ALMA MATER STUDIORUM — UNIVERSITÁ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TITOLO DELL'ELABORATO

Model Checking of Software Defined Networks using Header Space Analysis

Elaborato in

Network Security

Relatore

Franco Callegati
Walter Cerroni

Presentata da

Alessandro Molari

Sessione **II**

Anno Accademico **2016/2017**

This work is for my parents **Claudio** and **Oriena**, who made my studies possible,
my wise brother **Luca**, my best friend, super-hero and source of inspiration,
my dear sister **Denise**, who supported me in the darkest moments,
my lovely cat **Paolo**, who kept me warm, with whom I shared many unique moments,
the Cyber Security and Hacking group **CeSeNA**,
and for all those who believed in me.

Contents

Abstract

This thesis investigates the topic of verifying network status validity with a Cyber Security perspective.

The fields of interest are dynamic networks like OpenFlow and Software Defined Networks, where these problems may have larger attack surface and greater impact.

The framework under study is called Header Space Analysis, a **formal model** and **protocol-agnostic framework** that allows to perform **static policy checking** both in classical `TCP/IP` networks and modern dynamic `SDN`.

The goal is to analyse some classes of network failure, declaring valid network states and recognizing invalid ones.

`HSA` has evolved in NetPlumber, to face problems caused by high dynamics of `SDN` networks. The main difference between `HSA` and NetPlumber is the incremental way that the latter performs checks and keeps state updated, verifying the actual state compliance with the expected state defined in its model, but the concept is the same: declare what's allowed and recognize states violating that model.

The second and main contribute of this thesis is to expand existing vision with the purpose of increasing the network security degree, introducing model-checking-based networks through the **definition of an abstraction layer** that provides a **security-focused model-checking service** to `SDN`.

The developed system is called `MCS` (Model Checking Service) and is implemented for an existing `SDN` solution called `ONOS`, using NetPlumber as underlying model-checking technology, but it's validity is general, uncoupled with any kind of `SDN` implementation.

Finally, the demo shows how some cases of well-known security attacks in modern networks can be prevented or mitigated using the reactive behavior of `MCS`.

CHAPTER 1

Software Defined Networks

1. Introduction

In the last years digitisation and interconnection of society brought a greater usage of traditional `TCP/IP` networks.

However the Internet protocol stack was designed looking society of the late '70s, not today. Use-cases changed and today's demand isn't the same as it was before.

Now, traditional IP networks are complex and very hard to manage. It is both difficult to configure the network according to predefined policies and to reconfigure it to respond to faults, load and changes.

The main criticism is *current networks are vertically integrated*: decisions about how traffic should be handled are coupled with its forwarding.

In this scenario, modern standards have been proposed, such as OpenFlow from Stanford University. OpenFlow is then evolved in a dynamic networking structure called Software Defined Network.

2. Planes

`SDN`[5] breaks vertical integration into two planes. Every plane has a well-defined set of responsibilities:

- **Control Plane** (`CP`):
  Decides how to handle network traffic.
- **Data Plane** (`DP`):
  Forwards traffic according to the decisions made by the Control Plane, without making any decision.
  In this scenario, OpenFlow is just a possible implementation of the Data Plane.

This approach allows to centralize decisions in a **logically centralized controller**.

Finally, **Management Plane** (`MP`) is defined as the set of applications that leverage functions offered by the `NOS` (in particular the Northbound Interface) to *define policies*, which are translated to southbound-specific instructions that *program the behaviour* of Forwarding Devices.

3. Principles

`SDN` architecture has four pillars:

(1) **Control Plane and Data Plane are decoupled**: network devices will become simple forwarding elements.
(2) **Forwarding decisions are flow-based**, instead of destination-based. A flow is defined by a set of packet field values acting as *match criterion* and a set of *actions* that define what to do. Flow abstraction allows unifying the behavior of different types of network devices, including switches, firewalls, routers, ...

(3) **Control logic is moved to an external entity** called SDN CONTROLLER (or NETWORK OPERATING SYSTEM), that provides resources and abstractions to facilitate the programming of forwarding devices based on a logically centralized *abstract network view*.

Logical centralization of control logic makes simpler and less error-prone to modify network policies through high-level languages and software components.

Also, centralization of control logic in a controller with global knowledge of the network state simplifies the development of more sophisticated VIRTUAL NETWORK FUNCTION.

(4) **Network is programmable** through software applications running on top of `NOS`, that interacts with the underlying DATA PLANE devices.

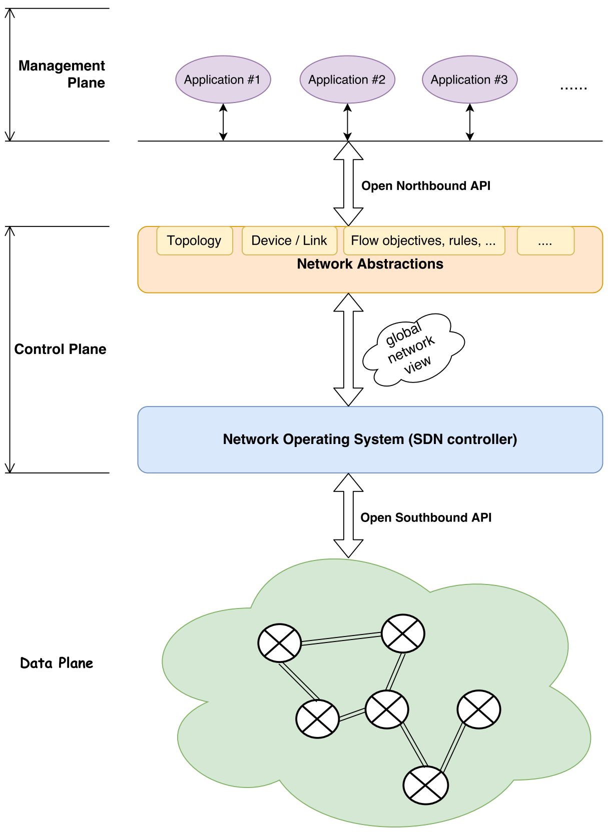


FIGURE 1. SDN architecture

4. Abstractions

From a *application point-of-view*, `SDN` can be defined by three abstractions:

(1) **Forwarding abstraction**: Should allow any forwarding behavior desired by network applications while *hiding details of the underlying hardware*. OPENFLOW is one example of such abstraction. It allows to reason about **FORWARDING DEVICES** (`FDs`) instead of physical devices: hardware or software-based DATA PLANE device that perform a set

of elementary operations through a well-defined *instruction set* (for example flow rules) used to *take actions* on the incoming packets.

(2) **Distribution abstraction**: Network applications should be shielded from the vagaries of distributed state, making the distributed control problem a *logically centralized one*. Its realization requires a *common distribution layer*, called `SDN` controller (or Network Operating System).

(3) **Specification abstraction**: Network applications should be able to express the desired network behavior *without being responsible for implementing that behavior itself*. This can be achieved because specification layer maps the global view of the physical network (provided by the `SDN` controller) into simplified abstract models that can be used by network applications.

5. Interfaces

Interaction between `SDN` layers is regulated by the following interfaces:

- **Southbound Interface** (`SBI`): The *instruction set* of the Forwarding Devices is defined by the southbound API. `SBI` can be seen as the connecting bridge between `NOS` and `FD`, the main instrument used to clearly separate the Control Plane with the Data Plane, thus these APIs are still tightly tied to the Forwarding Devices of the underlying physical (or virtual) infrastructure.
- **Northbound Interface** (`NBI`): The Network Operating System can offer an API to application developers. This API represents the Northbound Interface.

CHAPTER 2

SDN Control Plane implementation

1. Southbound Interface: OpenFlow

1.1. Introduction. OpenFlow[2] is a possible (and commonly used) Southbound Interface. It allows interoperability and the deployment of vendor-agnostic network devices.

1.2. OpenFlow switch. An OpenFlow switch consists of:

- One or more Flow Tables
- A Group Table
- A OpenFlow channel

The Flow Tables and Group Table are used to perform packet lookups and forwarding.
The OpenFlow channel is used to communicate with the SDN Controller, both to *inform about new events* and to let SDN Controller *manage* the OpenFlow switch via the OpenFlow protocol.

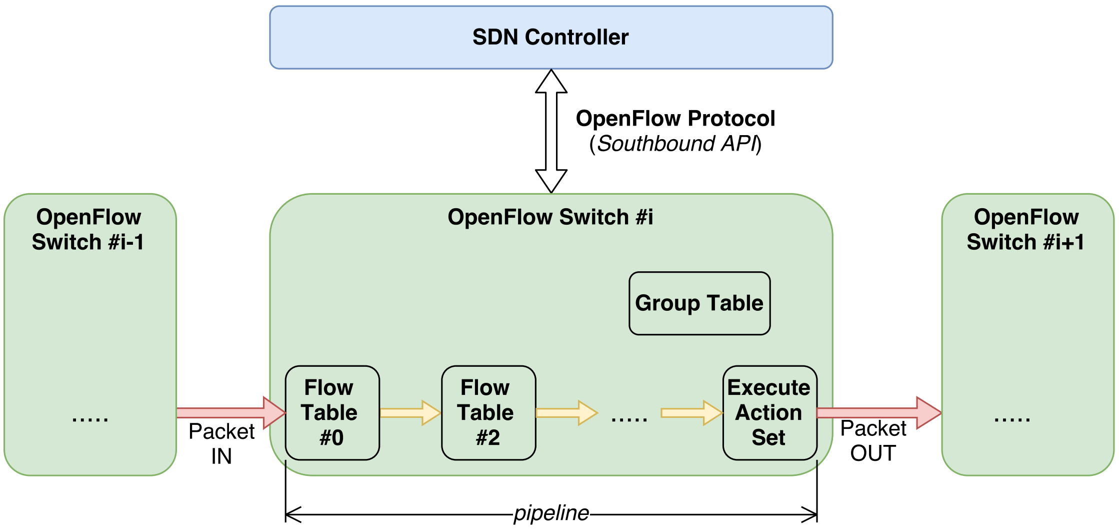


Figure 1. OpenFlow switch

The OpenFlow pipeline contains multiple Flow Tables. Flow Tables are *sequentially numbered*, starting from 0. Each Flow Table contains multiple Flow Entries.
Pipeline processing defines how incoming packets are consumed by Flow Tables and what result is produced.
When there is an incoming packet in a OpenFlow switch, it is matched against the Flow Entries of the first Flow Table to select a Flow Entry.

The following figure shows how a FLOW ENTRY is structured:

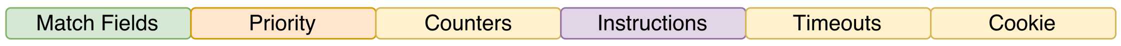


FIGURE 2. FLOW ENTRY

Where:

- `Match Fields`: Used to match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.
- `Priority`: Matching precedence of the FLOW ENTRY.
- `Counters`: Updates for matching packets.
- `Instructions`: instructions that should be added to the ACTION SET.
- `Timeouts`: maximum amount of time before flow is expired
- `Cookie`: opaque data value chosen by the controller. It may be used to filter flow statistics, modification, deletion, ... It is not used when processing packets.

Pipeline processing typically go through next tables until the last table is reached and the instructions available in the ACTION SET will be executed.

If a matching FLOW ENTRY does not direct packets to another FLOW TABLE, pipeline processing stops at this table. When a pipeline processing stops, the packet is processed with its ACTION SET and usually forwarded.

If a packet does not match a FLOW ENTRY in a FLOW TABLE, there is a **TABLE MISS**.

The behavior of a TABLE MISS depends on the table configuration, for example: dropping packets, passing them to another table, sending them to SDN CONTROLLER via messages.

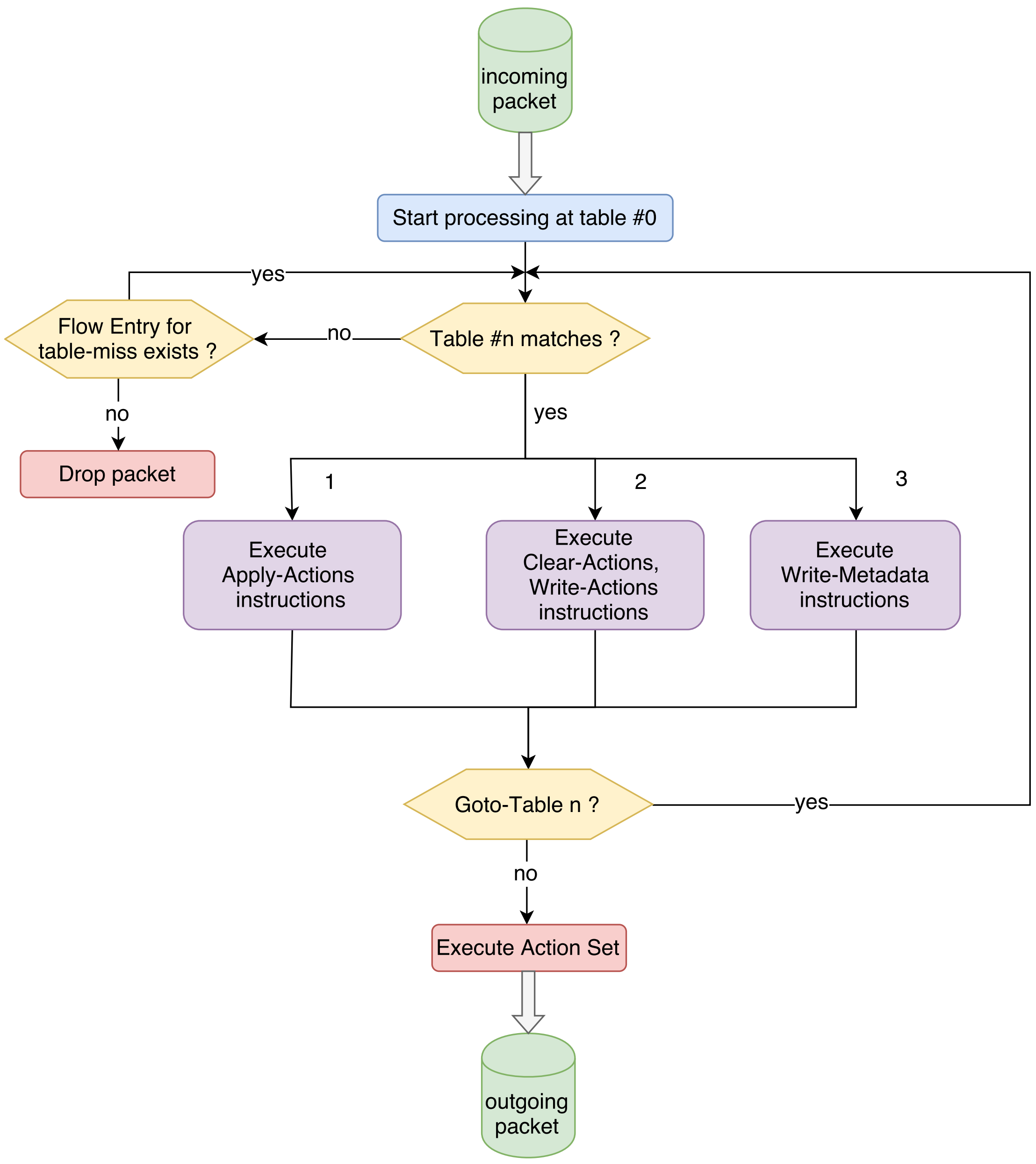

incoming packet
Start processing at table #0
yes
Flow Entry for table-miss exists ?
no
Table #n matches ?
no
Drop packet
yes
1
2
3
Execute Apply-Actions instructions
Execute Clear-Actions, Write-Actions instructions
Execute Write-Metadata instructions
Goto-Table n ?
yes
no
Execute Action Set
outgoing packet

Figure 3. Packet processing

Possible instructions include:

- `Meter` *`meter-id`* (*optional*): Direct packet to the specified meter.
- `Apply-Actions` *`actions`* (*optional*): Applies the specific actions *immediately*, without changing the Action Set.
- `Write-Actions` *`actions`* (*required*): Merges the specified actions into the current Action Set.

- `Clear-Actions` (*optional*): Clears all the actions in the Action Set.
- `Write-Metadata` ***metadata*** (*optional*): Writes the masked metadata value into the metadata field.
- `Goto-Table` ***next-table-id*** (*required*): Indicates the next table in the processing pipeline (must be greater than the current table). Flow Entries of the last Flow Table cannot include this instruction.

1.3. OpenFlow protocol. The provides three information sources for Network Operating System:

- **Controller-to-Switch Messages**: Messages *initiated by the controller* and may or may not require a response from the switch.
  Examples include:
  - `Features`: Message to request switch capabilities.
  - `Modify-State`, `Read-State`: Messages to read and modify switch state.
  - `Configuration`: Message to change switch configuration.
  - `Packet OUT`: Message to send packets in the specified port on the switch and, thus, to forward packets received via a `Packet IN` message. Message must also contain a list of actions that the switch should apply for that packet.
  - `Barrier`: Barriers are request/reply messages used to *ensure message dependencies have been met* or to receive notifications for completed operations.
- **Asynchronous Messages**: Messages sent without a controller soliciting them from a switch. Switches send asynchronous to controllers to denote an event, such as packet arrival, switch state change, error, ...
  Examples include:
  - `Packet IN`: Transfer control of a packet to the controller. This kind of event is always sent for (1) packets forwarded to port `CONTROLLER` (2) table-miss Flow Entry is hit.
  - `Flow-Removed`: Inform the controller about the removal of a Flow Entry from a Flow Table.
  - `Port-status`: Inform the controller of a change on a port.
- **Symmetric Messages**: Messages sent *without solicitation, in either direction.*

A typical `SDN` controller manages multiple OpenFlow channels, each one to a different OpenFlow switch.

An OpenFlow switch typically have one OpenFlow channel to a single SDN Controller.

Once connection between a switch and the SDN Controller has been established, they negotiate the protocol version to be used. Then, other messages following the negotiated protocol version can be exchanged through the channel.

2. SDN Controller: ONOS

2.1. Architecture. ONOS[6] is a SDN controller solution organized as a multi-module OSGi project, where modules are managed as OSGi bundles.
OSGi defines an architecture for developing and deploying modular systems.
Its architecture is *layered*:

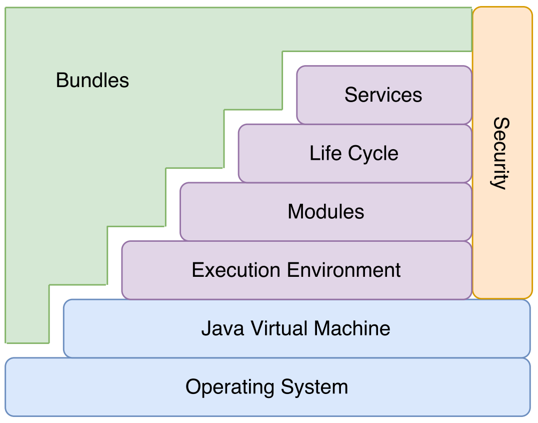


FIGURE 4. architecture

Where:

- **Bundles**: Bundles are the OSGi components made by the developers.
- **Services**: The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
- **Life-Cycle**: The API to install, start, stop, update, and uninstall bundles.
- **Modules**: The layer that defines how a bundle can import and export code.
- **Security**: The layer that handles the security aspects.
- **Execution Environment**: Defines what methods and classes are available in a specific platform.

Every bundle has its own lifecycle:

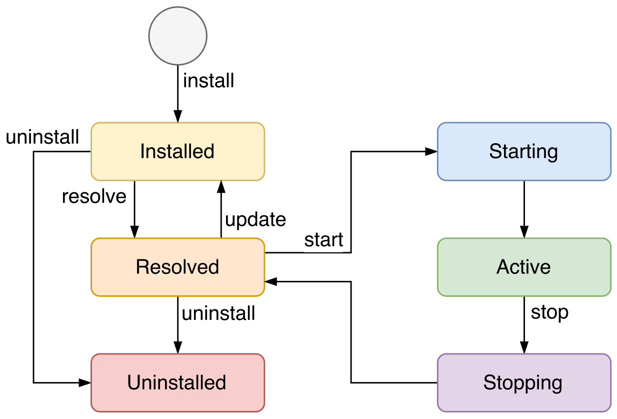


FIGURE 5. Bundle lifecycle

In ONOS this allows to build an open system where modules can be added on-demand and are decoupled from the core that handles the main NOS functionality and provides main network abstractions.

Inside a bundle, multiple OSGi components and services can be defined.

This perfectly maps to the concept of ONOS service. In fact, both have dependencies and a lifecycle. For this reason, most of ONOS services are implemented as OSGi services.

2.2. Tiers, Sub-systems, Components. ONOS system is organized in **tiers**. Each tier hosts some sub-systems and a well-defined interface. Tier interfaces are used to perform inter-tier interaction. Each sub-system is a collection of components making up a service that provides the desired functionality.

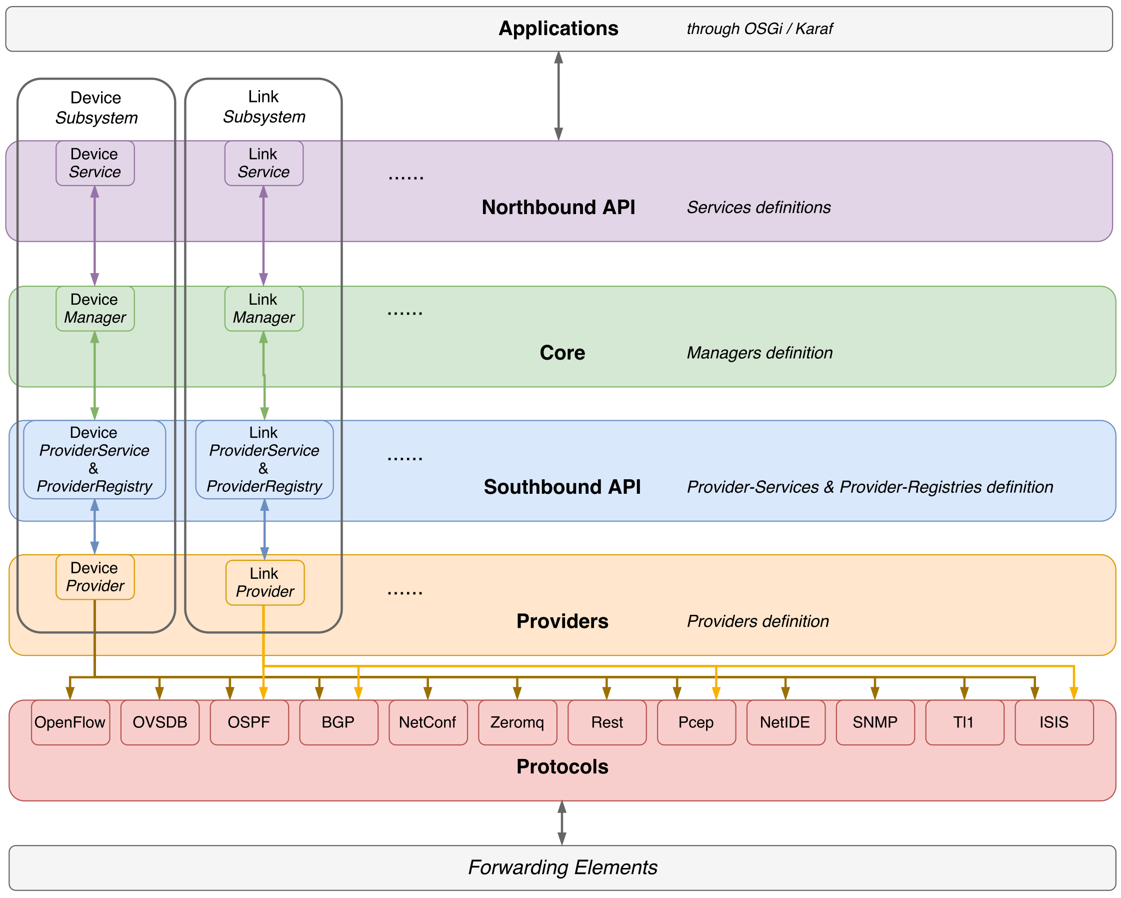


Figure 6. ONOS Tiers

As shown above, a sub-system can span over multiple layers.

In particular:

- **Northbound API Tier** allows to the applications to use defined services.
- **Core Tier** defines Manager abstraction. It allows to translate low-level services defined in Southbound Interface to high-level services defined in Northbound Interface and to receive information from applications.

- **Southbound API Tier** allows to the Core Tier to interact with Providers Tier via two main abstractions: `ProviderService` and `ProviderRegistry`.
- **Providers Tier** defines protocol-specific implementation of the desired functionalities. For example for devices sub-system there are: `OpenFlowDeviceProvider`, `OVSDBDeviceProvider`, ...
- **Providers Tier** defines protocol-specific implementation of the desired functionalities.
- **Protocols Tier** allows to interact with forwarding elements, by implementing needed communication protocols.

CHAPTER 3

Security in SDN

1. Introduction

In early days of SDN, security wasn't considered as an important factor, thus SDN resulted to be very insecure by design.

The architecture of SDN introduces radical changes in the vertical network integration model by decoupling Data Plane from the Control Plane. This poses new threats both from internal and external actors and a much more broader attack surface than in static networks.

In particular, the logical centralization of the network intelligence put the entire network at risk if the SDN Controller is compromised.

Moreover, even if SDN Controllers aren't compromised, but just a single network box's security fails, it can compromise the SDN Controller state and thus the entire network.

This is possible because pure SDN solutions available today don't have the concept of security isolation. On the other hand, each layer of SDN has its own security requirements and implications.

Since a single point of control can gain entire network control, a single point of failure can possibly exploit the entire network.

Imagine if you request a bank transaction and your sensitive information is stolen just because somewhere, not even between you and the bank, a network box is compromised, injecting malicious flow-rules to remove you from the network.

Imagine if your pacemaker monitoring solution stops working, and cannot communicate with your medical institute about health problems, just because networking failed due to a compromised SDN Controller that has infected the remaining parts of the network system.

These are just simple Denial-of-Service (DoS) attacks, but also more complex attacks are feasible.

The centralized knowledge base can be used for further attacks, like *reprogramming the entire network* to modify the network flow or *to steal sensitive information.*

Here, the principle "Security Is Everyone's Responsibility" can't be applied: a compromised network box can't compromise the entire network. Compromised sources need to be recognized and they shouldn't interfere with the rest of the network.

Security must be built as part of SDN architecture to ensure the integrity of the network system.

As always, *we need to face a technology that isn't secure by design.* SDN research moved from analysing network problems to security problems.

Many ideas and solutions have been proposed but none of them became the de-facto standard.

In the rest of this thesis we will try to use a formal model (called Header Space Analysis) and its evolution (called NetPlumber) to address network security issues in Software Defined Network.

2. Security vulnerabilities, attacks and challenges

SDN networks provide advanced and complex solutions that involve many actors to interact in a way that is programmable.

This amplifies the attack surface of existing threats and leads to new vulnerabilities in various SDN layers[1]:

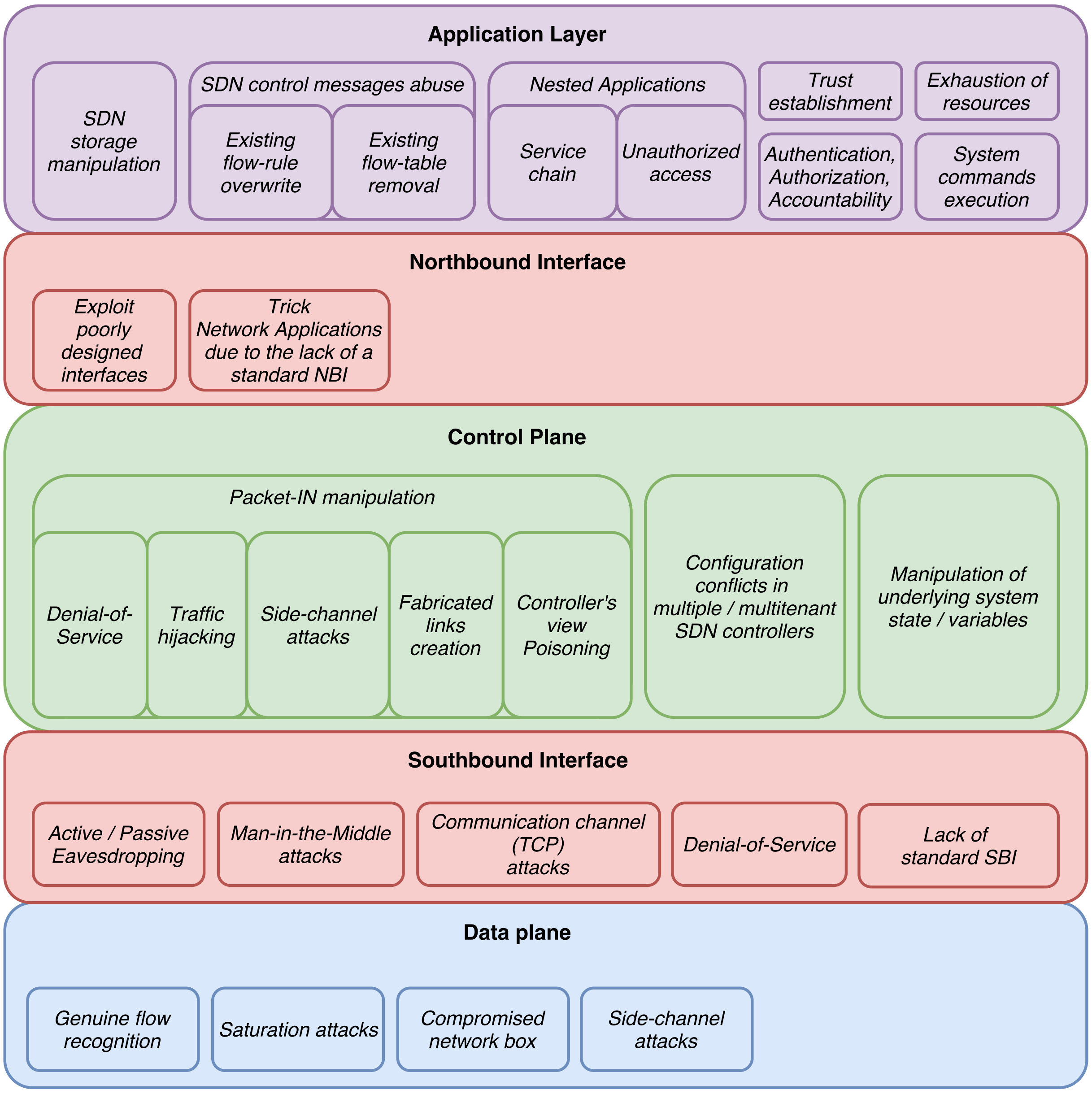

Application Layer
SDN storage manipulation
SDN control messages abuse
Existing flow-rule overwrite
Existing flow-table removal
Nested Applications
Service chain
Unauthorized access
Trust establishment
Authentication, Authorization, Accountability
Exhaustion of resources
System commands execution
Northbound Interface
Exploit poorly designed interfaces
Trick Network Applications due to the lack of a standard NBI
Control Plane
Packet-IN manipulation
Denial-of-Service
Traffic hijacking
Side-channel attacks
Fabricated links creation
Controller's view Poisoning
Configuration conflicts in multiple / multitenant SDN controllers
Manipulation of underlying system state / variables
Southbound Interface
Active / Passive Eavesdropping
Man-in-the-Middle attacks
Communication channel (TCP) attacks
Denial-of-Service
Lack of standard SBI
Data plane
Genuine flow recognition
Saturation attacks
Compromised network box
Side-channel attacks

FIGURE 1. SDN threats

3. Some practical cases / examples

- Malicious or compromised Network Application:
  - DoS chained applications
  - sidesteps to use chained applications authorizations
  - modifies SDN Controller database to change its perception of the underlying network
  - retrieves SDN Controller information as part of the reconnaissance stage of infection
  - overwrites flow rule or clear a flow table to cause unexpected network behaviour and trigger attacks at lower levels, like network boxes DoS, Man-in-the-Middle attacks, ...
  - exhausts all the available system resources and affect performance or DoS other Network Applications
  - exploits SDN Controller and thus can execute system commands, like `exit` to terminate the controller
- If a Network Application is compromised and its information is used somewhere else, for example a DBMS containing user sensitive data, other applications may use those information without knowing that the source shouldn't be trusted because it was compromised. That is due to compelling mechanisms to establish trust to certify SDN Controller and Network Applications don't exist yet
- Malicious or compromised network box:
  - floods a target SDN Controller with Packet IN, in order to compromise it
  - forges and sends a ARP packet relayed as Packet IN, fooling the SDN Controller into installing malicious flow rules to divert traffic flows, possibly for eavesdropping, thereby allowing a malicious host to intercept traffic intended for another host
  - along with another accomplice, initiate arbitrary flows to fool the OpenFlow switch and the SDN Controller into installing flow rules that create loops or black-holes in the network
  - uses the Southbound Interface to modify Control Messages exchanged between the Control Plane and the Data Plane, such as flow-rule messages to corrupt network behaviour
  - steals sensitive information by sniffing the Control Channel
  - floods the SBI's communication interface to DoS other network boxes (thus the Data Plane) and the SDN Controller (thus the Control Plane)

4. Summing up

We've just named few examples, but we clearly can see there is an entire world of possible attacks, which one has many ways to be triggered and executed.

In this scenario, it's not feasible to analyse and resolve all possible single vulnerabilities and threats, but we need a generic and pervasive long-range approach to solve classes of threats.

We will focus later on a broad-based approach to verify network state consistency that will provide a way to recognise some classes of threats and possibly prevent them, by eliminating root causes of related problems.

CHAPTER 4

Header Space Analysis

1. Introduction

Header Space Analysis[3] is a generic framework which provides a set of tools to model and check networks for classes of failures in a protocol-independent way.

`HSA` relies on a formal model, built as a *geometric model*, where:

- packets are points in a geometric space
- network boxes are transfer functions on the *same* geometric space

2. Taxonomy

The following taxonomy defines concepts used later:

2.1. header (h). A *flat sequence of ones and zeros.* It doesn't necessarily correspond to the packet header, but it could:

- include *just a part* of the header
- include *all* of the header
- include the header and part of the packet payload: in this case *the payload affects* packet *processing*, that might be useful when modelling a Intrusion Detection System

2.2. header space (H). A **geometric space**, where:

- *Every* point *is a* header
- *A* flow *is a region* in the $\{0,1\}^L$ space, where L is an *upper-bound* on the header length

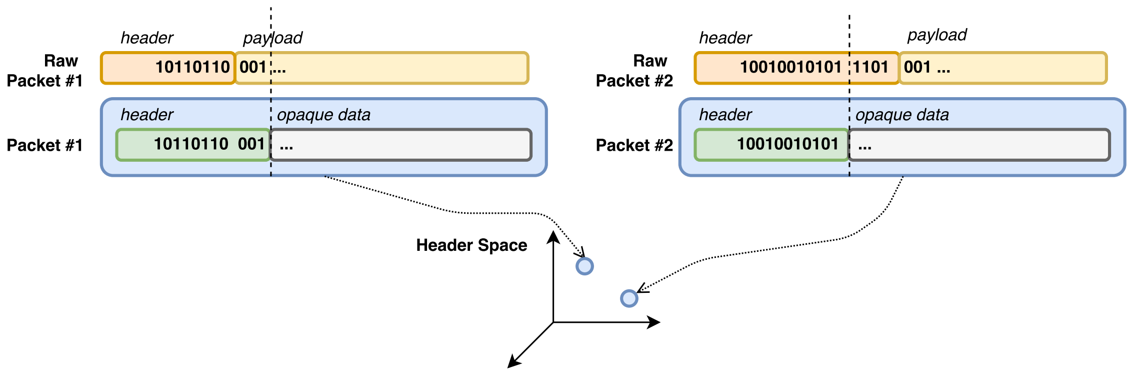


Figure 1. Building a header space

2.3. wildcard expression (w). A *sequence of L bits*, where each bit can assume values in $\{0, 1, x\}$.

Each wildcard expression corresponds to a hypercube in H.

2.4. flow (f). A *union* of WILDCARD EXPRESSION

2.5. network box (n). Any possible middlebox inside the network (also called SWITCH). It has external interfaces called PORTS.

Every PORT *has a unique identifier*. A NETWORK BOX is a node that can be modelled using its TRANSFER FUNCTION, called SWITCH TRANSFER FUNCTION (T), that maps HEADER h arriving on PORT p:

$$T(h,p) = (h,p) \rightarrow \{(h1,p1),(h2,p2),\ldots\} \tag{1}$$

Notice that T depends on input/output PORT to model PORT-specific behavior, for example Load Balancing.

2.6. network space (N). The space of all possible PACKET HEADERS, localized at all possible input PORTS in the network, defined as the cartesian product:

$$\{0,1\}^L x\{1,\ldots,P\} \tag{2}$$

Where $\{1,\ldots,P\}$ is the list of all ports in the network.

Every point in this space represents a packet traversing on a link.

2.7. network transfer function (Ψ). When a PACKET traverses the network, *it is transformed from one* POINT *in the* NETWORK SPACE *to another* POINT. The function that describes the operation is *protocol-independent* and it can be obtained combining all NETWORK BOXES behavior together we create a composite TRANSFER FUNCTION describing the overall behavior of the network, called NETWORK TRANSFER FUNCTION:

$$\Psi(h,p) = \begin{cases} T_1(h,p) & \text{if } p \in \text{switch}_1 \\ T_{\ldots}(h,p) & \text{if } p \in \text{switch}_{\ldots} \\ T_n(h,p) & \text{if } p \in \text{switch}_n \end{cases} \tag{3}$$

2.8. topology transfer function (Γ). Models *behavior of links* in the network:

$$\Gamma(h,p) = \begin{cases} \{(h,p*)\} & \text{if } p \text{ is connected to } p* \\ \{\} & \text{if } p \text{ is not connected} \end{cases} \tag{4}$$

We can apply the following function at each hop, to model a packet that traverses the network:

$$\Phi(.) = \Psi(\Gamma(.)) \tag{5}$$

2.9. slice transfer function (Ψ_s). Captures behavior of all rules installed by the Control Plane of SLICE S

2.10. permission (P). A subset of $\{\text{read}(r), \text{write}(w)\}$.

2.11. slice network space (Υ). A subset of the NETWORK SPACE N controlled by the SLICE S.

2.12. slice (S). The tuple (Ψ_s, P, Υ).

3. Modelling NETWORK BOXES with HSA

In this section we'll use the taxonomy defined above to model some kind of networking devices with HSA, transforming all of them in a single entity called NETWORK BOX, where the difference between them is encapsulated into the TRANSFER FUNCTION.

3.1. IPv4 Router.

A IPv4 ROUTER can be seen as a NETWORK BOX that implements the following main high-level functions:

- Rewrite source and destination MAC address
- Decrement TTL
- Update checksum
- Forward to outgoing port

Thus the TRANSFER FUNCTION is:

$$T_{\text{IPv4}}(.) = T_{\text{fwd}}(T_{\text{chksum}}(T_{\text{ttl}}(T_{mac}(.)))) \tag{6}$$

Where:

- $T_{\text{fwd}} = \{(h, \text{ip\_lookup(ip\_dst(h))})\}$, where: $\text{ip\_lookup}(.) : \text{ip\_dst} \rightarrow \text{port}$
- $T_{\text{mac}}(.)$ looks up the next hop MAC address and updates source and destination MAC addresses
- $$T_{\text{ttl}}(.) = \begin{cases} \text{drop\_packet} & \text{if } \text{ip\_ttl}(h) = 0 \\ \text{rewrite}(h, \text{ip\_ttl}(), \text{ip\_ttl}(h) - 1) & \text{if } \text{ip\_ttl}(h) \neq 0 \end{cases}$$
- $T_{\text{chksum}}(.)$ updates the IP checksum

If we focus just on IP routing, we simplify the model in:

$$T_{\text{IPv4}}(.) = T_{\text{fwd}}(.) = \begin{cases} \{(h, p_1)\} & \text{if } \text{ip\_dst}(h) \in S_1 \\ \{(h, p_2)\} & \text{if } \text{ip\_dst}(h) \in S_2 \\ \dots & \dots \\ \{(h, p_n)\} & \text{if } \text{ip\_dst}(h) \in S_n \\ \{\} & \text{otherwise} \end{cases} \tag{7}$$

Where S_i is the subnet where the packet should be forwarded.

3.2. Firewall.

A firewall can be modelled as following:

- Extract IP and TCP headers
- Match extracted headers against the WILDCARD EXPRESSIONS which model ACCESS CONTROL LIST rules
- If the header matches, forward the relative packet
- Otherwise, drop it

3.3. NAT.

A NAT can be implemented using rewrite rules in packet headers.

4. HSA algebra

In order to implement algorithms based on HSA framework, the following basic operations on HEADER SPACE H are defined:

4.1. Intersection. Given two headers, intersection is defined as the following bit-wise operation:

	0	**1**	**x**
0	0	z	0
1	z	1	1
x	0	1	x

Where `z` means the bitwise intersection is empty. It is an **annihilator**: if any bit returns `z`, the intersection of all bits is empty.

4.2. Union. In general, a union of WILDCARD EXPRESSIONS cannot be simplified. This is why a HEADER SPACE is defined as a union of WILDCARD EXPRESSIONS.

However, there are some specific cases where it can be simplified:

$$\begin{aligned} &1011\text{xxxx} \cup 1011\text{xxxx} = 1011\text{xxxx} \\ &1010\text{xxxx} \cup 1000\text{xxxx} = 10\text{x}0\text{xxxx} \end{aligned} \tag{8}$$

4.3. Complementation. The complement of HEADER `h` is the union of all headers that do *not* intersect with `h`.

Example:

$$(01011\text{xxx})' = 1\text{xxxxxxx} \cup \text{x}0\text{xxxxxx} \cup \text{xx}1\text{xxxxx} \cup \text{xxx}0\text{xxxx} \cup \text{xxxx}0\text{xxx} \tag{9}$$

4.4. Difference. The difference is defined using *complement* and *intersect* operations:

$$A - B = A \cap B' \tag{10}$$

It can be used to *check if one header is a subset of another*:

$$A \subseteq B \Leftrightarrow A - B = \emptyset \tag{11}$$

Example:

(12)

$$\begin{aligned} 101\text{xxx}0\text{x} - 01011\text{x}0\text{x} &= 101\text{xxx}0\text{x} \\ &\cap (1\text{xxxxxxx} \cup \text{x}0\text{xxxxxx} \cup \text{xx}1\text{xxxxx} \cup \text{xxx}0\text{xxxx} \cup \text{xxxx}0\text{xxx} \cup \text{xxxxxx}1\text{x}) \\ &= 101\text{xxx}0\text{x} \cup 101\text{xxx}0\text{x} \cup 101\text{xxx}0\text{x} \cup 1010\text{xx}0\text{x} \cup 101\text{x}0\text{x}0\text{x} \cup \emptyset \\ &= 101\text{xxx}0\text{x} \cup 101\text{xxx}0\text{x} \cup 101\text{xxx}0\text{x} \cup 1010\text{xx}0\text{x} \cup 101\text{x}0\text{x}0\text{x} \end{aligned}$$

In this case `101xxx0x` is not a subset of `01011x0x`.

5. Reachability use-case

One common networking and security problem is nodes reachability.

In particular, in security, absence of reachability could mean:

- One or mode nodes have been compromised
- Flow tables are altered and don't work anymore as designed
- One or more applications have injected rules into the system that affects communication
- A denial-of-service attack is in progress

Thus, reachability check is a great use-case for our domain of interest, because it allows us to face a large class of security problems.

The reachability function `R` between `a` and `b` is:

$$R_{a\to b} = \bigcup_{a\to b \text{ paths}} \{T_n(\Gamma(T_{n-1}(\dots(\Gamma(T_1(h,p))))))\} \tag{13}$$

Where:

- For each path $a \to b$ the TRANSFER FUNCTIONS along the path are: $T_1, T_2, \ldots, T_{n-1}, T_n$
- Γ functions are needed to compute which is the input port of the next TRANSFER FUNCTION that is linked with the output port returned by the previous TRANSFER FUNCTION

The result is a *union of* WILDCARD EXPRESSIONS that describe which headers can reach from `a` to `b`.

If the result is empty ($\emptyset$), it means there aren't any paths that can flow packets from that source to that destination with the given network topology.

6. HSA issues

Header Space Analysis has been developed without considering the dynamic networking needs of SDN.

In fact HSA is unable to stay in sync with the network state *incrementally*, but instead it needs to recompute all transformations every time network changes. This means it doesn't scale well with typical SDN lifecycle operations.

CHAPTER 5

NetPlumber

1. Plumbing Graph

NetPlumber[4] is a verification tool based on Header Space Analysis that provides *real-time model-based policy checking* for dynamic networks.

NetPlumber registers all possible paths of flows through the network in a graph, called Plumbing Graph:

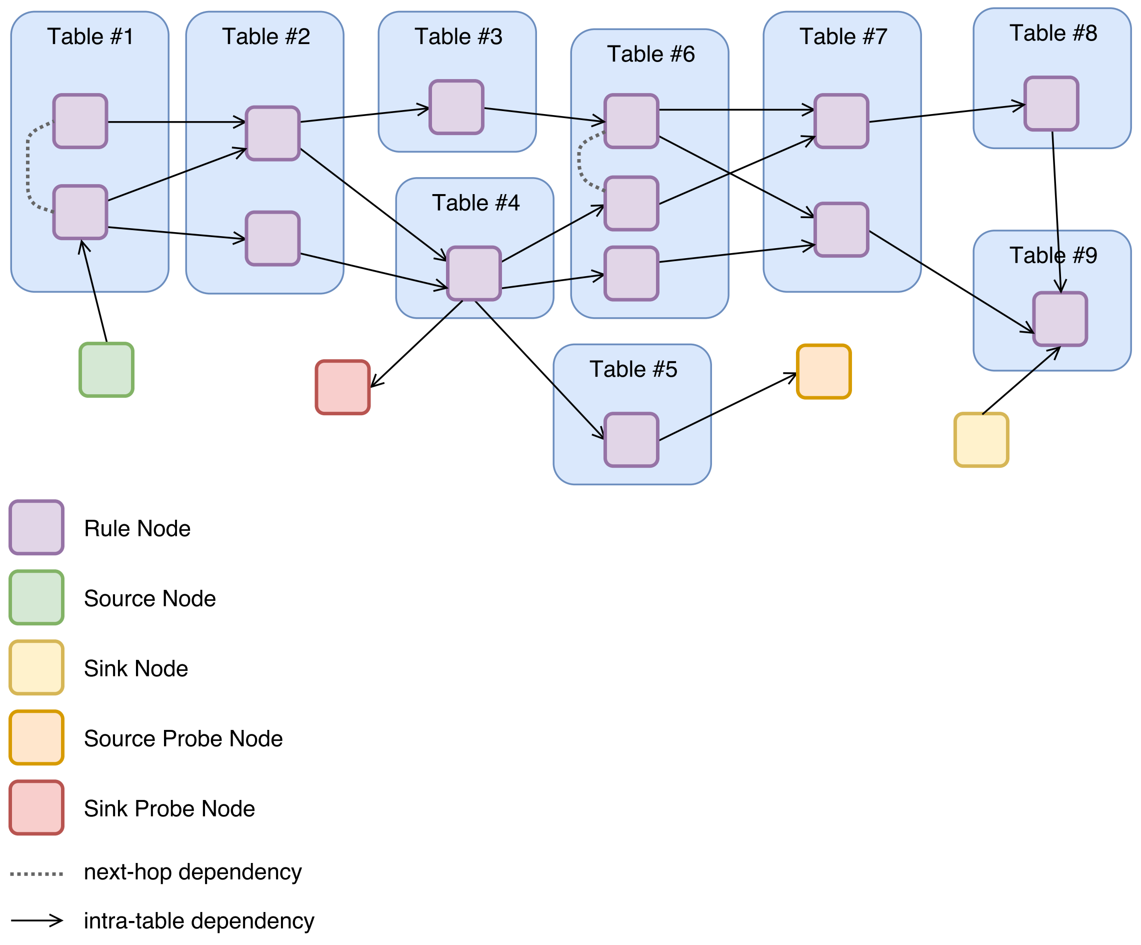


Figure 1. Plumbing Graph

Where:

- A rule node represents the rules in the network. A rule is a OpenFlow-like tuple (match, action) and it can *forward*, *rewrite*, *encapsulate* or *decapsulate* incoming packets. In fact, every rule has:
  - A **match** wildcard expression: used to match incoming flow
  - A set of **in-ports**: used to select which input traffic should be considered or ignored
  - A **out-port**: used to propagate traffic to the next hop
  - A **rewrite** wildcard expression used to perform changes in the flow that passes through the rule
- A next-hop dependency from node `A` to `B` represents a link from rule `A`'s network box to rule `B`'s network box
- Forwarding rules have priorities. A intra-table dependency is a way to represent priority between rules of the same table and thus which should be considered first when flows comes into the table
- A source node is a node capable to generate flow. This flow generator can be used to propagate flow in rule nodes of the Plumbing Graph, useful to check whether policies and assertions are met
- A sink node is the dual of a source node. It generates *sink flow* which traverses the Plumbing Graph in the reverse direction. When reaching a rule node, a sink flow is processed by the inverse of the rule. sink nodes don't increase the expressiveness of NetPlumber but can be used to *optimize or simplify some policies*
- probe nodes are used to check policy or invariants. Typically they process flow coming from sink nodes or source nodes and check the path and header of the received flows for violations of expected behavior. The policies and assertions are declared using a language called Flowexp

2. Sync with SDN

One of the main reasons because NetPlumber is very suited for SDN is that it can easily respond to SDN events:

SDN	**NetPlumber**
Flow-rule added in a network box	Add rule node and potentially change intra-table dependencies and next-hop dependencies
Flow-rule deleted in a network box	Delete rule node, associated intra-table dependencies
network box added	Plumbing Graph remains unchanged
network box removed	Delete all rule nodes of the relative table
Link added	Add next-hop dependencies between rule nodes with ports affected by the link
Link removed	Remove relative next-hop dependencies between affected rule nodes

3. Policy check

NetPlumber uses source nodes (or sink nodes) to generate flow, probe nodes to get the generated flow and check conditions.

A probe node is configured with:

- A *filter* Flowexp: used to decide which flows should be considered
- A *test* Flowexp: used to test whether the policy is met

Conditions and filters are declared in probe nodes with a language called Flowexp Language, a Domain Specific Language capable of expressing policies and invariants.

Flowexp Language is a *regular expression language* designed to check constraints on the history of flows received by probe nodes.

The following table will show the BNF syntax of Flowexp Language:

$$
\begin{array}{rl}
\textit{Constraint} \rightarrow & \text{True} \mid \text{False} \\
\mid & \textit{!Constraint} \\
\mid & (\textit{Constraint} \mid \textit{Constraint}) \\
\mid & (\textit{Constraint} \; \& \; \textit{Constraint}) \\
\mid & \textit{PathConstraint} \\
\mid & \textit{HeaderConstraint} \\
\textit{PathConstraint} \rightarrow & \text{list}(\textit{Pathlet}) \\
\textit{Pathlet} \rightarrow & \text{Port Specifier } [p \in \{P_i\}] \\
\mid & \text{Table Specifier } [t \in \{T_i\}] \\
\mid & \text{Skip Next Hop } [.] \\
\mid & \text{Skip Zero or More Hops } [.*] \\
\mid & \text{Beginning of Path } [\wedge] \\
\mid & \text{End of Path } [\$] \\
\textit{HeaderConstraint} \rightarrow & H_{received} \cap H_{constraint} \mid \\
H_{received} \subset H_{constraint} \mid & H_{received} = H_{constraint}
\end{array}
$$

First production rule (*Constraint*) gives us the possibility to create composite rules starting from simple constraints and combining them into more complex rules.

Second production rule defines what is the main content of a Flowexp: an ordered list of elementary building blocks called pathlets.

A pathlet is a way to specify constraints on the path taken by a flow. pathlets are sequentially checked and if a constraint fails, all the Flowexp fails.

Third production rule (*PathConstraint*) defines how a pathlet can be made:

- **Port Specifier**: Constraint on which port the flow should pass through
- **Table Specifier**: Constraint on which table the flow should pass through
- **Skip Next Hop**: Don't apply any constraint on the next hop in the flow's path
- **Skip Zero or More Hops**: Same as *Skip Next Hop* but with different cardinality
- **Beginning of Path**: Constraint that specifies the flow should begin its path with that pathlet and thus next flow's hop should match the next pathlet
- **End of Path**: Constraint that specifies the flow should end its path with that pathlet and thus there aren't any flow hops left

Last production rule (*HeaderConstraint*) defines constraints on the packet's content of interest (called header in `HSA` and NetPlumber).

As we'll see in the next chapter, that language has been mapped into `ONOS` in order to allow `ONOS` applications to create flow expressions based on `SDN` terms. The system will, underneath, translate `SDN` terms into respective NetPlumber terms.

4. Reachability check use-case

In previous sections NetPlumber framework and relative abstractions have been introduced.

Now we'll try to model a **Reachability Check Policy** with NetPlumber.

The reachability problem from node `A` to node `B` can be formulated as: "there should exist some flow that can travel from node `A` to node `B`".

A possible solution is to use:

- A source node that generates a *wildcarded flow* at each of node `A` ports
- A source probe node connected to destination port of `B` with the following condition:

$$\exists f : f.path \sim [\wedge(p \in \{P_1, P_2, \ldots, P_n\})] \tag{14}$$

Where $\{P_1, P_2, \ldots, P_n\}$ are the source ports of node `A`

CHAPTER 6

Model Checking Service

1. Vision

In the last chapters, we've seen many security issues related to the current state of `SDN` platform.

In fact there are a lot of threats regarding vulnerabilities or weak designs that could lead to attacks and exploitation activity.

However, let's try to analyse those problems from a different perspective. Instead of focusing on vulnerabilities or weaknesses, *let's focus on the consequences and the effects.* Let's try to answer to the question "What happens if vulnerability is exploited and attack succeeds?", instead of "How can I recognize every vulnerability and attack surface?"

This vision puts us in an abstract conceptual space suitable for the system that we want to realize, *without having to consider every single source of potential security issues, but instead focusing on side-effects.*

2. Goals

Let's define some high-level concrete goals:

2.1. The development process.

We've chosen an *incremental* and *model-driven* development process.

In the first iteration we want to build a basic artifact, with at least the Case Study working, while keeping the system open for extension, in order to implement more case studies in possible later works.

2.2. Think about problems, not solutions.

The purpose of this thesis is also to *define a generic framework to face up new security problems in* `SDN` *networks, without having to reinvent the wheel* every time a new security issue appears.

Since **technologies become obsolete, but problems don't**, we want to design abstraction layers to *decouple* technology-specific parts with technology-independent ones.

Following this idea, we should define:

(1) A formalization of `MCS` requirements in the section **REQUIREMENTS ANALYSIS**
(2) A formalization of `MCS` problem in the section **PROBLEM ANALYSIS**
(3) A formalization of `MCS` design in the section **PROJECT**

2.3. First things first.

Important things should always have higher priority.

Since we have limited resources, in this thesis we will follow this ordered list of priorities:

(1) PROBLEM ANALYSIS
(2) PROJECT
(3) Main abstractions implementation
(4) Implementation of technology-independent parts

(5) Implementation of technology-dependent parts

2.4. Encourage software reuse.
We should use as much as possible existing analysis to popular problems and relative designs.
For this reason we'll use **Design Patterns** and **Best Practices** to speed up the development process and to create a better final artifact.

3. Requirements

Let's try to be our customers and to formulate requirements in a human-language:

```
Develop a model-checking based policy system, that allows to build a network
model and check if relative policies are met.

The model should be kept in sync automatically with the underlying Software
Defined Network state, without any manual intervention.

The system should allow to insert parametrized policies to perform checks on
the network state.
They should be inserted manually, by human operators called Network Policy
Designers.

When network changes, the changes should be automatically propagated to the
underlying network model to the policy system.
In particular when devices, links and interaction rules between them are added,
updated or deleted, the network model should reflect changes accordingly.

Policy checks should rely just on the model, because network couldn't be
accessible at the time when check is done and because network probing could lead
to false-positives or false-negatives.

Policy checks should be real-time, thus network changes should trigger
re-evaluation of existing policies.

The output of policy checks should be available to external operators. Those
operators, called Network Policy Response Operator, have different role than
Network Policy Designers, in fact they shouldn't be allowed to add new policies
but just check the state of existing policies.

Possibly the system should be flexible enough to add new kinds of policy checks
without having to reinvent the wheel, but relying on the existing system,
instructing it with the new policy behavior.
```

In successive steps, during REQUIREMENTS ANALYSIS, we'll try to refine and formalize the requirements defined above.

4. Case Study

In addition to the generic requirements, we want to consider a specific case-study, relevant for as many security aspects as possible.

However, since the model-checking framework is generic, the security expert can place as many different policies as needed, in order to enforce the desired network behavior.

This is just another strength of `MCS`: it doesn't implement any policy by itself, but provides the security expert the needed tool to define specific policy to meet specific security needs.

A good case study is a **Reachability Check**, because it allows to detect a wide range of problems seen in previous chapters:

- Denial-of-Service attacks can be detected with a **Reachability Check**, because if an intermediate or edge node is under `DoS`, the nodes under test aren't reachable. Also the check is real-time, allowing an operator or an automatic system to automatically *mitigate* the attack as soon as it happens, without any significant delay between the attack and the detection
- SDN Controller attacks that can possibly result in database corruption, changing its perception of the underlying network
- One or more network boxes have been compromised and flow-rules don't permit nodes reachability anymore
- Attacks that could cause inner loops in the network, difficult to be detected by normal real network scans, can be detected easily with a model-based check
- System resources exhaustion sometimes can lead to `DoS` and thus can be detected
- Some kind of SDN Controller exploitation, when forwarding activity is compromised

What does not detect:

- Malware infection in SDN Controller that leads to some database information leakage or other side effects that don't affect nodes connectivity
- Any kind of exploitation both in SDN Controller and network boxes without changing or compromising traffic forwarding
- `SBI` communication flooding. In that case Plumbing Graph isn't updated and is unable to detect the attack

5. REQUIREMENTS ANALYSIS

5.1. Glossary.

Let's define meaning of terms used in the requirements above in the following glossary, in order to reduce ambiguity between terms:

Term	**Synonym**	**Meaning**
Model Check	Policy Check	Check if a single or a set of policies is met
NETWORK MODEL	Policy Model	State of the network model
Network State	Physical Network State	State of the physical network, how devices are connected and linked, which flow-rules are installed
Software Defined Network	SDN	Standard name used to refer to the underlying network technology
Assertion	—	Rule about a condition that the NETWORK MODEL can respect
Policy	—	Parametrized set of assertions that is met if every assertion is evaluated to true
Policy Behavior	—	The behavior of a policy: what the policy is supposed to check
Network Policy Designer	—	Professional role that has responsibility to instantiate policies with custom parameters
Network Policy Response Operator	—	Professional role that can take a look at NETWORK MODEL output and policies state
Real-time	—	Function should take effect immediately, when needed inputs and dependencies are available, without having to restart the system or part of it
Policy State	Policy output, output of policy checks	Status of a policy: if it is met or not

5.2. Requirements Model.

The model formalizes the requirements declared in Requirements section.

This model has been introduced to overcome the necessity to glue the requirements document with the rest of the development process.

In particular:

- We formalize the FUNCTIONAL REQUIREMENTS, that will be used later in the USE-CASES
- We formalize the NON-FUNCTIONAL REQUIREMENTS, that will be used later in the PROJECT

5.3. FUNCTIONAL REQUIREMENTS.

FUNCTIONAL REQUIREMENTS are the functions of `MCS`. A function is described by a set of inputs, the behavior and the outputs. In other words, FUNCTIONAL REQUIREMENTS are *what the system is supposed to do.*

This model expresses three main concepts:

- **Features**: Describe pieces of functional behavior that yield a specific result
- **Business Rules**: It's a catalogue of explicit business rules which are required to be implemented within the current project. Business Rules are typically executed during program execution and control the processing of information and transactions. They can be seen as constraints that has to be respected in order to allow correct program execution (in terms of the customer expectations)
- **User Interface**: Contain high-level descriptions of end-user visible screens and forms which are *required* to support the proposed system

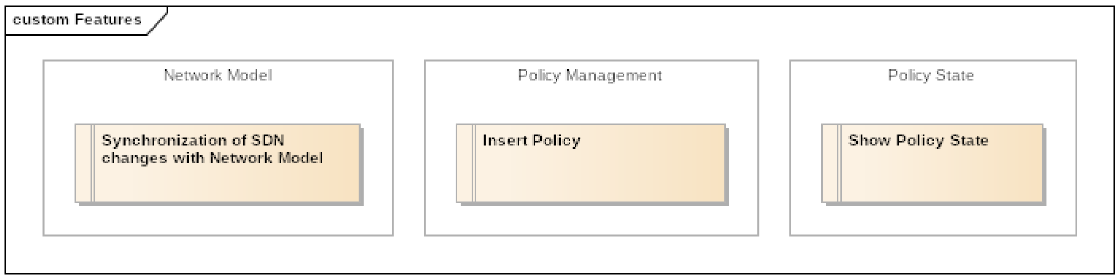


FIGURE 1. FUNCTIONAL REQUIREMENTS — **Features**

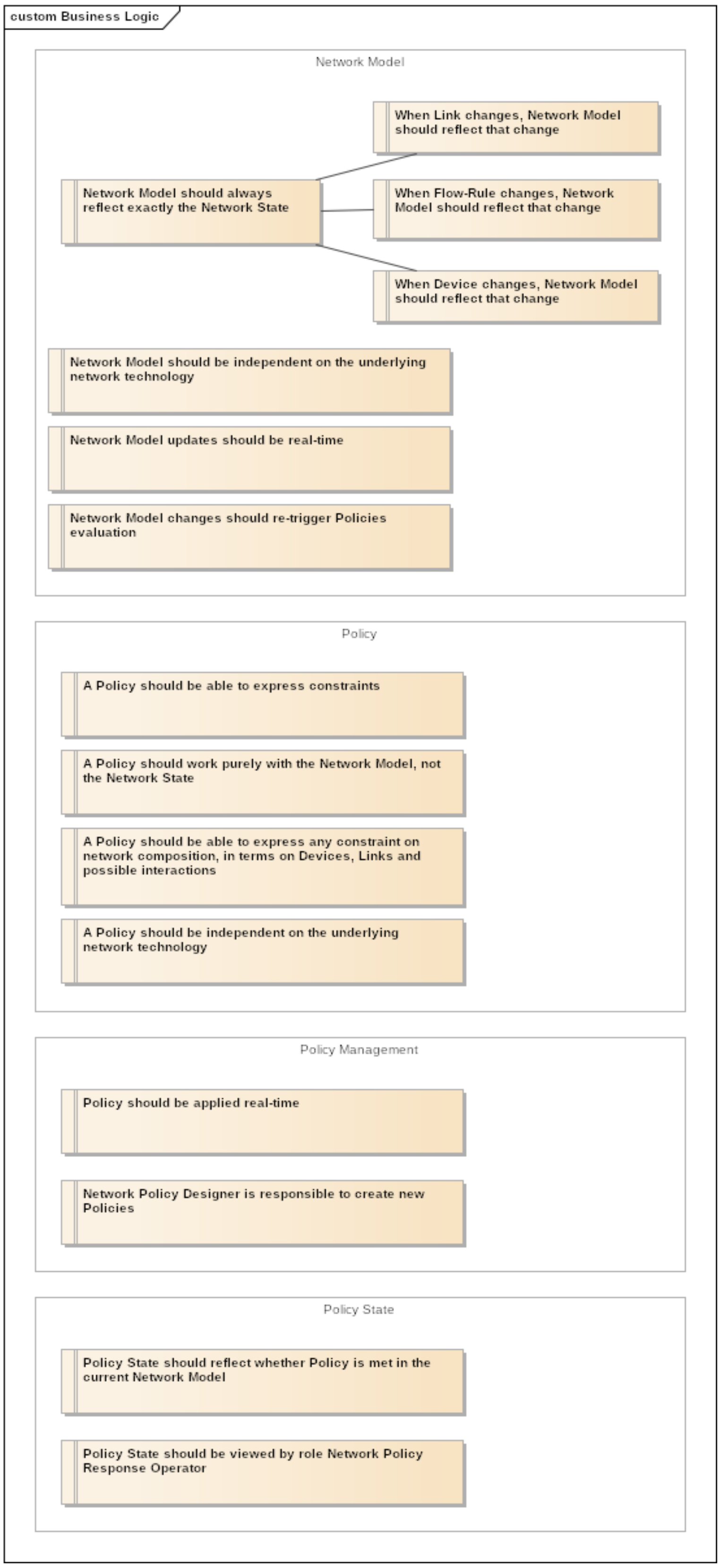

custom Business Logic
Network Model
When Link changes, Network Model should reflect that change
Network Model should always reflect exactly the Network State
When Flow-Rule changes, Network Model should reflect that change
When Device changes, Network Model should reflect that change
Network Model should be independent on the underlying network technology
Network Model updates should be real-time
Network Model changes should re-trigger Policies evaluation
Policy
A Policy should be able to express constraints
A Policy should work purely with the Network Model, not the Network State
A Policy should be able to express any constraint on network composition, in terms on Devices, Links and possible interactions
A Policy should be independent on the underlying network technology
Policy Management
Policy should be applied real-time
Network Policy Designer is responsible to create new Policies
Policy State
Policy State should reflect whether Policy is met in the current Network Model
Policy State should be viewed by role Network Policy Response Operator

Figure 2. functional requirements — **Business Logic**

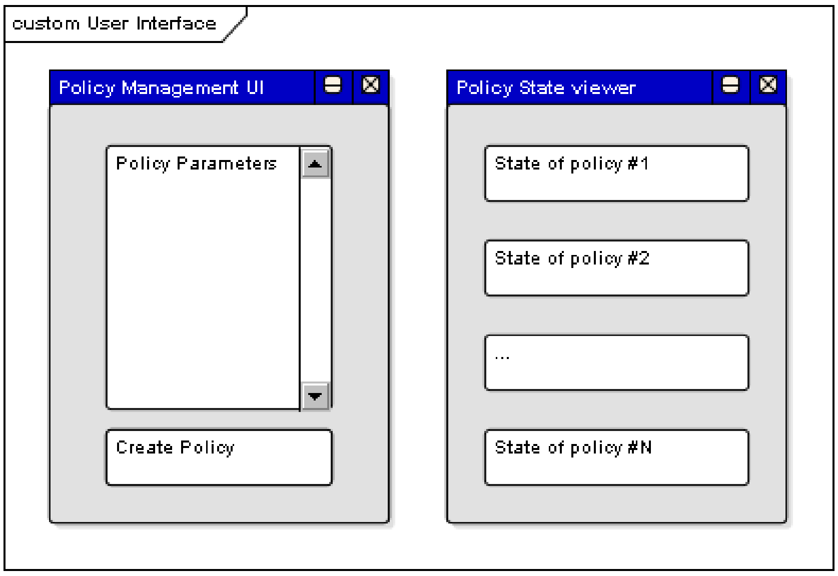


FIGURE 3. FUNCTIONAL REQUIREMENTS — **UI**

5.4. NON-FUNCTIONAL REQUIREMENTS.

NON-FUNCTIONAL REQUIREMENTS specify criteria that can be used to judge the quality of the software artifacts.

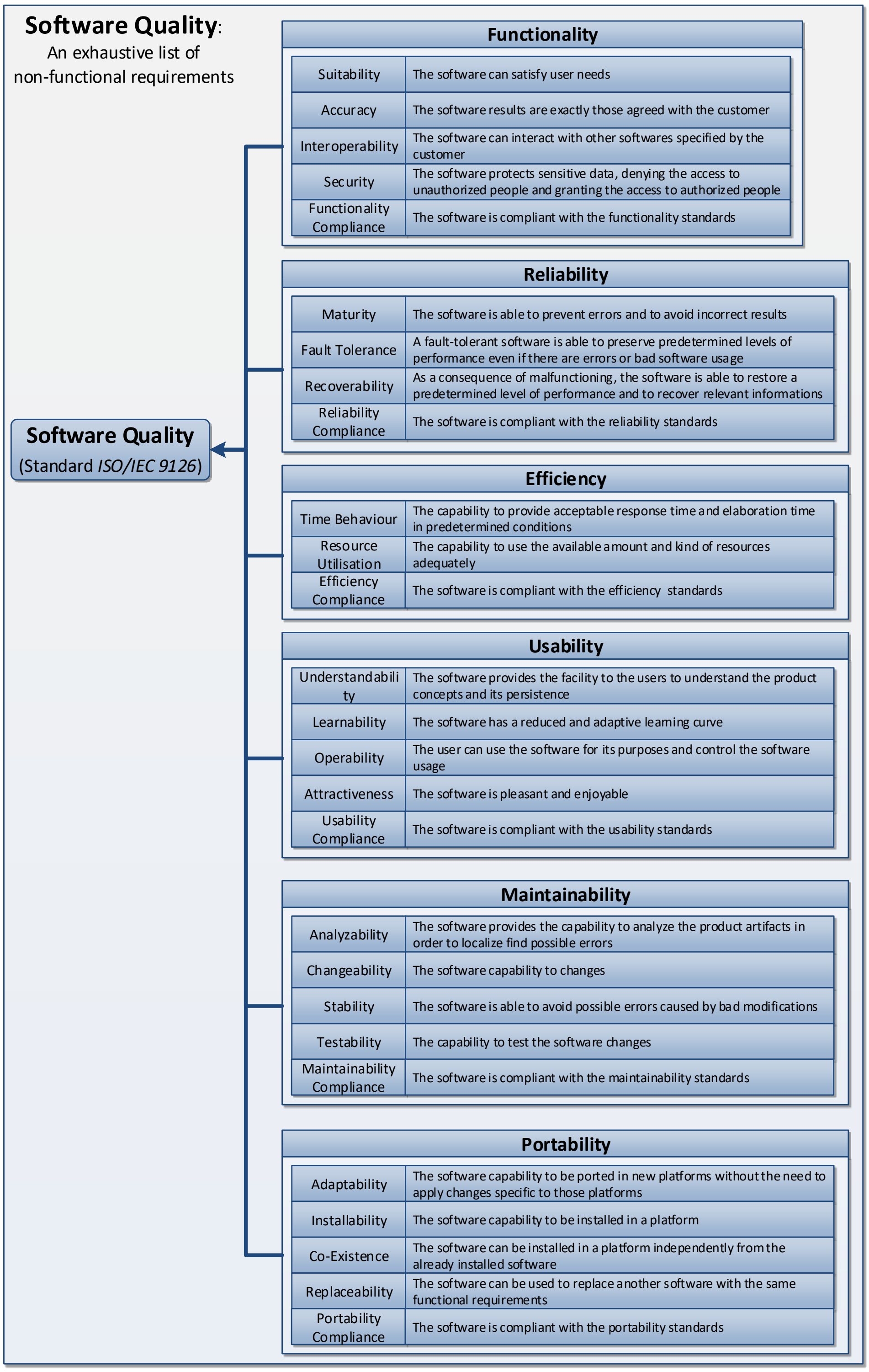
In other words, NON-FUNCTIONAL REQUIREMENTS are *how the system does* what is supposed to do.

The model below supplies a taxonomy of the available NON-FUNCTIONAL REQUIREMENTS, as defined in the standard `ISO/IEC 9126`.

We remind that every well made system should respect all of those requirements, at least with some degree of compliance.

However our particular software `MCS` pays particular attention to:

- **Functionality**:
  - **Accuracy**
  - **Interoperability**
  - **Functionality Compliance**
- **Efficency**:
  - **Time Behavior**
- **Usability**:
  - **Learnability**
  - **Operability**
- **Usability**:
  - **Analyzability**
  - **Changeability**
  - **Testability**
- **Portability**:
  - **Adaptability**
  - **Installability**
  - **Co-Existence**



Software Quality: An exhaustive list of non-functional requirements

Software Quality (Standard *ISO/IEC 9126*)

Functionality

Suitability	The software can satisfy user needs
Accuracy	The software results are exactly those agreed with the customer
Interoperability	The software can interact with other softwares specified by the customer
Security	The software protects sensitive data, denying the access to unauthorized people and granting the access to authorized people
Functionality Compliance	The software is compliant with the functionality standards

Reliability

Maturity	The software is able to prevent errors and to avoid incorrect results
Fault Tolerance	A fault-tolerant software is able to preserve predetermined levels of performance even if there are errors or bad software usage
Recoverability	As a consequence of malfunctioning, the software is able to restore a predetermined level of performance and to recover relevant informations
Reliability Compliance	The software is compliant with the reliability standards

Efficiency

Time Behaviour	The capability to provide acceptable response time and elaboration time in predetermined conditions
Resource Utilisation	The capability to use the available amount and kind of resources adequately
Efficiency Compliance	The software is compliant with the efficiency standards

Usability

Understandabili ty	The software provides the facility to the users to understand the product concepts and its persistence
Learnability	The software has a reduced and adaptive learning curve
Operability	The user can use the software for its purposes and control the software usage
Attractiveness	The software is pleasant and enjoyable
Usability Compliance	The software is compliant with the usability standards

Maintainability

Analyzability	The software provides the capability to analyze the product artifacts in order to localize find possible errors
Changeability	The software capability to changes
Stability	The software is able to avoid possible errors caused by bad modifications
Testability	The capability to test the software changes
Maintainability Compliance	The software is compliant with the maintainability standards

Portability

Adaptability	The software capability to be ported in new platforms without the need to apply changes specific to those platforms
Installability	The software capability to be installed in a platform
Co-Existence	The software can be installed in a platform independently from the already installed software
Replaceability	The software can be used to replace another software with the same functional requirements
Portability Compliance	The software is compliant with the portability standards

Figure 4. non-functional requirements

6. USE-CASES

The USE-CASE model is a catalogue of system functionality described using UML USE-CASES. Each USE-CASE represents a single, repeatable interaction that an ACTOR experiences when using the system.

ACTORS are the users of the system being modeled. Each ACTOR will have a well-defined role, and in the context of that role have useful interactions with the system.

A person may perform the role of more than one ACTOR, although they will only assume one role during one USE-CASE interaction. An ACTOR role may be performed by a non-human system, such as another computer program.

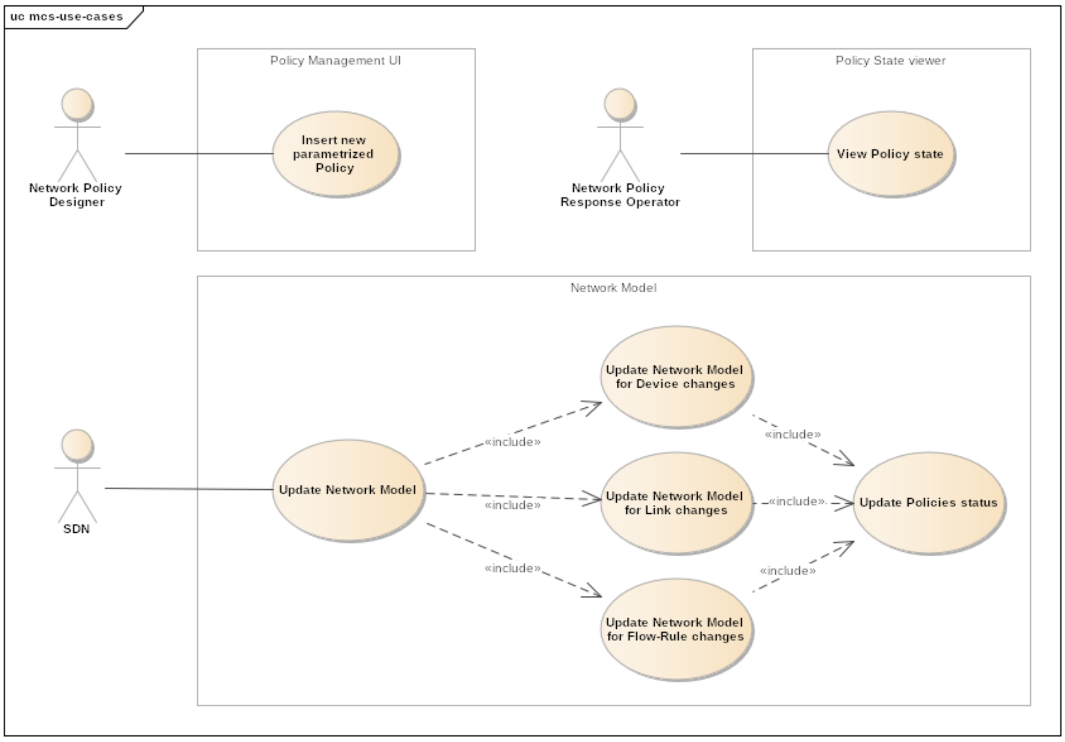

uc mcs-use-cases
Policy Management UI
Network Policy Designer
Insert new parametrized Policy
Policy State viewer
Network Policy Response Operator
View Policy state
Network Model
SDN
Update Network Model
«include»
Update Network Model for Device changes
Update Network Model for Link changes
Update Network Model for Flow-Rule changes
Update Policies status

FIGURE 5. USE-CASES

They involve the following actors:

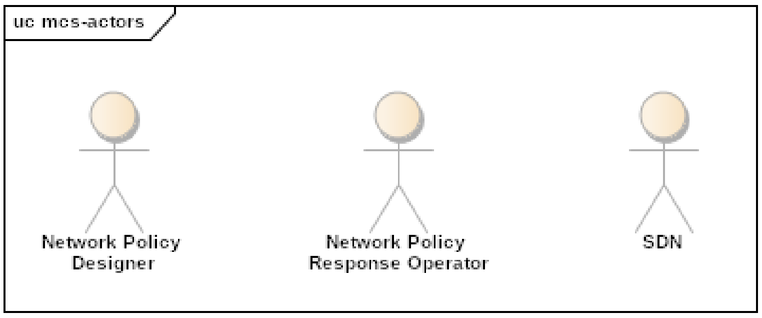


FIGURE 6. ACTORS

7. domain model

The domain model is a view of all the objects that make up an area of interest, and their relationships. It is used here to capture the significant objects within MCS system:

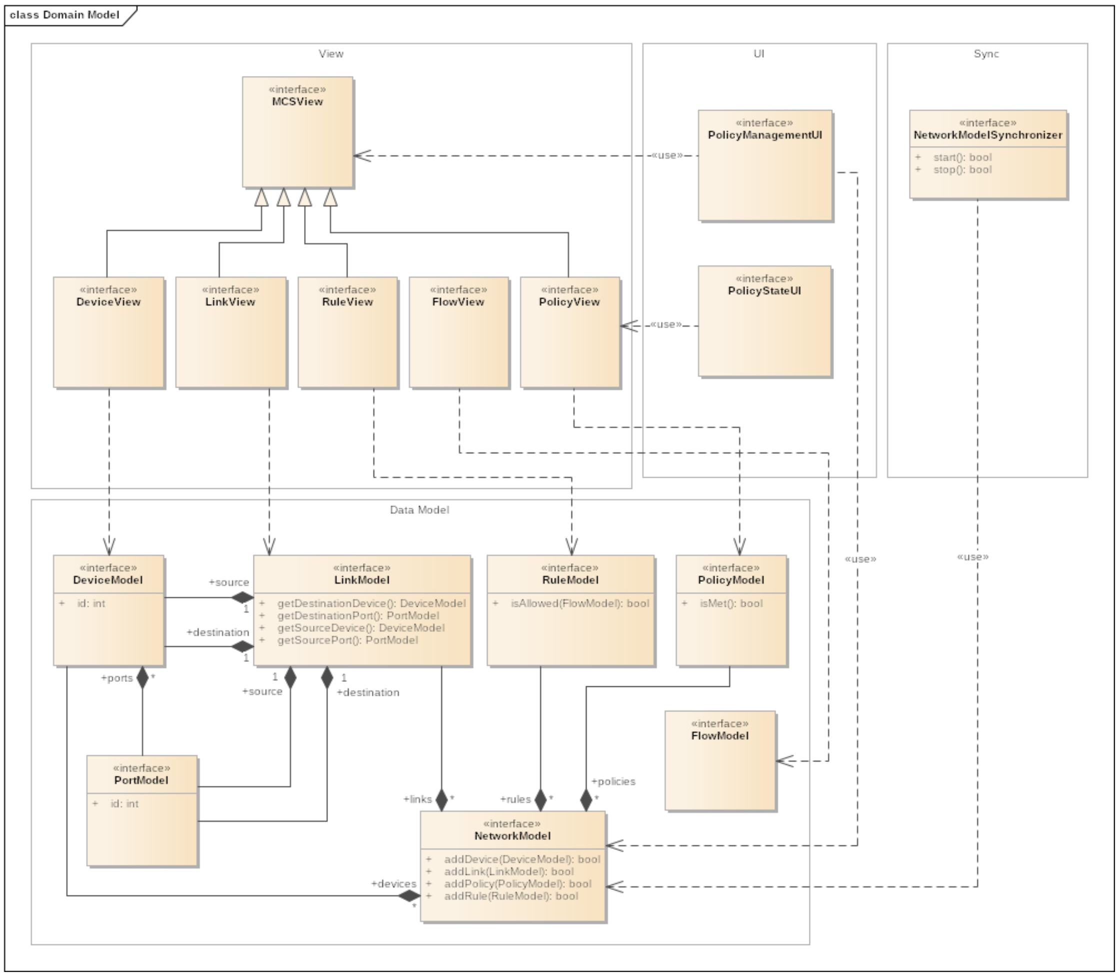


Figure 7. domain model

8. PROBLEM ANALYSIS (LOGIC ARCHITECTURE)

The LOGIC ARCHITECTURE shows the logical structure and interaction between the system components.

Let's start considering the artifacts coming from the REQUIREMENTS ANALYSIS:

- **USE-CASES**
- **DOMAIN MODEL**

The FUNCTIONAL REQUIREMENTS defined from those models are now considered to start defining what's the system architecture from a logical point of view.

The following UML Class diagrams show the structure of main `MCS` components.

`MCS` structure can be seen as union of four subsystems:

- **UI**: Permit user to interact with `MCS`
- **Services**: Expose operations and `MCS` functionality to **UI**
- **Operations**: Map incoming requested operations into appropriate business logic, that change or query the **NETWORK MODEL**
- **NETWORK MODEL**: Model-checking abstraction, that allows to manipulate the model with abstract terms and concepts, *decoupled from the underlying technology.* It also allows to define policies that check model properties

8.1. UI Structure.

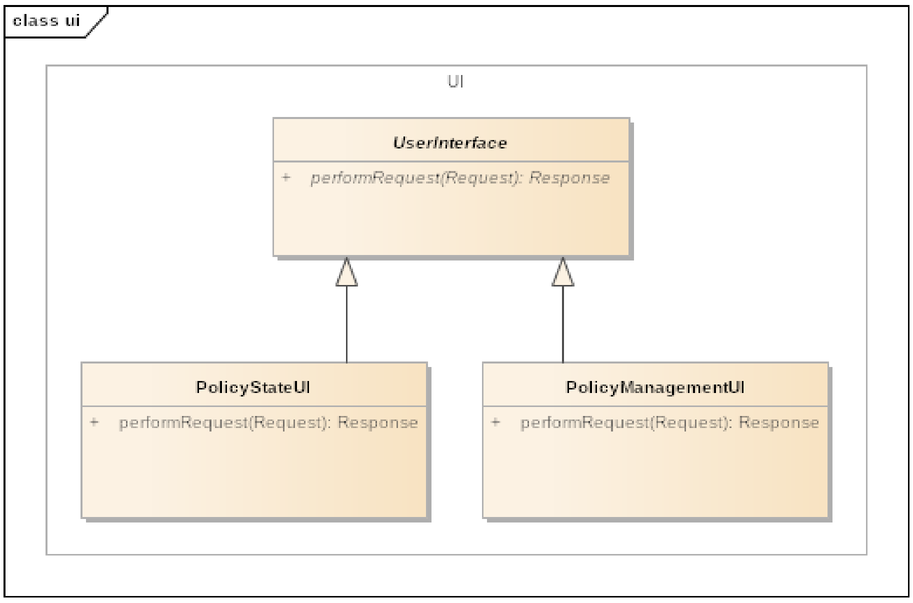

class ui
UI
UserInterface
+ performRequest(Request): Response
PolicyStateUI
+ performRequest(Request): Response
PolicyManagementUI
+ performRequest(Request): Response

FIGURE 8. LOGIC ARCHITECTURE — Structure — UI

8.2. Services Structure.

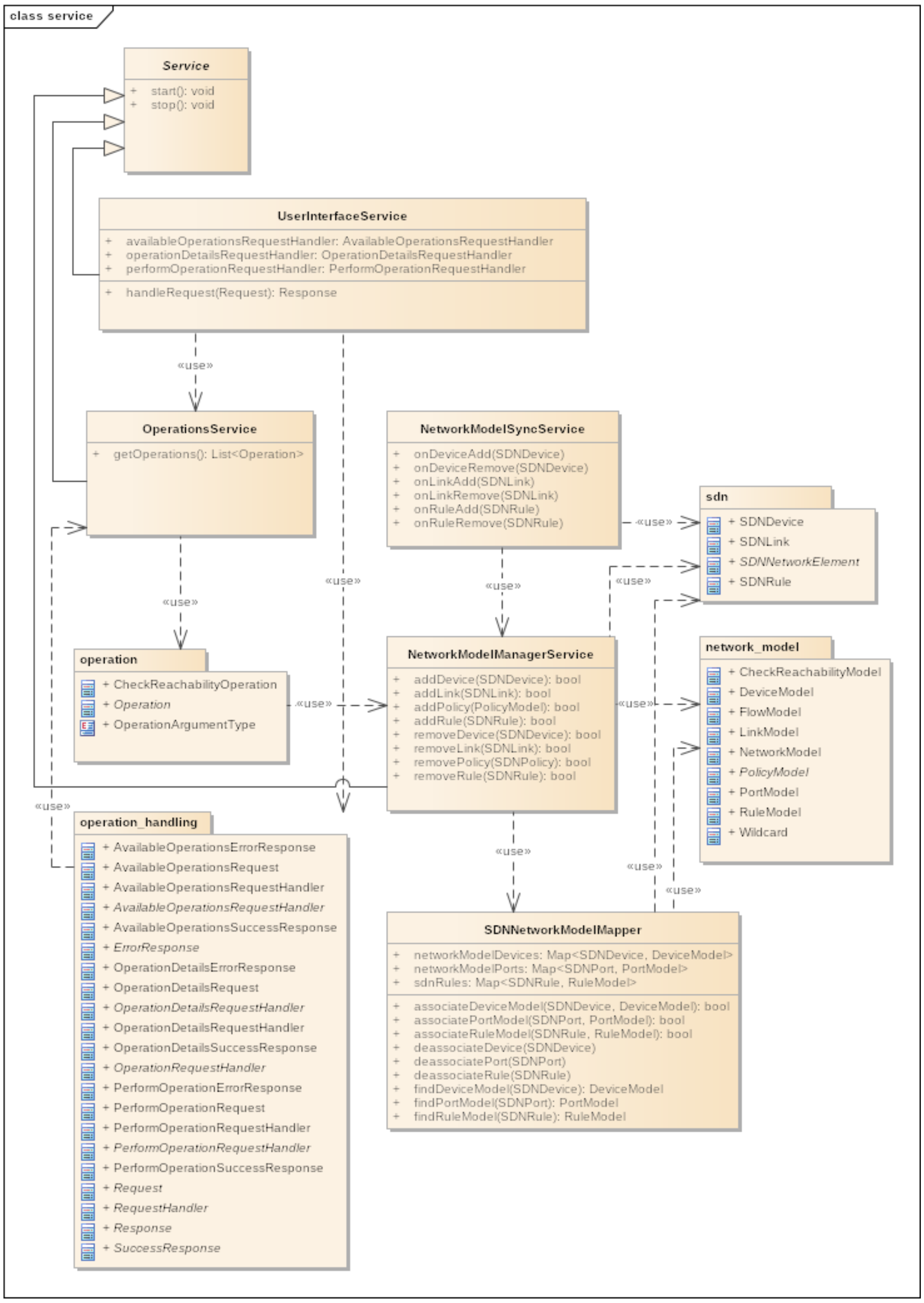

class service
Service
+ start(): void
+ stop(): void
UserInterfaceService
+ availableOperationsRequestHandler: AvailableOperationsRequestHandler
+ operationDetailsRequestHandler: OperationDetailsRequestHandler
+ performOperationRequestHandler: PerformOperationRequestHandler
+ handleRequest(Request): Response
«use»
OperationsService
+ getOperations(): List<Operation>
NetworkModelSyncService
+ onDeviceAdd(SDNDevice)
+ onDeviceRemove(SDNDevice)
+ onLinkAdd(SDNLink)
+ onLinkRemove(SDNLink)
+ onRuleAdd(SDNRule)
+ onRuleRemove(SDNRule)
sdn
+ SDNDevice
+ SDNLink
+ SDNNetworkElement
+ SDNRule
operation
+ CheckReachabilityOperation
+ Operation
+ OperationArgumentType
NetworkModelManagerService
+ addDevice(SDNDevice): bool
+ addLink(SDNLink): bool
+ addPolicy(PolicyModel): bool
+ addRule(SDNRule): bool
+ removeDevice(SDNDevice): bool
+ removeLink(SDNLink): bool
+ removePolicy(SDNPolicy): bool
+ removeRule(SDNRule): bool
network_model
+ CheckReachabilityModel
+ DeviceModel
+ FlowModel
+ LinkModel
+ NetworkModel
+ PolicyModel
+ PortModel
+ RuleModel
+ Wildcard
operation_handling
+ AvailableOperationsErrorResponse
+ AvailableOperationsRequest
+ AvailableOperationsRequestHandler
+ AvailableOperationsRequestHandler
+ AvailableOperationsSuccessResponse
+ ErrorResponse
+ OperationDetailsErrorResponse
+ OperationDetailsRequest
+ OperationDetailsRequestHandler
+ OperationDetailsRequestHandler
+ OperationDetailsSuccessResponse
+ OperationRequestHandler
+ PerformOperationErrorResponse
+ PerformOperationRequest
+ PerformOperationRequestHandler
+ PerformOperationRequestHandler
+ PerformOperationSuccessResponse
+ Request
+ RequestHandler
+ Response
+ SuccessResponse
SDNNetworkModelMapper
+ networkModelDevices: Map<SDNDevice, DeviceModel>
+ networkModelPorts: Map<SDNPort, PortModel>
+ sdnRules: Map<SDNRule, RuleModel>
+ associateDeviceModel(SDNDevice, DeviceModel): bool
+ associatePortModel(SDNPort, PortModel): bool
+ associateRuleModel(SDNRule, RuleModel): bool
+ deassociateDevice(SDNDevice)
+ deassociatePort(SDNPort)
+ deassociateRule(SDNRule)
+ findDeviceModel(SDNDevice): DeviceModel
+ findPortModel(SDNPort): PortModel
+ findRuleModel(SDNRule): RuleModel

FIGURE 9. LOGIC ARCHITECTURE — Structure — Services

8.3. Operation Handling Structure.

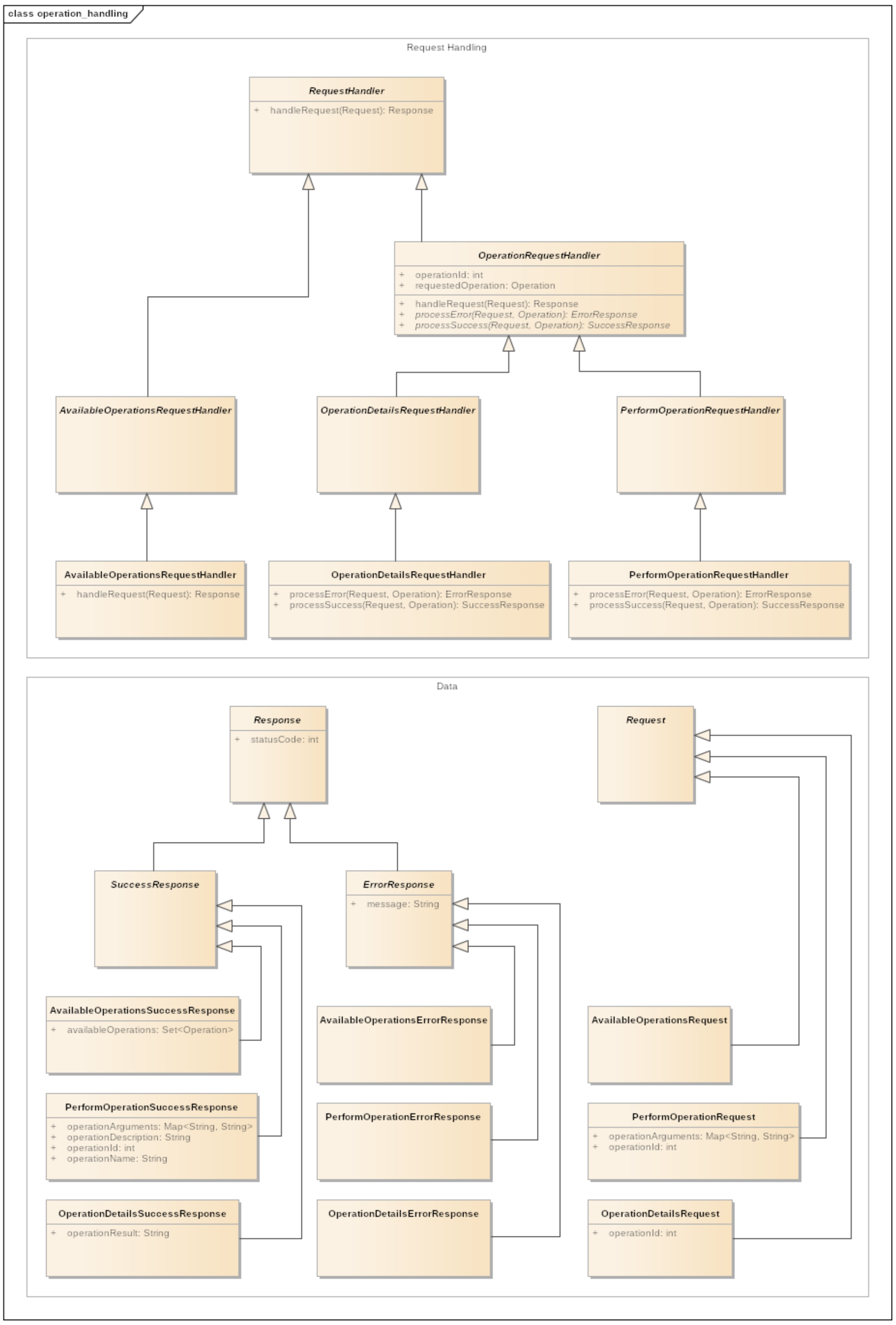

class operation_handling
Request Handling
RequestHandler
+ handleRequest(Request): Response
OperationRequestHandler
+ operationId: int
+ requestedOperation: Operation
+ handleRequest(Request): Response
+ processError(Request, Operation): ErrorResponse
+ processSuccess(Request, Operation): SuccessResponse
AvailableOperationsRequestHandler
OperationDetailsRequestHandler
PerformOperationRequestHandler
AvailableOperationsRequestHandler
+ handleRequest(Request): Response
OperationDetailsRequestHandler
+ processError(Request, Operation): ErrorResponse
+ processSuccess(Request, Operation): SuccessResponse
PerformOperationRequestHandler
+ processError(Request, Operation): ErrorResponse
+ processSuccess(Request, Operation): SuccessResponse
Data
Response
+ statusCode: int
Request
SuccessResponse
ErrorResponse
+ message: String
AvailableOperationsSuccessResponse
+ availableOperations: Set<Operation>
PerformOperationSuccessResponse
+ operationArguments: Map<String, String>
+ operationDescription: String
+ operationId: int
+ operationName: String
OperationDetailsSuccessResponse
+ operationResult: String
AvailableOperationsErrorResponse
PerformOperationErrorResponse
OperationDetailsErrorResponse
AvailableOperationsRequest
PerformOperationRequest
+ operationArguments: Map<String, String>
+ operationId: int
OperationDetailsRequest
+ operationId: int

FIGURE 10. LOGIC ARCHITECTURE — Structure — Operation Handling

8.4. Operation Data Types.

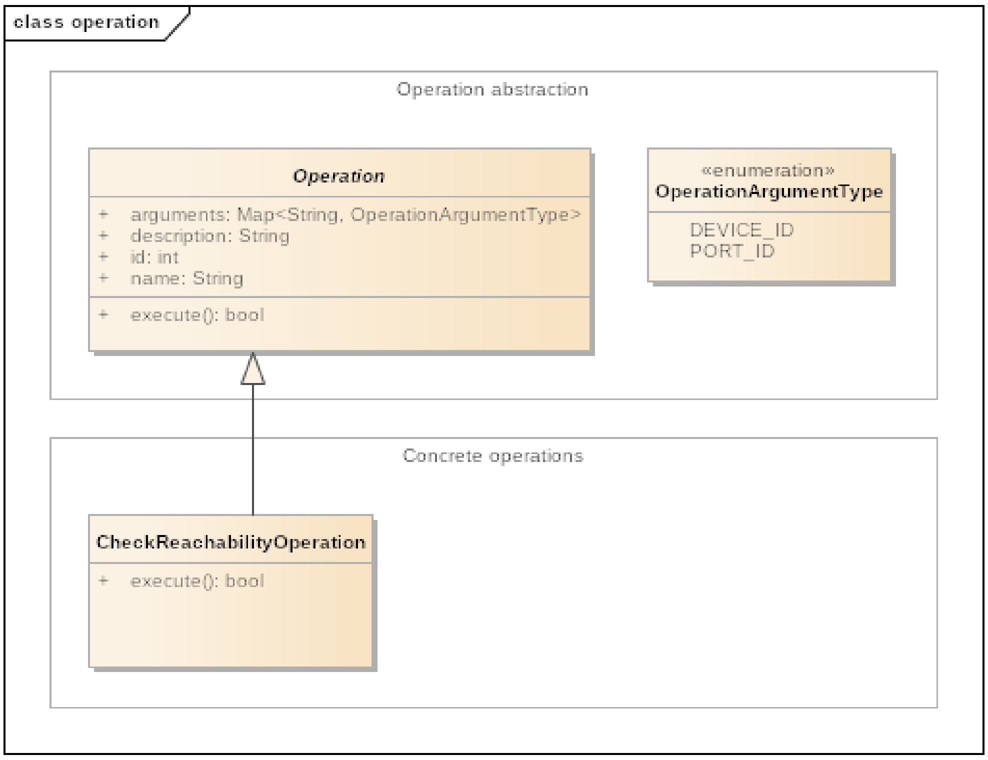


FIGURE 11. LOGIC ARCHITECTURE — Structure — Operation Data Types

8.5. NETWORK MODEL Data Types.

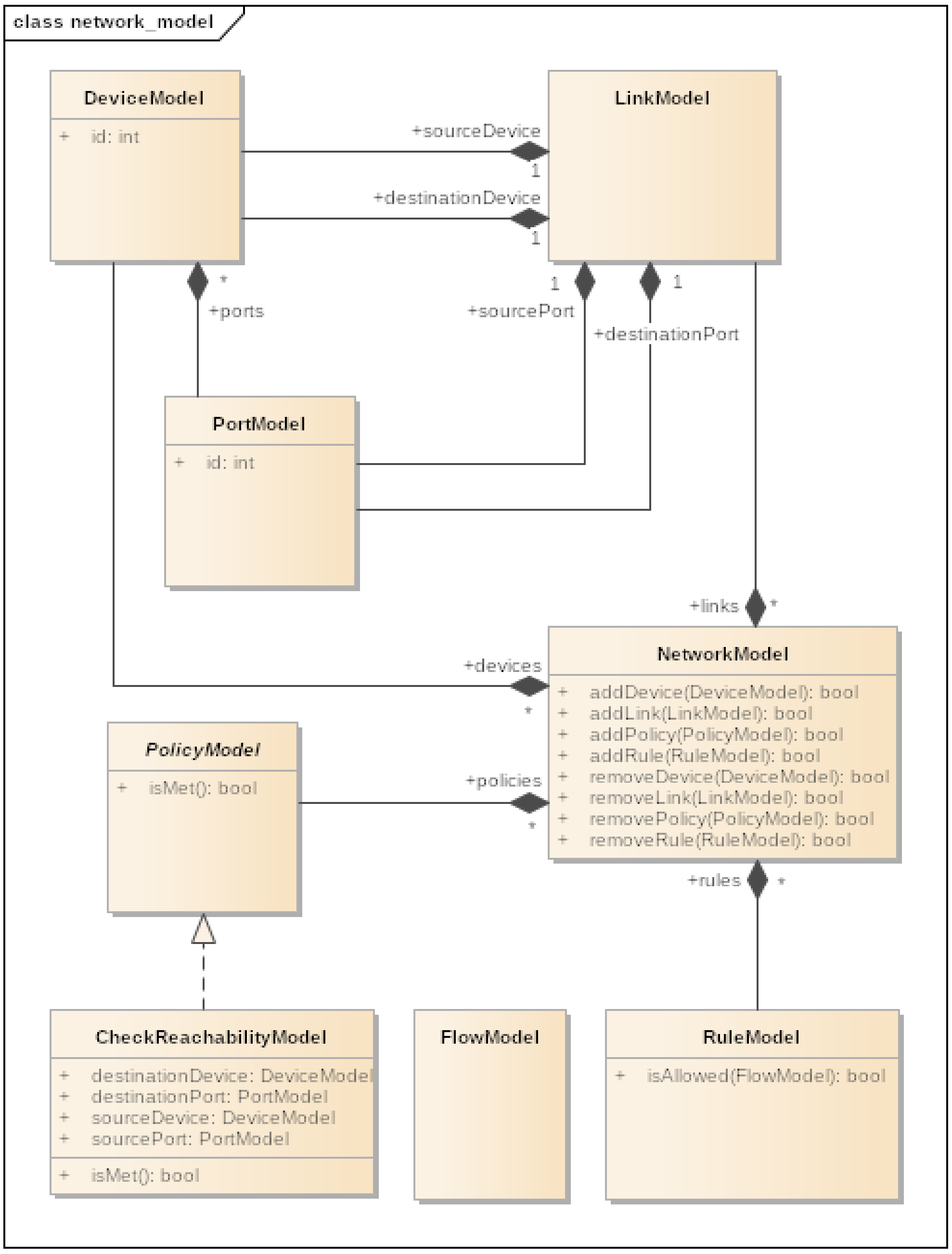

class network_model
DeviceModel
+ id: int
LinkModel
+sourceDevice
1
+destinationDevice
1
*
+ports
1
+sourcePort
1
+destinationPort
PortModel
+ id: int
+links
*
NetworkModel
+devices
*
+ addDevice(DeviceModel): bool
+ addLink(LinkModel): bool
+ addPolicy(PolicyModel): bool
+ addRule(RuleModel): bool
+ removeDevice(DeviceModel): bool
+ removeLink(LinkModel): bool
+ removePolicy(PolicyModel): bool
+ removeRule(RuleModel): bool
PolicyModel
+ isMet(): bool
+policies
*
+rules
*
CheckReachabilityModel
+ destinationDevice: DeviceModel
+ destinationPort: PortModel
+ sourceDevice: DeviceModel
+ sourcePort: PortModel
+ isMet(): bool
FlowModel
RuleModel
+ isAllowed(FlowModel): bool

FIGURE 12. LOGIC ARCHITECTURE — Structure — NETWORK MODEL Data Types

8.6. SDN Data Types.

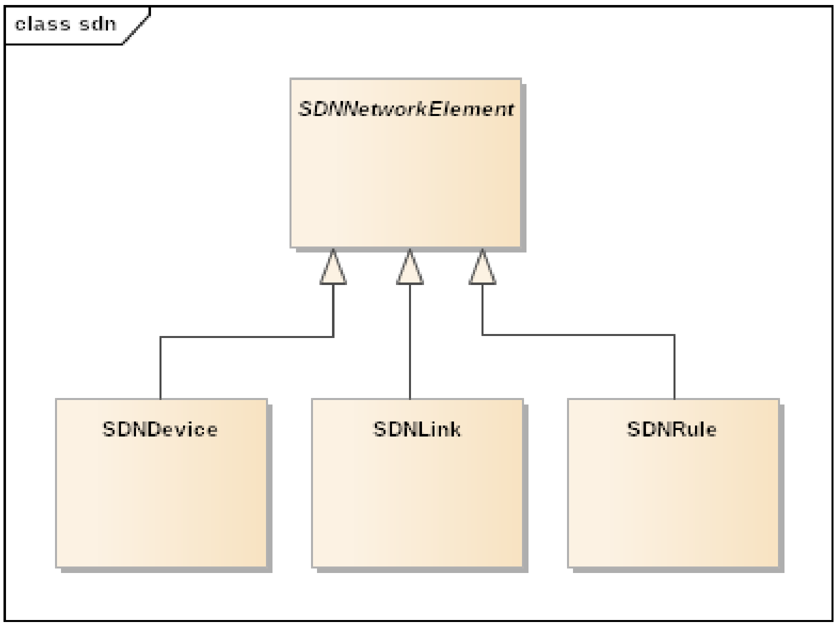


Figure 13. logic architecture — Structure — SDN Data Types

8.7. Execute-Operation Interaction.

The following UML sequence diagram shows the basic interaction flow when a Network Policy Designer triggers an Operation to create a new Policy:

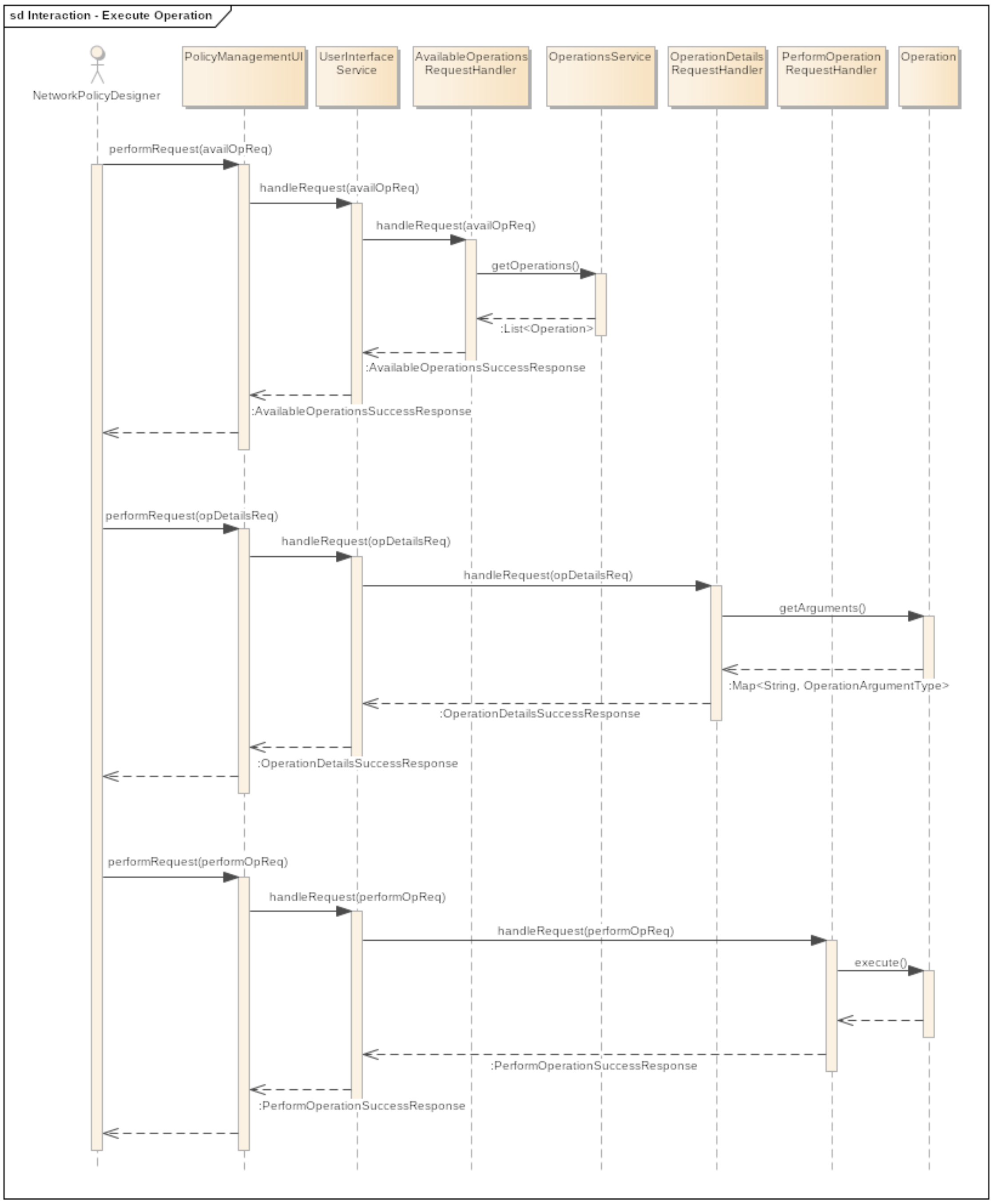

sd Interaction - Execute Operation
NetworkPolicyDesigner
PolicyManagementUI
UserInterface Service
AvailableOperations RequestHandler
OperationsService
OperationDetails RequestHandler
PerformOperation RequestHandler
Operation
performRequest(availOpReq)
handleRequest(availOpReq)
handleRequest(availOpReq)
getOperations()
:List<Operation>
:AvailableOperationsSuccessResponse
:AvailableOperationsSuccessResponse
performRequest(opDetailsReq)
handleRequest(opDetailsReq)
handleRequest(opDetailsReq)
getArguments()
:Map<String, OperationArgumentType>
:OperationDetailsSuccessResponse
:OperationDetailsSuccessResponse
performRequest(performOpReq)
handleRequest(performOpReq)
handleRequest(performOpReq)
execute()
:PerformOperationSuccessResponse
:PerformOperationSuccessResponse

Figure 14. logic architecture — Interaction — Execute Operation

8.8. Show-Policy-State Interaction.
In the interaction to show policies state, we've decided to stay as much generic as possible, since the solution details will be defined in later steps:

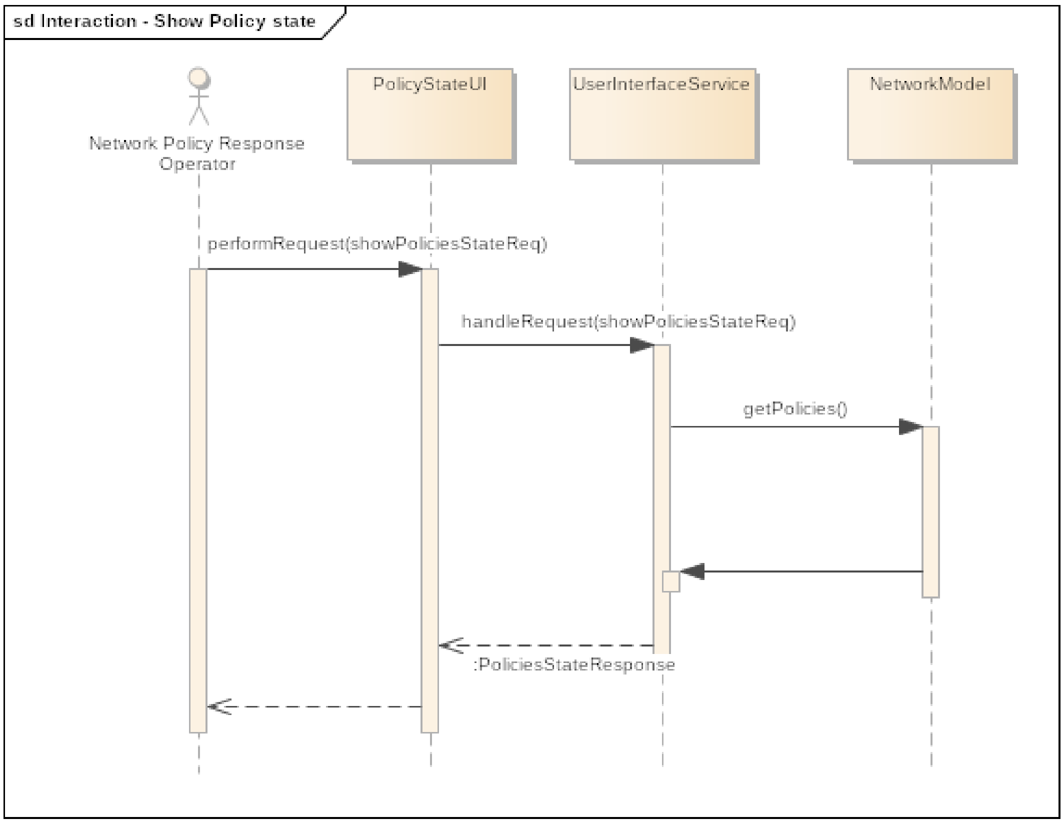

sd Interaction - Show Policy state
Network Policy Response Operator
PolicyStateUI
UserInterfaceService
NetworkModel
performRequest(showPoliciesStateReq)
handleRequest(showPoliciesStateReq)
getPolicies()
:PoliciesStateResponse

Figure 15. logic architecture — Interaction — Show Policy State

8.9. NETWORK MODEL Interaction.

When there are changes in the SDN, the NETWORK MODEL should be kept in sync with the changes. The following interaction diagram shows an example of synchronization, when a SDN element is added and the change should be propagated to the model (the other interactions, like remove and change, are very similar):

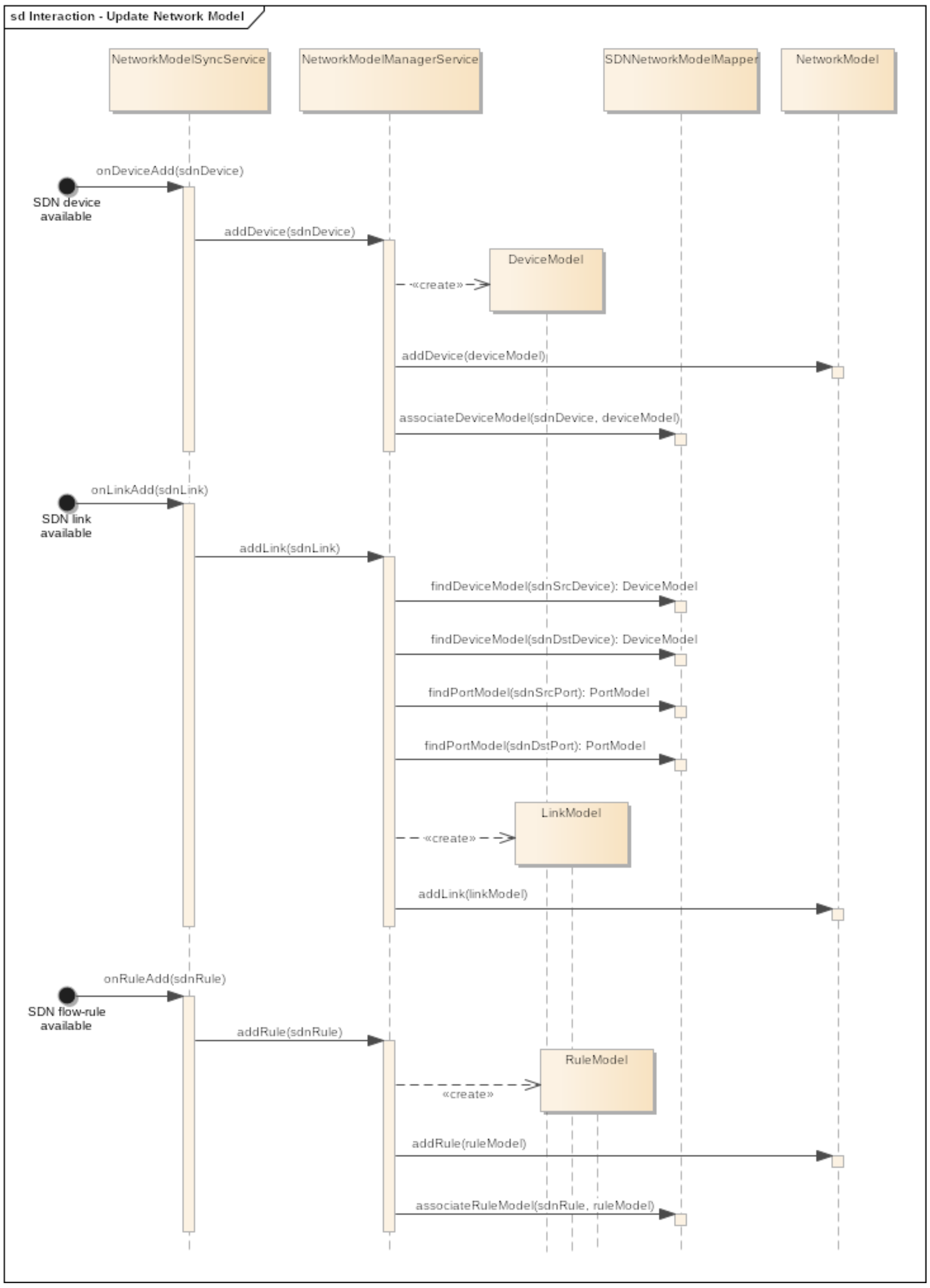


FIGURE 16. LOGIC ARCHITECTURE — Interaction — Sync SDN with NETWORK MODEL

8.10. Check-Reachability Interaction.
The following interaction example shows how a "Check Reachability" policy is created and added to the NETWORK MODEL:

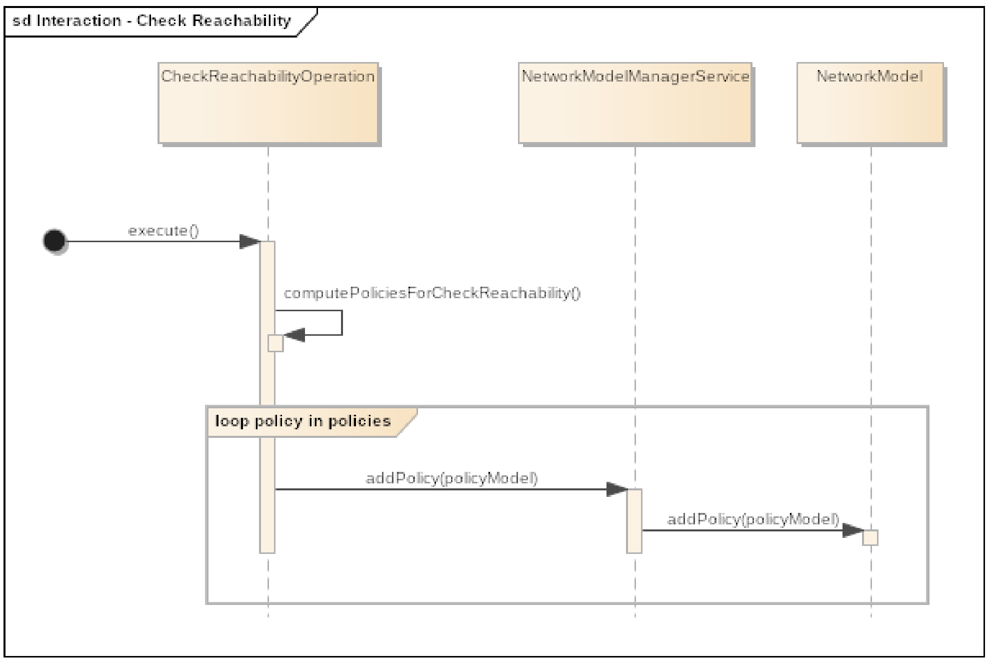

sd Interaction - Check Reachability
CheckReachabilityOperation
NetworkModelManagerService
NetworkModel
execute()
computePoliciesForCheckReachability()
loop policy in policies
addPolicy(policyModel)
addPolicy(policyModel)

Figure 17. LOGIC ARCHITECTURE — Interaction — Check Reachability

9. Abstraction Gap

The Southbound Interface of choice will be OpenFlow. This seems to be a reasonable choice, since it's the precursor of dynamic networks and it's the most widespread technology for `SBI`.

`ONOS` will be the SDN Controller because it's production-ready and provides all the needs for our proof-of-concept. Underlying network will be simulated using Mininet. It allows to work on different and custom network topologies.

NetPlumber has been shaped for `SDN`, thus it is particularly suited for implementing underlying model-checking mechanisms in our domain of interest.

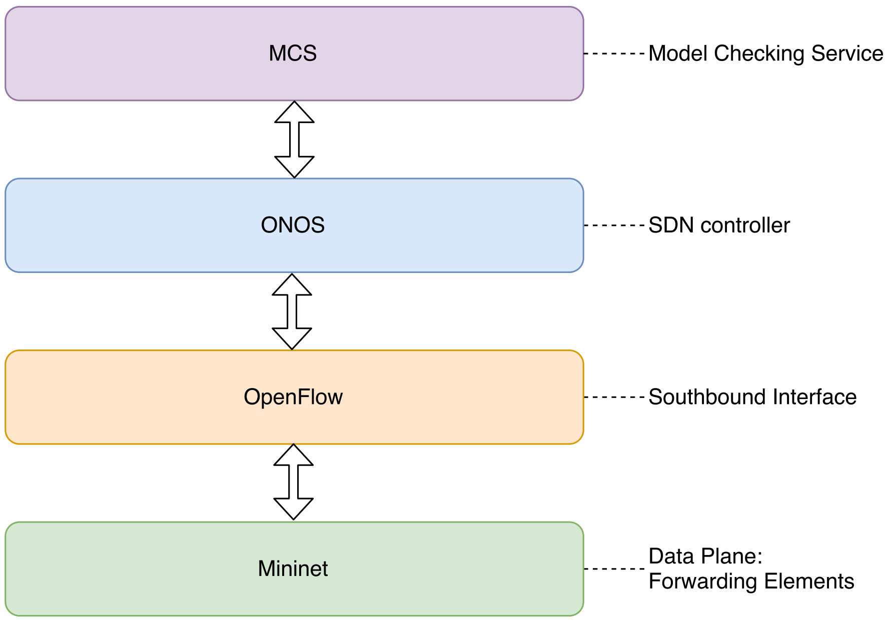


FIGURE 18. Reference Platform

Thus, from now on, we will focus on those technologies, while trying to be as much generic as possible.

10. Risk Analysis

The abstraction gap between the problem and the **Reference Platform** chosen above *is sustainable*.

11. PROJECT

In this step we introduce technology-specific solutions, in particular:

- `ONOS` will be used as `SDN` solution
- NETPLUMBER graph (called PLUMBING GRAPH) will be used to implement the NETWORK MODEL

11.1. Services Structure.

The PROJECT structure of the **Service** subsystem starts from the LOGIC ARCHITECTURE and applies the following changes:

- `UserInterfaceService` is abstract, the concrete implementation (called `ONOSUserInterfaceService`) is the one relying on `HTTP` communication protocol, used by `ONOS`
- Services don't work with abstract `SDN` and NETWORK MODEL data, but instead with `ONOS` network elements abstractions and PLUMBING GRAPH nodes representation, respectively

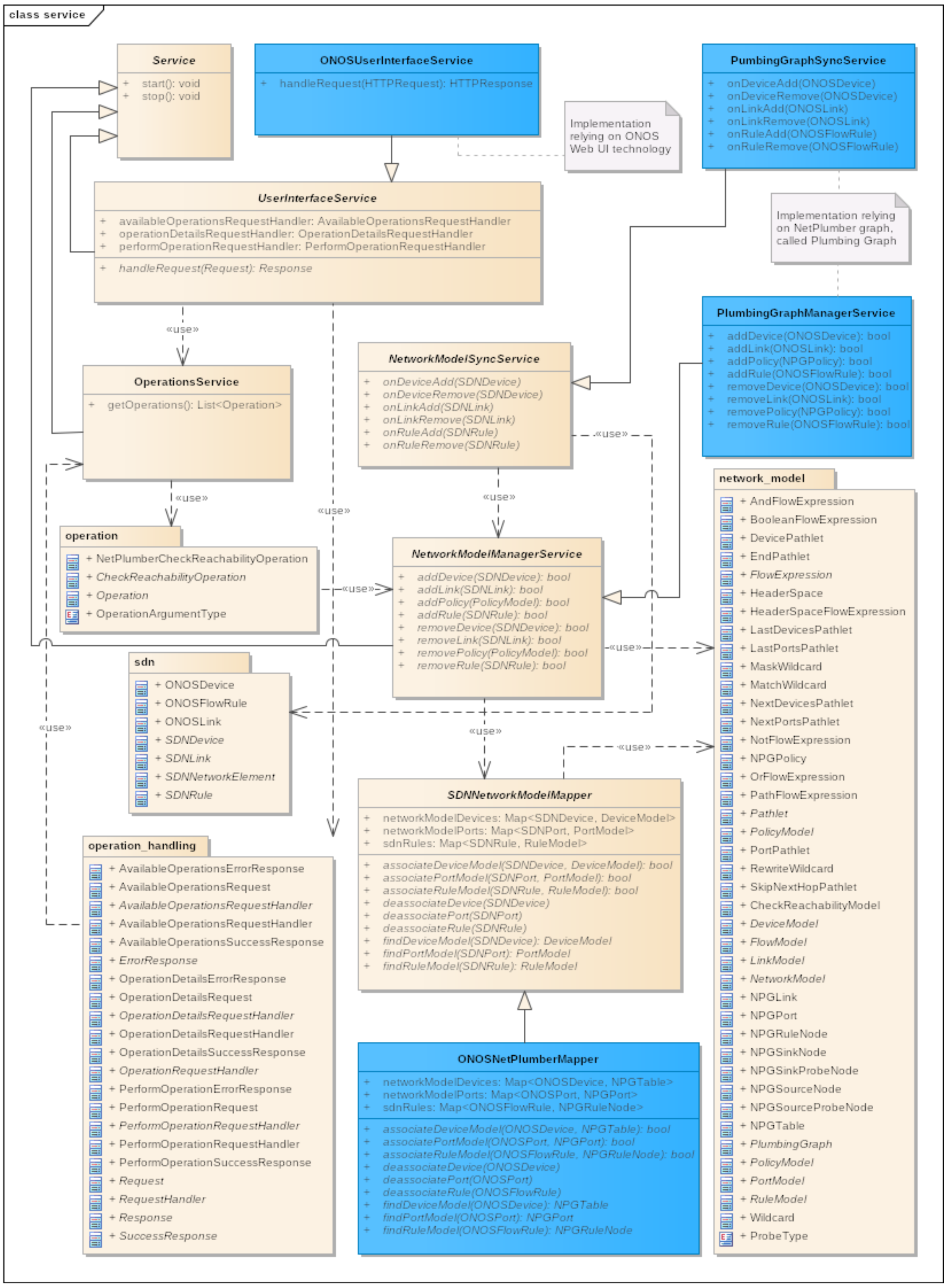

class service
Service
+ start(): void
+ stop(): void
ONOSUserInterfaceService
+ handleRequest(HTTPRequest): HTTPResponse
Implementation relying on ONOS Web UI technology
PumbingGraphSyncService
+ onDeviceAdd(ONOSDevice)
+ onDeviceRemove(ONOSDevice)
+ onLinkAdd(ONOSLink)
+ onLinkRemove(ONOSLink)
+ onRuleAdd(ONOSFlowRule)
+ onRuleRemove(ONOSFlowRule)
Implementation relying on NetPlumber graph, called Plumbing Graph
UserInterfaceService
+ availableOperationsRequestHandler: AvailableOperationsRequestHandler
+ operationDetailsRequestHandler: OperationDetailsRequestHandler
+ performOperationRequestHandler: PerformOperationRequestHandler
+ handleRequest(Request): Response
«use»
PlumbingGraphManagerService
+ addDevice(ONOSDevice): bool
+ addLink(ONOSLink): bool
+ addPolicy(NPGPolicy): bool
+ addRule(ONOSFlowRule): bool
+ removeDevice(ONOSDevice): bool
+ removeLink(ONOSLink): bool
+ removePolicy(NPGPolicy): bool
+ removeRule(ONOSFlowRule): bool
OperationsService
+ getOperations(): List<Operation>
NetworkModelSyncService
+ onDeviceAdd(SDNDevice)
+ onDeviceRemove(SDNDevice)
+ onLinkAdd(SDNLink)
+ onLinkRemove(SDNLink)
+ onRuleAdd(SDNRule)
+ onRuleRemove(SDNRule)
«use»
network_model
+ AndFlowExpression
+ BooleanFlowExpression
+ DevicePathlet
+ EndPathlet
+ FlowExpression
+ HeaderSpace
+ HeaderSpaceFlowExpression
+ LastDevicesPathlet
+ LastPortsPathlet
+ MaskWildcard
+ MatchWildcard
+ NextDevicesPathlet
+ NextPortsPathlet
+ NotFlowExpression
+ NPGPolicy
+ OrFlowExpression
+ PathFlowExpression
+ Pathlet
+ PolicyModel
+ PortPathlet
+ RewriteWildcard
+ SkipNextHopPathlet
+ CheckReachabilityModel
+ DeviceModel
+ FlowModel
+ LinkModel
+ NetworkModel
+ NPGLink
+ NPGPort
+ NPGRuleNode
+ NPGSinkNode
+ NPGSinkProbeNode
+ NPGSourceNode
+ NPGSourceProbeNode
+ NPGTable
+ PlumbingGraph
+ PolicyModel
+ PortModel
+ RuleModel
+ Wildcard
+ ProbeType
«use»
«use»
«use»
operation
+ NetPlumberCheckReachabilityOperation
+ CheckReachabilityOperation
+ Operation
+ OperationArgumentType
NetworkModelManagerService
+ addDevice(SDNDevice): bool
+ addLink(SDNLink): bool
+ addPolicy(PolicyModel): bool
+ addRule(SDNRule): bool
+ removeDevice(SDNDevice): bool
+ removeLink(SDNLink): bool
+ removePolicy(PolicyModel): bool
+ removeRule(SDNRule): bool
«use»
«use»
sdn
+ ONOSDevice
+ ONOSFlowRule
+ ONOSLink
+ SDNDevice
+ SDNLink
+ SDNNetworkElement
+ SDNRule
«use»
«use»
«use»
SDNNetworkModelMapper
+ networkModelDevices: Map<SDNDevice, DeviceModel>
+ networkModelPorts: Map<SDNPort, PortModel>
+ sdnRules: Map<SDNRule, RuleModel>
+ associateDeviceModel(SDNDevice, DeviceModel): bool
+ associatePortModel(SDNPort, PortModel): bool
+ associateRuleModel(SDNRule, RuleModel): bool
+ deassociateDevice(SDNDevice)
+ deassociatePort(SDNPort)
+ deassociateRule(SDNRule)
+ findDeviceModel(SDNDevice): DeviceModel
+ findPortModel(SDNPort): PortModel
+ findRuleModel(SDNRule): RuleModel
operation_handling
+ AvailableOperationsErrorResponse
+ AvailableOperationsRequest
+ AvailableOperationsRequestHandler
+ AvailableOperationsRequestHandler
+ AvailableOperationsSuccessResponse
+ ErrorResponse
+ OperationDetailsErrorResponse
+ OperationDetailsRequest
+ OperationDetailsRequestHandler
+ OperationDetailsRequestHandler
+ OperationDetailsSuccessResponse
+ OperationRequestHandler
+ PerformOperationErrorResponse
+ PerformOperationRequest
+ PerformOperationRequestHandler
+ PerformOperationRequestHandler
+ PerformOperationSuccessResponse
+ Request
+ RequestHandler
+ Response
+ SuccessResponse
ONOSNetPlumberMapper
+ networkModelDevices: Map<ONOSDevice, NPGTable>
+ networkModelPorts: Map<ONOSPort, NPGPort>
+ sdnRules: Map<ONOSFlowRule, NPGRuleNode>
+ associateDeviceModel(ONOSDevice, NPGTable): bool
+ associatePortModel(ONOSPort, NPGPort): bool
+ associateRuleModel(ONOSFlowRule, NPGRuleNode): bool
+ deassociateDevice(ONOSDevice)
+ deassociatePort(ONOSPort)
+ deassociateRule(ONOSFlowRule)
+ findDeviceModel(ONOSDevice): NPGTable
+ findPortModel(ONOSPort): NPGPort
+ findRuleModel(ONOSFlowRule): NPGRuleNode

Figure 19. project — Structure — Services

11.2. Operation Handling Structure.

It's generic, since it doesn't depend on particular technologies chosen.
The structure is the same as defined in LOGIC ARCHITECTURE.

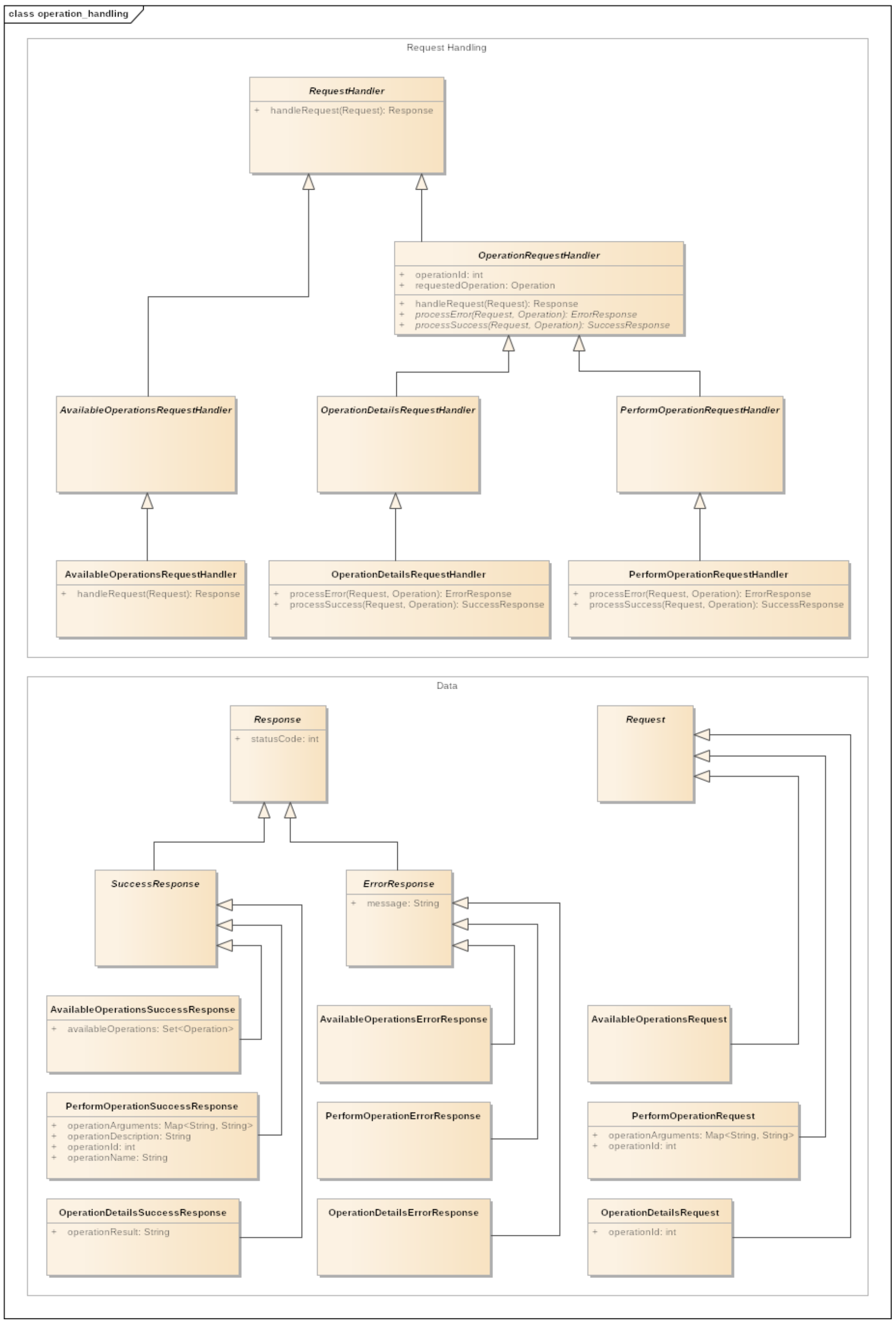

class operation_handling
Request Handling
RequestHandler
+ handleRequest(Request): Response
OperationRequestHandler
+ operationId: int
+ requestedOperation: Operation
+ handleRequest(Request): Response
+ processError(Request, Operation): ErrorResponse
+ processSuccess(Request, Operation): SuccessResponse
AvailableOperationsRequestHandler
OperationDetailsRequestHandler
PerformOperationRequestHandler
AvailableOperationsRequestHandler
+ handleRequest(Request): Response
OperationDetailsRequestHandler
+ processError(Request, Operation): ErrorResponse
+ processSuccess(Request, Operation): SuccessResponse
PerformOperationRequestHandler
+ processError(Request, Operation): ErrorResponse
+ processSuccess(Request, Operation): SuccessResponse
Data
Response
+ statusCode: int
Request
SuccessResponse
ErrorResponse
+ message: String
AvailableOperationsSuccessResponse
+ availableOperations: Set<Operation>
AvailableOperationsErrorResponse
AvailableOperationsRequest
PerformOperationSuccessResponse
+ operationArguments: Map<String, String>
+ operationDescription: String
+ operationId: int
+ operationName: String
PerformOperationErrorResponse
PerformOperationRequest
+ operationArguments: Map<String, String>
+ operationId: int
OperationDetailsSuccessResponse
+ operationResult: String
OperationDetailsErrorResponse
OperationDetailsRequest
+ operationId: int

FIGURE 20. PROJECT — Structure — Operation Handling

11.3. Operation Data Types.

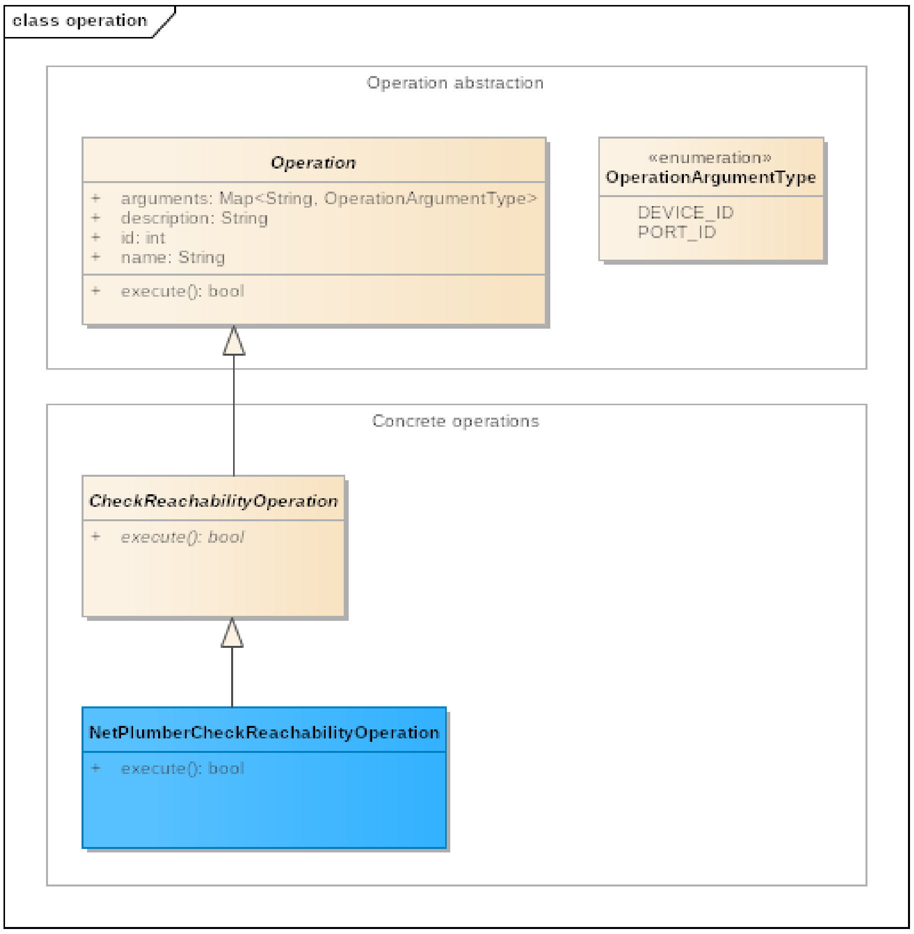


FIGURE 21. PROJECT — Structure — Operation

11.4. NETWORK MODEL Structure.

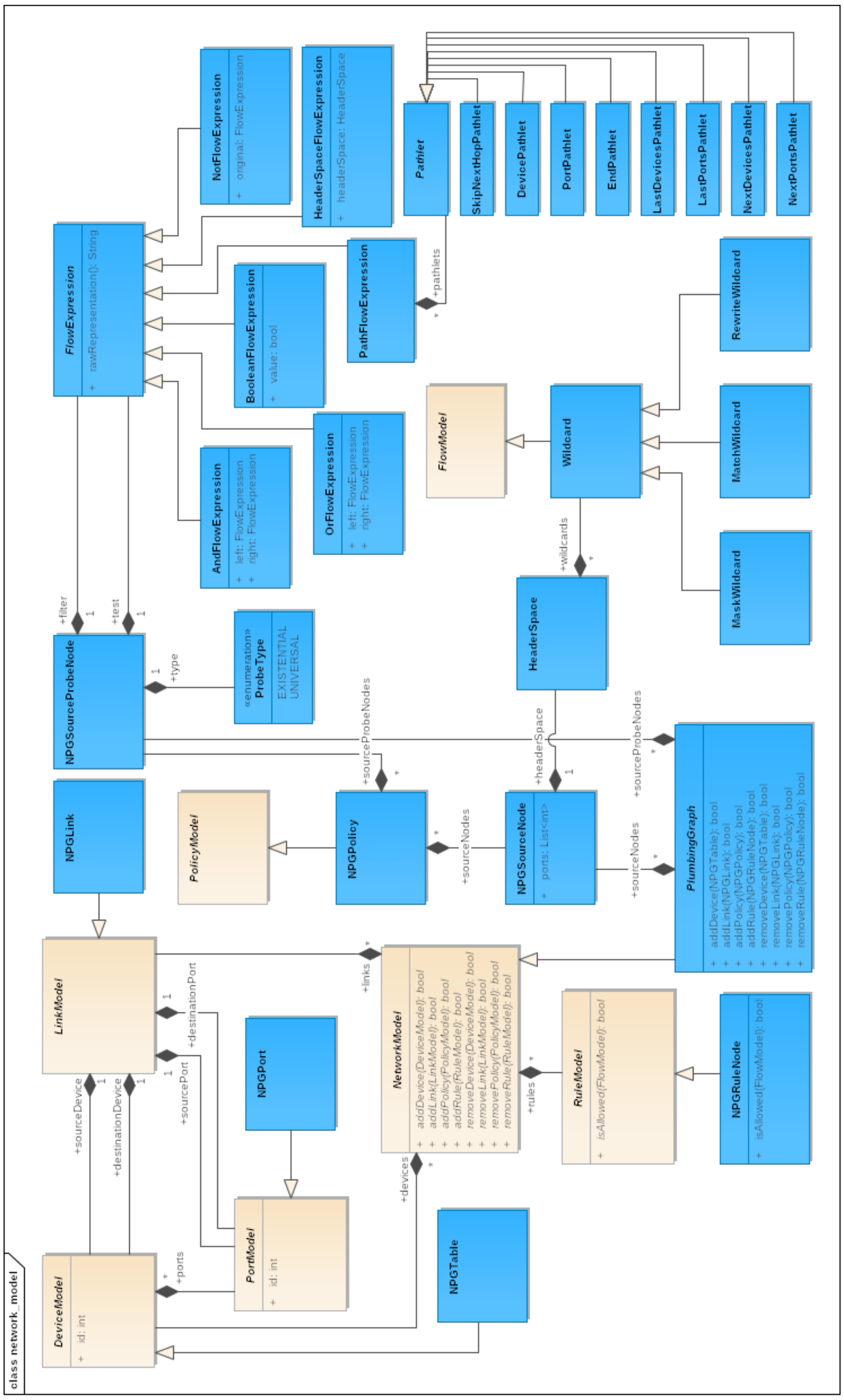


FIGURE 22. PROJECT — Structure — NETWORK MODEL Structure

11.5. `SDN` **Data Types.**

`SDN` data types are realized with `ONOS` respective data types.

In fact `ONOS` provides a subsystem called **Device Subsystem**, that is responsible for discovering and tracking the devices that comprise the network, and for enabling operators and applications to control them.

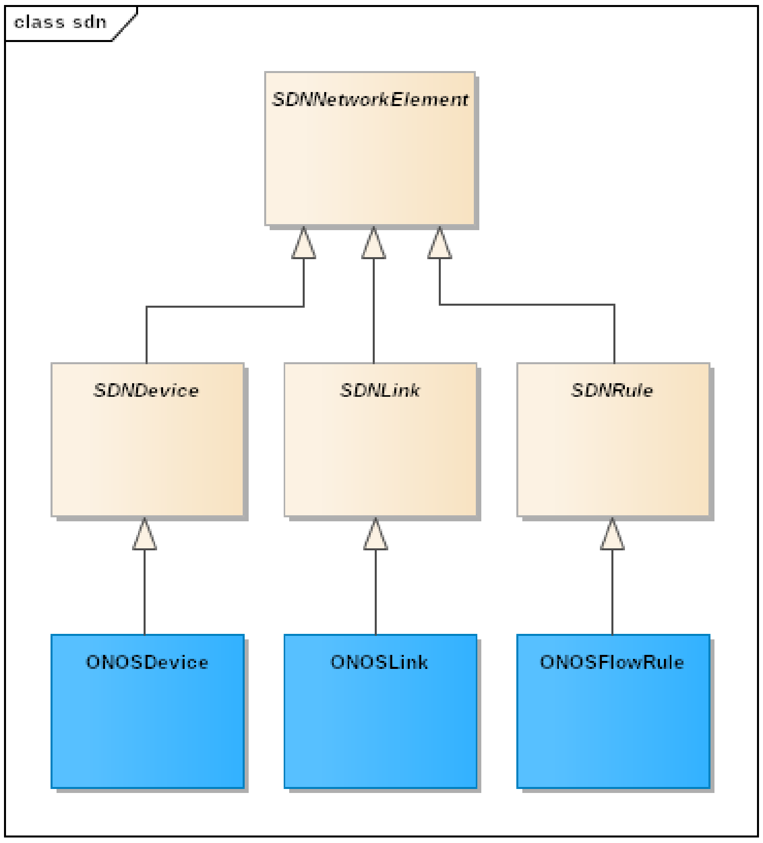


Figure 23. project — Structure — `SDN` Data Types

`ONOS` will then map the defined data types with the underlying network, to keep the state consistent. In our case with OpenFlow data types:

DeviceManager	**OpenFlowDeviceProvider**
`Device`	`OpenFlowSwitch`
`DeviceId / ElementId`	`Dpid`
`Port`	`OFPortDesc`

11.6. UI Interaction.

The chosen solution for User-Interface is the ONOS Web UI. It relies on modern technologies like Angular 2 to reduce the abstraction gap between the client technology and modern UI needs, in term of changes reaction and client structure.

Interactions between client and server follow the pattern **request / response**, thus they'll use the HTTP protocol, designed to handle that interaction pattern well.

UI updates need to be reactive, thus we will use Web Sockets as underlying communication technology.

11.7. Check-Reachability Interaction.

Considering the Check Reachability case-study, the main interactions are:

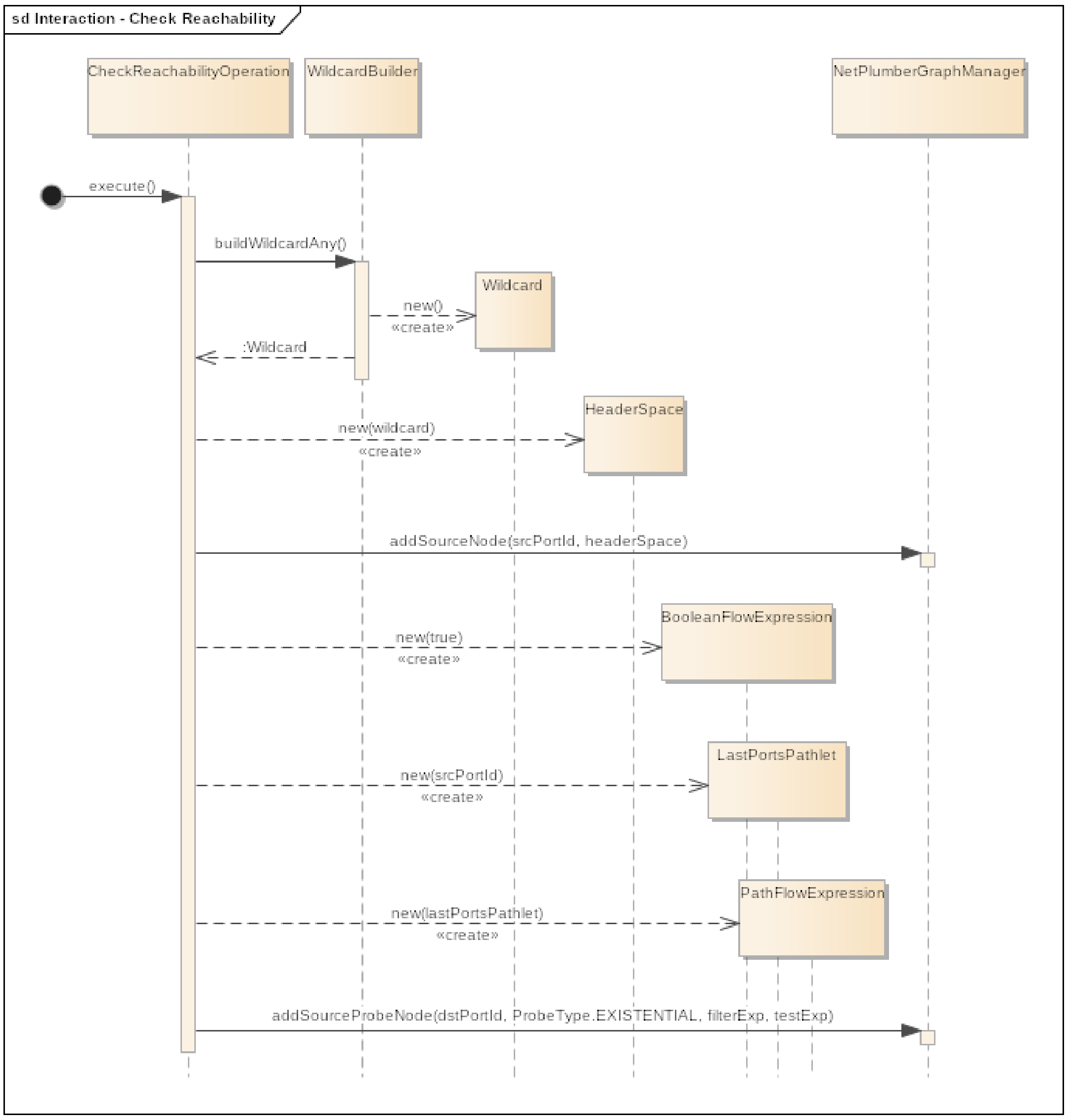


FIGURE 24. PROJECT — Interaction — Check Reachability

12. Implementation

Considering previous artifacts defined in the `MCS` PROJECT, we've successfully implemented the **Model Check Service**.

To get a copy of the source code and relative documentation, ask the author at `molari dot alessandro at gmail dot com`.

13. Tests

Core parts of `MCS` have been tested using jUnit testing framework.
Tests are organized in:

- **60 integration tests**: to test *interaction* between `MCS` and the underlying NetPlumber service
- **18 unit tests**: to test single *isolated functionality* of critical parts

For more details on tests structure and implementation, you can take a look at: `https://github.com/alem0lars/mcs/tree/master/src/test/java/me/alem0lars/mcs`.

14. Demo

A virtual machine called `MCS VM` has been created. It features:

- A working instance of `ONOS`
- Networking based on Mininet
- Customized utility scripts to interact with `ONOS` and perform deployment of `MCS`

To get a copy, ask to `molari dot alessandro at gmail dot com`.
To uncompress it run: `unzip mcs-vm.zip`
Now you can import the `OVA` file in VirtualBox.
The following ports need to be available:

- `22` to connect to `VM` via `SSH`
- `8181` to access the Web UI

So, if network interface is a NAT, you need to configure port forwarding accordingly.
You also need to get a copy of `MCS` at: `https://github.com/alem0lars/mcs`.
Build the project to generate the application package.
Now you can deploy the application into the `ONOS` instance available in the `MCS VM`.
Here are some useful commands:

- Build `ONOS`:
  ```
  cd $ONOS_ROOT && ./tools/build/onos-buck build onos --show-output
  ```
- Build `ONOS` internal test, to check the installation works correctly:
  ```
  cd $ONOS_ROOT && ./tools/build/onos-buck test
  ```
- Start `ONOS`:
  ```
  cd $ONOS_ROOT && ./tools/build/onos-buck run onos-local -- clean debug
  ```
- Attach to the Apache Karaf console of `ONOS`:
  ```
  cd $ONOS_ROOT && ./tools/test/bin/onos localhost
  ```
- Start mininet:
  ```
  sudo mn --controller=remote --topo=linear,10
  ```
  Where:
  - `-controller=remote` means that mininet OpenFlow nodes should use be connected to an external controller (with default port)
  - `-topo=TOPOLOGY` allows to choose an existing topology with custom parameters

14.1. Results. The `MCS` reachability check has been run in a *linear topology* with four switches and four hosts: The kind and size of topology chosen don't affect the test, so we've decided to keep it as much simple as possible:

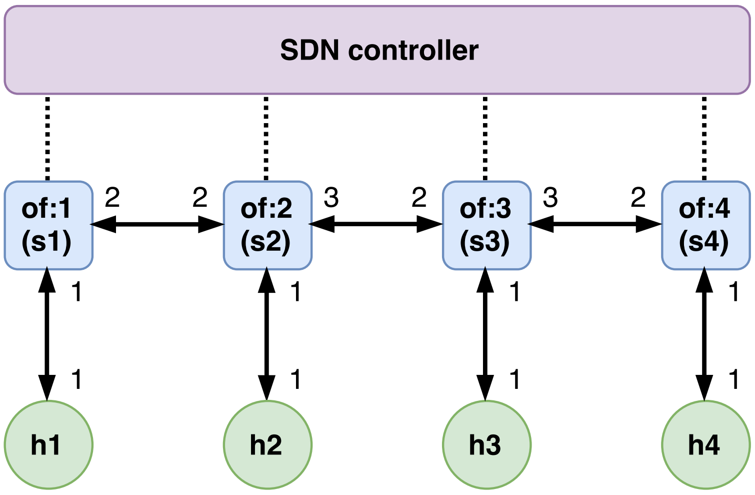
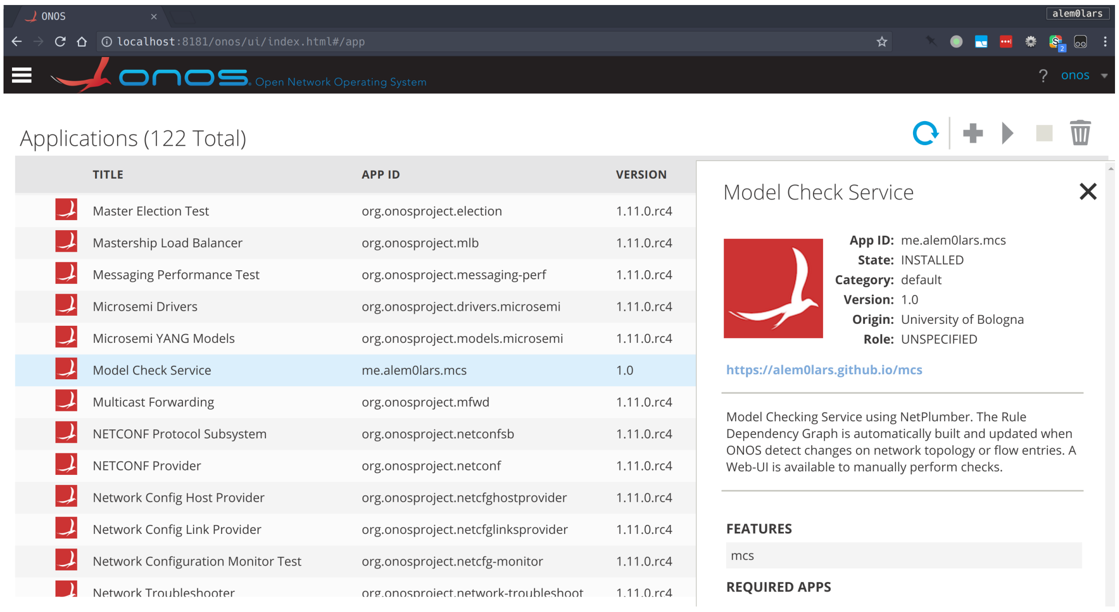


FIGURE 25. SDN DATA PLANE topology

First of all, we need to install the `MCS` application:


ONOS
alem0lars
localhost:8181/onos/ui/index.html#/app
onos Open Network Operating System
onos

Applications (122 Total)

TITLE	APP ID	VERSION
Master Election Test	org.onosproject.election	1.11.0.rc4
Mastership Load Balancer	org.onosproject.mlb	1.11.0.rc4
Messaging Performance Test	org.onosproject.messaging-perf	1.11.0.rc4
Microsemi Drivers	org.onosproject.drivers.microsemi	1.11.0.rc4
Microsemi YANG Models	org.onosproject.models.microsemi	1.11.0.rc4
Model Check Service	me.alem0lars.mcs	1.0
Multicast Forwarding	org.onosproject.mfwd	1.11.0.rc4
NETCONF Protocol Subsystem	org.onosproject.netconfsb	1.11.0.rc4
NETCONF Provider	org.onosproject.netconf	1.11.0.rc4
Network Config Host Provider	org.onosproject.netcfghostprovider	1.11.0.rc4
Network Config Link Provider	org.onosproject.netcfglinksprovider	1.11.0.rc4
Network Configuration Monitor Test	org.onosproject.netcfg-monitor	1.11.0.rc4
Network Troubleshooter	org.onosproject.network-troubleshoot	1.11.0.rc4

Model Check Service

App ID:	me.alem0lars.mcs
State:	INSTALLED
Category:	default
Version:	1.0
Origin:	University of Bologna
Role:	UNSPECIFIED

https://alem0lars.github.io/mcs

Model Checking Service using NetPlumber. The Rule Dependency Graph is automatically built and updated when ONOS detect changes on network topology or flow entries. A Web-UI is available to manually perform checks.

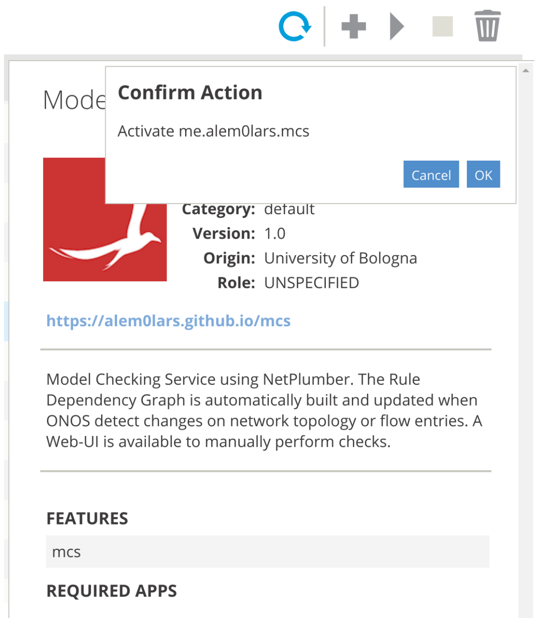
FEATURES

mcs

REQUIRED APPS

FIGURE 26. Install MCS application

Now the `MCS` bundle has been installed and can be activated:


Confirm Action

Activate me.alem0lars.mcs

Category: default
Version: 1.0
Origin: University of Bologna
Role: UNSPECIFIED

https://alem0lars.github.io/mcs

Model Checking Service using NetPlumber. The Rule Dependency Graph is automatically built and updated when ONOS detect changes on network topology or flow entries. A Web-UI is available to manually perform checks.

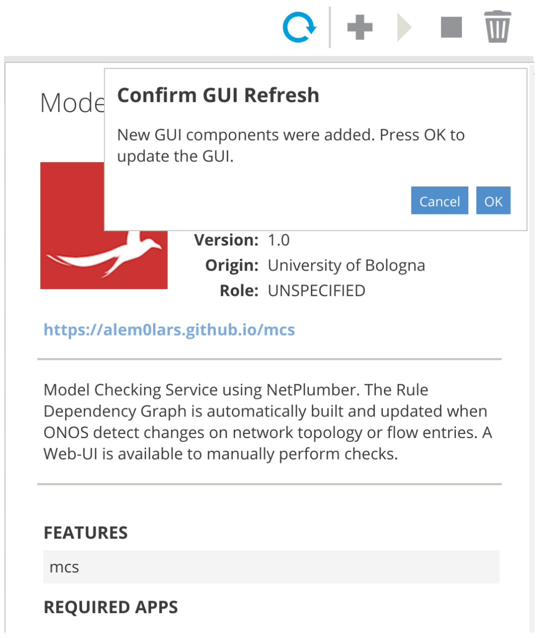
FEATURES

mcs

REQUIRED APPS

Figure 27. Activate MCS application

ONOS recognizes that MCS defines a custom Web UI, so it asks to enable it:


Confirm GUI Refresh

New GUI components were added. Press OK to update the GUI.

Version: 1.0
Origin: University of Bologna
Role: UNSPECIFIED

https://alem0lars.github.io/mcs

Model Checking Service using NetPlumber. The Rule Dependency Graph is automatically built and updated when ONOS detect changes on network topology or flow entries. A Web-UI is available to manually perform checks.

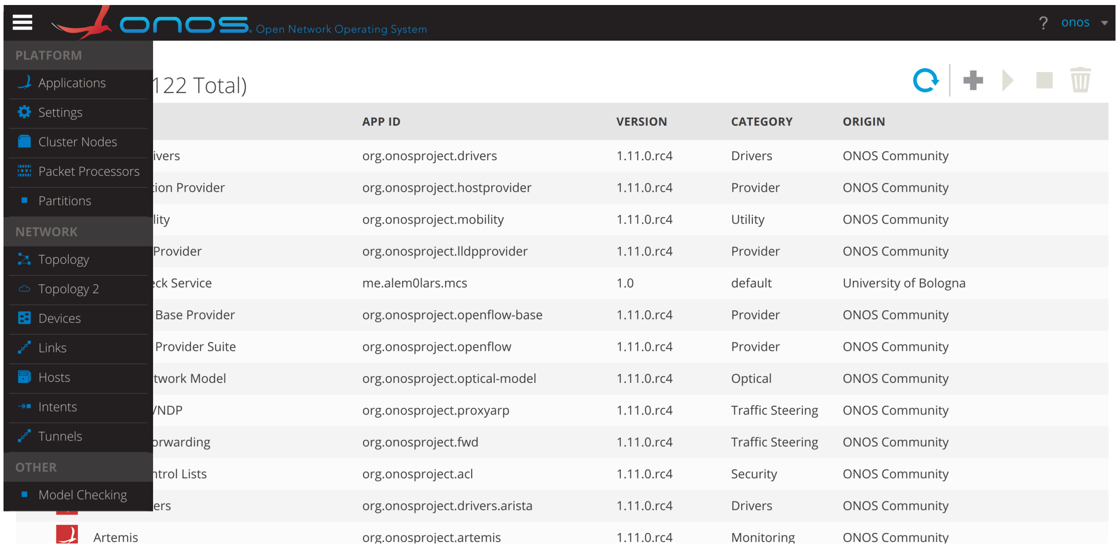
FEATURES

mcs

REQUIRED APPS

Figure 28. Enable MCS Web UI

The MCS UI is available in the **Model Checking** tab:

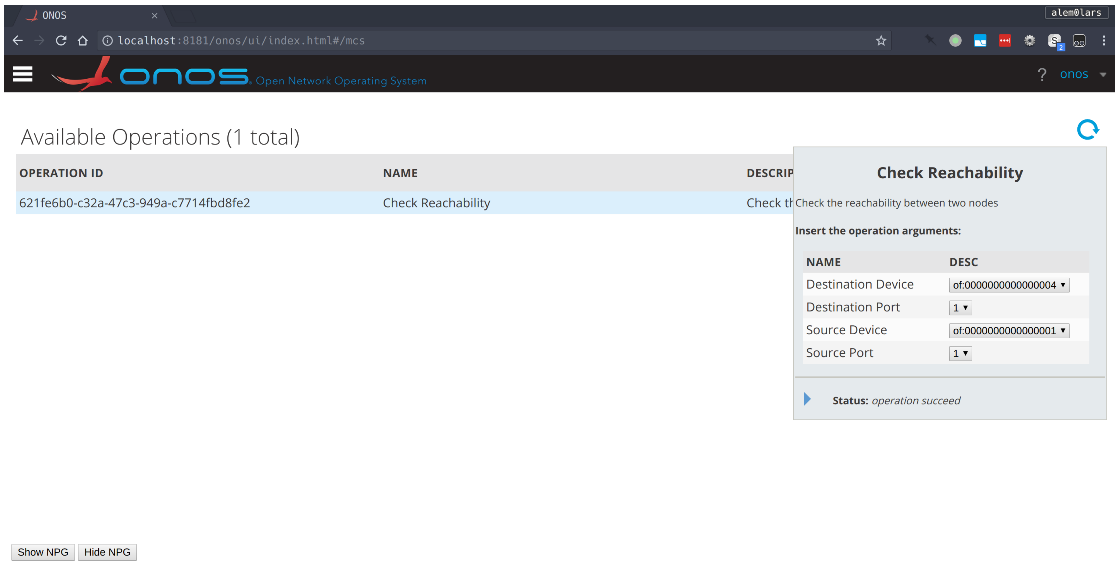


122 Total)

	APP ID	VERSION	CATEGORY	ORIGIN
ivers	org.onosproject.drivers	1.11.0.rc4	Drivers	ONOS Community
ion Provider	org.onosproject.hostprovider	1.11.0.rc4	Provider	ONOS Community
lity	org.onosproject.mobility	1.11.0.rc4	Utility	ONOS Community
Provider	org.onosproject.lldpprovider	1.11.0.rc4	Provider	ONOS Community
ck Service	me.alem0lars.mcs	1.0	default	University of Bologna
Base Provider	org.onosproject.openflow-base	1.11.0.rc4	Provider	ONOS Community
Provider Suite	org.onosproject.openflow	1.11.0.rc4	Provider	ONOS Community
twork Model	org.onosproject.optical-model	1.11.0.rc4	Optical	ONOS Community
/NDP	org.onosproject.proxyarp	1.11.0.rc4	Traffic Steering	ONOS Community
orwarding	org.onosproject.fwd	1.11.0.rc4	Traffic Steering	ONOS Community
ntrol Lists	org.onosproject.acl	1.11.0.rc4	Security	ONOS Community
ers	org.onosproject.drivers.arista	1.11.0.rc4	Drivers	ONOS Community
Artemis	org.onosproject.artemis	1.11.0.rc4	Monitoring	ONOS Community

Figure 29. MCS UI "Model Checking" tab

The UI is automatically mapped with the predefined operations that `MCS` allows. In fact, when opened, it shows the available operations, in this case **Check Reachability**:


ONOS
localhost:8181/onos/ui/index.html#/mcs
alem0lars
onos
Open Network Operating System
? onos
Show NPG
Hide NPG

Available Operations (1 total)

OPERATION ID	NAME	DESCRIP
621fe6b0-c32a-47c3-949a-c7714fbd8fe2	Check Reachability	Check th

Check Reachability

Check the reachability between two nodes

Insert the operation arguments:

NAME	DESC
Destination Device	of:0000000000000004
Destination Port	1
Source Device	of:0000000000000001
Source Port	1

Status: *operation succeed*

Figure 30. Model Check operations UI

If we select that operations, a panel is prompted to insert the *operation parameters*.

In the example we've started `MCS` with devices and links already added, but without any flows installed.

This doesn't affect solution generality, and also shows how NetPlumber and `MCS` *can easily adapt when underlying network changes.*

The following logs show the startup process, that involves external dependencies to be started just before the start of `MCS` components:

```
2017-10-31 14:47:22,846 |
    Starting NetPlumber server ...
    bound to ‘0.0.0.0:9000‘
    with wildcard size of‘224‘
    using log configuration at ...

2017-10-31 14:47:23,300 | Started OSGi component

...
```

After components have been started and activated, the `MCS` service called `NetPlumberSyncManager` will perform initial synchronization of `SDN` devices and links with the Plumbing Graph:

```
2017-10-31 14:47:23,303 |
    Performing initial NPG synchronization

2017-10-31 14:47:23,310 |
    Synchronizing NPG: ‘of:0000000000000003‘ with ports=‘[1, 2, 3]‘ added
                       ==> adding ‘Table{id=1, ports=[1, 2, 3]}‘ to NPG
2017-10-31 14:47:23,316 |
    Synchronizing NPG: ‘of:0000000000000004‘ with ports=‘[1, 2]‘ added
                       ==> adding ‘Table{id=2,ports=[4, 5]}‘ to NPG
2017-10-31 14:47:23,321 |
    Synchronizing NPG: ‘of:0000000000000001‘ with ports=‘[1, 2]‘ added
                       ==> adding ‘Table{id=3,ports=[6, 7]}‘ to NPG
2017-10-31 14:47:23,326 |
    Synchronizing NPG: ‘of:0000000000000002‘ with ports=‘[1, 2, 3]‘ added
                       ==> adding ‘Table{id=4, ports=[8, 9, 10]}‘ to NPG

2017-10-31 14:47:23,331 |
    Synchronizing NPG:
      Link{src=of:0000000000000003/3,
                  dst=of:0000000000000004/2,
                  type=DIRECT,
                  state=ACTIVE,
                  expected=false} added
      ==> adding ‘Link{src_port=3 <-> dst_port=5}‘ to NPG

2017-10-31 14:47:23,339 |
    Synchronizing NPG:
      Link{src=of:0000000000000002/3,
                  dst=of:0000000000000003/2,
                  type=DIRECT,
                  state=ACTIVE,
                  expected=false} added
      ==> adding ‘Link{src_port=10 <-> dst_port=2}‘ to NPG

2017-10-31 14:47:23,342 |
    Synchronizing NPG:
      Link{src=of:0000000000000002/2,
                  dst=of:0000000000000001/2,
                  type=DIRECT,
                  state=ACTIVE,
                  expected=false} added
      ==> adding ‘Link{src_port=9 <-> dst_port=7}‘ to NPG

2017-10-31 14:47:23,344 |
    Synchronizing NPG:
      Link{src=of:0000000000000004/2,
                  dst=of:0000000000000003/3,
                  type=DIRECT,
                  state=ACTIVE,
                  expected=false} added
      ==> adding ‘Link{src_port=5 <-> dst_port=3}‘ to NPG

2017-10-31 14:47:23,347 |
    Synchronizing NPG:
      Link{src=of:0000000000000003/2,
                  dst=of:0000000000000002/3,
                  type=DIRECT,
                  state=ACTIVE,
                  expected=false} added
      ==> adding ‘Link{src_port=2 <-> dst_port=10}‘ to NPG

2017-10-31 14:47:23,350 |
    Synchronizing NPG:
      Link{src=of:0000000000000001/2,
                  dst=of:0000000000000002/2,
                  type=DIRECT,
                  state=ACTIVE,
                  expected=false} added
      ==> adding ‘Link{src_port=7 <-> dst_port=9}‘ to NPG
```

After initial synchronization, the resulting state is the following:

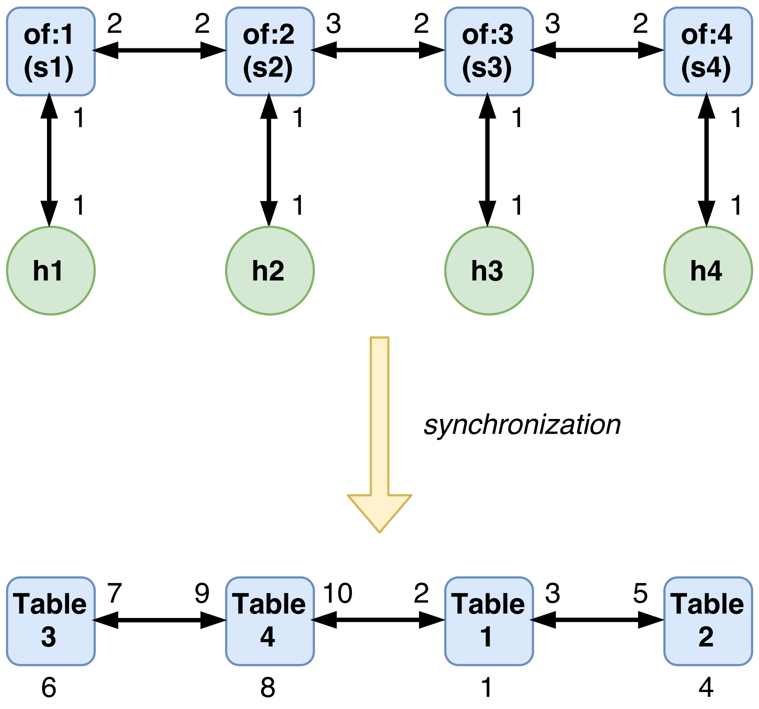


Figure 31. Plumbing Graph synchronized with SDN topology

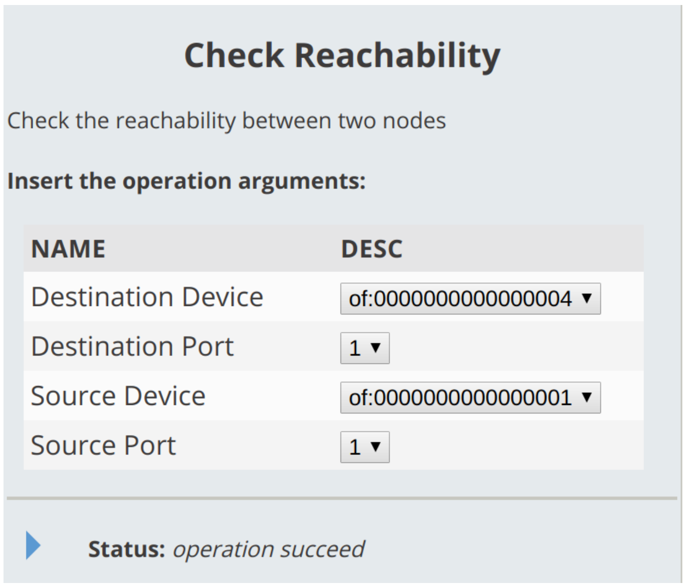
The following logs show the client asked to MCS what are the available operations that can be performed, and the server responded with "Check Reachability" operation:

```
2017-10-31 14:47:29,137 |
    Processing request: operationDetailsRequest

2017-10-31 14:47:29,138 |
    Providing operation response for
      Optional[CheckReachabilityOperation{
        fields={name=Check Reachability,
                description=Check the reachability between two nodes,
                id=621fe6b0-c32a-47c3-949a-c7714fbd8fe2}}]

2017-10-31 14:47:29,196 |
    Sending response: operationDetailsResponse
```

Let's try to submit a new reachability check between devices `of:1` at port `1` and `of:4` at port `4`, so basically we're checking if *host 1 can communicate with host 4*:



Check Reachability

Check the reachability between two nodes

Insert the operation arguments:

NAME	DESC
Destination Device	of:0000000000000004 ▾
Destination Port	1 ▾
Source Device	of:0000000000000001 ▾
Source Port	1 ▾

Status: *operation succeed*

Figure 32. Check Reachability operation parameters

In the UI the user can insert the ONOS devices and ports that should be checked whether they're reachable or not.

In particular the user reasons in terms of SDN concepts and not with NetPlumber concept. This is the result of MCS abstractions layers, defined to hide internal interaction details to the user, so if the internal model-checking framework changes, the user experience isn't affected at all.

When the user presses the arrow button, MCS adds needed source nodes and source probe nodes needed to perform reachability check and the underlying NetPlumber daemon updates the probes state every time Plumbing Graph changes:

```
2017-10-31 14:48:08,581 |
    Linking Source Node to NPG: Link{src_port=11 <-> dst_port=6}

2017-10-31 14:48:08,590 |
    Synchronizing NPG:
      Adding SourceNode{
        ports=[11],
        HeaderSpace{
          wildcards=[MatchWildcard{bits=xxxxxxxx..., size=224}],
          diff=[[]]}}

2017-10-31 14:48:08,603 |
    Linking Source Probe Node to NPG: Link{src_port=12 <-> dst_port=4}

2017-10-31 14:48:08,606 |
    Synchronizing NPG:
      Adding SourceProbeNode{
        ports=[12],
        probe_type=ProbeType{label=existential},
        filter=BooleanFlowExpression{value=true},
        test=PathFlowExpression{
          pathlets=[LastPortsPathlet{
            ports=[PortId{device=of:0000000000000001,
                                        port=1}]}]}}‘

2017-10-31 14:48:08,628 |Sending response: performOperationResponse
```

Right after adding SOURCE PROBE NODE for reachability check, we can see the condition isn't met, since there aren't any flow-rules and thus RULE NODES:

```
2017-10-31 14:48:08,624
    Existential Probe 2 Activated after event Start Source Probe:
       Started in False State
```

Now let's add flow-rules that allows devices to communicate through the established links:

```
2017-10-31 14:49:02,958 |
    Synchronizing NPG: FlowRule{
      id=6f0000302f74d4,
      deviceId=of:0000000000000004,
      priority=10,
      selector=[IN_PORT:2,
                ETH_DST:36:A7:89:EB:C0:E2,
                ETH_SRC:72:3A:B9:F8:00:AB],
      treatment=TrafficTreatment{immediate=[OUTPUT:1],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509457742870,
      payLoad=null} added
    ==> adding RuleNode{
      table_id=2,
      index=-1,
      in_ports=[4, 5],
      out_ports=[4],
      match=01001110,01011100,10011101,00011111,00000000,
            11010101,01101100,11100101,10010001,11010111,
            00000011,01000111,xxxxxxxx,xxxxxxxx,...} to NPG

2017-10-31 14:49:02,976 |
    Synchronizing NPG: FlowRule{
      id=6f000004381f02,
      deviceId=of:0000000000000001,
      priority=10,
      selector=[IN_PORT:1,
                ETH_DST:36:A7:89:EB:C0:E2,
                ETH_SRC:72:3A:B9:F8:00:AB],
      treatment=TrafficTreatment{immediate=[OUTPUT:2],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509457742861,
      payLoad=null} added
    ==> adding RuleNode{
      table_id=3,
      index=-1,
      in_ports=[6, 7],
      out_ports=[7],
      match=01001110,01011100,10011101,00011111,00000000,
            11010101,01101100,11100101,10010001,11010111,
            00000011,01000111,xxxxxxxx,...} to NPG

2017-10-31 14:49:02,979 |
    Synchronizing NPG: FlowRule{
      id=6f0000eaafcfe5,
      deviceId=of:0000000000000004,
      priority=10,
      selector=[IN_PORT:1,
                ETH_DST:72:3A:B9:F8:00:AB,
                ETH_SRC:36:A7:89:EB:C0:E2],
      treatment=TrafficTreatment{immediate=[OUTPUT:2],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509457742877,
      payLoad=null} added
    ==> adding RuleNode{
      table_id=2,
      index=-1,
      in_ports=[4, 5],
      out_ports=[5],
      match=01101100,11100101,10010001,11010111,00000011,
            01000111,01001110,01011100,10011101,00011111,
            00000000,11010101,xxxxxxxx,...} to NPG

2017-10-31 14:49:02,989 |
    Synchronizing NPG: FlowRule{
      id=6f0000a344c5c7,
      deviceId=of:0000000000000002,
      priority=10,
      selector=[IN_PORT:2,
                ETH_DST:36:A7:89:EB:C0:E2,
                ETH_SRC:72:3A:B9:F8:00:AB],
      treatment=TrafficTreatment{immediate=[OUTPUT:3],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509457742865,
      payLoad=null} added
    ==> adding RuleNode{
      table_id=4,
      index=-1,
      in_ports=[8, 9, 10],
      out_ports=[10],
      match=01001110,01011100,10011101,00011111,00000000,
            11010101,01101100,11100101,10010001,11010111,
            00000011,01000111,xxxxxxxx,...} to NPG

2017-10-31 14:49:02,993 |
    Synchronizing NPG: FlowRule{
      id=6f0000aef00501,
      deviceId=of:0000000000000001,
      priority=10,
      selector=[IN_PORT:2,
                ETH_DST:72:3A:B9:F8:00:AB,
                ETH_SRC:36:A7:89:EB:C0:E2],
      treatment=TrafficTreatment{immediate=[OUTPUT:1],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509457742881,
      payLoad=null} added
    ==> adding RuleNode{
      table_id=3,
      index=-1,
      in_ports=[6, 7],
      out_ports=[6],
      match=01101100,11100101,10010001,11010111,00000011,
            01000111,01001110,01011100,10011101,00011111,
            00000000,11010101,xxxxxxxx,...} to NPG

2017-10-31 14:49:02,997 |
    Synchronizing NPG: FlowRule{
      id=6f00001713d4f1,
      deviceId=of:0000000000000003,
      priority=10,
      selector=[IN_PORT:2,
                ETH_DST:36:A7:89:EB:C0:E2,
                ETH_SRC:72:3A:B9:F8:00:AB],
      treatment=TrafficTreatment{immediate=[OUTPUT:3],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509457742868,
      payLoad=null} added
    ==> adding RuleNode{
      table_id=1,
      index=-1,
      in_ports=[1, 2, 3],
      out_ports=[3],
      match=01001110,01011100,10011101,00011111,00000000,
            11010101,01101100,11100101,10010001,11010111,
            00000011,01000111,xxxxxxxx,...} to NPG

2017-10-31 14:49:03,004 |
    Synchronizing NPG: FlowRule{
      id=6f000020290603,
      deviceId=of:0000000000000003,
      priority=10,
      selector=[IN_PORT:3,
                ETH_DST:72:3A:B9:F8:00:AB,
                ETH_SRC:36:A7:89:EB:C0:E2],
      treatment=TrafficTreatment{immediate=[OUTPUT:2],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509457742878,
      payLoad=null} added
    ==> adding RuleNode{
      table_id=1,
      index=-1,
      in_ports=[1, 2, 3],
      out_ports=[2],
      match=01101100,11100101,10010001,11010111,00000011,
            01000111,01001110,01011100,10011101,00011111,
            00000000,11010101,xxxxxxxx,...} to NPG

2017-10-31 14:49:03,008 |
    Synchronizing NPG: FlowRule{
      id=6f000091eb3b33,
      deviceId=of:0000000000000002,
      priority=10,
      selector=[IN_PORT:3,
                ETH_DST:72:3A:B9:F8:00:AB,
                ETH_SRC:36:A7:89:EB:C0:E2],
      treatment=TrafficTreatment{immediate=[OUTPUT:2],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509457742880,
      payLoad=null} added
    ==> adding RuleNode{
      table_id=4,
      index=-1,
      in_ports=[8, 9, 10],
      out_ports=[9],
      match=01101100,11100101,10010001,11010111,00000011,
            01000111,01001110,01011100,10011101,00011111,
            00000000,11010101,xxxxxxxx,...} to NPG
```

After those flow-rules events have been catch by `NetPlumberSyncManager` and, using `NetPlumberGraphManager`, Plumbing Graph has been updated, we can see the source probe node state changed:

```
2017-10-31 14:48:08,624
    Existential Probe 2 Activated after event Start Source Probe:
       Started in False State

2017-10-31 14:49:03,002
    Existential Probe 2 Activated after event Add Rule:
       Met Probe Condition

2017-10-31 14:49:03,002
    Existential Probe 2 Activated after event Add Rule:
       More Flows Met Probe Condition
```

This clearly states that after adding RULE NODES relative to the added flow-rules, *the reachability check policy has been met.*

As a counter-proof, we can try to remove the flow rules added before:

```
2017-10-31 14:58:59,232 |
    Synchronizing NPG: FlowEntry{rule=FlowEntry{
      id=6f0000a344c5c7,
      deviceId=of:0000000000000002,
      priority=10,
      selector=[IN_PORT:2,
                ETH_DST:36:A7:89:EB:C0:E2,
                ETH_SRC:72:3A:B9:F8:00:AB],
      treatment=TrafficTreatment{immediate=[OUTPUT:3],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509458339170,
      payLoad=null},
      life=596000000000,
      liveType=UNKNOWN,
      packets=579,
      bytes=56742,
      errCode=-1,
      errType=-1,
      lastSeen=1509458339170} removed
    ==> removing RuleNode{
      table_id=4,
      index=-1,
      in_ports=[8, 9, 10],
      out_ports=[10],
      match=01001110,01011100,10011101,00011111,00000000,11010101,
            01101100,11100101,10010001,11010111,00000011,01000111,
            xxxxxxxx,...} from NPG

2017-10-31 14:58:59,261 |
    Synchronizing NPG: FlowEntry{rule=FlowEntry{
      id=6f00001713d4f1,
      deviceId=of:0000000000000003,
      priority=10,
      selector=[IN_PORT:2,
                ETH_DST:36:A7:89:EB:C0:E2,
                ETH_SRC:72:3A:B9:F8:00:AB],
      treatment=TrafficTreatment{immediate=[OUTPUT:3],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                cleared=false,
                                metadata=null},
      tableId=0,
      created=1509458339181,
      payLoad=null},
      life=596000000000,
      liveType=UNKNOWN,
      packets=579,
      bytes=56742,
      errCode=-1,
      errType=-1,
      lastSeen=1509458339181} removed
    ==> removing RuleNode{
      table_id=1,
      index=-1,
      in_ports=[1, 2, 3],
      out_ports=[3],
      match=01001110,01011100,10011101,00011111,00000000,11010101,
            01101100,11100101,10010001,11010111,00000011,01000111,
            xxxxxxxx,...} from NPG

2017-10-31 14:58:59,273 |
    Synchronizing NPG: FlowEntry{rule=FlowEntry{
      id=6f0000302f74d4,
      deviceId=of:0000000000000004,
      priority=10,
      selector=[IN_PORT:2,
                ETH_DST:36:A7:89:EB:C0:E2,
                ETH_SRC:72:3A:B9:F8:00:AB],
      treatment=TrafficTreatment{immediate=[OUTPUT:1],
                                deferred=[],
                                transition=None,
                                meter=None,
                                cleared=false,
                                metadata=null},
      tableId=0,
      created=1509458339177,
      payLoad=null},
      life=596000000000,
      liveType=UNKNOWN,
      packets=579,
      bytes=56742,
      errCode=-1,
      errType=-1,
      lastSeen=1509458339177} removed
    ==> removing RuleNode{
      table_id=2,
      index=-1,
      in_ports=[4, 5],
      out_ports=[4],
      match=01001110,01011100,10011101,00011111,00000000,11010101,
            01101100,11100101,10010001,11010111,00000011,01000111,
            xxxxxxxx,...} from NPG

2017-10-31 14:58:59,280 |
    Synchronizing NPG: FlowEntry{rule=FlowEntry{
      id=6f000091eb3b33,
      deviceId=of:0000000000000002,
      priority=10,
      selector=[IN_PORT:3,
                ETH_DST:72:3A:B9:F8:00:AB,
                ETH_SRC:36:A7:89:EB:C0:E2],
      treatment=TrafficTreatment{immediate=[OUTPUT:2],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509458339170,
      payLoad=null},
      life=596000000000,
      liveType=UNKNOWN,
      packets=579,
      bytes=56742,
      errCode=-1,
      errType=-1,
      lastSeen=1509458339170} removed
    ==> removing RuleNode{
      table_id=4,
      index=-1,
      in_ports=[8, 9, 10],
      out_ports=[9],
      match=01101100,11100101,10010001,11010111,00000011,01000111,
            01001110,01011100,10011101,00011111,00000000,11010101,
            xxxxxxxx,...} from NPG

2017-10-31 14:58:59,283 |
    Synchronizing NPG: FlowEntry{rule=FlowEntry{
      id=6f000004381f02,
      deviceId=of:0000000000000001,
      priority=10,
      selector=[IN_PORT:1,
                ETH_DST:36:A7:89:EB:C0:E2,
                ETH_SRC:72:3A:B9:F8:00:AB],
      treatment=TrafficTreatment{immediate=[OUTPUT:2],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509458339169,
      payLoad=null},
      life=596000000000,
      liveType=UNKNOWN,
      packets=579,
      bytes=56742,
      errCode=-1,
      errType=-1,
      lastSeen=1509458339169} removed
    ==> removing RuleNode{
      table_id=3,
      index=-1,
      in_ports=[6, 7],
      out_ports=[7],
      match=01001110,01011100,10011101,00011111,00000000,11010101,
            01101100,11100101,10010001,11010111,00000011,01000111,
            xxxxxxxx,...} from NPG

2017-10-31 14:58:59,293 |
    Synchronizing NPG: FlowEntry{rule=FlowEntry{
      id=6f0000eaafcfe5,
      deviceId=of:0000000000000004,
      priority=10,
      selector=[IN_PORT:1,
                ETH_DST:72:3A:B9:F8:00:AB,
                ETH_SRC:36:A7:89:EB:C0:E2],
      treatment=TrafficTreatment{immediate=[OUTPUT:2],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509458339177,
      payLoad=null},
      life=596000000000,
      liveType=UNKNOWN,
      packets=579,
      bytes=56742,
      errCode=-1,
      errType=-1,
      lastSeen=1509458339177} removed
    ==> removing RuleNode{
      table_id=2,
      index=-1,
      in_ports=[4, 5],
      out_ports=[5],
      match=01101100,11100101,10010001,11010111,00000011,01000111,
            01001110,01011100,10011101,00011111,00000000,11010101,
            xxxxxxxx,...} from NPG

2017-10-31 14:58:59,296 |
    Synchronizing NPG: FlowEntry{rule=FlowEntry{
      id=6f0000aef00501,
      deviceId=of:0000000000000001,
      priority=10,
      selector=[IN_PORT:2,
                ETH_DST:72:3A:B9:F8:00:AB,
                ETH_SRC:36:A7:89:EB:C0:E2],
      treatment=TrafficTreatment{immediate=[OUTPUT:1],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509458339169,
      payLoad=null},
      life=596000000000,
      liveType=UNKNOWN,
      packets=579,
      bytes=56742,
      errCode=-1,
      errType=-1,
      lastSeen=1509458339169} removed
    ==> removing RuleNode{
      table_id=3,
      index=-1,
      in_ports=[6, 7],
      out_ports=[6],
      match=01101100,11100101,10010001,11010111,00000011,01000111,
            01001110,01011100,10011101,00011111,00000000,11010101,
            xxxxxxxx,...} from NPG

2017-10-31 14:58:59,301 |
    Synchronizing NPG: FlowEntry{rule=FlowEntry{
      id=6f000020290603,
      deviceId=of:0000000000000003,
      priority=10,
      selector=[IN_PORT:3,
                ETH_DST:72:3A:B9:F8:00:AB,
                ETH_SRC:36:A7:89:EB:C0:E2],
      treatment=TrafficTreatment{immediate=[OUTPUT:2],
                                 deferred=[],
                                 transition=None,
                                 meter=None,
                                 cleared=false,
                                 metadata=null},
      tableId=0,
      created=1509458339181,
      payLoad=null},
      life=596000000000,
      liveType=UNKNOWN,
      packets=579,
      bytes=56742,
      errCode=-1,
      errType=-1,
      lastSeen=1509458339181} removed
    ==> removing RuleNode{
      table_id=1,
      index=-1,
      in_ports=[1, 2, 3],
      out_ports=[2],
      match=01101100,11100101,10010001,11010111,00000011,01000111,
            01001110,01011100,10011101,00011111,00000000,11010101,
            xxxxxxxx,...} from NPG
```

Now, taking a look at SOURCE PROBE NODE state, we see that reachability check conditions aren't met anymore:

```
2017-10-31 14:58:59,237 -
    Existential Probe 2 Activated after event Remove Rule:
       Fewer Flows Met Probe Condition

2017-10-31 14:58:59,238 -
    Existential Probe 2 Activated after event Remove Rule:
       Failed Probe Condition
```

14.2. Summing up. In this section we've seen a basic demo of the `MCS` operation *Reachability Check*:

(1) Initially there weren't any flow-rules installed, so the policy wasn't met
(2) Then we've added the flow-rules and we've seen the synchronized model started to match the policy
(3) Finally we've tried to remove flow-rules and we've seen that the policy wasn't met anymore, as expected

This was a simple example that showed how model-checking can be useful to perform security and networking checks on the underlying network *without having to directly interact with the network itself.*

However, the developed application is generic, and it can be used for different operations and/or different underlying model-checking technologies, different than NetPlumber.

Bibliography

[1] Abnan Akhunzada et al. *Secure and dependable software defined networks*. 2015.

[2] Open Networking Foundation. *OpenFlow Switch Specification v.1.3.0*. 2012.

[3] Peyman Kazemian, George Varghese, and Nick McKeown. *Header Space Analysis: Static Checking For Networks*.

[4] Peyman Kazemian et al. *Real Time Network Policy Checking using Header Space Analysis*.

[5] Diego Kreutz et al. *Software-Defined Networking: A Comprehensive Survey*. 2014.

[6] *ONOS Wiki*. URL: `https://wiki.onosproject.org`.