

6. Supported VNFs

- n1:

$$n1@h1 \Rightarrow \{"FIREWALL"\} \subset \{"FIREWALL","CACHE"\}$$

- n2:

$$
\begin{aligned}
n2@h1 &\Rightarrow \{"DPI"\} \subset \{"FIREWALL","CACHE"\} \\
n2@h2 &\Rightarrow \{"DPI"\} \subset \{"FIREWALL","DPI",\}
\end{aligned}
$$

- n3:

$$
\begin{aligned}
n3@h2 &\Rightarrow \{"CACHE"\} \subset \{"FIREWALL","DPI"\} \\
n3@h3 &\Rightarrow \{"CACHE"\} \subset \{"DPI","CACHE"\}
\end{aligned}
$$

Not all of the previous constraints can be satisfied for the considered scenario. The task of z3 is to discern the deployments that can fulfill the requirements and those that cannot; then, to find among those that meet the requirements, the one that is the optimal.
In this example the set $S \subseteq D$, that contains all the deployment conditions that are true, would be:

$$S = \{n1@h1, n2@h2, n3@h3\}$$

as can be easily derived seeing only the constraint 6.7.

6.4.4 Isolation Policy

The first policy Verifoo has supported is the reachability policy which tests if there is at least one path that a packet can follow to reach a specific destination from a specific source. An extension, made during this thesis work, has been to implement a new type of policy, the isolation policy, and make it compliant with all the available VNFs. The introduction of this new policy has been possible thanks to the extension of the XML schema and the addition of new formulas in the internal Verifoo model. The example below shows how to request to verify an isolation policy between a client and a server.

Listing 6.35. Policy Example

```xml
<PropertyDefinition>
      <Property graph="0" name="IsolationProperty" src="nodeA" dst="nodeB" src_port="
        ↪ 5000" dst_port="80">
         <HTTPDefinition url="polito.it" body="cats"/>
      </Property>
</PropertyDefinition>
```

The isolation policy checks if not exists any packet that from the specified source is able to reach the specified destination. The type of packet that is checked to be reachable can be enriched with various characteristics (e.g. ports, application protocol, etc.) as better described in the chapter about the Verifoo XML schema. As regards the model, the isolation policy has been modelled as follows:

$$
\begin{aligned}
&\forall \{n_0, p_0\}: \\
&\qquad recv(n_0, dest, p_0) \implies p_0.origin \neq src
\end{aligned}
\tag{6.8}
$$

$$\begin{aligned}&\exists\{n_1,p_1\}:\\&\qquad send(src,n_1,p_1)\wedge p_1.dest == dest\end{aligned} \tag{6.9}$$

Where *src* and *dest* are the source and destination specified in the XML. The formula (6.8) states that if the destination specified in the XML has received a packet, it must not have been originated from the source specified in the XML. The formula (6.9), instead, ensure that exists a packet that is generated from the source with the specified destination. This last constraint is useful because, without it, z3 will simply not generate any packet from the source making the first constraint true, thus mistakenly satisfying the isolation policy.
Another constraint has been added for each VNF to ensure the correctness implementation of the new policy. Each VNF has an additional constraint that can be generalized as follows:

$$\begin{aligned}&\forall\{n_0,p_0\}:\\&\qquad recv(n_0,vnf,p_0)\wedge satisfy\_vnf\_constraints(p_0)\implies\\&\qquad\quad \exists\{n_1\}:send(vnf,n_1,p_0)\end{aligned} \tag{6.10}$$

The *satisfy_vnf_constraint* function is a function that depends on the specific VNF and tests if the packet is allowed to be forwarded. For instance, for a firewall it translates into the function that checks if a packet matches a rule in the ACL, while for an antispam, it translates into the function that ensures that a packet has not been sent from an address contained in the blacklist, and so on. In general, formula (6.10) obliges a VNF to forward a packet that is received, if it does not break any VNF rules. This constraint is useful because without it, in order to satisfy formula (6.8), any VNF would avoid forwarding the received packets.
A future improvement could be further increase the number of policies that can be requested.

6.4.5 Extended Model

Another assumption the Verifoo model introduced, was that the type of service graphs that it could deploy on physical topologies was not properly a graph but a simple chain. In order to include also the possibility to have a real service graphs, some modification have been made on the formulas computed by z3, extending the work presented in [9]. As described in that paper, in Verifoo each VNF in the network service has its behaviour modelled with a set of formulas. These formulas represent the routing tables and specify which node is the next hop in order to reach a destination. However, the approach described in the paper only considered chains, so the presence of more than one next hop in the routing table was not expected. To take into account the possibility that a node can have more than one neighbour that reaches a specific destination, the formulas have been enhanced to allow z3 to choose between multiple nodes. Formally, declaring:

- $n0$, $n1$ as generic nodes
- $p0$ as a generic packet that goes through a node
- *dest* as the final destination of the packet
- N_n as the set of all the neighbours of $n1$ that can reach the destination

$\forall\{n0,p0\}$:

$$[send(n1,n0,p0)\wedge destAddr(p0,dest)]\Rightarrow\bigvee_{\forall i|n_i\in N_n} n0=n_i \tag{6.11}$$

The previous formula states that every packet $p0$ that arrives at $n1$ with destination equal to *dest*, must have as next hop $n0$, that is one of its neighbours. This formula modifies the network behaviour, ensuring that every packet follows only paths that are expected in the requested service graph, but it does not affect in any way the deployment. To have a deployment that is coherent with the service graph, i.e. all the sequential nodes are deployed either on the same host or on hosts that are directly reachable, the following formula is used. Declaring:

- N as the set of the nodes

- N_i as the set of all the neighbours of a generic node i
- H as the set of the hosts
- d_{ij} as a generic deployment condition that specify on which host (j) is deployed which node (i), and it is true if that node is actually deployed on that host
- $int(d_{ij})$ as a function that translate the boolean condition d_{ij} into its integer value, following the convention that true is equal to one and false to zero
- D as the set of all the deployment condition
- D_{pair} as the set that contains all the couple of deployment conditions that refer to sequential nodes
- $D_{pair_{ik}}$ as a sub set of D_{pair} where all the deployment conditions refer to the same nodes i and k, which must be sequential, but have different hosts

$\forall i : n_i \in N :$

$$\bigwedge_{\forall k | n_k \in N_i} \left[\left(\sum_{D_{pair_{ik}}} int(d_{pair_{ik}}) \right) = 1 \right] \tag{6.12}$$

This second formula states that for each node, every one of its neighbours must be deployed on eligible hosts and that only one configuration among all is chosen (i.e. every neighbour must be deployed on one and only one host).
To have a practical demonstration of the previous formulas, the scenario in Figure 6.10 will be considered. It could be seen as a section of a bigger topology but, for the sake of simplicity, only this part will be taken into account.

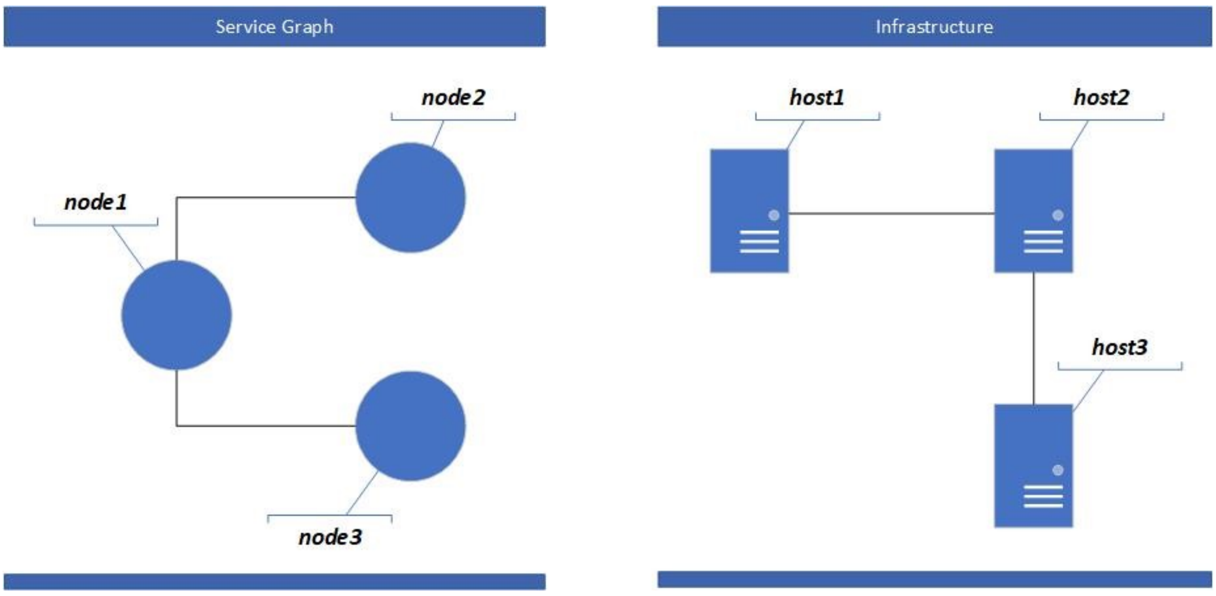


Figure 6.10. Example Scenario

The various sets that can be extracted from the previous scenario are:

- the set of all the nodes $N = \{node1, node2, node3\}$
- the set of the neighbour nodes $N_1 = \{node2, node3\}, N_2 = N_3 = \{node1\}$
- the set of hosts $H = \{host1, host2, host3\}$

- the set of deployment condition

  $$D = \{node1@host1, node2@host1, node2@host2, node3@host1, node3@host2\}$$

  ; since only a small part of the topology is considered, these last conditions can not be retrieved from the picture itself, but assuming that node1 can only be on host1, the other conditions can be immediately deduced because both node2 and node3 must be deployed on hosts that are directly reachable from node1

- $D_{pair} =$ $\{(node1@host1, node2@host1), (node1@host1, node2@host2), (node1@host1, node3@host1), (node1@host1, node3@host2)\}$

Taking into consideration the formulas only for node1, equation 6.11 becomes

$$[send(node1, n0, p0) \land destAddr(p0, dest)] \Rightarrow (n0 = node2 \lor n0 = node3)$$

while equation 6.12 becomes

$$[int(node1@host1 \land node2@host1) + int(node1@host1 \land node2@host2) = 1] \\ \land \\ [int(node1@host1 \land node3@host1) + int(node1@host1 \land node3@host2) = 1]$$

Similar formulas are computed also for node2 and node3.
The application of the previous two equations to all the nodes present in a service graph ensures its correct the deployment on any physical topology.

6.4.6 Packet Extensions and wildcards

Initially, in Verifoo, an IP address was modelled as single entity, like a symbolic name, thus allowing an IP address to be for example node1, node2, etc., but excluding the possibility to handle an IP. Moreover, since Verifoo was in early development, for the sake of simplicity the packets in the z3 were modelled without considering the source and destination ports. One of the extension made on Verifoo, changed the handling of IP addresses and included the source and destination ports in the packet model. At the moment, the only VNF that exploits these new additions is the firewall, hence a future improvement could be to extend their compatibility also to the other VNFs whose behaviour could take advantage of those additions, in order to get closer to their real counterpart. Thanks to the IP extension, using a real IP address like 10.0.0.1, has now a real impact on a firewall on how it handles the rules. In fact, with a real IP, it has been also possible to introduce the concept of wildcards, even though in a simplistic way that does not use the netmasks. Nevertheless it is an approach supported also by some commercial firewall products [16]. Both the wildcards and the ports can now be used to define a firewall rule (as shown below) to decide if a packet needs to be dropped or not.

Listing 6.36. Firewall Configuration Example

```xml
<configuration name="confFW">
  <firewall>
    <elements>
      <source>20.0.-1.-1</source>
      <destination>30.0.5.-1</destination>
      <src_port>5000-8000</src_port>
      <dst_port>80-110</dst_port>
    </elements>
  </firewall>
</configuration>
```

In Verifoo a wildcard is considered to be the number -1, which can replace any of the four decimal number an IP address is composed of. So, for instance, the address that represents any address (*.*.*.*)

is translated into -1.-1.-1.-1. The reason behind this translation is that the variable assigned to the IP is an integer, so it cannot be a character like "*". To correctly handle this new concept of IP, also the equality between two addresses has been revised, introducing the following constraint:

$$\bigwedge_{\forall i \in \{0,1,2,3\}} [ip1_i == ip2_i \lor ip1_i == -1 \lor ip2_i == -1] \tag{6.13}$$

The formula (6.13) states that two IP addresses are equal if the value of the numbers in the same position, is exactly the same or at least one among the two is equal to -1. The formula (6.13) has been the key to allow a firewall to be able to understand rules that use the wildcards during the verification. In this way, in a scenario similar to the one presented in Figure 6.11, the packets of both clients will be dropped.

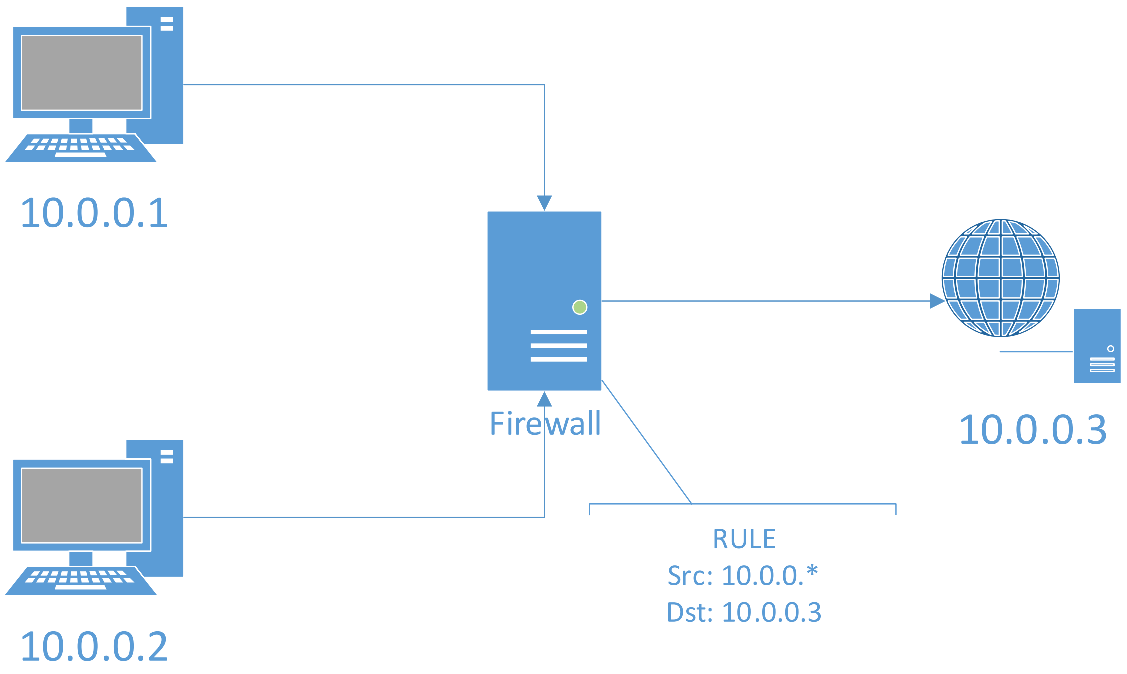


Figure 6.11. Wildcards Example Scenario

The corresponding firewall configuration in the input XML is the following:

Listing 6.37. Firewall Configuration Example

```xml
<configuration name="confFW">
  <firewall>
    <elements>
      <source>10.0.0.-1</source>
      <destination>30.0.5.2</destination>
    </elements>
  </firewall>
</configuration>
```

If, for a node, a symbolic name is used instead of a real IP (for example “node1”), Verifoo derives an IP address from the hash code of that symbolic name which will be used internally for the computation, however, externally, it will be always shown the symbolic name. The use of a hash code could also introduce collision problems but since the number of the nodes in a real service graph is usually restrained, the probability of collision is very low.

With the introduction of the wildcards has also been possible to explore the possibility of including it in the declaration of a policy in the source or in the destination field as shown in listing 6.38 with the idea of expanding further the input XML expressivity.

Listing 6.38. Property Definition Example

```xml
<PropertyDefinition>
        <Property graph="0" name="IsolationProperty" src="10.0.0.-1" dst="
            ↪ 100.100.100.-1"/>
</PropertyDefinition>
```

Inserting any of the two field with a wildcard without any further processing into z3 caused some issues. In fact, those field are used to formulate the conditions on the network behaviour through the *send* and the *recv* functions which expect that the node specified as source (and/or destination) is one of the node included in the service graph. Moreover, the concept of a wildcards is obviously not built-in in the z3 code, so when z3 tries to satisfy the condition $send(\text{“}10.0.0.-1\text{”}, nexthop, p0)$, it tries to satisfy it for a possible node referred as “10.0.0.-1” without knowing that it can also be referring to, for instance, 10.0.0.1. Since the *send* and the *recv* functions are the core of the network behaviour, a modification in those functions could have meant rethinking a significant part of the network model. To avoid this process, a workaround has been implemented, specifically a pre-processing task has been added that translates every policy that involves a wildcard in the source or in the destination, to a list of policy that enumerates the nodes in the network included by that wildcard. Taking as example the scenario depicted in Figure 6.12, the policies transforms following the pattern shown in listing 6.39.

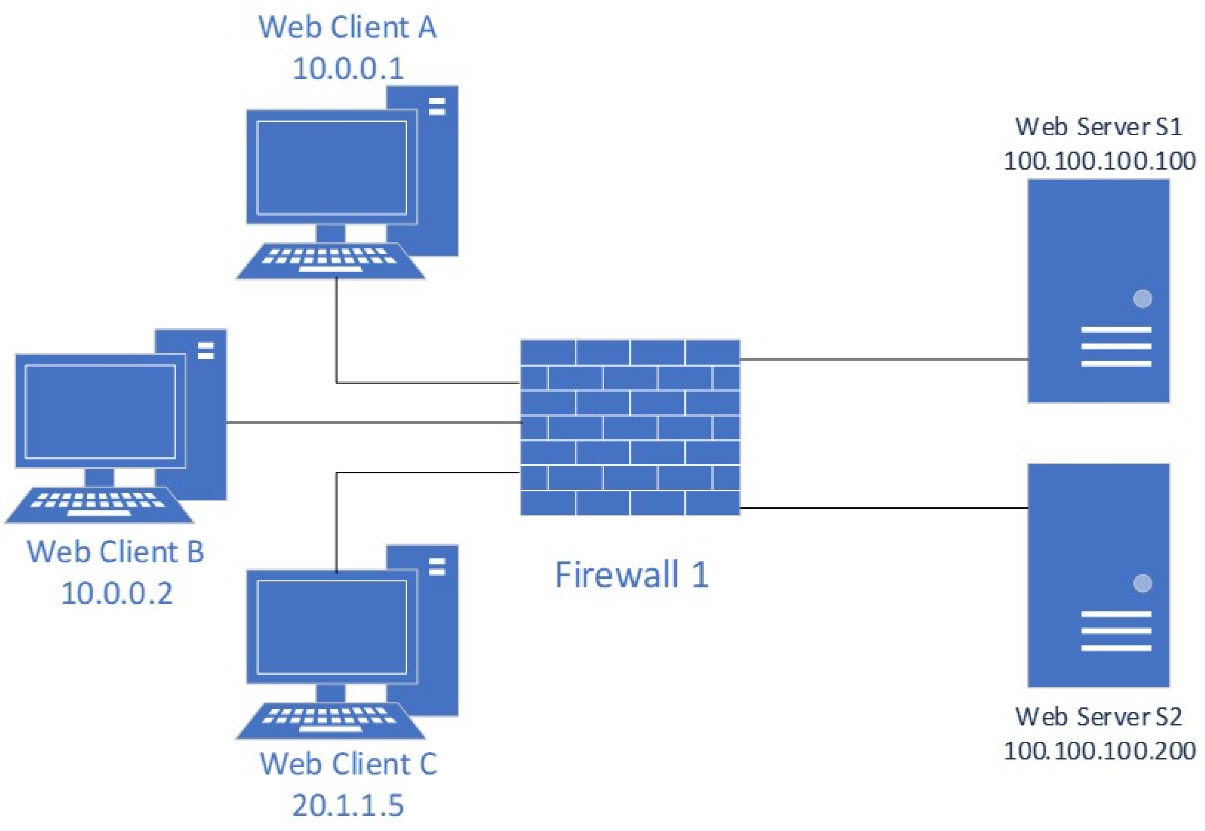


Figure 6.12. Example Scenario

Listing 6.39. Property Definition Transformation

```xml
<PropertyDefinition>
        <Property graph="0" name="IsolationProperty" src="10.0.0.1" dst="
            ↪ 100.100.100.100"/>
        <Property graph="0" name="IsolationProperty" src="10.0.0.1" dst="
            ↪ 100.100.100.200"/>
        <Property graph="0" name="IsolationProperty" src="10.0.0.2" dst="
            ↪ 100.100.100.100"/>
        <Property graph="0" name="IsolationProperty" src="10.0.0.2" dst="
            ↪ 100.100.100.200"/>
</PropertyDefinition>
```

Regarding the TCP/UDP ports, they have been added as packet fields, as well as in the firewall rules, in order to model more realistic scenarios. In a firewall rule, the ports indications are optional and when none is specified, the firewall uses a more general approach that considers only the addresses. It is also possible to have partial rule in which only the destination port or the source port is specified, giving the possibility to have a wider variety of rules types. To request the verification of a policy for a well-defined set of ports, the correspondent attributes in the input XML must be set. An example could be the following:

Listing 6.40. Policy Example

```xml
<PropertyDefinition>
        <Property graph="0" name="IsolationProperty" src="A" dst="B" src_port="
            ↪ 1000-2000" dst_port="80"/>
</PropertyDefinition>
```

In this case, the property definition imposes that all the packets generated from the client nodeA must contain those ports. If a rule in the firewall matches the fields in the packet, it will apply the specified rule action (forward or drop).
A possibility that the addition of the ports has opened up, was to specify different policies with the same source and destination differing from other characteristics, as can be seen in the XML example in listing 6.41.

Listing 6.41. Policies Example

```xml
<PropertyDefinition>
        <Property graph="0" name="IsolationProperty" src="A" dst="B" src_port="
            ↪ 1000-2000" dst_port="80"/>
        <Property graph="0" name="ReachabilityProperty" src="A" dst="B" src_port="
            ↪ 4000-5000" dst_port="443"/>
</PropertyDefinition>
```

Internally, applying this type of policies creates a contradiction that is caused by the following condition:

$$\begin{aligned}&\forall p_0 : \\ &\quad send(endhost, nexthop, p_0) \implies p_0\ has\ well\ defined\ characteristics\end{aligned} \tag{6.14}$$

As shown in formula (6.14), to ensure that z3 sends the right packet from a specific endhost, a universal quantifier is used. However, this also limits the possibility to have different packets sent from the same endhost. For this reason, to overcome this problem the solution found uses a pre-processing task that creates from a single node used as a source in a certain number of policies, an equal number of abstract nodes with each its own policy.

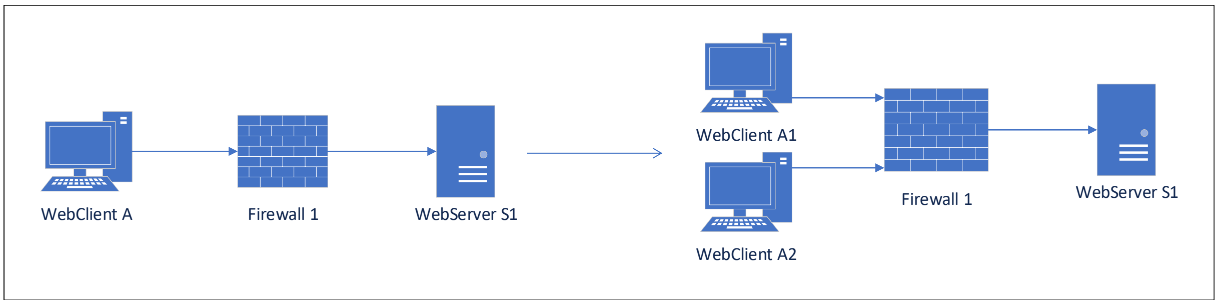


Figure 6.13. Pre-Processing Task

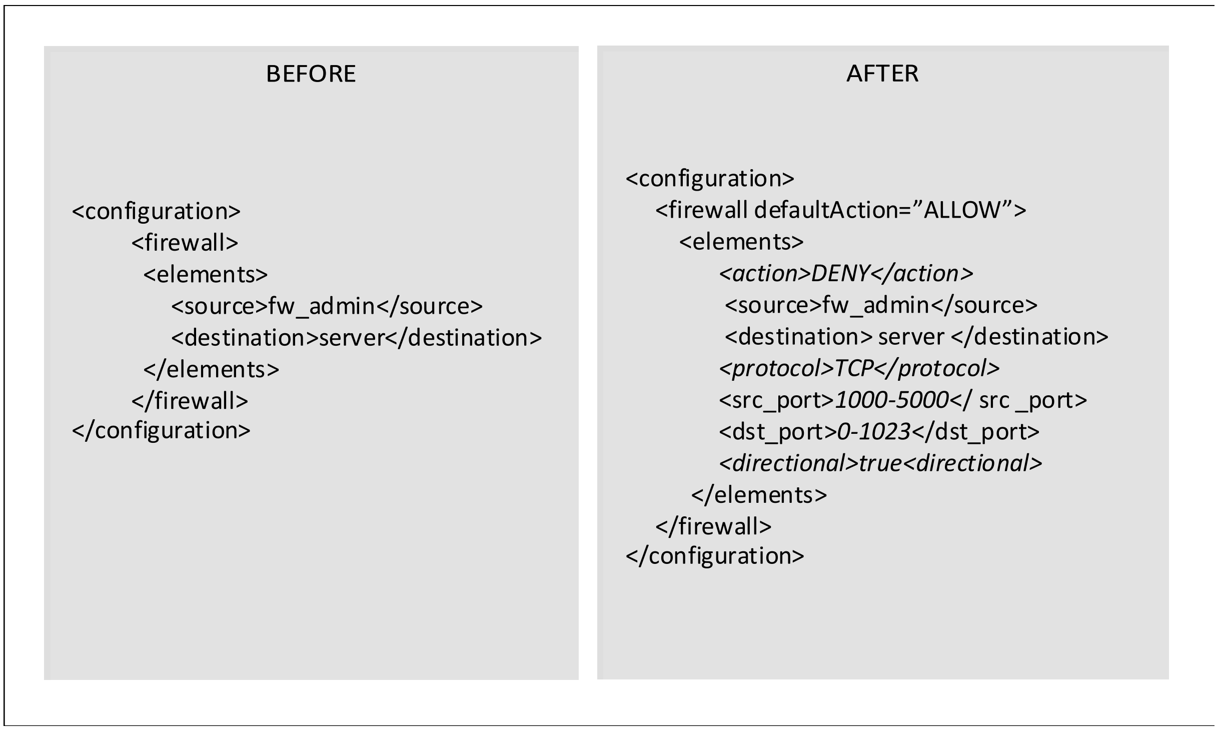
Listing 6.42. Policies translated

```
<PropertyDefinition>
        <Property graph="0" name="IsolationProperty" src="A1" dst="B" src_port="
            ↪ 1000-2000" dst_port="80"/>
        <Property graph="0" name="ReachabilityProperty" src="A2" dst="B" src_port="
            ↪ 4000-5000" dst_port="443"/>
</PropertyDefinition>
```

Each of the abstract node can be seen as a single flow of packets that comes from a specific endhost, hence after the z3 computation, if all the policies that refer to the abstract nodes are satisfied, the original problem can also be considered satisfied. Before showing the output, the abstract nodes are then regrouped in the original node, hiding this internal complexity from the final user.

6.4.7 New Firewall Model

A firewall is one of the VNFs that Verifoo supports and models with with FOL (First Order Logic) formulas. It is possible to build a network service with one or more firewalls, specifying for each of them its configuration with a high-level abstraction (i.e. a set of firewall rules declared following the XML schema) and let Verifoo perform its evaluation to know if their current configurations are enough to satisfy a set of requested policy. Initially, the firewall was modelled to work only in blacklist mode, hence the rules specified which packets to drop, while the firewall let all the packets that were not covered by any of the rules pass. Moreover, the rules only included the source and the destination as packet fields to check in order to decide if it was needed to drop a packet or not. During this thesis work, the VNF of a firewall has been completely revamped, improving its model in order to have the possibility to work also in a whitelist mode and enriching its rules expressivity with other packet fields to let the administrator set up a finer behaviour. The input XML has been modified accordingly to reflect those changes as depicted in Figure 6.14. As shown, in the configuration element of a firewall the default action can be specified and represents the behaviour that the firewall adopts for all the packets that don't match any of the specified rules. The default action can be ALLOW (forward the packets that are not included in any of the rules) or DENY (drop the packets) and if no indication is given, the default action is assumed to be DENY in order to have a more conservative behaviour. In addition, the rules contain a field that explicitly specifies the action (always ALLOW or DENY) for a packet with certain characteristics. More information about the XML schema can be found in chapter 3.1. Specifying a type of default action and listing only rules of the opposite type allows the definition of firewall that works in blacklist mode or whitelist mode in a simple way. However, it is also possible to define rules with mixed types of actions to have a more complex firewall behaviour. Internally, Verifoo translates the indications given in the XML in formulas that can be understood by z3. The forwarding actions of a firewall (i.e. if the firewall has to forward a packet or not) is determined by a function declared in the z3 context called $acl\_func()$. This function is a function that returns a boolean, true or false that respectively map the ALLOW or DENY action. Each received packet is forwarded only if $acl\_func()$ returns true. Therefore, correctly modelling that function translates in the right forwarding behaviour.



BEFORE

```
<configuration>
        <firewall>
          <elements>
              <source>fw_admin</source>
              <destination>server</destination>
          </elements>
        </firewall>
</configuration>
```

AFTER

```
<configuration>
    <firewall defaultAction=”ALLOW”>
       <elements>
            <action>DENY</action>
             <source>fw_admin</source>
             <destination> server </destination>
            <protocol>TCP</protocol>
            <src_port>1000-5000</ src _port>
            <dst_port>0-1023</dst_port>
            <directional>true<directional>
         </elements>
    </firewall>
</configuration>
```

Figure 6.14. Firewall XML Schema Extensions

For each rule the following constraint is added:

$$\begin{aligned}&\forall\{p_0\}:\\&\qquad rule.match(p_0) \implies acl\_func(p_0) == rule.action\end{aligned} \tag{6.15}$$

where the $rule.match(*)$ expression is an abstraction of the following formula:

$$\begin{aligned}&p_0.src == rule.src \land\\&p_0.dst == rule.dst \land\\&p_0.lv4proto == rule.lv4proto \land\\&p_0.src\_port.start \geq rule.src\_port.start \land\\&p_0.src\_port.end \leq rule.src\_port.end \land\\&p_0.dst\_port.start \geq rule.dst\_port.start \land\\&p_0.dst\_port.end \leq rule.dst\_port.end \land\end{aligned} \tag{6.16}$$

The formula (6.15) models the firewall behaviour in order to act correctly based on the specified rules. In conjunction with it, there is one more condition, added only once for each firewall, to model the behaviour of the default action:

$$\begin{aligned}&\forall\{n_0, p_0\}:\\&\qquad recv(n0, fw, p_0) \land (\nexists rule : rule.match(p_0)) \implies\\&\qquad\qquad acl\_func(p_0) == fw.defaultAction\end{aligned} \tag{6.17}$$

At the moment, the model does not consider the possibility of conflict between overlapping rules. This means that if for the same packet two or more rules match with it, the actions of those rules must be the same otherwise it would create a contradiction, which would return an UNSAT result regardless of all the other constraints. A future development could be to assign different priorities to the various rules and apply only the one that has the highest priority. However, modelling the concept of priority directly in z3 could be very difficult, or even impossible. In fact, z3 only provides to the developer hard and soft constraints to solve MaxSAT problems. Obviously, using only hard constraints creates the contradiction problem mentioned previously. A direct evolution of this approach would be to model the rules using the soft constraints, giving more weight to the rules with higher priority. In addition, there could be one hard constraint to ensure that, for each packet only one soft constraint at a time (i.e. one rule) is satisfied to avoid inconsistencies. The problem introduced by this approach is that the soft constraints are like suggestions given to z3, which may also prefer to satisfy a soft constraint with lower priority just to avoid contradicting some hard constraints. An example of this behaviour can be observed in the following example:

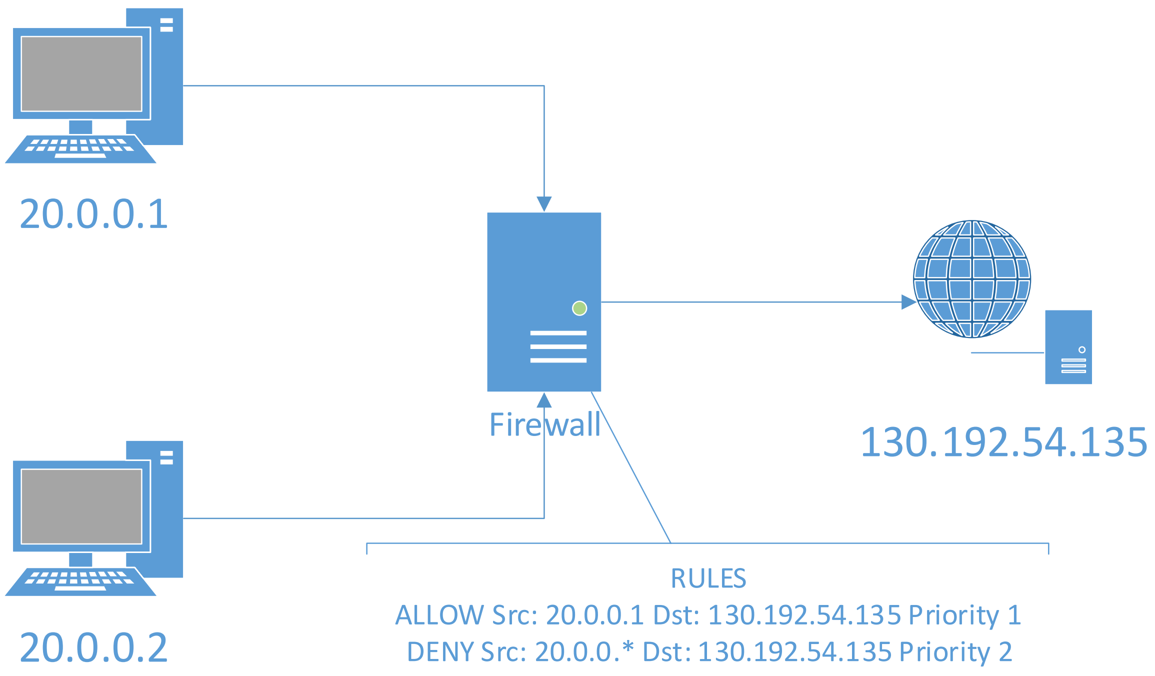

20.0.0.1
20.0.0.2
Firewall
130.192.54.135
RULES
ALLOW Src: 20.0.0.1 Dst: 130.192.54.135 Priority 1
DENY Src: 20.0.0.* Dst: 130.192.54.135 Priority 2

Figure 6.15. Example Scenario

The rules modelled as soft constraints are:

A) ALLOW(from 20.0.0.1 to 130.192.54.135) weight:2 (higher priority)

B) DENY(from 20.0.0.* to 130.192.54.135) weight:1 (lower priority)

In addition there is one hard constraint to ensure that there are no inconsistencies: A + B = 1 If, among the other policies (in Verifoo they are modelled as hard constraints), there is one similar to:

- Isolation Policy between 20.0.0.1 and 130.192.54.135

It will happen that in order to avoid a contradiction with the hard constraint introduced by the policy, the soft constraint A is falsified, allowing B to be true, thus the result is SAT even though it should not because the node 20.0.0.1 should not be blocked.
Another approach could be to create a pre-processing algorithm in java that delivers to z3 a conflict-free set of rules.

6.4.8 Auto-Configuration

An important extension added to Verifoo is the possibility to have VNFs without any configuration or with partial configuration, giving to the tool itself the task of providing the missing configurations at the end of the computation. The tool will generate the configuration with the objective of satisfying all the

requested policies while minimizing the number of generated rules in order to achieve it. For instance, for a firewall, this translates into the generation of the rules that decide if a received packet needs to be dropped or not. The auto-configuration does not affect the deployment in any way, since all the VNFs that are declared in the service graph will be deployed onto a host in the physical topology even if they still have an empty configuration after the z3 computation (this would mean that even without adding any rules, the policies are satisfied).
Internally, Verifoo describes the VNFs that require the autoconfiguration differently from the already configured counterpart of the same type. It uses a set of soft clauses that can be falsified, that have "null" as a default value (or a value that has the same meaning for the specific context, e.g. zero for a TCP port, more information are given below). In conjunction with these soft clauses, also new hard constraints are declared to model the behaviour of the specific VNF. The general idea is that following the VNF model, z3 decides which soft clauses will have values different from their default ones in order to satisfy the requested policies. Not using the default value will introduce a penalty that z3 will try to minimize, hence the resultant number of significant rules will be the least possible. In particular, for a firewall, the rules that are generated express an action (ALLOW or DENY) that is always the opposite of the default one in order to avoid any conflicts.
The soft clauses that are declared for a firewall for each rule are the following:

$$\begin{aligned}
&Soft(src == null, k, \text{“}rules\text{“})\\
&Soft(dst == null, k, \text{“}rules\text{“})\\
&Soft(protocol == 0, k, \text{“}rules\text{“})\\
&Soft(src\_port == null, k, \text{“}rules\text{“})\\
&Soft(dst\_port == null, k, \text{“}rules\text{“})
\end{aligned} \tag{6.18}$$

where the first argument of the $Soft$ function represents the constraint that can be falsified, k is a constant that define its weight and the third argument is a label assigned to differentiate between the classes of soft clauses (the weights of the constraints in each class are optimized independently from each other). The previous soft clauses represent all the fields that will compose a firewall rule. In (6.18), only $protocol$ is an integer variable and is mapped in the output XML with an enumeration to have a human-readable string. The src, dst, src_port and dst_port variables are DatatypeSort, instead. A DatatypeSort is a particular type that z3 provides to the developers that allows the definition of more complex data structures. In fact, the src and dst variables are an abstraction of an IP address and, as such, they are made of four different decimal numbers (integers) that compose the address DatatypeSort. Moreover, also the port fields are declared as a particular DatatypeSort in order to consider the possibility to have an interval (e.g. 10-80) and not only a single value. This latter DatatypeSort is therefore composed by two integers that represent the start and the end of the interval. To instruct z3 on which values are possible for the mentioned DatatypeSort, new hard constraints have been added.
For the IPs:

$$\begin{aligned}
\forall\{n0, n1, p0\}:&\\
&recv(n0, n1, p0) \implies\\
&\quad p0.src.\_0 > 0 \land p0.src.\_0 < 255 \land\\
&\quad p0.src.\_1 > 0 \land p0.src.\_1 < 255 \land\\
&\quad p0.src.\_2 > 0 \land p0.src.\_2 < 255 \land\\
&\quad p0.src.\_3 > 0 \land p0.src.\_3 < 255 \land\\
&\quad p0.dst.\_0 > 0 \land p0.dst.\_0 < 255 \land\\
&\quad p0.dst.\_1 > 0 \land p0.dst.\_1 < 255 \land\\
&\quad p0.dst.\_2 > 0 \land p0.dst.\_2 < 255 \land\\
&\quad p0.dst.\_3 > 0 \land p0.dst.\_3 < 255
\end{aligned} \tag{6.19}$$

The formula (6.19) ensures that every packet that is exchanged between the VNFs has correct IPs. This is because for every packet that is received, there is one that is sent and vice versa, if it doesn't break any other constraints (e.g. the packet has some blacklisted fields). Therefore, it would be redundant to repeat the same constraint also for the send function since whenever is possible, if there is a send, there is also a receive. The only exceptions for this assumption are the clients and the server. The former sends a packet without receiving one, while the latter receive a packet but it doesn't send any. For this reason, between the two function, the implication has been based on the recv one instead of the send, because for the client (which sends only), the code builds the packet, thus it is certain to be correct. On the other hand, the server receive a packet that is built by z3 though the various conditions and implications, hence it is necessary to specify the range of the addresses otherwise some unwanted results may appear. So with (6.19) only, all of the cases are covered with the minimum number of additional conditions. For the ports intervals:

$$\begin{aligned}
&\forall\{n0,n1,p0\}: \\
&\qquad recv(n0,n1,p0) \implies \\
&\qquad\quad p0.src\_port.start > 0 \land p0.src\_port.end < MAX\_PORT \\
&\qquad\quad \land p0.dst\_port.start > 0 \land p0.dst\_port.end < MAX\_PORT
\end{aligned} \tag{6.20}$$

where MAX_PORT is defined as a constant and it's equal to 65535.
For the IP related fields, the assignment in (6.18) can be therefore considered as a view at a higher level of the following formulas (which are the ones that effectively are fed into z3):

$$\begin{aligned}
&Soft(src_0 == 0, k, \text{``}rules\text{``}) \\
&Soft(src_1 == 0, k, \text{``}rules\text{``}) \\
&Soft(src_2 == 0, k, \text{``}rules\text{``}) \\
&Soft(src_3 == 0, k, \text{``}rules\text{``})
\end{aligned} \tag{6.21}$$

with $src = (src_0, src_1, src_2, src_3)$. To keep this consistency, one hard constraint is added to correlate the *src* variable, that will be used to check for matching fields in the packets as shown afterwards, with the src_i, to which a value will be assigned by z3.
To improve the potential of the autoconfiguration task, the decimal numbers of the IP addresses can also be assigned to be equal to a wildcard in order to allow z3 to generate rules that exploit that feature. This obviously leads to a further minimization of the total number of rules. To express this possibility in z3, the assumption that has been made is that the value "-1" is considered to be the wildcard. Therefore, to implement this feature the following declarations must be added:

$$\begin{aligned}
&Soft(src_0 == -1, c, \text{``}wildcards\text{``}) \\
&Soft(src_1 == -1, c, \text{``}wildcards\text{``}) \\
&Soft(src_2 == -1, c, \text{``}wildcards\text{``}) \\
&Soft(src_3 == -1, c, \text{``}wildcards\text{``})
\end{aligned} \tag{6.22}$$

In (6.22) it is important to notice that the class is different from the previous declaration (the weight *c* can also be different from the previous *k* but it doesn't matter as they are in different classes). Using a different class makes z3 capable of distinguish between a "null" rule and a rule with wildcards otherwise it will use them indistinctly. Here only the src variable is shown but deriving the variables for the dst one is straightforward.
A generic rule is then declared as a boolean condition that returns true if the fields of a generic packet p0 match the soft clauses declared in (6.18). At a high level, the rule declaration can be seen as follows:

$$\begin{aligned}
rule = (&p0.src == src \land \\
&p0.dst == dst \land \\
&p0.protocol == protocol \land \\
&p0.src\_port == src\_port \land \\
&p0.dst\_port == dst\_port)
\end{aligned} \tag{6.23}$$

where the equalities $p0.scr == src$ and $p0.dest == dst$, also consider the possibility to have wildcards (e.g. the comparison between 10.0.0.1 and 10.0.0.-1 returns true). This is achieved transforming the mentioned equalities as follows:

$$\bigwedge_{\forall i \in \{0,1,2,3\}} p0.src_i == src_i \lor src_i == -1 \tag{6.24}$$

The condition declared in (6.23) defines the way a packet matches a firewall rule. For each requested policy, one rule is created to work as a placeholder considering the worst-case scenario (i.e. every policy needs one rule to be satisfied), however z3 will always assign significant values only to the least number of rules. Nevertheless, every firewall in the service graph will have a number of placeholder rules equal to the number of the policies even if a particular firewall is never traversed by some of the flows declared in those policies (i.e. the firewall is in neither of the paths that link a source to a destination specified in a policy, thus it will never be able to affect that specific flow) leading to the declaration of some unnecessary variables. A brief discussion about the performance is carried out in the next section.
At this point the firewall behaviour is modelled with a set of hard constraints that use the rules declared in (6.23). The model differs based on the default action that has been declared in the input XML (if no default action is declared by the user, to have a more conservative approach, the default action is DENY, i.e. drop every packet). If the default action is ALLOW, the firewall is then modelled by means of the following formulas:

$$\begin{aligned}&\forall\{next, p0\}:\\ &\qquad send(firewall, next, p0) \implies\\ &\qquad\quad \exists\{previous\}|recv(previous, firewall, p0) \land \neg rule\end{aligned} \tag{6.25}$$

$$\begin{aligned}&\forall\{previous, p0\}:\\ &\qquad recv(previous, firewall, p0) \land \neg rule \implies\\ &\qquad\quad \exists\{next\}|send(firewall, next, p0)\end{aligned} \tag{6.26}$$

The formula (6.25) states that if a firewall wants to send a packet p0, that same packet must have been received and the rule condition returns false (this means that p0 does not match the rule). The formula (6.26) obliges the firewall to send every packet p0 that has been received and does not match the rule. The previous formulas assume that there is only one placeholder rule (i.e. only one requested policy). If there are more than one policy, instead of the *rule* condition there is

$$\bigvee_{\forall i|r_i \in R} r_i \tag{6.27}$$

where R is the set of all the placeholder rules. This ensures that the correct behaviour is applied even if there are multiple rules to consider. If the default action is DENY the constraints are the same as before but there is no negation in front of the *rule* condition as shown below:

$$\begin{aligned}&\forall\{next, p0\}:\\ &\qquad send(firewall, next, p0) \implies\\ &\qquad\quad \exists\{previous\}|recv(previous, firewall, p0) \land rule\end{aligned} \tag{6.28}$$

$$\begin{aligned}&\forall\{previous, p0\}:\\ &\qquad recv(previous, firewall, p0) \land rule \implies\\ &\qquad\quad \exists\{next\}|send(firewall, next, p0)\end{aligned} \tag{6.29}$$

In (6.28) and (6.29) the logic is inverted respect to the one described previously to fit the opposite default action.
When enforcing the requested policies, one last condition is added to enforce the constraints of the specified flow. The constraint for each policy is the following:

$$\begin{aligned}&\forall\{next, p0\}:\\ &\qquad send(firewall, next, p0) \implies\\ &\qquad\quad p0.lv4proto == specified\_lv4proto\\ &\qquad\quad \land p0.src\_port == specified\_scr\_port\\ &\qquad\quad \land p0.dst\_port == specified\_dst\_port\end{aligned} \tag{6.30}$$

where the $specified_*$ variables are extracted from the XML element of that particular policy. Having the possibility to falsify the assignments in (6.18), z3 can choose the right value for a soft clause in order to satisfy the requested policies in accordance with the default action of the firewalls, using as few as possible rules.
To have a practical example, the scenario in Figure 6.16 will be considered. For the sake of simplicity, the assumption made in this scenario is that both firewalls have been configured by the user with a default action of ALLOW, so that all the packets that do not match any rule are forwarded, and the auto generated rules only contain the IPs.

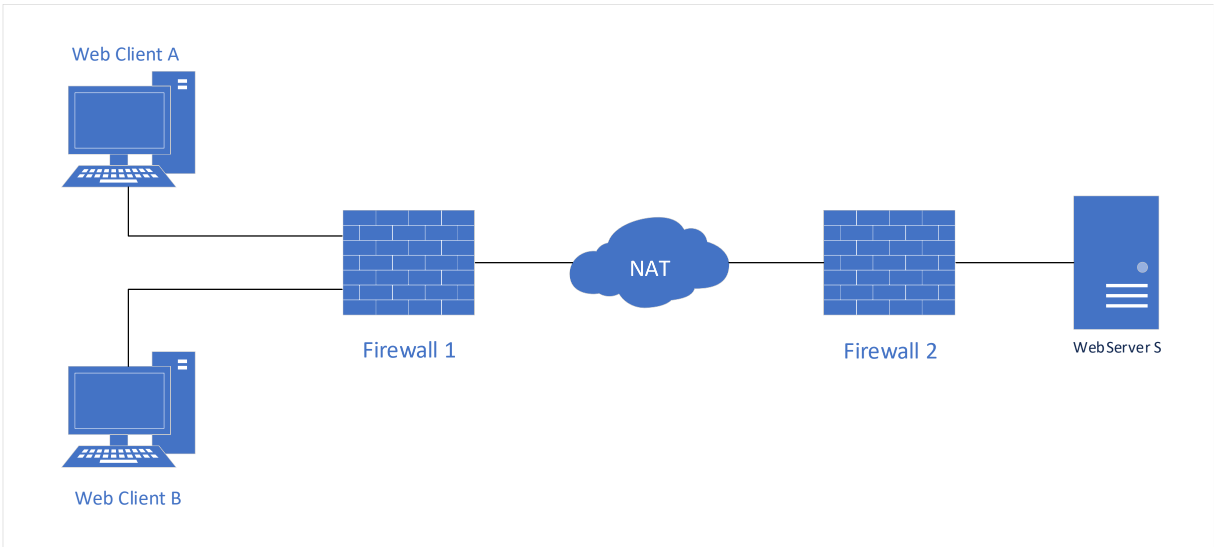


Figure 6.16. Example Scenario

Requiring that both A and B can reach S, will result in no auto-configured rule, while, when the requested policies require that S should be reachable from A, but not from B, there will be one auto-configured rule in the first firewall that drops all the packets from B that are directed to S. The optimization capability of z3 can be seen when the policies state that neither A nor B must be able to reach S. In fact, the presence of a NAT that hides both clients, allows to have a single rule in the second firewall that is able to drop the packets of both clients which will have as source address the NAT address (another possibility would be to have a single rule in the first firewall with a wildcard). A summary of all the cases can be seen in Figure 6.17. The same scenario with both firewalls configured with a default action of DENY, (i.e. if a packet does not match any rule, it is dropped) will give the results shown in Figure 6.18.

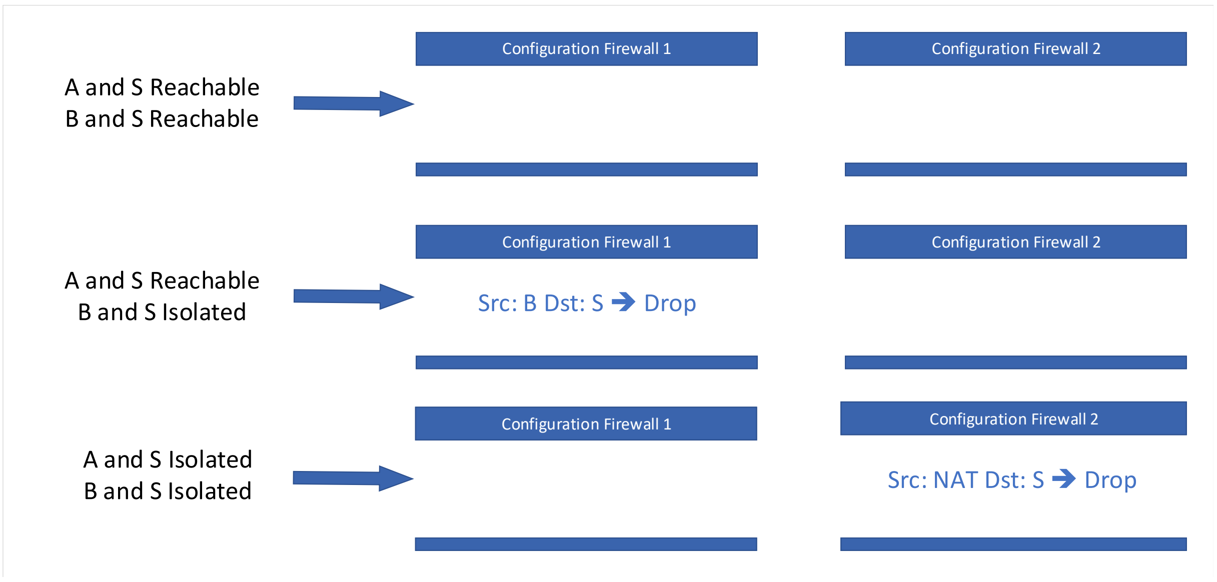


Figure 6.17. Autoconfiguration Result

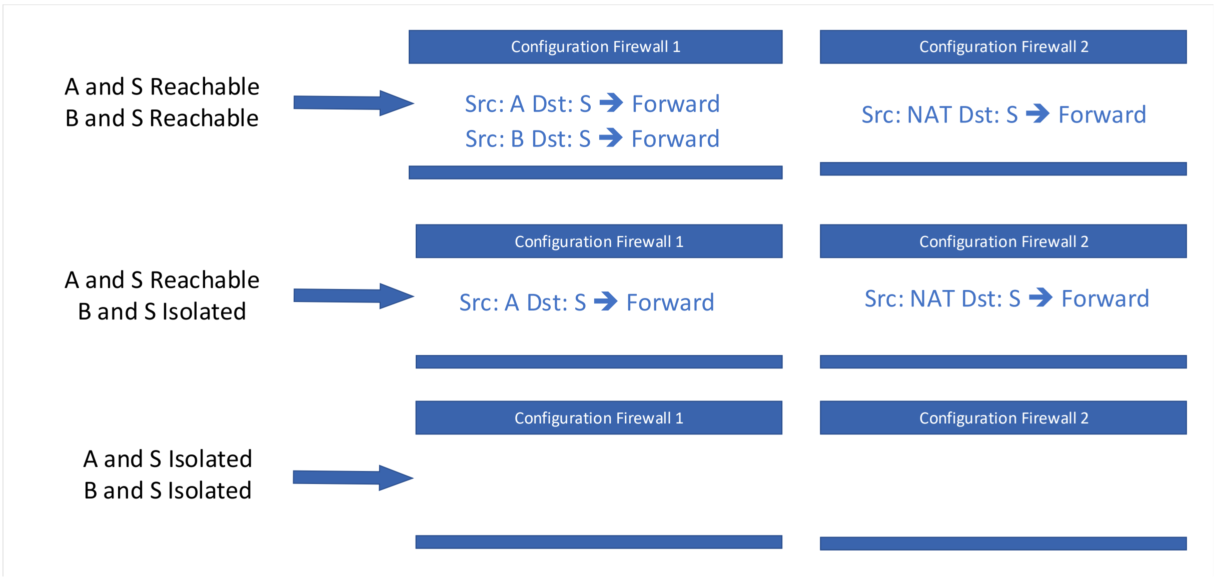


Figure 6.18. Autoconfiguration Result

6.4.9 Auto-placement

Directly correlated with the auto-configuration feature described in the previous chapter, there is the auto-placement. With the auto-configuration, Verifoo can generate the rules needed in a VNF in order to satisfy the requested policies and with the auto-placement it extends further its capabilities. In fact, with the proper input, Verifoo can understand that not all the VNFs are needed in the final solution and it can decide which one can be omitted keeping the policies satisfaction unaltered. A VNF is considered to be unnecessary if the set of its auto-configured rules is empty and there is an alternative path towards the destination that does not include that VNF (otherwise it will be deployed even if it has an empty configuration because its presence is requested for network needs). At the moment this feature, like the auto-configuration, has been implemented only for the firewall, the DPI and the antispam. A node of one of those functional types, can be enhanced with the auto-placement feature if properly indicated in the input XML, as shown below:

Listing 6.43. XML Example

```xml
<graphs>
  <graph id="0">
    ...
    <node functional_type="FIREWALL" name="fw1">
      <neighbour name="client"/>
      <neighbour name="nat"/>
      <configuration description="A simple description" name="conf1">
        <firewall/>
      </configuration>
    </node>
    ...
  </graph>
</graphs>
<Constraints>
    <NodeConstraints>
      <NodeMetrics node="fw1" optional="true"/>
    </NodeConstraints>
    <LinkConstraints/>
</Constraints>
```

Thanks to the indication in the NodeMetrics XML element, Verifoo will keep track of the optional nodes and of all the paths that include those nodes. These paths will be available only if the correspondent optional node is actually used. This last aspect has been enforced modifying the network behaviour formulas to correlate the choice of a next hop with its state (i.e. used or not used). This choice needs to be made only on those nodes that have an optional node as a possible next hop.
Formally, declaring:

- n_1 as a generic node that is directly connected with optional nodes
- p_0 as a generic packet that goes through a node
- $dest$ as the final destination of the packet
- $N_{n_1\_opt}$ as the set of all the neighbours of n_1 that can reach the destination and are optional
- $N_{n_1\_no\_opt}$ as the set of all the neighbours of n_1 that can reach the destination and are not optional
- $n_i.used$ to refer to the boolean variable that defines if n_i is used or not

the formulas that model a correct network behaviour are the following:

$$\begin{aligned}&\forall\{n_0, p_0\}:\\ &\qquad [send(n_1, n_0, p_0) \wedge p_0.dest == dest]\\ &\qquad\quad \Rightarrow \left[\bigvee_{\forall i | n_i \in N_{n_1\_opt}} n_0 == n_i \wedge n_i.used\right]\end{aligned} \tag{6.31}$$

$$\begin{aligned}&\forall\{n_0, p_0\}:\\ &\qquad [send(n_1, n_0, p_0) \wedge p_0.dest == dest]\\ &\qquad\quad \Rightarrow \left[\bigvee_{\forall i | n_i \in N_{n_1\_no\_opt}} n_0 == n_i\right] \wedge \left[\bigwedge_{\forall j | n_j \in N_{n_1\_opt}} \neg(n_j.used)\right]\end{aligned} \tag{6.32}$$

The previous two formulas are an extension of the ones presented in [9] and ensure that a packet will only be sent to nodes that are actually used. The formula (6.31) ensures that if an optional node n_i is chosen as a next hop for a certain packet, the correlated boolean value of its *used* attribute is true. For a generic node n_1, the formula (6.32) excludes the possibility that if a non-optional node has been selected as a next hop, the *used* attribute of the optional nodes directly connected with n_1 is set as true (i.e. if a non-optional neighbour is used, all the optional neighbours must be unused). The previous two formulas are required to have a correct network behaviour. Further extensions have been made in order to model

how z3 decides if a VNF is used or not, and how the deployment of the nodes onto the physical topology is coherent with this new variable.
As said previously, the *used* variable is a boolean expression, handled by z3, present only in those VNFs that support the auto-placement feature. Its actual value is decided by z3 based on the following formulas where n_1 represents a VNF that supports the auto-placement:

$$\forall n_1 : \quad \bigvee_{\forall rule \in R} (rule \neq null) \implies n_1.used \tag{6.33}$$

$$\forall n_1 : \quad \neg(n_1.used) \implies \bigwedge_{\forall rule \in R} (rule == null) \tag{6.34}$$

where R is the set of all the auto-generated rules, while the meaning of a "null" rule depends on the specific type of VNF (e.g. for a firewall this translates in a rule that has source and destination equal to 0.0.0.0). Since this feature is an extension of the auto-configuration, how *rule* is generated follows the same pattern as in the previous chapter. Defined the firewall default action (ALLOW or DENY), z3 generates only rules of the other type (e.g if the default action is ALLOW, all the rules express a DROP action) in order to satisfy the requested policy.
In addition to the auto-configuration, the previous formulas force a not used VNF to have all its rules equals to null (6.34) and ensure that if at least one rule is not null, the VNF is used (6.33).
As concerns the deployment behaviour, an extension has been made on the assertions presented in [9]. The variables that indicate on which host h_j a node n_i can be deployed, have been modified (in [9] they were referred to as x_{ij} while here are referred to as $d_{n_ih_j}$, but they have the same meaning). In particular, for an optional node:

$$\begin{gathered} d_{n_ih_j} \\ becomes \\ d_{n_ih_j} \vee \neg(n_i.used) \end{gathered} \tag{6.35}$$

Therefore:

$$\begin{gathered} \left[\sum_{\forall i | n_i \in M_{h_j}} int(d_{n_ih_j}) \right] = 1 \\ becomes \\ \left[\sum_{\forall i | n_i \in M_{h_j}} int\left(d_{n_ih_j} \vee \neg(n_i.used)\right) \right] = 1 \end{gathered} \tag{6.36}$$

where $M_{h_j} \subseteq N$ and represents the collection that contains all the nodes that can be deployed on the host h_j (N represents the set of all nodes, instead). This modification ensures that even if a node is not used (i.e. all the $d_{n_ih_j}$ are equal to false) the sum is still equal to one, avoiding an inaccurate z3 behaviour.
Finally, to have the minimum number of deployed VNFs a soft constraint is added for each optional node n_1:

$$\forall n_1 : \quad Soft([int(\neg n_1.used) == 1], k, \text{“}placement\text{“}) \tag{6.37}$$

To have a practical example, the scenario in Figure 6.19 will be considered. For the sake of simplicity, only the graph and the network behaviour will be analyzed since the deployment is its direct consequence (i.e. only the the used VNFs are deployed). An intuitive view of the scenario is depicted in the following image:

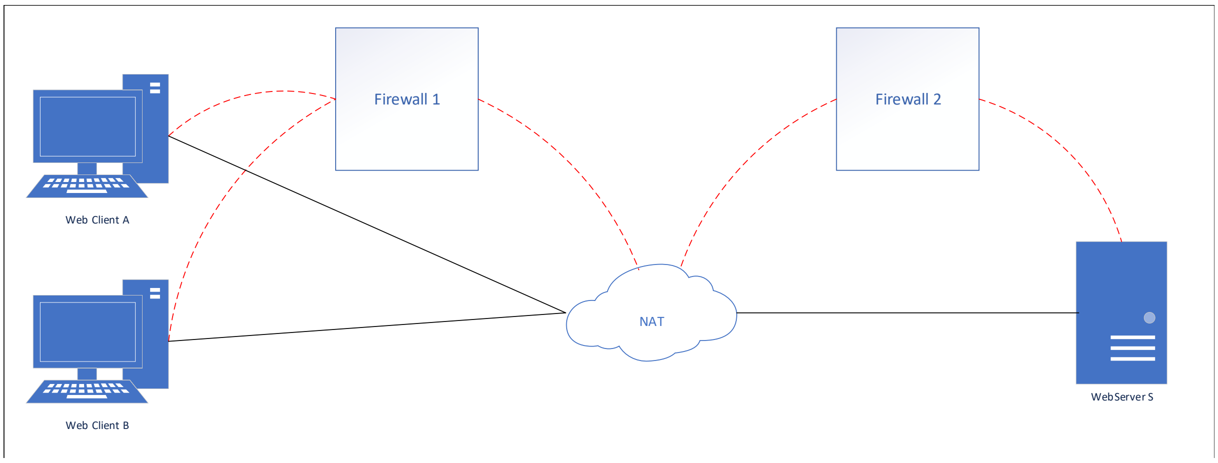


Figure 6.19. Example Scenario

The firewalls are both optional VNFs and the dashed lines, in contrast to the normal ones, highlight the path that can be taken only if the optional node is used. It is important to notice that even without the firewalls, the clients are still able to reach the destination thanks to the direct links with the NAT. If these links were not present, the firewall would be deployed even without having significant rule, only for a network necessity. The XML that represents the scenario in Figure 6.19 is the following:

Listing 6.44. XML Example

```
<graphs>
    <graph id="0">
     <node functional_type="WEBCLIENT" name="a">
        <neighbour name="fw1"/>
        <neighbour name="nat"/>
        <configuration description="A simple description" name="confA">
          <webclient nameWebServer="s"/>
        </configuration>
      </node>
      <node functional_type="WEBCLIENT" name="b">
        <neighbour name="fw1"/>
        <neighbour name="nat"/>
        <configuration description="A simple description" name="confA">
          <webclient nameWebServer="s"/>
        </configuration>
      </node>
      <node functional_type="FIREWALL" name="fw1">
        <neighbour name="a"/>
        <neighbour name="b"/>
        <neighbour name="nat"/>
        <configuration description="A simple description" name="conf1">
          <firewall/>
        </configuration>
      </node>
       <node functional_type="NAT" name="node2">
        <neighbour name="a"/>
        <neighbour name="b"/>
        <neighbour name="fw1"/>
        <neighbour name="fw2"/>
        <neighbour name="s"/>
        <configuration description="A simple description" name="conf2">
          <nat>
            <source>nodeA</source>
            <source>nodeC</source>
          </nat>
        </configuration>
      </node>
       <node functional_type="FIREWALL" name="fw2">
        <neighbour name="nat"/>
        <neighbour name="s"/>
        <configuration description="A simple description" name="conf1">
          <firewall/>
        </configuration>
      </node>
      <node functional_type="WEBSERVER" name="s">
        <neighbour name="nat"/>
        <neighbour name="fw2"/>
        <configuration description="A simple description" name="confB">
          <webserver>
            <name>s</name>
          </webserver>
        </configuration>
      </node>
    </graph>
  </graphs>
  <Constraints>
      <NodeConstraints>
        <NodeMetrics node="fw1" optional="true"/>
        <NodeMetrics node="fw2" optional="true"/>
      </NodeConstraints>
      <LinkConstraints/>
  </Constraints>
```

The automated generation of the rules follows the same pattern as for the auto-configuration as shown below (there is the assumption that the firewalls have been configured with a default action of ALLOW):

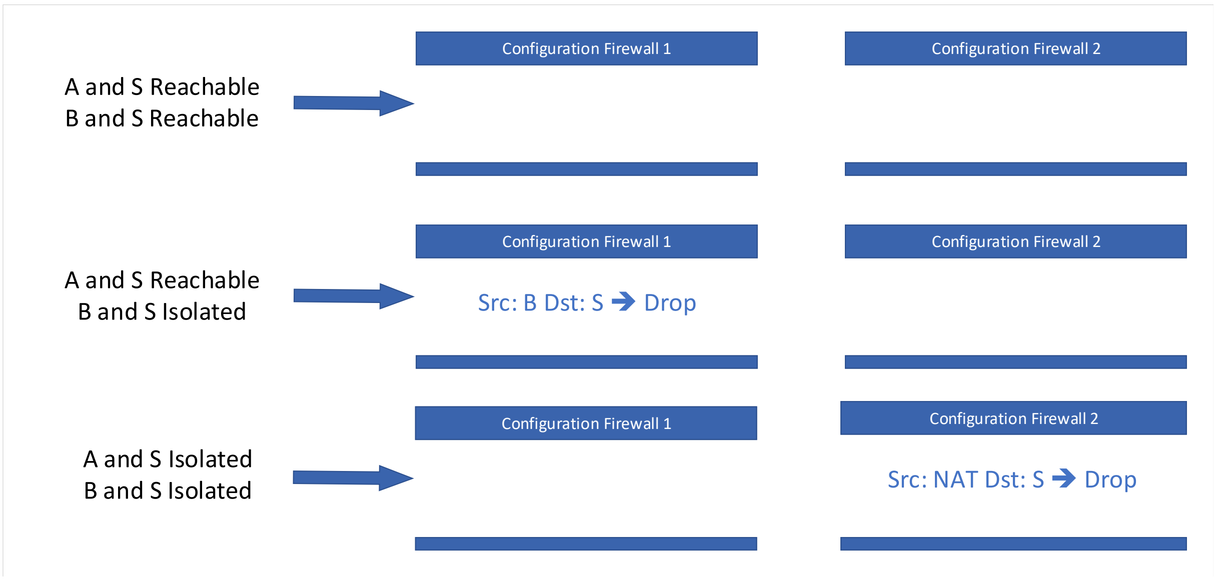


Figure 6.20. Auto generated rules

The only difference is that, a firewall with no rules is excluded from the final graph as can be seen in Figure 6.21 that depicts the situation where the policies require that S should be reachable from A, but not from B (similar behaviours can be seen for the other situations).

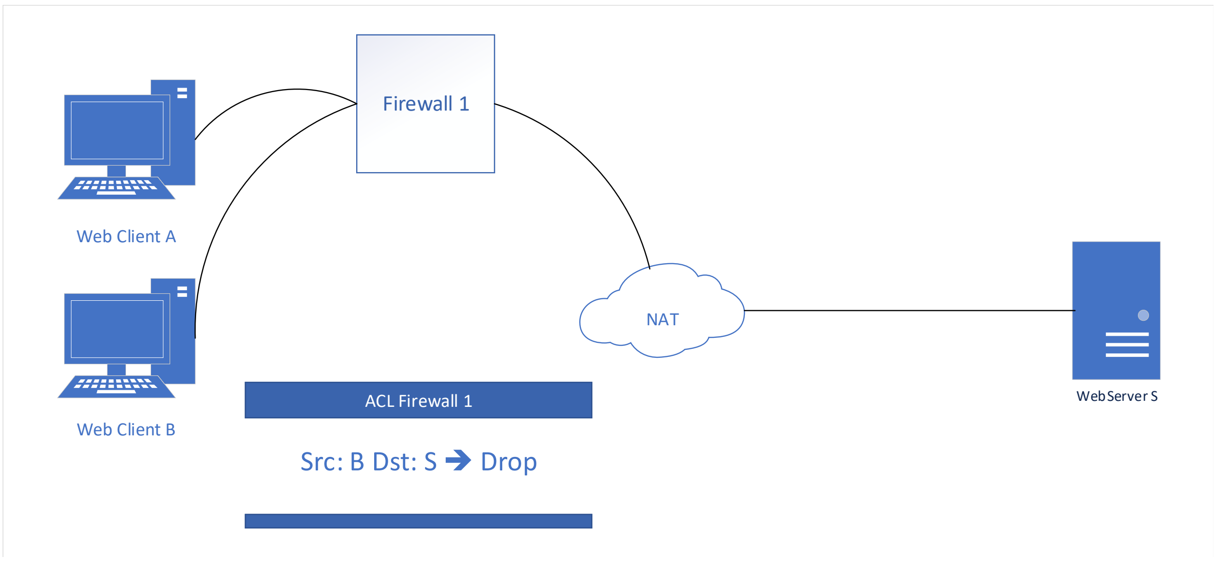


Figure 6.21. Auto generated rules

Chapter 7

Experimental Results

7.1 Verification & Deployment Performance

The new model has been evaluated by performing a number of experiments with real data sets. The data set consists of real physical topology characterized by a different number of hosts and connections between hosts. For each topology scenario, the evaluation has been performed by trying to deploy a defined number of VNFs, changing the position of the fixed end points. The result is the average time taken by all iterations. The need of a series of iterations is crucial to avoid that fluctuations caused by other operation on the machine affect the final result. Moreover, the z3 computation itself has a certain degree of variance for the resolution of the same problem.
All experiments have been executed in a virtual machine on a workstation with 32GB of RAM and an Intel i7-6700 CPU at 3.40GHz, in the JVM environment version 8.

7.1.1 Old Model and New Model

In the first experiment, the average time taken using the new model, is compared with the time taken by the old one and the results have been reported in table 7.1.1.

Topology	Hosts	Connections	VNFs	Old Model	New Model	Difference in %
Internet2	12	15	4	2.857 s	2.876 s	+0.67 %
GEANT	23	74	4	5.256 s	5.103 s	-2.91 %
UNIV1	23	43	4	10.571 s	10.515 s	-0.53 %

The results show that the introduction of the new model affects the average time in a negligible way. This is very important as it introduces no need to repeat all the past evaluations which are still valid.

7.1.2 Deployment Constraints

The next experiment provides a comparison between the different time taken by the new model considering an increasing number of types of deployment constraints for the VNFs (e.g. the memory that the VNF will occupy, or the minimum number of cores it requires). This constraints are added for each node, so the total number of hard constraints that are added to the z3 computation is equal to the number of types of constraints multiplied by the number of nodes that require a deployment. The constraints have values that do not prevent any of the deployment to be considered (e.g. all nodes require 1GB of RAM while every host has more than enough RAM to satisfy the needs of all the nodes), as this experiment only wants to retrieve the additional computational time needed by z3 to verify these further constraints. If there are not enough resources available for the placement plan then the solver returns UNSAT without any model. The methodology applied for the experiment, as well as the physical topologies used, are the same that were described previously.

Topology	0 Constr.	1 Constr.	2 Constr.	3 Constr.	4 Constr.	5 Constr.
Internet2	2.876 s	3.519 s (+22.36%)	3.498 s (+21.63%)	3.250 s (+13.00%)	3.628 s (+26.15%)	3.401 s (+18.25%)
GEANT	5.103 s	6.257 s (+22.61%)	6.331 s (+24.06%)	6.368 s (24.79%)	6.599 s (29.32%)	6.919 s (+31.64%)
UNIV1	10.515 s	11.855 s (+12.74%)	11.433 s (+8.7%)	11.854 s (+12.73%)	11.228 s (+6.78%)	11.621 s (+10.52%)

In the table only the number of constraints is specified because similar performances are obtained regardless of the type of constraints. Moreover, the results show that adding the constraints led to a performance degradation, but the main difference is made by the presence or absence of constraints and not by their number. This is due to the internal optimization of the z3 tool respect to the hard constraints.

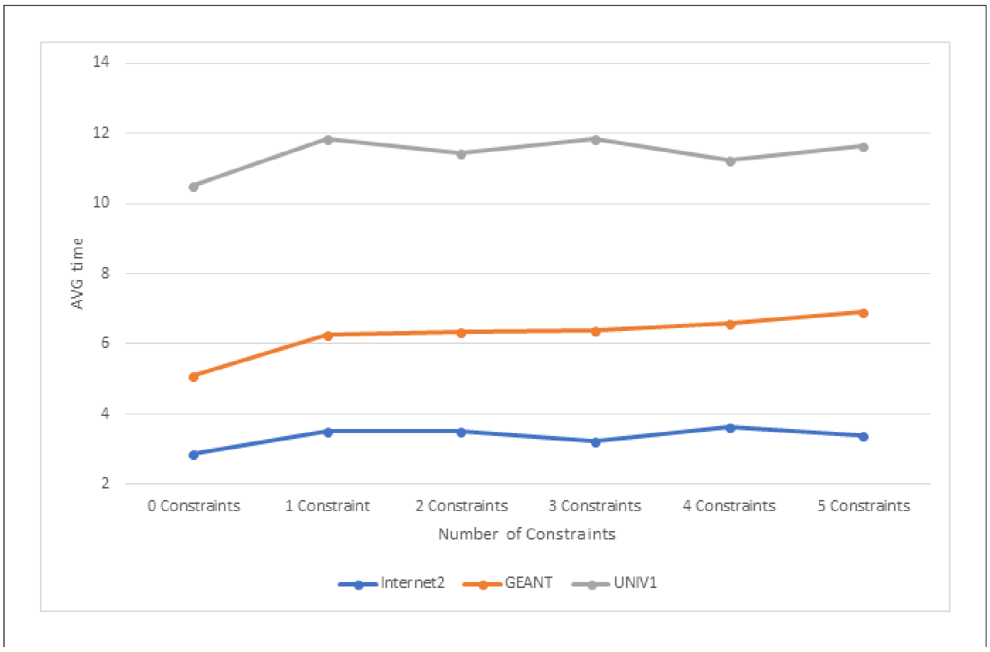


Figure 7.1. Constraints Performance Results

7.1.3 Service Graph and Chains

The third experiment evaluates the performance of Verifoo when the service graph that is being deployed is an actual graph and not only a chain, applying no other constraints. The service graph in Figure 7.2 is taken as example.

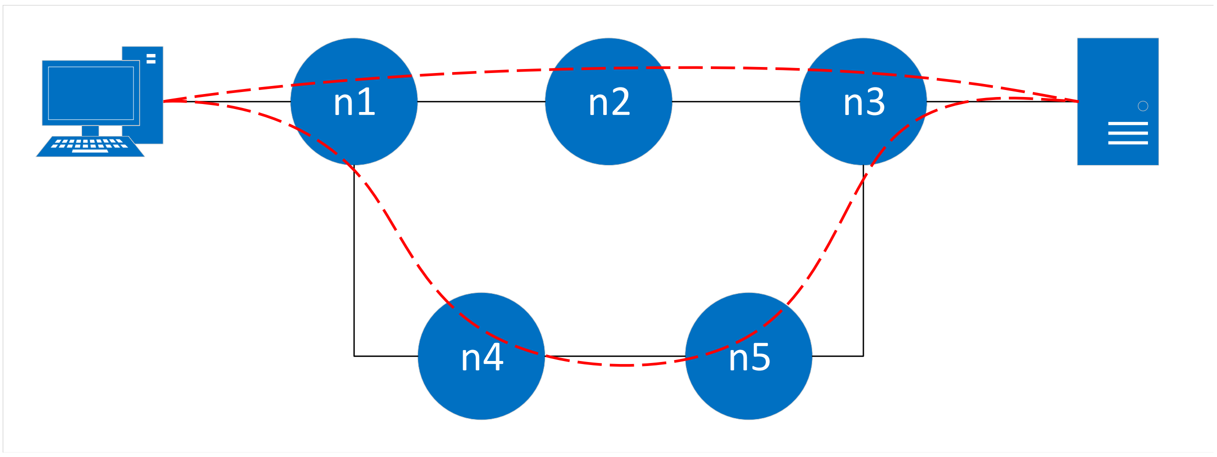


Figure 7.2. Graph Example

As shown in the figure, deploying this service graph is comparable to deploying two different chains, i.e. one formed by the nodes n1-n2-n3 and the other formed by n1-n4-n5-n3. In the following experiment the two chains are deployed sequentially, with the result of the first deployment that impacts on the second one, i.e. if n1 has been deployed on a certain host after the first deployment, z3 will be forced to have it there during the computation of the second deployment. Taken this into account, table 7.1.3 provides the experimental results.

Topology	Two Chain	Service
Internet2	3.308 s	2.429 s (-26.57 %)
GEANT	7.341 s	3.995 s (-53.75%)
UNIV1	11.092 s	9.224 s (-16.84%)

The results show a great performance improvement that can also be explained by the fact that a graph has some intrinsic constraints on how the nodes can be deployed (e.g. n1 must be deployed on a host that can directly reach both the hosts on which n2 and n4 are deployed), resulting in a fewer number of possible deployment scenario and consequently in a fewer number of formulas computed by z3. Moreover, the total number of the nodes in the two chains is greater than the total number of the nodes in the graph, as n1 and n3 are shared among the chains but they are treated as if they were different. The new model of Verifoo ensures that the graph is deployed as a whole and not as the consequential deployment of more chains, contributing to the performance boost.

7.2 Scalability Performance

A series of experiments have been performed to evaluate the scalability of the current model of Verifoo. For these tests, a set of random inputs, generated using a random generator developed exactly for Verifoo, has been used. For all the input for which it was possible to define a deployment (i.e. z3 returned a SAT result), the total computational time was registered. All the shown data represents the average computation time, determined on scenarios with different number of nodes and hosts.

7.2.1 General Model Performance

In this section, the performance tests, and their obtained results, all refer to scenarios in which there is no auto-configurable VNF and the task given to Verifoo is to find the optimal placement for a network service onto a physical topology, performing also a verification of the requested policies. These assumptions are important because some scalability issues appear when using the autoconfiguration for a firewall, but this will be discussed afterwards. The evaluation has been performed separately for hard and soft constraints because they introduce different level of complexity. In fact, while the former is simpler for z3 because it only checks if a constraint is verified or not, the latter introduces a higher complexity since z3 also tries to optimize the weight of non-falsified constraints.

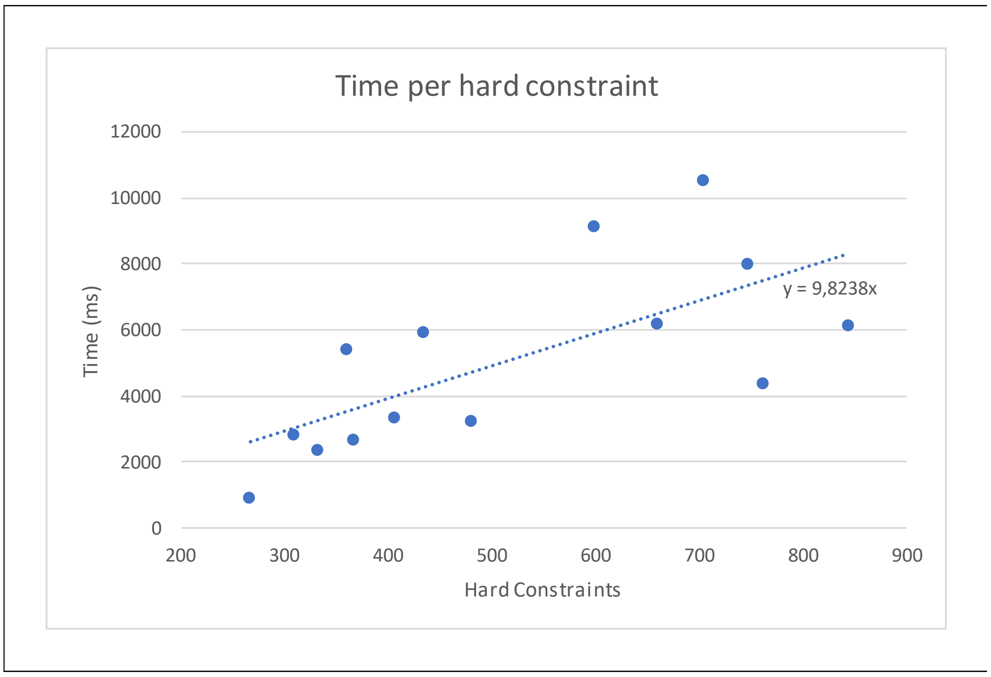


Figure 7.3. Performance results for hard constraints

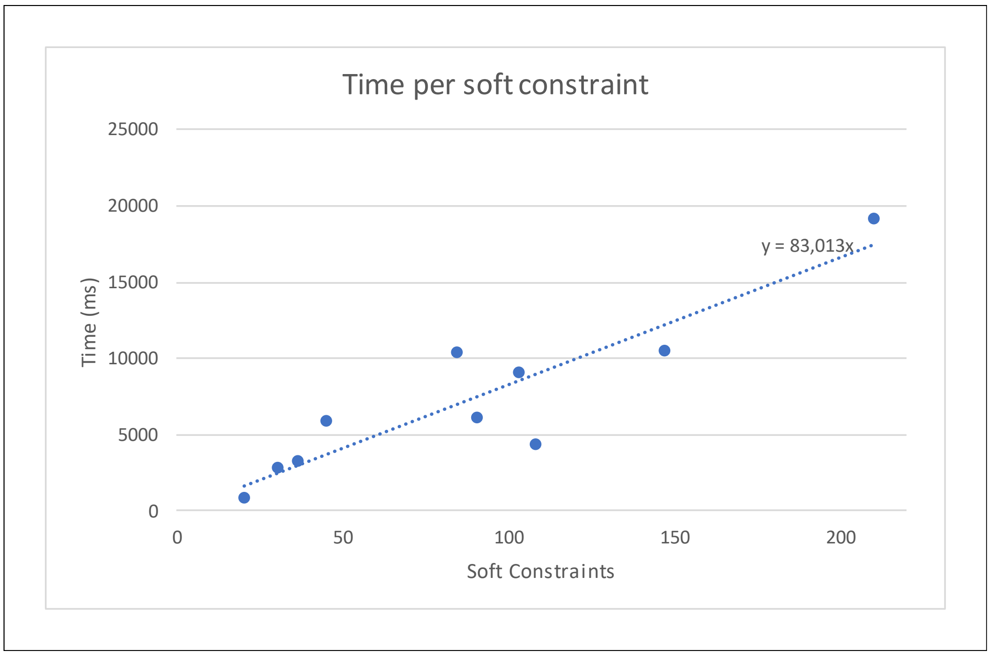


Figure 7.4. Performance results for soft constraints

As shown in the previous charts, with the current models implemented in Verifoo, one hard constraint adds on average 9ms to the total computation, so only a significant increase in their number can cause a perceivable slowdown in performance. Regarding the soft constraints, instead, the correspondent chart shows that one of them slows down the execution of Verifoo of about 83ms, an order of magnitude greater respect to the hard constraints. The practical implications this value has, can be seen considering figure 7.5.

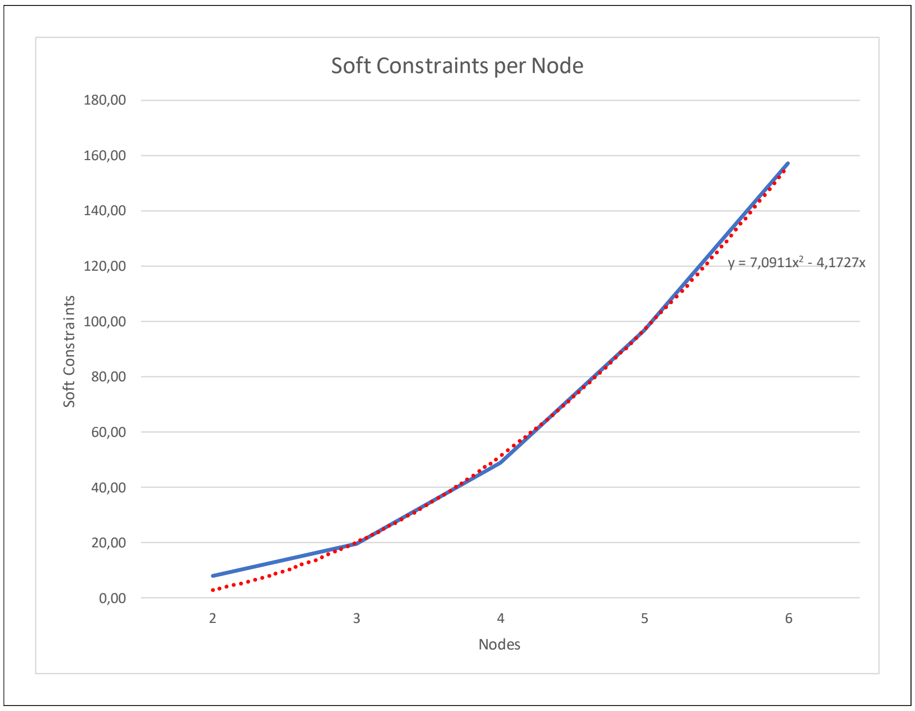


Figure 7.5. Soft Constraints per node

This chart shows how the number of soft constraints increases with the number of VNFs present in the network service. The relation is not linear because the deployment of a node affects also the one of its neighbours. For example, increasing the nodes from 3 to 4, adds on average 25 soft constraints which, considering figure 7.4, brings about a slowdown of 2 seconds.
The presented data shows that, even though z3 is a powerful tool that allows to model very complex realities, the performance do not scale well with bigger scenarios when using the z3Opt module.

7.2.2 Autoconfiguration Performance

In this section, the autoconfiguration performance will be discussed with a particular focus on the scalability issues that currently affect the firewall. In fact, performance-wise, experimental results show that, for a firewall, enabling all the features described in the autoconfiguraton chapter, causes a significant slowdown. To understand how the performance degrades, some charts will be shown. In these charts, the plotted lines represent the temporal trend of the autoconfiguration task when the number of policies increases (the number of firewalls is fixed to one). To have an idea of which feature causes the greatest slowdown, the measurements have been performed considering various types of autoconfiguration, which are:

- Basic autoconfiguration, where the firewall generates rules that are composed only by IPs, without the possibility of using the wildcards (in this case the addresses are modelled in z3 using an EnumSort instead of a DatatypeSort)
- Quintuple autoconfiguration, which adds the possibility to generate rules that include also the protocol and the source and destination ports (the IPs are still EnumSort).
- Wildcards autoconfiguration, where the addresses are considered as a more complex data structure (here the DatatypeSort is used) and the firewall can use the wildcards to further minimize the number of rules. However, in this configuration only the IPs are present in the generated rule, therefore there are no protocol nor ports.
- All features autoconfiguration, in which the generated rules contain IPs, protocol and ports, with the IPs that can also have wildcards.

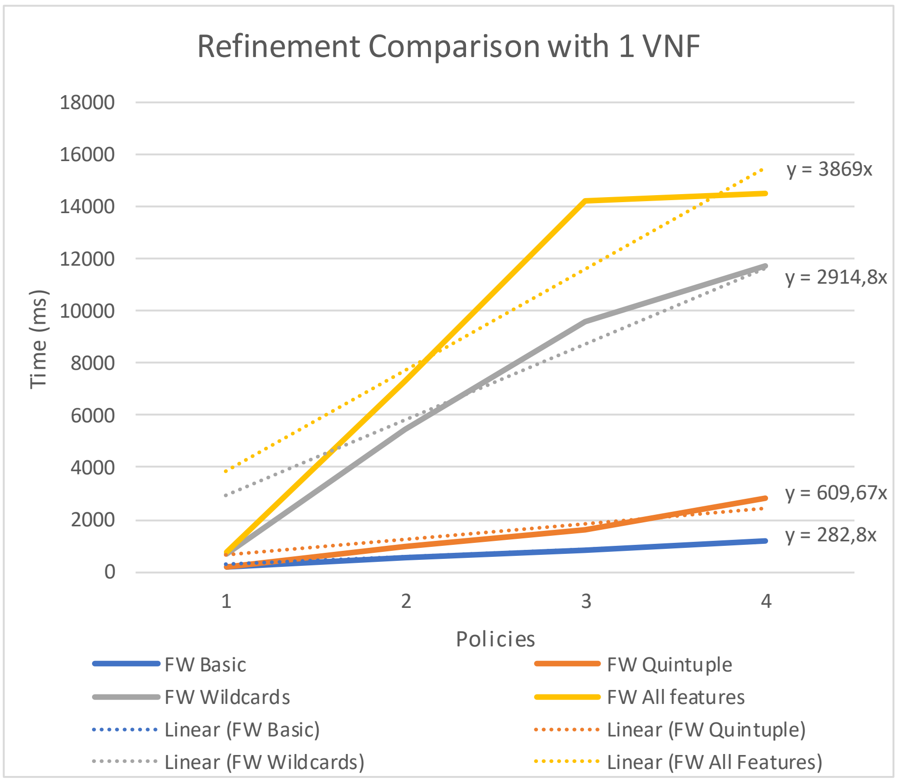


Figure 7.6. Performance results

As shown in chart 7.6, using a DatatypeSort to model the IPs (wildcard autoconfiguration), causes a significant slowdown with respect to its counterpart with a simple EnumSort (basic autoconfiguration). This concept can be further analyzed in the next chart. In that chart, both the basic and the quintuple autoconfigurations are the same as in the chart above, however, in addition, two other lines have been plotted. One of them refers to the temporal trend of the autoconfiguration task for a DPI, whose model is exactly the same as the firewall one, but the packet field that the DPI checks is set to be the body of the packet (modelled as an integer). The other added line is the result of a firewall that checks only the protocol field, which is also an integer field. This last configuration has been evaluated only for testing purposes and has obviously no real application.

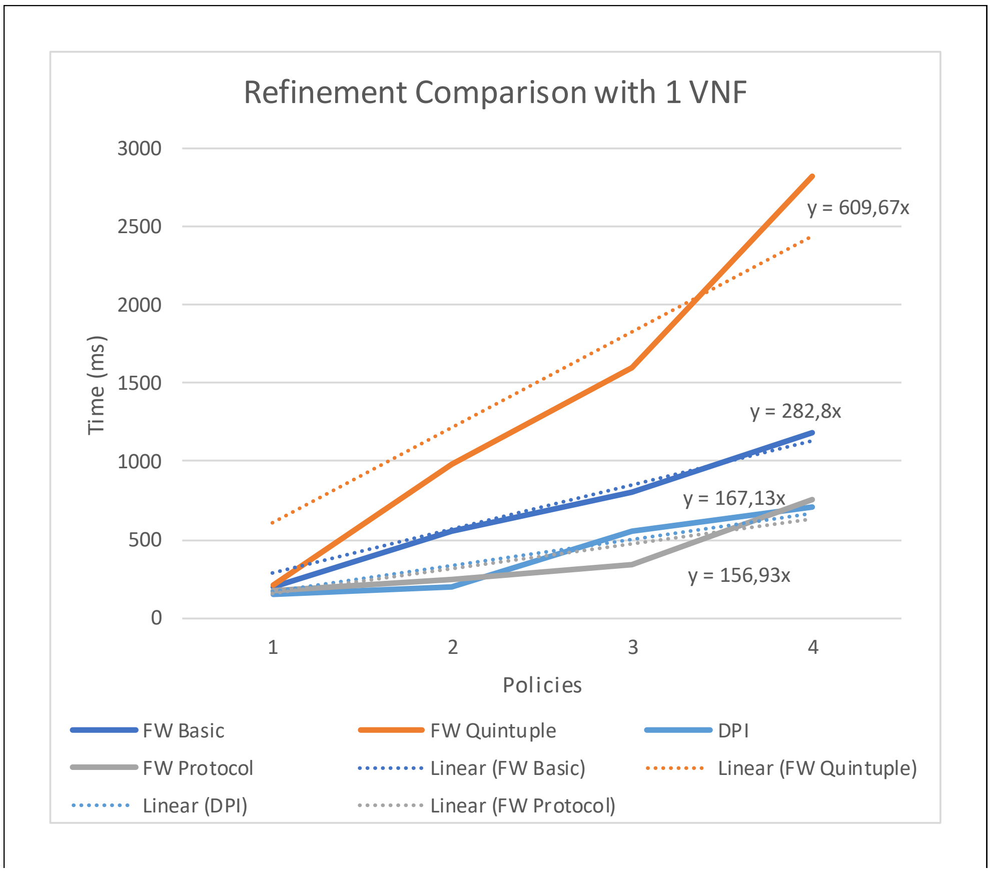


Figure 7.7. Performance results

As shown in chart 7.7, firewall and DPI have comparable performance when they both check an integer field. The performance begins to degrade when the field is an EnumSort to the point where the firewall has serious scalability issues when a DatatypeSort is used (the wildcards autoconfiguration in the previous chart).
In order to limit the scalability problem some modifications have been made on the behavioural model of the involved VNFs. In particular, these modifications improved the VNF formulas that were using the existential quantifier ∃ when referring to some neighbour in a *send* or *recv* function. For example, for a firewall:

$$\begin{aligned} &\forall\{next, p0\}: \\ &\qquad send(firewall, next, p0) \implies \\ &\qquad\quad \exists\{previous\}|recv(previous, firewall, p0) \wedge \neg rule \end{aligned} \tag{7.1}$$

To resolve this kind of pattern, z3 uses the skolemization in order to work only with universally quantified formulas. In fact, the skolemization is a special procedure that substitute every existentially quantified variable with a free function f based on the variable quantified by the universal quantifier that precedes the existential quantifier. To have an example, consider the following formulas:

$$\begin{gathered} \forall x\,[someConditionOn(x) \wedge \exists\{y\}|someOtherConditionOn(y)] \\ becomes \\ \forall x\,[someConditionOn(x) \wedge someOtherConditionOn(f(x))] \end{gathered} \tag{7.2}$$

Resolving this type of formulas can be heavy on the performance.
Considering formula (7.1), since in a graph the neighbours of a node are known, the existential quantifier can be replaced by the enumeration of the neighbours. Currently in Verifoo there is no distinction between previous nodes and next nodes, therefore in the enumeration are listed all the neighbours, however this does not creates any inconsistencies in the results because of how it is modelled the network

behaviour (i.e. the formulas introduced by the network behaviour excludes the possibility to send a packet to a specific destination back to a previous node). The formula becomes:

$$\forall p0: \left[\bigvee_{\forall i|n_i \in N} send(firewall, n_i, p0)\right] \implies \left[\bigvee_{\forall j|n_j \in N} recv(n_j, firewall, p0)\right] \wedge \neg rule \tag{7.3}$$

where N is the set of all the neighbours of the firewall. The performance before the modifications can be seen in figure 7.8, while the changes introduced by them can be observed in the figure 7.9. The performance has been evaluated with the simpler type of autoconfiguration, the BASIC one which is composed only by IPs and there is no possibility of using the wildcards. As can be seen, the modifications have halved the computational for the more complex scenario (more firewall and more policies), but the improvement effect fades in the simpler one.

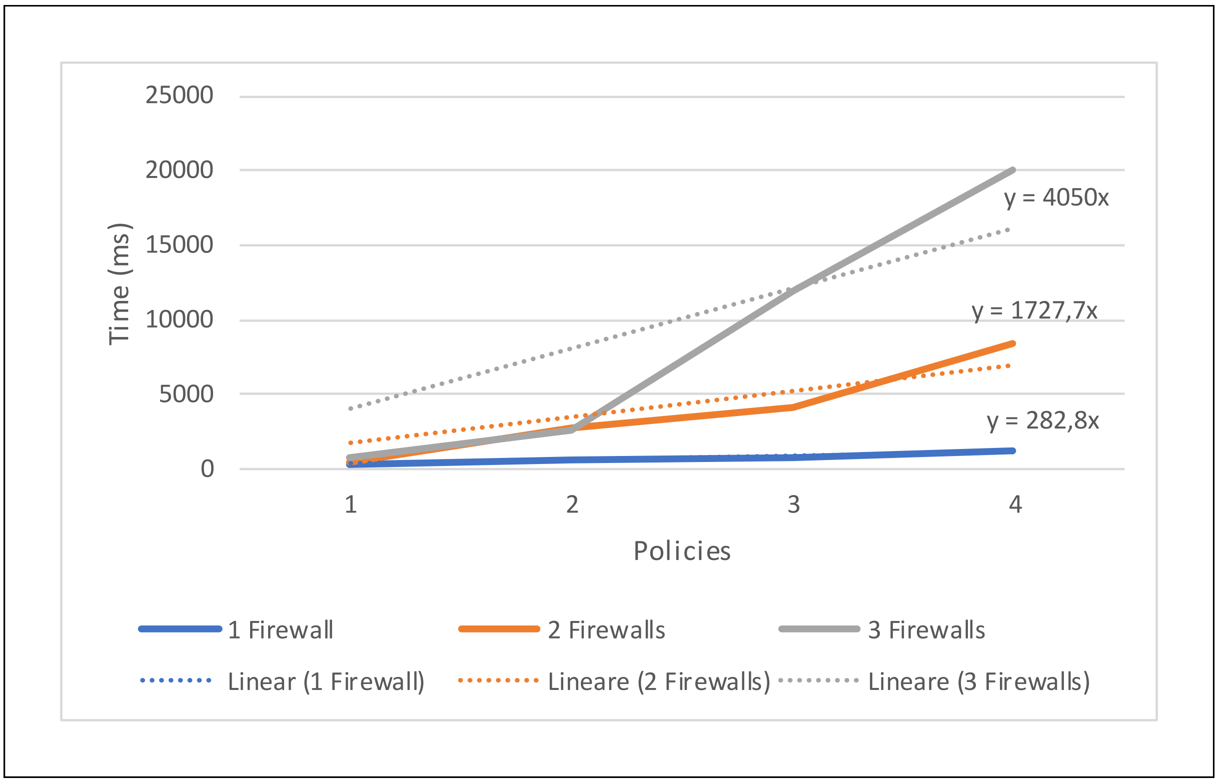


Figure 7.8. Firewall with BASIC autoconfiguration and no modifications

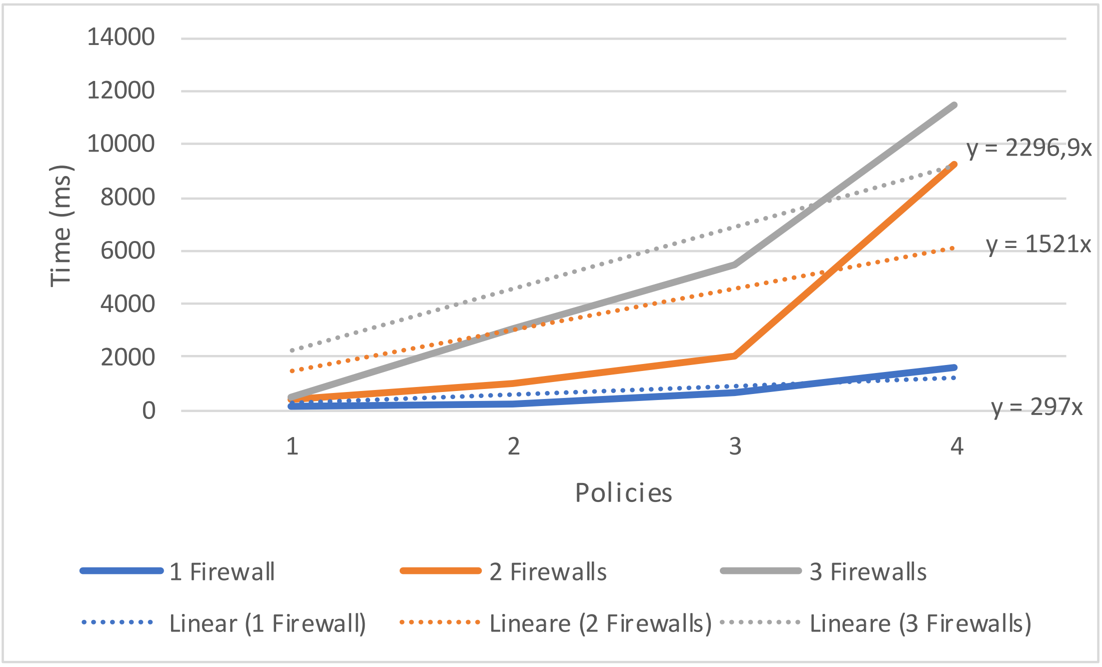


Figure 7.9. Firewall with BASIC autoconfiguration and modifications

Chapter 8

Conclusions and future improvements

This thesis work gave the possibility to enhance a tool, Verifoo, that was in its early stages of development. The tool fits into the new virtualization paradigms of SDN and NFV. In particular, it can be seen as a modular addition to an NFV Orchestrator in order to enhance its functionalities and cover those problems that the ETSI standardization left open as study items. Exploiting a theorem prover software like z3 and using Verigraph as a starting point, Verifoo sets its field of operations mainly in the deployment domain, obtaining the optimal solution with mathematical certainty.

The foundation of the work has been the developing of an XML schema that conveys the various information needed for the internal computation. Its structure presents a high degree of flexibility and modularity that allows future extensions with very few additions. Anyhow, in its current version, the XML schema already ensures the possibility to express a wide range of scenarios that can reproduce realistic network infrastructures and network services. This also allowed to improve the deployment algorithm, adding a series of new variables that enable a finer control on which resources of the physical hosts should be taken into consideration.

The first important extension implemented in the Verifoo model has been to extend the support for complex graphs with the possibility to include multiple endpoints, thus removing the previous limitation of being capable to handle only chains with one client and one server. Non-sequential graphs are widely used in an SDN architecture because they provide a globally consistent view of the network in real time which is aligned with the centralized approach of SDN. Therefore, the model extension in Verifoo allows to natively elaborate services designed with advanced virtualization technology.

An important part of this thesis work has been to explore new solutions for modelling the behaviour of a firewall. A first step towards this objective has been to improve the network simulation model computed by z3, adding some new packet headers like the level 4 protocol and the TCP/UDP ports. Moreover, in order to push to the limits z3 and have even more realistic use cases, the concept of wildcards has been introduced and modelled. However, the current formulas present in Verifoo only allow a classful addressing, but this can be obviously improved in future works evolving the wildcards into the concept of netmasks.

The aforementioned additions, allowed the evolution of the firewall model from a blacklist-only type to a more freely configurable one that behaves almost like the real counterpart. An open challenge remains though, namely the impossibility of using overlapping conflicting rules, an issue that is still debated for the real firewall implementations. Following the traces of the plentiful present literature, a series of solutions have been theorized and they could be used as a starting point for future works. The use of some more high-level concept, like the wildcards, caused conflicts when indicating some type of policies with the low level z3 implementation on which Verifoo was built on. To overcome these issues, a pre-processing task has been implemented in order to hide from the user the implementation details, thus avoiding limiting the expressivity of the tool.

A completely new opportunity arisen during the developing of the tool, was the idea of exploiting the already built framework to introduce the possibility to automatically generate configurations for the VNFs and remove from the final solution those that receive an empty one at the end of the computation. The auto-configuration module provides the rules, while the auto-placement module checks that the VNFs that are removed with empty configuration are indeed not necessary also from a network point of view. A study was conduct to explore the real applicability of these functionalities. Unfortunately, the optimization nature of the computation led to some unsatisfactory results since solving a NP-complete problem caused the execution time to grow exponentially with the increasing complexity.

Besides the extensions applied to the tool itself, two interesting interface modules towards external tools have been developed. The integration with SONATA highlighted the feasibility to include Verifoo in an NFV framework as a verification and deployment service that can directly instruct a VIM. A future development could be to explore the possibility to integrate Verifoo with the Resource Orchestrator of the OSM project to have a completely automated interaction.

An initial integration of Neo4j has also been explored. Currently, it serves only as a debugging tool but helpful indications were given on how to expand further its utilization in the design chapter.

The performance tests executed on the final solution showed that the extended network model for service graphs and the new constraints add a reasonable amount of computational time. However, as mentioned, for the auto-configuration module the experimental results express a clear impracticality of the proposed methodology for the solution space exploration.

In conclusion, this work tried to extend the capabilities of Verifoo preserving a high-level of modularity in order to delineate the guidelines needed to keep the tool evolving. Particular attention was paid to the key features of the NFV and SDN paradigms that are seen as the founding stones of this project. However, Verifoo is to be considered still in development and is not free of limitations. Future works can further improve the software by polishing some of the outlined issues and improving its interoperability with the other NFV products present on the market.

Bibliography

[1] NFV ETSI, “GS NFV 002 V1.1.1 Network Functions Virtualisation (NFV); Architectural Framework,” 2013. https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf.

[2] “OpenMANO Github Page.” https://github.com/nfvlabs/openmano. [Online; accessed 19-04-2018].

[3] “Floodlight Website.” http://www.projectfloodlight.org/floodlight/. [Online; accessed 26-08-2018].

[4] “ISG NFV community publications.” https://www.etsi.org/standards-search#page=1&search=&title=1&etsiNumber=1&content=0&version=1&onApproval=1&published=1&historical=0&startDate=1988-01-15&endDate=2018-08-26&harmonized=0&keyword=&TB=789,,832,,831,,795,,796,,800,,798,,799,,797,,828&stdType=&frequency=&mandate=&collection=&sort=3:.

[5] “Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV,” 2014. ETSI GS NFV 003 V1.2.1.

[6] “Z3 Github Page.” https://github.com/Z3Prover/z3. [Online; accessed 20-04-2018].

[7] F. L. Bjørner Nikolaj, Dung Phan Anh, “vZ - An Optimizing SMT Solver,” Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015, pp. 194–199, 2015.

[8] “Verigraph Github Page.” https://github.com/netgroup-polito/verigraph. [Online; accessed 20-04-2018].

[9] G. Marchetto, R. Sisto, J. Yusupov, A. Ksentiniy, “Virtual network embedding with formal reachability assurance,” p. 5, 2017.

[10] H. K. M. Peuster and S. v. Rossem, “MeDICINE: Rapid Prototyping of Production-Ready Network Services in Multi-PoP Environments,” IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), (Palo Alto, CA, USA), pp. 148–153, 2016.

[11] “Containernet Github Page.” https://github.com/containernet/containernet. [Online; accessed 28-05-2018].

[12] “Neo4j Website.” https://neo4j.com/. [Online; accessed 21-04-2018].

[13] “Neo4j Graph Algorithms.” https://neo4j.com/developer/graph-algorithms/. [Online; accessed 28-08-2018].

[14] “Neo4j OGM Introduction.” https://neo4j.com/docs/ogm-manual/current/introduction/. [Online; accessed 28-08-2018].

[15] “Bolt Protocol.” https://en.wikipedia.org/wiki/Bolt_(network_protocol). [Online; accessed 26-04-2018].

[16] “Barracuda Firewall.” https://campus.barracuda.com/product/nextgenfirewallf/doc/70586675/how-to-create-wildcard-network-objects/. [Online; accessed 12-04-2018].

Appendices

Appendix A

Developer's guide to future extensions

The Verifoo project includes a detailed javadoc which describes all the classes and methods. In this chapter an overview of the most important packages will be presented, with a particular focus on those classes that might have a central role in future extensions.

A.1 it.polito.verifoo.random

This package contains all the classes that are assigned to the generation of a random input for Verifoo. It contains four classes, each deals with a different part of the input:

- **RandomGraph**: it generates a random service graph with random requirements and set of policies. This is a slightly modified version of the graph generator available for Verigraph
- **RandomTopology**: it generates a random physical topology ("Hosts" element and "Connections" element)
- **RandomConstraints**: it generates a random set of requirements for a service graph
- **RandomInputGenerator**: it wraps the creation of a completely random input file

To create a new input, it is enough to create a new instance of the RandomInputGenerator and then call the *getRandomInput()* method on it that return the NFV object. The class constructor accepts various parameters that defines the dimensions of the network service graph and the topology. In the debugging phase, the object randomly created can be serialized to have the actual XML file.

Listing A.1. Random Input Generation Example

```java
RandomInputGenerator r = new RandomInputGenerator(maxClients, maxServers,
    ↪ maxInternalNodes, maxProperties, maxHosts);
NFV root = r.getRandomInput();
JAXBContext jc= JAXBContext.newInstance( "it.polito.verifoo.rest.jaxb" );
Marshaller m = jc.createMarshaller();
OutputStream out = new FileOutputStream("./random.xml");
m.marshal( root, out );
```

This package could be used to help during a debugging phase of the project to check if there are some corner case that are not addressed.

A.2 it.polito.verifoo.rest.common

This package includes the classes that deal with the translation from the XML input to the z3 formulas and vice versa.

Among all the classes, those that are noteworthy are:

- **PhyResourceModel**: this is the interface that describes the actions for an interaction with a VIM. For now only a SONATA module has been developed that implements this interface. Other modules that set up the same objective should also implement this.
- **VerifooProxy**: this is the main class that coordinates the XML deserialization in z3 formulas. A useful method is *setConditions()* which gathers all the formulas that express the resource allocation constraints (e.g. disk storage, CPU, memory, etc.). If new variables will be considered, it would be appropriate to add there the new constraints.
- **Translator**: this class extracts all the information from the z3 model after the computation. If new data needs to be retrieved, new methods should be implemented here.
- **z3Translator**: this class provides all the patterns that match the strings in the z3 model to retrieve the important information. It also declares an enumeration class that gathers all the complex data types used in the model, which should be extended with all the future ones. Moreover, if the version of the z3 libraries used by Verifoo will be updated, this class should be replaced with one that provides the new patterns.

A.3 it.polito.verifoo.rest.medicine

This package contains the classes that set up the SONATA simulation environment. Modifications in newer versions of the tool should be propagated also in these classes. The package is composed by the following Java classes:

- **MedicineSimulator**: this class runs the SONATA simulation for a physical topology. In particular in the *deployVNF()* method, SONATA CLI commands are used. Future modification in the libraries should be reflected also here.
- **PhysicalTopology**: this class creates the file that represents the topology that will be simulated
- **ServiceDescriptor**: this class create the service descriptor for a specific graph. The descriptor version is the 1.0. Future modifications to the ETSI standard should be applied also to this class.
- **VNFDeployment**: this class creates the custom placement file that SONATA will read to know how to deploy the service.
- **VNFDescriptor**: this class creates a VNF descriptor file following the ETSI specifications.
- **TopologyDB**: this class implements the singleton pattern to store the information about the running SONATA simulation. The singleton is used to have only one simulation at a time, but it can be generalized to more than one. The simple method to do that would be to differentiate the ports for the SONATA service REST API among different simulations.

A.4 it.polito.verigraph.mcnet.components

This package include all the classes that model the formulas which simulate the network behaviour. It derives from Verigraph but there are some unique feature implemented for Verifoo. The most useful classes in this package are the following:

- **NetContext**: this class represents the foundation of the z3 model since all the basic constraints are declared here. For instance, it declares formulas that avoid receiving unwanted behaviour from z3 (e.g. the range of the IP addresses that needs to be from 0 to 255, how the packet exchange should happen without modifying the destination field, etc.). Other future fundamental constraints should be added in the *baseCondition()* function. In addition to that, in this class are also initialized all the complex data type that z3 needs to know. This is done in the *mkTypes(...)* function. New types should be added preferably here. All the constraints declared in this class are added to the z3 context in the *addConstraints()* function.

- **Network**: this is the class that models the network behaviour. All the conditions are built in the *routingOptimizationSGOptional(...)* that includes the support for complex service graph with optional nodes (which the auto-placement feature tries to minimize). Other changes on how the network behaves should be introduced in this function. This constraints are added to the z3 context in the *addConstraints()* function.
- **Checker**: it is the class that formalizes the policies constraints and triggers the z3 resolution process. The general pattern followed in Verifoo, as well as in Verigraph, is to maintain a list of constraints in each class and fill it as the deserialization of the input goes on. For instance, there is one list of constraints for each of the VNFs that are instantiated and it is filled when the model of that VNF is initialized, after reading its configuration. Only at the end, all the constraints are added to the *solver*, which is an instance of the Optimize class that is an interface towards the z3 libraries. The Checker class has a knowledge on what are the objects that have a constraints list and before triggering the z3 resolution, it ensures that all of them are included in the problem. This is achieved in the *addConstraints()* method which calls in cascade the *addConstraints()* function of the other classes. This adds all the constraints to the same solver, thus creating a single context for the z3 execution. An extract of the code can be found below.

Listing A.2. Constraints addition pattern

```
public void addConstraints() {
    // add the constraints of the different network object (i.e. VNFs)
    for(NetworkObject el:network.elements) el.addConstraints(solver);
    // add the constraints about the network routing
    network.addConstraints(solver);
    // add the constraints about the service deployment
    netContext.addConstraints(solver);
}
```

The only thing that needs to be paid attention to, is the fact that in z3 the order of the constraints addition is meaningful for the soft clauses (it is not for the hard clauses). In particular, the classes of soft constraints that are added before, are the first that are optimized.

Listing A.3. Constraints ordering example

```
// for example, adding the constraints in this order
    network.addConstraints(solver);
    netContext.addConstraints(solver);
// or in this order
    netContext.addConstraints(solver);
    network.addConstraints(solver);
// gives totally different results
```

When introducing a new policy, the point of contact with the XML deserialization is designed to be the *propertyAdd(...)* method which then calls the specific function that models each policy as highlighted below.

Listing A.4. Property addition pattern

```
switch (property) {
    case ISOLATION:
        addIsolationProperty(...);
        break;
    case REACHABILITY:
        addReachabilityProperty(...);
    break;
}
```

The same pattern should be maintained also for future policies.

A.5 Other packages

In this section some other interesting minor packages are described, emphasizing what is their role in the project.

it.polito.verifoo.rest.neo4j

This package should contain all the classes that interact with Neo4j. At the moment, only the **Neo4jClient** class is present and contains a simple proof of concept on how to use the Cypher Query Language to store graph information in the database.

it.polito.verifoo.rest.test

This package contains all the classes that exploit the jUnit framework to automate the tests. Each class tests a different feature of Verifoo to have a more precise indication on what is working and what not.

it.polito.verifoo.rest.webservice

This package contains all the classes that build the web service of Verifoo. The REST API of the deployment service are collected in the **RestFoo** class, while the APIs that allow to interact with the SONATA simulation are in the **RestMeD** class.