A Semantic Model to Assist Policy Refinement Mechanisms for NFV-MANO Systems

Michel Bonfim[1], **Fred Freitas**[1], **Stênio Fernandes**[1]

[1]Centro de Informática (CIn) – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brazil

{msb6,fred,sflf}@cin.ufpe.br

Abstract. *Management in NFV scenarios is a complex task. In this scenario, automated policy refinement can be used to enforce NFV-MANO functions to deal with the increased complexity. However, existing solutions do not perform policy analysis. Therefore, in this work, we propose a semantic model in OWL 2, named Onto-Planner, to assist the policy refinement process for NFV-MANO systems. Preliminary results show that Onto-Planner provides policy analysis when a DL reasoner is applied.*

1. Introduction

Management in Network Function Virtualization (NFV) scenarios is a complex task since delivering such Virtual Network Functions (VNFs) and Network Services (NSs) over the physical infrastructure requires flexible and adaptable NFV Management and Orchestration (NFV-MANO) systems [ETSI 2014]. The increasing complexity of the NFV-MANO has widened the gap between human intention and the managed system behavior. Therefore, new solutions are necessary to reduce this gap and improve these complex systems' management.

In this context, Policy-based Management (PBM) systems can be used to enforce NFV-MANO functions as a way to deal with the increased complexity. PBM systems are the key enablers to provide flexibility and adaptability in NFV-MANO systems. Assisted by policies, NFV-MANO functions can be provided in an automated fashion, aiming to meet the dynamic requirements of Network Service Orchestration (NSO) and Resource Orchestration (RO). An NFV-PBM system refers to the management of rules governing the behavior of NFV-MANO. However, PBM is not a straightforward task [Riekstin et al. 2016]. This situation is exacerbated when considering its application in the management of NFV systems since NFV-MANO must provide several functionalities in a flexible and adaptable way. In this scenario, a key issue regarding PBM systems arises, namely, policy refinement.

Policy Refinement is the process of transforming high-level policies (Goals or Intents), which are not directly executable in a management system, into directly enforceable, low-level policies (Event-Condition-Action (ECA) rules). In most systems, policy refinement is done manually. Automated policy refinement is a nontrivial process, and it remains a much-neglected research area. It has been severely dismissed due to its inherent complexity [Machado et al. 2017].

According to [Riekstin et al. 2016], one of the key issues for automated policy refinement is **Policy Analysis**. A refinement solution must ensure that the refined specification achieves the requirements and is consistent with system properties and limitations,

as well as with existing policies [Craven et al. 2010]. Since disputes occur among a set of policies, pairwise detection will not suffice [Bouten et al. 2016]. To perform policy analysis, a semantic model (ontology) to describe the goals and low-level rules may be created. Such ontology would be expressed in Web Ontology Language (OWL) Description Logic (DL) [Krötzsch et al. 2012], which - besides the well-defined, non-ambiguous semantics above mentioned for the vocabulary being defined - provides decidable, sound and complete inference for some reasoning tasks, such as inconsistency checking.

Aiming at moving forward on the Network Management Research Group (NMRG) standardization threads, this work proposes the use of a semantic model in OWL 2 [Krötzsch et al. 2012] to assist the policy refinement process for NFV-MANO systems, called **Onto-Planner**. In summary, our main contributions are as follows:

1. A language for describing goals in NFV environments;
2. The Onto-Planner ontology, which offers a vocabulary with its complex relations and constraints of the domain, expressive enough to describe different types of alarms, goals, and ECA rules that will be used in the refinement process;
3. The Onto-Planner axioms and Semantic Web Rule Language (SWRL)[1] rules that provides policy analysis, when a DL reasoner is applied.

2. Network Function Virtualization (NFV)

NFV is transforming the computer and communication networks industry. Rather than dealing with proprietary hardware equipment, which includes a limited set of specific network applications, NFV allows users to transfer network functions from specific to Common Off-The-Shelf (COTS) hardware using virtualization technologies [ETSI 2014]. These network functions are then called VNFs and perform different network operations: Firewall, Network Address Translation (NAT), Deep Packet Inspection (DPI), Load Balancing, among others.

NFV aims at creating NSs that interconnect one or more VNFs to support the creation and deployment of end-to-end network services [ETSI 2014]. Some benefits of deploying network services as virtual functions are: (i) flexibility in the allocation of network functions in general-purpose hardware; rapid implementation and deployment of new network services; support of multiple versions of service and multi-tenancy scenarios; reduction in Capital Expenditure (CAPEX) costs by managing energy usage efficiently; and automation of the operational processes, thus improving efficiency and reducing Operational Expenditure (OPEX) costs.

The NFV architecture comprises three main functional blocks: VNF, NFV Infrastructure (NFVI), and NFV-MANO. VNF is the virtualization of a specific network function, which should operate independently of the others. A particular VNF can also be divided into several sub-functions called VNF Components (VNFCs), where there is one Virtual Machine (VM) implementing each VNFC. On the other hand, NFVI comprises all hardware and software required to deploy, operate, and monitor VNFs. To this end, NFVI has a virtualization layer necessary for abstracting the hardware resources (processing, storage, and network connectivity). Finally, NFV-MANO performs both NSO and RO. Such functionalities include the following non-exhaustive list of operations [ETSI 2014]:

[1] https://www.w3.org/Submission/SWRL/

VNF and NS lifecycle management, VNF and NS scaling, resource orchestration, and management in multi-domain scenarios, access control, and performance and fault management.

For these purposes, NFV-MANO comprises three components.

- The Virtualized Infrastructure Manager (VIM), which manages and controls the interaction of VNFs with physical resources under its control (e.g., allocation, deallocation, and inventory);
- The VNF Manager (VNFM), which is responsible for managing the VNF lifecycle (e.g., initialization, suspension, and termination); and
- The NFV Orchestrator (NFVO), which is responsible for realizing NS on NFVI. It also performs NFVI monitoring as a way to collect information for operations and performance management.

3. Related Work

In NFV systems, we can identify two types of policies: application-specific and management-specific. The former consists of domain-specific policies that can be set for both a single NS and globally for all services such as network function precedence, location constraints, and resource usage. The latter consists of the rules that will be generated by the refinement process and stored in NFV-PBM to govern system behavior. We only found works that perform policy analysis to detect conflicts between application-specific policies [Bouten et al. 2016, Bonfim et al. 2019], and one of them is our previous work [Bonfim et al. 2019]. Therefore, we argue that Onto-Planner is the first semantic model that perform policy analysis to detect conflicts between management-specific policies.

We found only two works that provide a solution for NFV policy refinement. [Scheid et al. 2017] proposed an Intent-Based Networking (IBN) solution, named IN-SpIRE. This solution rely on a Non-Functional Requirement (NFR) framework to perform the refinement procedure. [Jacobs et al. 2019] proposed an intent definition language, called Nile, to fill the gap between human intention and the network configurations. Nile-based code is compiled to generate the lower-level configuration commands.

It is worth mentioning that none of the above works perform **policy analysis**. They did not take into account the effects of actions/policies have over the managed objects (system state) during the refinement process. In this context, Onto-Planner can be adapted to support such solutions to avoid conflicts between different intents and low-level policies.

4. Onto-Planner

To compose Onto-Planner classes, we reused two information models: NOVI Policy from the NOVI (Network Innovation over Virtualized Infrastructures) Information Model[2], and Policy Core Information Model (PCIM)[3]. It is worth noting that we reused them to define policy rules and their different parts in a vendor-independent manner.

Onto-Planner enables the description of various objects that make up the state of a policy refinement domain for NFV. With Onto-Planner at hand, an operator or

[2]https://www.sciencedirect.com/science/article/abs/pii/S0167739X13002811

[3]https://tools.ietf.org/html/rfc3060

NFV-MANO may record different types of alarms and goals that will be used in the refinement process. Besides, at the end of this process, Onto-Planner may be used to describe the generated enforceable policies (ECA rules). Finally, conflicts between these elements (alarms, goals, ECA rules) are carried out through DL inconsistency verification when a DL semantic reasoner is relied upon.

We organized the Onto-Planner classes into two categories: classes used before and after the refinement process. The first category encompasses all classes used for the following descriptions:

- **NFV-MANO operators**: deals with the described NFV-MANO elements that are manageable (e.g., NFVO) and the operators that allow them to be managed, defined according to European Telecommunications Standards Institute (ETSI) specifications. The following classes are used for this description: ***ManagedEntity*** and ***ManagedEntityMethod***;
- **Alarms**: deals with the description of different types of alarms. To describe an alarm, the operator (via Threshold API) should specify one of the predefined metrics (instances of the ***Metric*** class) for the alarm to watch, define the threshold (***Threshold*** class) for that metric, and the type of action (from a predefined list - ***Action*** class) to perform when the alarm is triggered. Onto-Planner currently allows alarm creation for auto-healing and auto-scaling. Further details will be presented in Section 4.1;
- **Goals**: deals with the description of different types of high-level goals. An individual of the ***Goal*** class indicates a goal's existence. Besides, a single goal must be associated with a single NS (***NS*** class), one or more VNFs (***VNF*** class) belonging to that NS, a single level (***Level*** class), and at least one service attribute (***Attribute*** class). Further details are presented in Section 4.2;
- **Events** and **Parameters**: deal with the description of possible events (***Event*** class) and parameters (***Parameter*** class) that can be used to compose the events and conditions, respectively, of the generated enforceable policies. Further details are in Section 4.3.

It is worth noting that all of the above descriptions are used in the refinement process.

The second category comprises all classes used after the refinement process for the description of enforceable policies, and subsequent conflict detection and diagnosis. There are classes that help the description of ECA rules such as ***PolicyRule***, ***PolicyEvent***, ***PolicyCondition***, and ***PolicyAction***. Furthermore, there are classes that assist an ECA rule to support alarm creation into the NFV-PBM. Since Onto-Planner only supports healing and scaling alarms, we create only the following classes: ***HealingGroup***, ***HealingPolicy***, ***HealingCriteria***, ***ScalingGroup***, ***ScalingPolicy***, and ***ScalingCriteria***. They offer a way of describing enforceable alarms that is supported by different MANO frameworks such as Open Source MANO[4]. Such classes only describe the alarms selected during refinement. Further details about enforceable policies description are in Section 4.3.

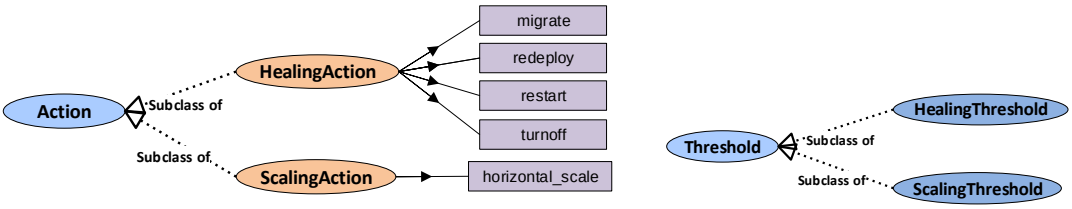
Additionally, Onto-Planner has all sibling classes disjoint because they describe different elements of the domain.

[4]https://osm.etsi.org/

4.1. Describing Alarms

Multiple NFV-MANO frameworks such as Open Baton[5] and Open Source MANO[6] support fault management. Such feature enables the creation and execution of alarms in the form of ECA Rules to deal with possible problems that may occur in NFVI. Alarms help attend to the requirement of dynamicity regarding current or future system state and topology changes. Therefore, in addition to enforceable policies that govern NFV-PBM behavior, we must also generate rules that include alarms in our refinement process.

Onto-Planner enables the operator to record different alarm types for the refinement process. In this context, one operator can specify the metric for the alarm to watch, the threshold for that metric, and the kind of action (from a predefined list) to be executed when the alarm fires. To this end, one prerequisite must be met: all available metrics must be previously recorded as ***Metric*** individuals. Such metrics should be accessible by the target NFV-MANO. The ontology maintainer should carry out this task, i.e., someone responsible for editing Onto-Planner (via, e.g., Protégé tool[7]).



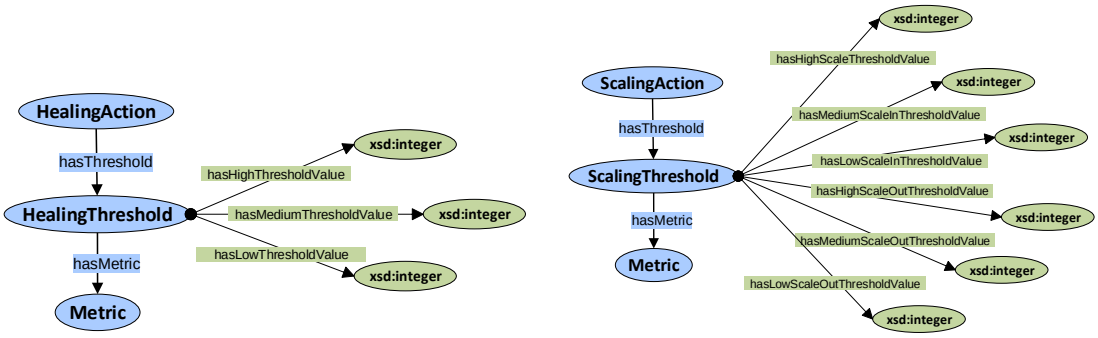
(a) Action subclasses. **(b)** Threshold subclasses.

Figure 1. Healing and Scaling subclasses.

Onto-Planner currently allows alarm creation for auto-healing and auto-scaling. Figure 1 shows the classes and subclasses available for these purposes. The ***Action*** class enables the description of all actions (class individuals) necessary to alarm creation (see Figure 1a). To group these operations, we created two defined classes [Krötzsch et al. 2012], subclasses of *Action*. The ***HealingAction*** class encompasses only healing-oriented actions. It is currently limited to the following individuals: ***migrate***, ***redeploy***, ***restart***, ***turnoff***. These individuals represent common healing actions [ETSI 2016]. The ***ScalingAction*** class includes only scaling-oriented actions. It is currently limited to only one individual: ***horizontal_scale***. Horizontal scaling means adding or removing virtualized computer resources into/from your pool of resources [ETSI 2015]. Finally, it is possible to define non-functional parameters for *Action* individuals such as its subject and target.

The ***Threshold*** class allows description of custom thresholds for both healing and scaling actions. For this, it has two subclasses: ***HealingThreshold*** and ***ScalingThreshold*** (see Figure 1b). Figure 2a shows how a healing alarm can be described. An individual of type *HealingAction* must have a relationship of type ***hasThreshold*** with an individual of type *HealingThreshold*. In its turn, the *HealingThreshold* individual must have three

[5]Open Baton website: https://openbaton.github.io/
[6]Open Source MANO website: https://osm.etsi.org/
[7]Protégé website: https://protege.stanford.edu/



(a) Healing alarms. **(b)** Scaling alarms.

Figure 2. Describing alarms.

data properties set with integer values. These data properties define thresholds for the three possible levels of a goal: *low*, *medium*, and *high*. Besides, the *HealingThreshold* individual must have a relationship of type ***hasMetric*** with an individual of type *Metric*.

Similarly, one can describe a scaling action, as shown in Figure 2b. A *ScalingThreshold* individual type must have six data properties set instead of three. The new data property is because horizontal scaling has two possible operations: scale in (i.e., removing resources) and out (i.e., adding resources). Therefore, we must define thresholds for both operations.

Finally, some restrictions (axioms) have been defined in the above classes to limit the scope in creating alarms. In fact, a *Threshold* element must be associated with at least one *Metric*. Furthermore, a *Threshold* element must have exactly one data property set for each level and each operation in case of *ScalingThreshold*.

4.2. Describing Goals

According to [Han and Lei 2012], the use of goals (or intents) allows the specification of the desired states instead of a sequence of actions. In this context, we can declare the high-level goals, such as Service Level Agreements (SLAs), and hence generate enforceable policies to govern NFV-MANO behavioral choices while satisfying the goals. Besides, goal-oriented refinement assists in reducing both Capital Expenditures (CAPEX) and Operational Expenditures (OPEX).

In this context, Onto-Planner enables NFVO to record goal policies extracted from Network Service Descriptor (NSD). It deals with the description of different types of high-level goals. To this end, we proposed the following language to define high-level goals in a NS Request (NS-Req) (Listing 1):

Listing 1. Goal Language Grammar

```
Language -> <Elements> must receive <Level> <Attributes>
Elements -> <Element> | <Element> <Connective> <Elements>
Element -> vnf-member-index
Level -> high | medium | low
Attributes -> <Attribute> | <Attribute> <Connective> <Attributes>
Attribute -> resiliency | manageability | security | performance
Connective -> and
```

A tenant can define one or more goals in the same NS-Req. Each goal has three (3) parts:

- ***Elements***: specifies which NS-specific VNFs the goal should be applied to. To identify a VNF, we set the parameter ***vnf-member-index***, which indicates the position of the VNF in the service chain;
- ***Level***: determines the goal's degree of importance, i.e., how critical this goal is for the NS (high, medium or low);
- ***Attributes***: defines one or more service attributes for the goal. According to ETSI [ETSI 2014], the overall NS attributes are reliability, availability, manageability, security and performance. For the grammar, we replaced reliability and availability by resilience since ETSI considers that resiliency is an aspect of QoS that can be characterized by the combination of reliability and availability [ETSI 2015].

For a better understanding, let us consider the following goal policy example (Listing 2):

Listing 2. Goal Example

```
1 and 2 must receive high performance and resilience
```

This goal establishes that NFV-MANO must provide high performance and resilience for the VNFs in positions 1 and 2 within the NS.

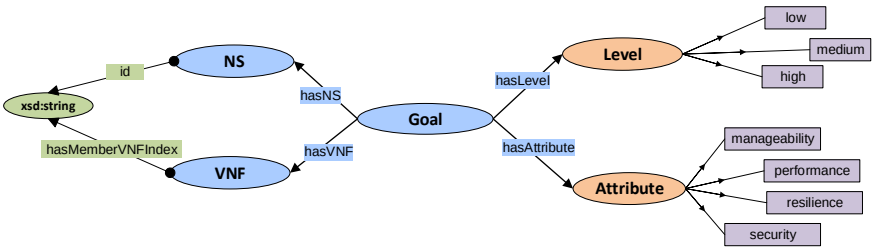


Figure 3. Describing high-level goals.

According to Figure 3, a ***Goal*** individual must be associated with a single NS (***NS*** class), one or more VNFs (***VNF*** class) belonging to that NS, a single level (***Level*** class), and at least one service attribute (***Attribute*** class). Table 1 lists the axioms that delimit these restrictions.

Both *Level* and *Attribute* classes are defined classes [Krötzsch et al. 2012]. The *Level* class is limited to the following individuals: ***low***, ***medium***, ***high***. They determine the degree of importance of the goal, i.e., how critical this goal is for the NS. In its turn, the *Attribute* class is restricted to the following individuals: ***resiliency***, ***manageability***, ***security***, and ***performance***, as defined by ETSI [ETSI 2014].

4.3. Describing Policy Rules

Once a Policy Refinement Mechanism executes, it generates a set of enforceable policies (ECA rules) and alarms for a previously registered goal policy. Those elements can be described in the Onto-Planner for inconsistency checking (policy analysis). In this section, we show how to describe ECA rules and enforceable alarms.

Table 1. Class Axioms for Goal description (Manchester Syntax).

Goal **SubClassOf** *hasNS max* 1 NS
Goal **SubClassOf** *hasVNF some* VNF
Goal **SubClassOf** *hasLevel max* 1 Level
Goal **SubClassOf** *hasAttribute some* Attribute

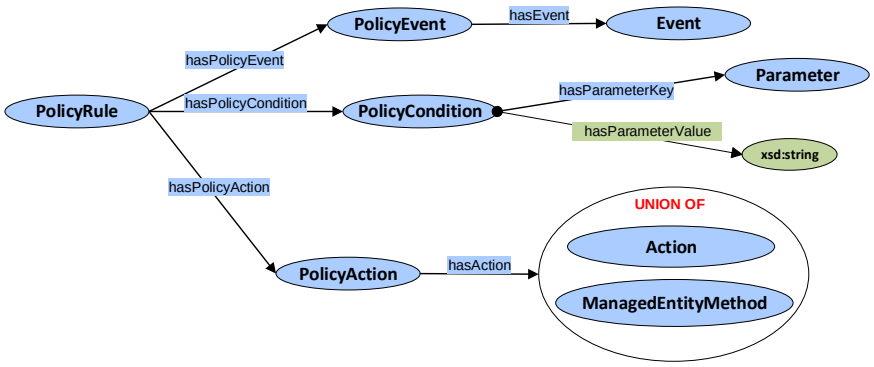


Figure 4. Describing low-level policy rules.

As shown in Figure 4, we define the following classes to describe ECA rules: ***PolicyRule***, ***PolicyEvent***, ***PolicyCondition***, and ***PolicyAction***. A *PolicyRule* individual must have relationship with exacly one *PolicyEvent*, via the ***hasPolicyEvent*** object property, and exacly one *PolicyAction*, via ***hasPolicyAction*** object property. Besides, a *PolicyRule* must have one or more *PolicyCondtion* elements; this relationship is defined by the ***hasPolicyCondition*** object property.

Furthermore, an individual of the *PolicyEvent* class must be associated (*hasEvent* property) with one or more ***Event*** individuals. The ***Event*** class deals with the description of all existing events in an NFV-MANO such as "network service instantiated" or "VNF instance scaled". It is noteworthy that possessing multiple *hasEvent* relationships in the same *PolicyEvent* indicates that all events must occur to trigger the rule.

Table 2. Class Axioms for ECA Rule description (Manchester Syntax).

PolicyRule **SubClassOf** hasPolicyAction exactly 1 PolicyAction
PolicyRule **SubClassOf** hasPolicyEvent exactly 1 PolicyEvent
PolicyEvent **SubClassOf** hasEvent some Event
PolicyAction **SubClassOf** hasAction some (Action or ManagedEntityMethod)
PolicyCondition **SubClassOf** hasParamenterKey exactly 1 Parameter

In addition, one *PolicyAction* may be related (***hasAction*** property) with one or more actions, which may be of type *Action* or *ManagedEntityMethod*. Finally, each *PolicyCondition* must be associated (***hasParameterKey*** property) with only one parameter type (*Parameter* class), which can include multiple string values (***hasParameterValue*** data property). The ***Parameter*** class deals with the description of all existing parameters in an NFV-MANO such as "network service id" or "alarm type". Table 2 lists the created axioms that reflect the above restrictions.

4.4. Managing Policy Conflicts

Onto-Planner enables the following conflict checks: Alarm Conflicts, Goal Conflicts, and ECA Rule Conflicts.

Concerning **Alarm Conflicts**, we want to detect if there are at least two alarms with the same metric and the same action, an obvious policy conflict. For this, we create the object property ***usedBySameActionAs***, relating two *Metric* individuals, that are used by alarms that apply the same action.

Nevertheless, we still have a critical challenge: the definition of which statements or propositions reflect the behavior of this function. For the property to reflect its expected behavior, it has to **Irreflexive**. For instance, let x and y be individuals of the *Metric* class. If x is used by the same action as y, then x cannot be y.

Besides, we specify an SWRL rule in (1) to generate implicit facts with the *usedBySameActionAs* object property: if we have an action x that has two different thresholds y and z, where y has the metric $m1$ and z has the metric $m2$, then there is a relationship of type *usedBySameActionAs* between these two metrics.

$$\begin{aligned} & differentFrom(y,z) \wedge Action(x) \\ \wedge & hasThreshold(x,z) \wedge hasThreshold(x,y) \\ \wedge & hasMetric(y,m1) \wedge hasMetric(z,m2) \\ \rightarrow & \; usedBySameActionAs(m1,m2) \end{aligned} \tag{1}$$

Concerning **Goal Conflicts**, we want to detect an intersection between one or more goals. Remarkably, intersections between goals can lead to duplicate policies at the end of the refinement process. Therefore, we should check this whenever a new goal is created. In this work, we consider two goals intersect when they share the same NS, one or more VNFs, and one or more attributes. Hence, again we create the ***intersectsWith*** object property to reflect this behavior. It relates two individuals of *Goal* class. Besides, we specify the SWRL rule in (2) to generate implicit facts with this object property.

$$\begin{aligned} & Goal(g1) \wedge Goal(g2) \wedge differentFrom(g1,g2) \\ \wedge & hasNS(g1,ns1) \wedge hasNS(g2,ns2) \wedge id(ns1,idt) \wedge id(ns2,idt) \\ \wedge & hasVNF(g1,vnf1) \wedge hasVNF(g2,vnf2) \\ \wedge & hasMemberVNFIndex(vnf1,index) \\ \wedge & hasMemberVNFIndex(vnf2,index) \\ \wedge & Attribute(a) \wedge hasAttribute(g1,a) \wedge hasAttribute(g2,a) \\ \rightarrow & \; intersectsWith(g1,g2) \end{aligned} \tag{2}$$

Suppose we have two goals x and y. Once we execute the DL reasoner, the following facts will be inferred: $intersectsWith(x,y)$ and $intersectsWith(y,x)$.

However, these facts alone are insufficient to generate inconsistency in ontology, which is the approach used to detect conflicts. The object property *intersectsWith* has to be **Asymmetric** to meet this constraint. For example, let x and y be individuals of the *Goal* class. If x intersects with y, then y cannot intersects with x. By applying this restriction, Onto-Planner is inconsistent whenever the rule in (2) infers facts with the *intersectsWith* object property.

Concerning **ECA Rule Conflicts**, both alarm and goal conflict detection are performed whenever one of these elements is created in the ontology. This cautionary ap-

proach seeks to prevent enforceable policies from being generated with conflicts after the refinement process. Nonetheless, there is still a kind of conflict to avoid: the duties' conflict. According to [Bandara et al. 2003], this conflict arises if the same subject performs a task that encompasses operations that, in the context of the application, are defined to be conflicting, for example, instantiating and terminating the same VNF in the same task.

To solve this issue, we need to detect if at least two *PolicyRule* individuals share the same events and conditions but include conflicting actions. To make this case, we created the ***conflictsWith*** object property to indicate that two actions are conflicting. This property links two individuals belonging to the set formed by the union of *ManagedEntityMethod* and *Action* classes. Before the refinement process, the ontology maintainer must define all conflicting actions, i.e., define facts with this object property. The ontology maintainer is then responsible for editing Onto-Planner without using the Protégé tool[8], for example.

Besides, we specify the SWRL rule in (3) to generate implicit facts with the *conflictsWith* object property.

$$\begin{aligned} & PolicyRule(r1) \wedge PolicyRule(r2) \\ & \wedge hasPolicyEvent(r1, pe1) \wedge hasPolicyEvent(r2, pe2) \\ & \wedge hasEvent(pe1, e) \wedge hasEvent(pe2, e) \\ & \wedge hasPolicyCondition(r1, pc1) \wedge hasPolicyCondition(r2, pc2) \\ & \wedge hasParameterKey(pc1, key) \wedge hasParameterKey(pc2, key) \\ & \wedge hasParameterValue(pc1, value) \\ & \wedge hasParameterValue(pc2, value) \\ & \wedge hasPolicyAction(r1, pa1) \wedge hasPolicyAction(r2, pa2) \\ & \wedge hasAction(pa1, a1) \wedge hasAction(pa2, a2) \\ & \wedge conflictsWith(a1, a2) \\ & \rightarrow conflictsWith(a2, a1) \end{aligned} \tag{3}$$

Suppose the following fact was previously created into Onto-Planner: $conflictsWith(intantiate\_vnf, terminate\_vnf)$.

Now, suppose that, after the refining process, two PolicyRule individuals were generated: x and y. These rules share the same events and conditions, but x includes on $instantiate\_vnf$ action and y, a $terminate\_vnf$ action. Thus, the DL reasoner infers the following fact: $conflictsWith(terminate\_vnf, intantiate\_vnf)$.

Analogously to other definitions above, *conflitsWith* is **Asymmetric**, so as to provoke an inconsistency. For example, let x and y be *Action* individuals. If x intersects with y, then y cannot intersects with x. Using this restriction, Onto-Planner will be inconsistent whenever the rule in (3) summon facts with the *conflitsWith* object property.

We emphasize that, unfortunately, we have not been able to define axioms using role chains. According to OWL 2.0 GRs[9], once we set an object property as irreflexive or asymmetric, we can only define simple object property expressions for it, and thus, property chains are prohibitive. This is the rationale behind the above SWRL rules employing property chains.

[8]Protégé website: https://protege.stanford.edu/

[9]OWL 2.0 GR website: `http://www.w3.org/TR/owl2-syntax/`

```
{ "name":"Conflict Thresholds",
  "healPols": [{
     "name":"Migrate CPU",
     "action":"migrate",
     "metric":"osm_pc_vim_cpu_load",
     "highThresholdValue":70,
     "mediumThresholdValue":80,
     "lowThresholdValue":100},
  {
     "name":"Migrate MEM",
     "action":"migrate",
     "metric":"osm_pc_vim_cpu_load",
     "highThresholdValue":60,
     "mediumThresholdValue":70,
     "lowThresholdValue":90
  }]
}
```

(a) JSON request Use Case 1.

1) **ObjectPropertyDomain:** ***hasThreshold*** *Action*
2) migrate ***hasThreshold*** conflict-thresholds-migrate-cpu
3) migrate ***hasThreshold*** conflict-thresholds-migrate-mem
4) conflict-thresholds-migrate-cpu ***hasMetric*** osm_pc_vim_cpu_load
5) conflict-thresholds-migrate-mem ***hasMetric*** osm_pc_vim_cpu_load
6) **SWRL Rule in (1)**
7) **IrreflexiveObjectProperty:** ***usedBySameActionAs***

(b) Explanation for the inconsistency detected in Use Case 1.

Figure 5. Detecting and Diagnosing conflicts between alarms.

5. Onto-Planner Validation

In this section, we present the process and results of Onto-NFV validation. The process was carried out in two phases.

In the first phase, we implement Onto-Planner using the Protégé tool[10]. The source code is available in the following link: `https://github.com/michelsb/atom/blob/master/database/onto-planner.owl`. Besides, we implement a Java-based RESTful API to manage Alarm, Goal and ECA rule individuals, relying on Spring Boot[11] and OWL API[12]. To create or remove those individuals, we use JSON as a data model. For conflict detection and diagnostic, we use the HermiT[13], a fully compliant OWL 2 Reasoner for inconsistency verification.

In the second phase, we simulate three use cases (UC) to evaluate the capacity of Onto-Planner to detect and diagnose conflicts between alarms, goals, and ECA rules. We describe them below.

5.1. Use Case 1 - Testing Conflicts between Alarms

As mentioned in Subsection 4.4, Onto-Planner allows the operator to verify if there are alarms in which the same metric applies twice for the same action. In this use case, we simulate conflict between two alarms by inserting the two alarms (see Figure 5a) into Onto-Planner.

We create two alarms that point to the same action (i.e., migrate) and the same metric (i.e., osm_pc_vim_cpu_load). In this case, when the reasoner is performed, it detects inconsistencies in the Onto-Planner and presents the explanations shown in Figure 5b.

Such explanations state that, since there is the SRWL rule (1), the reasoner infers the following relationships **(line 6)**: $usedBySameActionAs(osm\_pc\_vim\_cpu\_load, osm\_pc\_vim\_cpu\_load)$.

[10]Protégé website: https://protege.stanford.edu/
[11]Spring Boot website: https://spring.io/projects/spring-boot
[12]OWL API website: http://owlapi.sourceforge.net/
[13]HermiT website: http://www.hermit-reasoner.com/

```
{ "name":"Conflict Goals",
  "goals": [{
     "name":"Goal 1",
     "nsrId": "2599ab88-aaab",
     "vnfMemberIndexes": ["1","2"],
     "level":"low",
     "attributes": ["resilience"]},
  {
     "name":"Goal 2",
     "nsrId": "2599ab88-aaab",
     "vnfMemberIndexes": ["2"],
     "level":"medium",
     "attributes": ["resilience"]
  }]
}
```

(a) JSON request Use Case 2.

1) goal-1 ***hasNS*** goal-1-ns-2599ab88-aaab
2) goal-2 ***hasNS*** goal-2-ns-2599ab88-aaab
3) goal-1-ns-2599ab88-aaab ***id*** “2599ab88-aaab”
4) goal-2-ns-2599ab88-aaab ***id*** “2599ab88-aaab”
5) goal-1 ***hasVNF*** goal-1-vnf-2
6) goal-2 ***hasVNF*** goal-2-vnf-2
7) goal-1-vnf-2 ***hasMemberVNFIndex*** “2”
8) goal-2-vnf-2 ***hasMemberVNFIndex*** “2”
9) goal-1 ***hasAttribute*** resilience
10) goal-2 ***hasAttribute*** resilience
11) **DifferentIndividuals:** goal-1, goal-2
12) **SWRL Rule in (2)**
13) **AsymmetricObjectProperty:** ***intersectsWith***

(b) Explanation for the inconsistency detected in Use Case 2.

Figure 6. Detecting and Diagnosing conflicts between Goals.

However, this assertion cannot be valid since *usedBySameActionAs* is irreflexive **(line 7)**, as mentioned in Subsection 4.4, thus generating inconsistencies.

5.2. Use Case 2 - Testing Conflicts between Goals

As mentioned in Subsection 4.4, Onto-Planner allows the NFVO to detect conflicts between goals that apply to the same NS and share the same attributes and VNFs. In this use case, we simulate such a conflict by inserting the two goals (see Figure 6a) into Onto-Planner.

In this scenario, we create two goals that point to the same NS with id “2599ab88-aaab” and share the same VNFs (i.e., member index 2) and attributes (i.e., resilience). In this case, when the reasoner is performed, it detects inconsistencies in the Onto-Planner and presents the explanations shown in Figure 6b.

Such explanations state that, since there is the SRWL rule (2), the reasoner infers the following relationships **(line 12)**: $intersectsWith(goal\text{-}1, goal\text{-}2)$ and $intersectsWith(goal\text{-}2, goal\text{-}1)$.

However, this assertion cannot be valid since *intersectsWith* is asymmetric **(line 13)**, as mentioned in Subsection 4.4, and the individuals *goal-1* and *goal-2* are different **(line 11)**, thus generating inconsistencies.

5.3. Use Case 3 - Testing Conflicts between Rules

As mentioned in Subsection 4.4, Onto-Planner allows the NFVO to detect conflict of duties. Such conflict will arise if the same subject should perform a task that encompasses operations that are explicitly defined to be conflicting in the context of the application. For example, instantiate and terminate the same VNF in the same task. Therefore, we simulate the detection and diagnosis of conflicts of duties in this use case.

To this end, first, we define a relationship *conflictsWith* between two previously created operations: *create_phy_nic_bonding* and *create_load_sharing_between_links*.

Besides, we created two ECA rules. Both rules were described to be triggered by the same events and conditions. In addition, one has *create_phy_nic_bonding* as action,

while the other has *create_load_sharing_between_links* as action. Finally, once executed, the reasoner detects inconsistencies in the Onto-Planner and presents the explanations shown in Figure 7.

1) create_phy_nic_bonding ***conflictsWith*** create_load_sharing_between_links
2) pol_create_load_sharing_between_links **Type** PolicyRule
3) pol_create_phy_nic_bonding **Type** PolicyRule
4) pol_create_load_sharing_between_links ***hasPolicyEvent*** object26
5) pol_create_load_sharing_between_links ***hasPolicyAction*** object25
6) object25 ***hasAction*** create_load_sharing_between_links
7) pol_create_phy_nic_bonding ***hasPolicyAction*** object21
...
11) object21 ***hasAction*** create_phy_nic_bonding
...
18) **SWRL Rule in (3)**
19) **AsymmetricObjectProperty: *conflictsWith***

Figure 7. Explanation for the inconsistency detected in Use Case 3.

Such explanations state that, since there is a relationship *conflictsWith* between the operations *create_phy_nic_bonding* and *create_load_sharing_between_links* **(line 1)**, and since there are two ECA rules, named pol_create_load_sharing_between_links **(line 2)** and pol_create_phy_nic_bonding **(line 3)**, that include those two operations **(lines 6 and 11)**, the reasoner infers the following relationship from the SRWL rule (2) **(line 18)**: $conflictsWith(create\_load\_sharing\_between\_links, create\_phy\_nic\_bonding)$

However, this assertion cannot be valid since *conflictsWith* is asymmetric **(line 19)**, as mentioned in Subsection 4.4, thus generating inconsistencies.

6. Conclusion

This paper proposed the Onto-Planner, a semantic model in OWL 2 that enables the description of different types of alarms and goals used in the refinement process. Besides, at the end of this process, Onto-Planner may be used to describe the generated ECA rules. Finally, when a DL semantic reasoner is relied upon, it performs policy analysis to detect conflicts between those elements. Preliminary experiments demonstrate that Onto-Planner can be used to assist the policy refinement process for NFV-MANO systems. However, it is worth mentioning that these experiments are only in their initial phase. In this case, for future work, it would be good to provide a performance evaluation.

It is noteworthy that the implementation of a Policy Refinement Mechanism and an NFV-PBM is not part of the scope of this work. Onto-Planner aims to assist and improve the operation of these solutions. An example of an NFV-PBM architecture can be found in [ETSI 2017].

Finally, taking into account the latest efforts made by IETF/IRTF, future activities include pursuing contributions to standardization along the way Onto-Planner can be used to support intelligent network management mechanisms (Intent-Based Networking) for NFV systems in an efficient and scalable way.

References

Bandara, A. K., Lupu, E. C., and Russo, A. (2003). Using event calculus to formalise policy specification and analysis. In *Proceedings of the 4th IEEE International Workshop*

on Policies for Distributed Systems and Networks, POLICY '03, pages 26–, Washington, DC, USA. IEEE Computer Society.

Bonfim, M., Freitas, F., and Fernandes, S. (2019). A semantic-based policy analysis solution for the deployment of nfv services. *IEEE Transactions on Network and Service Management*, 16(3):1005–1018.

Bouten, N., Claeys, M., Mijumbi, R., Famaey, J., Latré, S., and Serrat, J. (2016). Semantic validation of affinity constrained service function chain requests. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 202–210.

Craven, R., Lobo, J., Lupu, E. C., Russo, A., and Sloman, M. (2010). Decomposition techniques for policy refinement. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM 2010, Niagara Falls, Canada, October 25-29, 2010*, pages 72–79.

ETSI (2014). Network Functions Virtualisation (NFV) - Architectural Framework. *ETSI GS NFV 002 V1.2.1*.

ETSI (2015). Network Functions Virtualisation (NFV) - Resiliency Requirements. *ETSI GS NFV-REL 001 V1.1.1*.

ETSI (2016). Network Functions Virtualisation (NFV) - Reliability - Report on Models and Features for End-to-End Reliability. *ETSI GS NFV-REL 003 V1.1.2 (2016-07)*.

ETSI (2017). Network functions virtualisation - management and orchestration - report on policy management in mano (release 3). *ETSI GR NFV-IFA 023 V3.1.1*.

Han, W. and Lei, C. (2012). A survey on policy languages in network and security management. *Computer Networks*, 56(1):477–489.

Jacobs, A. S., Pfitscher, R. J., Ferreira, R. A., and Granville, L. Z. (2019). Refining network intents for self-driving networks. *SIGCOMM Comput. Commun. Rev.*, 48(5):55–63.

Krötzsch, M., Patel-Schneider, P., Rudolph, S., Hitzler, P., and Parsia, B. (2012). OWL 2 web ontology language primer (second edition). Technical report, W3C. http://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

Machado, C. C., Wickboldt, J. A., Granville, L. Z., and Filho, A. E. S. (2017). ARKHAM: an advanced refinement toolkit for handling service level agreements in software-defined networking. *J. Network and Computer Applications*, 90:1–16.

Riekstin, A. C., Januario, G. C., Rodrigues, B. B., Nascimento, V. T., de Brito Carvalho, T. C. M., and Meirosu, C. (2016). A survey of policy refinement methods as a support for sustainable networks. *IEEE Communications Surveys and Tutorials*, 18(1):222–235.

Scheid, E. J., Machado, C. C., Franco, M. F., dos Santos, R. L., Pfitscher, R. P., Schaeffer-Filho, A. E., and Granville, L. Z. (2017). Inspire: Integrated nfv-based intent refinement environment. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 186–194.