An EC-Based Formalism for Policy Refinement in Software-Defined Networking

Cristian Cleder Machado, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Alberto Schaeffer-Filho
Institute of Informatics – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
Email: {ccmachado, jwickboldt, granville, alberto}@inf.ufrgs.br

***Abstract*—Software-Defined Networking (SDN) provides a sophisticated and accurate solution for managing network traffic. SDN logically centralizes, in devices called controllers, part of the decision-making logic of flow processing and packet routing. The whole network is controlled according to rules written and deployed in the controller device. However, the large amount of network devices, links, and services also gives rise to a large number of rules to be managed in the controller. Policy-Based Network Management (PBNM) can be used to manage complex network infrastructures through policies rather than specifying device-by-device configurations. Particularly, policy refinement techniques can be used to automatically translate high-level policies into a set of low-level ones. In this paper, we define a formal representation of high-level SLA policies using Event Calculus (EC) and apply logical reasoning to model both the system behavior and the policy refinement process for SDN management. We also describe the implementation of this formal model in Prolog, which enables the automatic inference of low-level policies from high-level ones, and present evaluation results.**

I. Introduction

Software-Defined Networking (SDN) introduces a novel architecture that simplifies network management by moving part of the decision-making logic of the network to a component called controller, while switching elements become simple packet forwarding devices [1]. As a result, controllers can have a global view of the network traffic, and switching elements can be configured through an open interface, such as the OpenFlow protocol [2], independently from the hardware. The configuration of forwarding devices is often coordinated by software written for specific situations directly into the controller. However, this programming model is susceptible to many complications, such as *(i)* high human workload due to the need to write a large set of rules, *(ii)* struggle to accommodate new services that were not anticipated when hard-coded rules were written, and *(iii)* low-level rules may not faithfully fulfill high-level policies, as network programmers might not be aware of business goals.

A possible solution to alleviate the issues above is the use of a Policy-Based Network Management (PBNM) approach. PBNM can be used to manage complex network infrastructures through a set of high-level policies rather than specifying device-by-device configurations [3]. In particular, policy refinement techniques can be used to automatically translate such high-level policies – *e.g.*, specified as a *Service Level Agreement (SLA)* – into a set of low-level rules, which can be enforced directly into network devices. Thus, network operation can be modified without the need to rewrite software, without human intervention or having to stop the system [4]. The use of PBNM and policy refinement in traditional networks has been investigated already [5], [6], [7]. However, these studies have been limited to the characteristics of traditional IP networks, such as best-effort packet delivery and distributed control state in forwarding devices. In addition, policy refinement in the field of SDN has been a neglected topic, in part, because refinement is a nontrivial process.

In this paper, we define a formal representation of high-level policies in the form of SLAs using Event Calculus (EC) and apply logical reasoning to model both the system behavior and the policy refinement process in SDN. We aim to assist infrastructure-level programmers to develop refinement tools and configuration approaches to achieve more robust SDN deployments, independent of the network controller implementation or policy language. Unlike previous work in the area of policy refinement [5], [6], [7], our formal representation for refinement of SLAs relies on the information captured from the SDN infrastructure. We chose EC [8] as the basis of our formalism because it supports logical reasoning [9]. On the one hand, we use inductive reasoning in a *top-down* process in order to conduct the policy decomposition at different levels of abstraction. On the other hand, we employ abductive reasoning in a *bottom-up* process in order to suggest policies that the infrastructure can support. We present a case-study that illustrates the decomposition of rules and demonstrates the performance of our refinement solution.

This paper is organized as follows. Section II describes some background and an overview of our policy refinement toolkit. Section III introduces our formalism for policy refinement. Section IV demonstrates and discusses a case-study. Section V presents the experimental evaluation of our formalism. Section VI outlines the related work and Section VII concludes the paper presenting final remarks and future work.

II. Policy Refinement Toolkit: An overview

In this section, we review our previous work on policy refinement. Because of space constrains, we have limited our scope by considering examples of policies written for QoS management. We represent high-level policies and business-level goals as Service Level Agreements (SLAs). Policy refinement is the process of computing low-level objectives/rules, so-called Service Level Objectives (SLOs), that must meet the high-level goals/policies, *i.e.* SLA $\rightarrow$ SLO_1, SLO_2, ..., SLO_n.

To perform the above translations, we developed a Policy Refinement Toolkit [10], [11] that consists of several components placed inside three fundamental elements: *(i)* an OpenFlow Controller, which collects information from the

978-1-4673-7194-0/15/$31.00 ©2015 IEEE

network infrastructure and that is the key to improve the refinement process, *(ii)* a Policy Authoring Framework where infrastructure-level programmers specify technical characteristics of services and where Business-level Operators write SLAs in a Controlled Natural Language (CNL), and *(iii)* a Repository to store information coming from both the OpenFlow Controller and the Policy Authoring Framework.

The refinement approach is split into two stages (Figure 1). The first stage, called *bottom-up*, consists of the network *information gathering process* (*e.g.*, available bandwidth, routes). A key element of this stage is the OpenFlow controller, which gathers information to store in the Repository. Based on this information, the framework uses *abductive reasoning* to indicate to the business-level operator what are the possible configurations, *i.e.*, QoS classes, that faithfully satisfy the requirements specified by a particular SLA.

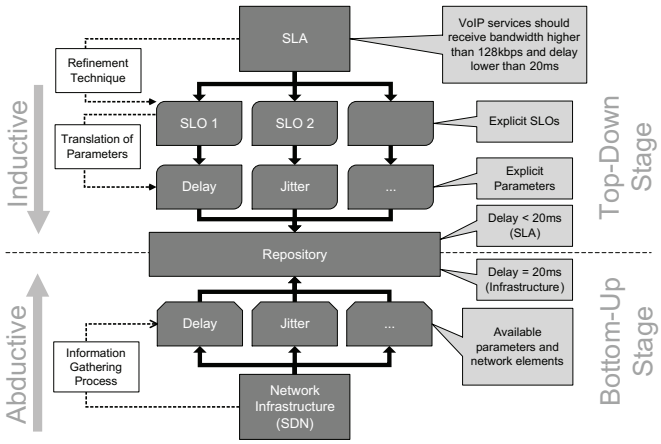

Fig. 1: Deriving SLOs/parameters from high-level goal and gathering network information.

The second stage, called *top-down*, employs a *refinement technique* to translate high-level goals extracted from SLAs into achievable objectives (SLOs). When business-level operators write SLAs, multiple configuration options are offered by the framework, so they can select or customize an existing configuration or even create a new configuration using *inductive reasoning*.

III. An EC-Based Formalism for Policy Refinement

In order to make our refinement model independent of the network controller implementation or policy language, we defined a formal representation of high-level SLA policies using Event Calculus (EC) and applied logical reasoning to model both the system behavior and the policy refinement process for SDN management. It is our aim with this work to assist infrastructure-level programmers to develop refinement tools and configuration approaches to achieve more robust SDN deployments. Due to space constraints some parts of the formalism had to be omitted, especially with respect to notation aspects such as the formal representation of the network infrastructure and QoS classes.

A. Extend Event Calculus

Event Calculus (EC) is a formalism that allows representing and reasoning about dynamic systems. We use the form described by Bandara *et al.* [8], consisting of *(i)* a set of event types, *(ii)* a set of properties (fluents) that can vary over the system lifetime, and *(iii)* a set of time points.

In order to achieve our goals we customized EC with new constants, variables, functions, and predicates as follow:

- **Constants –** these can be defined as SLAs (SLA), services (Serv), classes (Class), parameters/requirements (Par), or objects (Obj_n). Obj may represent a set of objects of the system where *n* represents a source object (Obj_{Src}), a destination object (Obj_{Dst}), an object of link – *i.e.*, the connection between an Obj_{Src} and Obj_{Dst} – (Obj_{Link}), or even a route (Obj_{Route}).
- **Variables –** define V_o to represent the attributes of objects and V_ρ to represent the parameters for the operations supported by objects.
- **Predicates –** specify what the object represents in the system, what is declared about it or relationships between objects. Table I presents the predicates.
- **Operations –** specify actions used with predicates. For example, a query in a repository or the triggering of a phase.

B. Policy Refinement Model

To provide a more targeted case-study, we concentrated our efforts in the support of policy configurations for QoS classes. In order to perform the process of decomposition/translation of an SLA, we use *regexes* as a concise and flexible way for identifying strings of interest such as particular characters (*e.g.*, $>$, $<$, $=$, $\neq$) or words (*e.g.*, high, low, P2P, VoIP, priority, delay, diamond, bronze). We defined the following types of *regexes*: *qos-regexes* to identify QoS classes (*e.g.*, Bronze, Diamond); *service-regexes* to identify services (*e.g.*, VoIP, FTP); *requirements-regexes* to identify service requirements (*e.g.*, Priority, Delay); *adjective-regexes* to identify adjectives in service requirements (*e.g.*, low, high, equal).

Our refinement process consists of a technique that extracts *regexes* from an SLA and decomposes them into Service Level Objectives (SLOs), *i.e.* $\text{SLA}_1 \rightarrow \text{SLO}_{1-1}$, $\text{SLA}_1 \rightarrow \text{SLO}_{1-2}$, ..., $\text{SLA}_1 \rightarrow \text{SLO}_{1-n}$. These SLOs are identified from a query to the repository of *requirements-regexes*, such as delay (D), jitter (J), bandwidth (B), and priority (P), or *qos-regexes*, such as Diamond, or *service-regexes*, such as HTTP. Thus, $\text{SLO}_{1-1} \rightarrow$D, $\text{SLO}_{1-2} \rightarrow$J, $\text{SLO}_{1-3} \rightarrow$B, $\text{SLO}_{1-4} \rightarrow$P, or $\text{SLO}_{1-1} \rightarrow$*qos-regexes*, or $\text{SLO}_{1-1} \rightarrow$*service-regexes*.

In our solution we first describe the SLAs in EC, and use logical reasoning to derive the SLOs and QoS classes based on a system model description. We also model the state of routes and links maintained by the controller, and use logical reasoning to match the best route based on the requirements of the SLA. *Requirements-regexes* are used to search for matching QoS classes (*qos-regexes*) that were previously registered in the repository. We use abductive reasoning to identify possible configurations that achieve the goals specified by the SLA. Thus we can maximize the number of inferences between *requirements-regexes* found in the SLA and the parameters of the QoS classes registered in the repository. The solution proposed performs the matching in the following order:

TABLE I: New Predicates for Event Calculus.

Predicates	Description
object(SLA/Serv/Class/Par/Obj_n)	Used to specify that Obj is an object in the system. Objects can be network elements such as routers, switches, controllers, links, or routes.
isElement(Obj_n)	Holds if Obj_n represents a network element, *e.g.*, switch, controller.
isLink(Obj_{Link}, Obj_{Src}, Obj_{Dst})	Holds if Obj_{Link} represents a link between an Obj_{Src} and Obj_{Dst}.
isRoute(Obj_{Route})	Holds if Obj_{Route} represents a route.
isMemberRoute(Obj_{Route}, Obj_{Link})	Holds if the object, Obj_{Link}, is a member of the route, Obj_{Route}.
isRouterParameter(Obj_{Route}, Par, V_o)	Holds if the object, Par, is a parameter of the route.
isSLA(SLA)	Holds if SLA represents an SLA.
isDescriptionSLA(Serv/Class/Par/Obj_n,SLA)	Defines if Serv/Class/Par/Obj_n is an object contained in the SLA description.
isService(Serv)	Holds if Serv represents a service, *e.g.*, HTTP, VoIP.
isClass(Class)	Holds if Class represents a QoS Class, *e.g.*, gold, silver.
isPar(Par)	Holds if Par represents a parameter/requirement, *e.g.*, delay, priority.
isMemberClass(Class, Serv)	Holds if the object, Serv, is a member of the QoS Class, Class.
isMemberParameter(Class/Serv/SLA, Par, V_o)	Holds if the object, Par, is a parameter of a QoS Class, Service or SLA.
newMemberParameter(Class/Serv/SLA, Par, V_o)	Holds if the object, Par, is a changing or addition of parameter value of a Class, Service or SLA.
attr(SLA/Serv/Class/Par/Obj_n, V_o)	Defines that V_o is an attribute of a Serv, Class, Par, or Obj_n.
state(Obj_n, V_o, Value)	Indicates the state of an object in the system.
operation(Obj_n, Action(V_ρ))	Indicates the operations and functions specified in a policy or event.
systemEvent(Event)	Indicates any event in the system. It is used to trigger the operations and functions.
doAction(Obj_{Src}, operation(Obj_{Dst}, Action(V_ρ)))	Indicates the action performed by Obj_{Src} in Obj_{Dst}.

λ_1 : isMemberParameter(SLA$_n$,P,V$_o$) $\wedge$ isMemberParameter(SLA$_n$,B,V$_o$)
$\wedge$ isMemberParameter(SLA$_n$,D,V$_o$) $\wedge$ isMemberParameter(SLA$_n$,J,V$_o$)

λ_2 : isMemberParameter(Class$_n$,P,V$_o$) $\wedge$ isMemberParameter(Class$_n$,B,V$_o$)
$\wedge$ isMemberParameter(Class$_n$,D,V$_o$) $\wedge$ isMemberParameter(Class$_n$,J,V$_o$)

λ_3 : isMemberParameter(Class$_n$,P,V$_o$) $\wedge$ isMemberParameter(Class$_n$,B,V$_o$)
$\wedge$ isMemberParameter(Class$_n$,D,V$_o$)

λ_4 : isMemberParameter(Class$_n$,B,V$_o$) $\wedge$ isMemberParameter(Class$_n$,P,V$_o$)

λ_5 : isMemberParameter(Class$_n$,P,V$_o$) $\wedge$ isMemberParameter(Class$_n$,D,V$_o$)

λ_6 : isMemberParameter(Class$_n$,B,V$_o$) $\wedge$ isMemberParameter(Class$_n$,D,V$_o$)

λ_7 : isMemberParameter(Class$_n$,P,V$_o$) $\vee$ isMemberParameter(Class$_n$,B,V$_o$)
$\vee$ isMemberParameter(Class$_n$,D,V$_o$) $\vee$ isMemberParameter(Class$_n$,J,V$_o$)

$\phi_1 : \lambda_1 \leftrightarrow \lambda_2$
$\phi_2 : (\lambda_1 \leftrightarrow \lambda_3) \leftarrow \neg\ \phi_1$
$\phi_3 : (\lambda_1 \leftrightarrow \lambda_4) \leftarrow \neg\ \phi_2$
$\phi_4 : (\lambda_1 \leftrightarrow \lambda_5) \leftarrow \neg\ \phi_3$
$\phi_5 : (\lambda_1 \leftrightarrow \lambda_6) \leftarrow \neg\ \phi_4$
$\phi_6 : (\lambda_1 \leftrightarrow \lambda_7) \leftarrow \neg\ \phi_5$

SLA$_n$ is an SLA
Class$_n$ is a QoS Class
V$_o$ is a value of a parameter
λ_n is a predicate

The set of λ rules are used to retrieve the QoS classes that satisfy the largest amount of requirements. Thus, λ_2 will retrieve QoS classes that satisfy all requirements (B, P, D, J), whereas λ_7 will retrieve QoS classes that satisfy at least one of the requirements. The set of ϕ rules is an order of matches that happens until an occurrence of ϕ matches the desired result. Thus, ϕ_1 is an ideal match where all requirements are satisfied while ϕ_6 is a match where at least one requirement is satisfied.

Ultimately, the result presented to the business-level operator is a set of QoS classes ordered by the highest amount of requirements found which can better meet the SLA. In the worst case, the refinement process will propose QoS class(es) which contain at least one of the parameters.

As mentioned previously, the controller, at startup, collects information about the network infrastructure to calculate all possible routes (Route) between two elements. The calculations are carried out for the paths using as weights the bandwidth (B), delay (D), jitter (J), and number of hops (NH) in each link of the network. Thus, the representation of this operation performed by the controller is:

σ_1 : isRouterParameter(Route$_n$,NH,V$_o$) $\wedge$ isRouterParameter(Route$_n$,B,V$_o$)
$\wedge$ isRouterParameter(Route$_n$,D,V$_o$) $\wedge$ isRouterParameter(Route$_n$,J,V$_o$)

σ_2 : isRouterParameter(Route$_n$,B,V$_o$) $\wedge$ isRouterParameter(Route$_n$,D,V$_o$)
$\wedge$ isRouterParameter(Route$_n$,J,V$_o$)

σ_3 : isRouterParameter(Route$_n$,NH,V$_o$) $\vee$ isRouterParameter(Route$_n$,B,V$_o$)
$\vee$ isRouterParameter(Route$_n$,D,V$_o$) $\vee$ isRouterParameter(Route$_n$,J,V$_o$)

Route$_n$ $\leftarrow \sigma_1$
Route$_{n+1}$ $\leftarrow \sigma_2$
Route$_{n+2}$ $\leftarrow \sigma_3$

Route$_n$ is a route

As can be observed, σ_1 is a rule that chooses the route that faithfully fulfills all the parameters of the SLA. σ_2 is an alternative route that does not consider the number of hops. We exclude the number of hops since many services do not recognize this as a main requirement for their proper functioning. Finally, σ_3 is a rule that selects all routes that satisfy at least one of the parameters. Route$_n$, Route$_{n+1}$, and Route$_{n+2}$ are possible routes sorted by the requirements presented in each σ_n. These routes are selected at runtime whenever competing SLAs are detected. This process is performed in order to establish a load balancing or to create a best route to satisfy the requirements of each SLA.

Thus, after any choice, our refinement model will match the indications performed by the business-level operator with the conditions of the network infrastructure as follows:

θ_1 : Class$_n$ $\leftrightarrow$ Route$_n$
θ_2 : Class$_n$ $\leftarrow$ Route$_{n+1}$ $\leftarrow \neg\ \theta_1$
θ_3 : Class$_n$ $\leftarrow$ Route$_{n+2}$ $\leftarrow \neg\ \theta_2$

We have configured the SLA requirements and grouped them into classes, because it is an easier way to deal with the increase in the number of policies that have common goals.

However, if at any time it is discovered that an SLA cannot be fulfilled or if some SLO cannot be achieved, the system is flexible enough to allow the necessary adjustments.

IV. Implementation and Discussion

In this section we demonstrate how our formal model can be used to refine an SLA into a set of low-level configurations. We use the following SLA as a case-study:

HTTP traffic should receive lowest priority and lowest bandwidth.

We represent our refinement model using EC notation and its standard predicates (*e.g., initiates*, *holdsAt*, *happens*) as presented in Bandara *et al.* [8]. For a better understanding of the formal representation of the refinement process, we use friendly names to indicate to the reader where each process occurs in our solution. For example, *PolicyAnalyzer* is used to indicate the module that performs the regex analyzing operation, and *PolicyAuthoringFramework* is used to indicate where this operation occurs. Also, we use lambda (λ_n) to indicate a set of predicates or operations used with predicates. As mentioned previously, when an operator inserts an SLA, our refinement model uses *regexes* – previously stored in the *Policy Repository* – to match the expressions written in semi-structured natural language, and suggests the more appropriate QoS class/classes to the SLA. As a result, suggestions will be displayed to the operator only after all possible matches.

Regarding the refinement process, on the one hand, a *top-down* stage initiates a policy analyzer process aiming to search *qos-regexes* explicitly written in the SLA. In this case, as there are no *qos-regexes* in the SLA, no compatible class will be returned by the query.

λ_n : initiates(doAction(PolicyAuthoringFramework,
operation(PolicyAuthoringFramework,
PolicyAnalyzer(qos-regexes))),state(http,status,enabled),T).

λ_{n+1} ← (happens(doAction(PolicyAuthoringFramework,
operation(Repository,request(qos-regexes))),T+1)
← λ_n).

Following, we search *service-regexes* in the SLA.

λ_{n+2} ← (holdsAt(operation(PolicyAuthoringFramework,
PolicyAnalyzer(service-regexes)),T+2)
← ¬ λ_{n+1}).

At this time, *service-regexes* "HTTP" is found. From this occurrence, we perform a query in the repository to find if a QoS class associated with the HTTP service exists.

λ_{n+3} ← (happens(doAction(PolicyAuthoringFramework,
operation(Repository,request(service-regexes(http))),T+3)
← λ_{n+2}).

λ_{n+4} ← (happens(doAction(Repository,
operation(PolicyAuthoringFramework,return(Class))),T+4)
← λ_{n+3}).

In this case, the HTTP service is not associated with any QoS class. Thus, a search for *requirements-regexes* and their *adjectives-regexes* is performed.

λ_{n+5} ← ((holdsAt(operation(PolicyAuthoringFramework,
PolicyAnalyzer(requirements-regexes)),T+5)
∧ holdsAt(operation(PolicyAuthoringFramework,
PolicyAnalyzer(adjectives-regexes)),T+5))
← λ_{n+4}).

In the given SLA, we found the *requirements-regexes* "priority (P)" and "bandwidth (B)" and the occurrences of *adjective-regexes* "lowest" and "lowest". As our refinement model associates the number of occurrences of *adjectives-regexes* with the number of occurrences of *requirements-regexes*, the toolkit needs to check what are the correct *adjectives-regexes* for the *requirements-regexes* found. We used a factor of proximity to analyze in which position of the SLA each *adjective-regexes* is located relative to the position of the *requirements-regexes*. At this point, we need to associate the *adjectives-regexes* "lowest" with some value in the SLA (*e.g.*, another service). As we have no explicit statement in the SLA indicating that priority and bandwidth in HTTP service must be lower than something, we assume that this service should receive the lowest priority and lowest bandwidth registered in the repository.

λ_{n+6} ← (((happens(doAction(PolicyAuthoringFramework,
operation(Repository,request(requirements-regexes(priority)))),T+6)
∧ happens(doAction(PolicyAuthoringFramework,
operation(Repository,request(adjectives-regexes(low)))),T+6))
∧ (happens(doAction(PolicyAuthoringFramework,
operation(Repository,request(requirements-regexes(bandwidth)))),T+6)
∧ happens(doAction(PolicyAuthoringFramework,
operation(Repository,request(adjectives-regexes(low)))),T+6)))
← λ_{n+5}).

Ultimately, our toolkit applies abductive reasoning to build a query based on *regexes* found in the SLA. This query will be executed and will return to the operator the QoS class that best matches the SLA requirements, in this case, Bronze QoS class as a top choice. Abductive reasoning reaches this conclusion because if the search is for "lowest priority" and "lowest bandwidth", in practice, Bronze QoS class is the class which has lowest priority and lowest bandwidth among the registered QoS classes. After the choice performed by the operator, our toolkit will register in the repository the SLA associating it with the Bronze QoS class.

λ_{n+7} ← (happens(doAction(PolicyAuthoringFramework,
operation(Repository,registerNew(sla(http)))),T+7)
∧ happens(doAction(PolicyAuthoringFramework,
operation(Repository,associateNewSLA(isMemberClass(bronze,http)))),T+7)
← λ_{n+6}).

terminates(λ_{n+7}, state(SLA,status,associated),T+7).

On the other hand, a *bottom-up* stage performs the controller stages. When starting the controller, it monitors the network in order to discover the network elements and their links (StartUp stage). Network devices are instructed to send Link Layer Discovery Protocol (LLDP) packets to report their location in the topology. The controller collects these packets and performs a routine for calculating all possible links between all elements. For each result of this calculation a standard rule is created. As a result, the controller comes to know the position of all network elements and what is the cost (per link) to reach them. This information about links between forwarding elements and their standard rules is stored in the repository.

σ_n : initiates(doAction(Controller,
operation(Controller,startupStage(packets)),state(Controller,status,on),T)).

σ_{n+1} ← (happens(operation(Controller,discoveryTopology(LLDP)),T+1)
∧ happens(operation(Controller,discoveryLinks(LLDP)),T+1))
← happens(doAction(Switch,operation(Controller,sendPacket(LLDP)),T+1))
← σ_n

σ_{n+2} ← (happens(doAction(Controller,
operation(Repository,registerSwitchId(idSwitch)),T+2))
∧ happens(doAction(Controller,
operation(Repository,registerSwitchLink(linkSwitch)),T+2)),
∧ happens(doAction(Controller,
operation(Repository,registerRule(standardRule)),T+2))
← σ_{n+1}.

σ_n is a predicate

Subsequently, the controller reads the QoS classes - which have been previously registered through the policy refinement process – from a repository that contains descriptions and requirements of all services that are initially scheduled to run in the network and the standard rules to address best efforts. The controller compares each QoS class with the links previously analyzed (when the controller was initialized) and creates a spanning tree with the links that have the best possibility to fulfill the needs of each QoS class. Each spanning tree is created by specific rules that will be set up in forwarding devices. These rules are composed by the flow priority (that identifies the order in which the packets should be processed), TCP/UDP destination port (that identifies to which service the packet is addressed), output port (that indicates to which port the switch will send the packet). Also, the controller uses the standard rule to create a spanning tree based on the best links between any two elements. This spanning tree aims to set up best-effort routes to initially address any service that appears on the network without causing transmission delay in the first packets while specific rules for each new service (not provided in an SLA) have not been established yet. Finally, the controller installs each rule in the flow tables of the switches.

$\sigma_{n+3.1}$ ← happens(doAction(Controller,
operation(Repository,request(QoSClass)),T+3))
← σ_{n+2}.

$\sigma_{n+3.2}$ ← happens(doAction(Controller,
operation(Repository,request(standardRule)),T+3))
← σ_{n+2}.

$\sigma_{n+4.1}$ ← happens(doAction(Repository,
operation(Controller,return(bronze)),T+4))
← $\sigma_{n+3.1}$.

$\sigma_{n+4.2}$ ← happens(doAction(Repository,
operation(Controller,return(standardRule)),T+4))
← $\sigma_{n+3.2}$.

$\sigma_{n+5.1}$ ← holdsAt(operation(Controller,
calculateSpanningTreeClass(bronze)),T+5)
← $\sigma_{n+4.1}$.

$\sigma_{n+5.2}$ ← holdsAt(operation(Controller,
calculateSpanningTreeStandard(standardRule)),T+5)
← $\sigma_{n+4.2}$.

$\sigma_{n+6.1}$ ← happens(doAction(Controller,
operation(Switch,registerRule(specificRule)),T+6))
← $\sigma_{n+5.1}$.

$\sigma_{n+6.2}$ ← happens(doAction(Controller,
operation(Switch,registerRule(standardRule)),T+6))
← $\sigma_{n+5.2}$.

$\sigma_{n+7.1}$ ← holdsAt(operation(Switch,writeRule(specificRule)),T+7)
← $\sigma_{n+6.1}$.

$\sigma_{n+7.2}$ ← holdsAt(operation(Switch,writeRule(standardRule)),T+7)
← $\sigma_{n+6.2}$.

Next, the controller enters a stage that stays in a loop awaiting the occurrence of events during the operation of the infrastructure. When running a specific service, such as HTTP, the source host (hostSrc) sends packets to the switch. If there is a specific rule for that type of service, the switch forwards the packet to destination host (hostDst). If there is no specific rule, the switch forwards the packet to hostDst and a copy of the packet to the controller. Subsequently, the controller performs a similar process, as outlined above, checking out from the repository if there is any QoS class establishing specific rules for this new service.

σ_{n+8} ← initiates(doAction(Controller,
operation(Controller,eventsStage(packet)),state(packet,status,on),T+8)).

σ_{n+9} ← happens(doAction(HostSrc,operation(Switch,sentPacket(packet)),T+9))
← σ_{n+8}.

σ_{n+10} ← holdsAt(operation(Switch,verifySpecificRule(packet)),T+10)
← σ_{n+9}.

σ_{n+11} ← (happens(doAction(Switch,operation(HostDst,sentPacket(packet)),T+11)),
∧ happens(doAction(Switch,operation(Controller,sentPacket(packet)),T+11)))
← initiallyFalse(σ_{n+10})

The purpose of the EC-based formalism described in this paper is to formally specify the operation of the policy authoring, controller, and policy refinement processes. Although the logical inferences presented in this work are not integrated with the refinement toolkit, it is our aim as part of our future work to incorporate a Prolog engine as part of the refinement toolkit prototype.

V. EC-Based Formalism Experimental Evaluation

In this section, our goal is to measure the amount of iterations and rules to find QoS classes that fulfill the requirements of different SLAs. The experimental evaluation was performed using Prolog 6.6.4. The experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory.

We created three SLAs by changing the number of expressions according to Table II. We applied the three SLAs to five different repositories A-E and populated the repository according to the number of classes, where, A=5, B=10, C=50, D=100, and E=250 classes. Each QoS class considers all QoS requirements, *i.e.*, priority, bandwidth, delay, and jitter. Each QoS requirement has different values[1].

TABLE II: Description of SLAs used in the experiments.

SLA	Description of SLAs
SLA_1	Streaming traffic should receive highest priority, lowest delay, lowest jitter, and highest bandwidth.
SLA_2	FTP traffic should receive highest bandwidth.
SLA_3	SNMP traffic should receive priority higher than 500 and bandwidth lower than 16kbps.

Regarding the experimental evaluation, each SLA (Table II) generated a set of Prolog rules to find at least one occurrence of a QoS class. The generated Prolog rules are based on the matching process presented in Section III-B.

[1]The values for QoS requirements were generated randomly: between 0 and 999 for priority (where 0 is highest priority and 999 is lowest priority), between 2 kbps and 2^{12} kbps for bandwidth, between 1 ms and 999 ms for delay, and 10% of the delay value for jitter.

Table III shows the relationship between the number of iterations generated by each rule and the number of classes found. As can be seen, in all cases at least one class has been found by a rule. In addition, some rules have found multiple classes such as S3R1. This happens because each rule aims to find an ideal match of parameters ignoring the other parameters registered in the class. Thus, a rule aiming values of priority = 12 and bandwidth = 128kbps can identify classes with the following information: $Class_1$: priority = 12, bandwidth = 128kbps; $Class_2$: priority = 12, bandwidth = 128kbps, and delay = 10ms; $Class_3$: priority = 12, bandwidth = 128kbps, delay = 10ms, and jitter = 1ms.

TABLE III: Number of iterations and classes found for each scenario.

SLA	Rule*	Number of Classes									
		5		10		50		100		250	
		NI**	NC***	NI	NC	NI	NC	NI	NC	NI	NC
1	S1R1	158	1	332	0	2617	0	4747	0	11573	1
	S1R2	N/A	N/A	332	0	2617	0	4747	0	N/A	N/A
	S1R3	N/A	N/A	226	0	2111	0	2232	1	N/A	N/A
	S1R4	N/A	N/A	252	1	2157	1	N/A	N/A	N/A	N/A
2	S2R1	68	1	289	2	1743	1	2137	2	13563	6
3	S3R1	14	2	33	5	125	20	293	45	718	107

* Rule is the short representation of a rule where S = SLA and R = Rule. Thus, S1R1 is the first rule for SLA_1. ** NI = number of iterations performed by each rule. *** NC = number of classes found in each rule. N/A means that a previous rule found a class.

As can be observed in Table III, by increasing the number of classes the amount of iterations for each repository grows. This increase is visible in all experiments performed with the SLAs. This behavior is expected, since the number of classes influences the number of iterations to obtain the ideal matches between SLAs and QoS classes. In addition, the SLAs requiring extreme values, *e.g.*, lowest delay, highest bandwidth also increase the number of iterations because they needed to find lowest or highest values.

VI. Related Work

The use of PBNM and policy refinement in computer networks has been investigated for over a decade. Bandara *et al.* [5] introduced the use of goal design and abductive reasoning to decompose strategies that target a specific high-level goal. Policies can be refined by grouping strategies with events and restrictions. The authors presented tool support for the refinement process, and used examples of DiffServ QoS management. Craven *et al.* [6] presented a policy refinement process for authorization and obligation that involves stages of decomposition, operationalization, re-refinement, and deployment. The work described in detail how a formalization of UML information on system objects, a high-level policy, and decomposition rules that relate actions can produce concrete low-level policies. Rubio-Loyola *et al.* [7] presented a goal-oriented approach for goal decomposition using KaOS. The refinement approach makes use of linear temporal logic and reactive systems analysis techniques, thus generating deployable policies in Ponder. Foster *et al.* [12] introduced a new language for network programming supporting OpenFlow called Frenetic. Frenetic has a set of operators for handling network traffic flows, and a runtime that abstracts the details related to (un)installing low-level rules in switches.

In summary, despite these efforts, several policy refinement techniques were limited by the characteristics of traditional IP networks, such as best-effort packet delivery and distributed control state. However, we argue in this paper that refinement techniques may be better explored by relying on the characteristics of SDN to enhance the policy refinement process.

VII. Concluding Remarks

In this paper we advocate that SDN, compared to traditional networks, can provide more features/information to perform more accurate policy refinement. We demonstrated and discussed an EC-based formalism and presented a case study and examples where we have described phases, processes, and operations executed by the controller, as well as the process of policy authoring. In addition, we performed experiments to analyze the amount of iterations and suggestion of classes for the rules created by our solution. As future work, we intend to analyze the generality of the formalism when managing other resources and types of services, such as access control, and load balancing.

References

[1] J. Wickboldt, W. Jesus, P. Isolani, C. Both, J. Rochol, and L. Granville, "Software-Defined Networking: Management Requirements and Challenges," *IEEE Communications Magazine - Network & Service Management Series*, January 2015.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[3] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, no. 1, pp. 477–489, 2012.

[4] D. C. Verma, "Simplifying network administration using policy-based management," *Network, IEEE*, vol. 16, no. 2, pp. 20–26, 2002.

[5] A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, and G. Pavlou, "Policy refinement for diffserv quality of service," *IEEE eTransactions on Network and Service Management*, vol. 3, no. 2, 2005.

[6] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Policy refinement: Decomposition and operationalization for dynamic domains," in *Network and Service Management (CNSM), 2011 7th International Conference on*, Oct 2011, pp. 1–9.

[7] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, "A functional solution for goal-oriented policy refinement," in *Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop on*, June 2006, pp. 133–144.

[8] A. Bandara, E. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, 2003, pp. 26–39.

[9] M. Shanahan, "An abductive event calculus planner," *The Journal of Logic Programming*, vol. 44, no. 1, pp. 207–240, 2000.

[10] C. C. Machado, L. Z. Granville, A. Schaeffer-Filho, and J. A. Wickboldt, "Towards SLA policy refinement for QoS management in software-defined networking," in *Advanced Information Networking and Applications (AINA-2014), 2014 28th IEEE International Conference on*. IEEE, May 2014.

[11] C. C. Machado, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho, "Policy authoring for software-defined networking management." *Proceedings of the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM 2015). 11-15 May 2015, Ottawa, Canada. (to appear)*, 2015.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," *SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, Sep. 2011.