Policy Authoring for Software-Defined Networking Management

Cristian Cleder Machado, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Alberto Schaeffer-Filho
Computer Networks Group – 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) permits centralizing part of the decision-logic in controller devices. Thus, controllers can have an overall view of the network, assisting network programmers to configure network-wide services. Despite this, the behavior of network devices and their configurations are often written for specific situations directly in the controller. As an alternative, techniques such as Policy-Based Network Management (PBNM) can be used by business-level operators to write Service Level Agreements (SLAs) in a user-friendly interface without the need to change the code implemented in the controllers. In this paper, we introduce a framework for Policy Authoring to *(i)* facilitate the specification of business-level goals and *(ii)* automate the translation of these goals into the configuration of system-level components in an SDN. We use information from the network infrastructure obtained through SDN features and logic reasoning for analyzing policy objectives. As a result, experiments demonstrate that the framework performs well even when increasing the number of expressions in an SLA or increasing the size of the repository.**

***Keywords*—policy authoring; policy refinement; PBNM; SDN;**

I. Introduction

Software-Defined Networking (SDN) [1] has simplified network administration by logically centralizing part of the decision making process in a controller element, such as an OpenFlow controller [2]. Controllers have an overall view of the network, which assists network operators in managing network-wide services [3]. However, we argue that SDN alone does not satisfactory improve the network operator's ability of writing concise yet expressive rules for network management. Despite the benefits of SDN, the intended network behavior is usually defined by static rules written to cope with specific situations [4][3]. This ends up hindering the development and deployment of new network services. Moreover, to deal with diverse situations, the amount of rules can become prohibitive.

An approach to tackle this problem is the use of Policy-Based Network Management (PBNM) [5][6]. In PBNM, an operator specifies infrastructure goals and constraints in the form of high-level policies to guide the behavior of the network. The use of PBNM aims to reduce the complexity of network management tasks [7]. Techniques such as policy refinement can be used to automatically translate high-level policies into a set of low-level policies for configuration of various devices of a system [8]. The use of PBNM in computer networks has been investigated for over a decade [8][9]. However, PBNM and policy refinement in the novel context of SDN has been much less explored [4][10]. We argue that PBNM for SDN management is still in its infancy, and the work in the area often ignores the vast literature on PBNM produced before the advent of SDN. Thus, several aspects of PBNM can be exploited toward more flexible SDN management.

We introduce in this paper a Policy Authoring framework for SDN management in which operators write high-level policies (expressed in a Controlled Natural Language - CNL [11]) that are refined into lower-level ones. In previous work, we have customized features of an OpenFlow controller aiming to collect information about the network infrastructure that can assist in improving the refinement process [12]. The policy authoring process introduced in this paper is a further step toward a comprehensive policy refinement toolkit for SDN which enables refining policies into a set of rules to be deployed by our customized OpenFlow controller. Policy refinement is accomplished by using logical reasoning [13] for analyzing policy objectives. On the one hand, inductive reasoning indicates the goals that should be extracted and fulfilled at lower-levels of abstraction. On the other hand, abductive reasoning confronts these goals with the network characteristics obtained from an SDN controller to indicate whether the network infrastructure can accommodate such goals. We developed a prototype as a proof-of-concept.

The main contributions of this paper are: *(i)* refined policies with minimal human intervention; *(ii)* analysis of the infrastructure's ability to fulfill the requirements of high-level policies; *(iii)* decreased amount of network rules coded into the controller; and *(iv)* management and deployment of new rules with minimal disruption to the network.

In this paper, we have limited the scope of our experiments and evaluation to QoS management. However, the principles of policy authoring presented here can be more generally applicable to other areas. We define three different Service Level Agreements (SLAs) by changing the number of expressions (amount of requirements, values, services, and QoS classes) and present experiments in five different scenarios. Results demonstrate that the policy authoring framework performs well, even when increasing the number of expressions in an SLA or increasing the size of the repository of rules.

This paper is organized as follows: in Section II we provide a briefly description of the main concepts used in our work. In Section III we present details of our policy authoring framework for SDN management. In Section IV we present the experiments and a discussion about the achieved results. In Section V, related work is discussed. Finally, in Section VI we conclude the paper with final remarks and future work.

II. Background: A Toolkit for Policy Refinement

In this section, we present an overview of the main concepts, techniques and elements used in our policy refinement toolkit. We also briefly describe our previous work [12], identifying the benefits of replacing traditional network architectures with SDN. In order to make the policy refinement toolkit independent of the network controller implementation or policy language, we defined a formal representation of high-level SLA policies using Event Calculus (EC) [14] and applied logical reasoning [13] to model both the system behavior and the policy refinement process for SDN management. The formalization aspects of our work are described elsewhere [15].

A. Policy Refinement Toolkit: An Overview

Our toolkit (see Figure 1) consists of three main elements: *(i)* a policy authoring framework (described in Section III), which is used by infrastructure-level programmers to specify the technical characteristics of services and by business-level operators to write SLAs in a *controlled natural language* (CNL) [11]; *(ii)* an OpenFlow controller, which collects information from the network infrastructure (which is the key to improve the refinement process); and *(iii)* a repository to store information from both the controller and the framework.

Our policy refinement toolkit is based on the research efforts of Bandara *et al.* [16] and Craven *et al.* [17]. These studies were limited by the characteristics of traditional networks, such as the notion of best-effort for QoS and the lack of a centralized control plane [3]. In traditional networks, the control plane is executed in each network device. Also, each device has its proprietary protocols thus becoming difficult to be programmed.

Instead, our approach introduces the use of Software-Defined Networking (SDN) [18][19] to enhance the refinement process. In SDN, these limiting factors of traditional networks can be overcome, since SDN is mainly characterized by a clear separation between the forwarding and control planes. Thus, differently from traditional networks, SDN has a logically centralized control plane which allows moving part of the decision-making logic of network devices to external controllers. This provides controller devices with the ability to have an overall view of the network and its resources, thus becoming aware of all the network elements and their characteristics [19][1]. Based on this centralization, network devices become simple packet forwarding elements, which can be programmed through an open interface, such as the OpenFlow protocol [2] SDN architecture in which network traffic information is centralized by a controller is valuable to our policy refinement approach. It makes it easier to retrieve information from the network infrastructure, and to validate SLA requirements more accurately.

We apply the concept of *logical reasoning* [13] to support the business-level operator in the refinement of an SLA. Logical reasoning has three modes:

- In *deductive reasoning*, a conclusion is reached by using a rule that analyzes a premise. For example, if streaming packets are transmitted the network becomes slower; streaming packets are being transmitted now; therefore, the network is slower.
- In *inductive reasoning*, the goal is to identify a rule, starting from a historical set of conclusions generated from a premise. For example, every time streaming packets are transmitted the network becomes slower; so, if streaming packets are transmitted tomorrow, the network will be slower.
- In *abductive reasoning*, starting from a conclusion and a known rule, we can explain a premise. For example, when streaming packets are transmitted the network becomes slower; the network is slower now; so, possibly streaming packets are being transmitted.

Our policy refinement toolkit uses two modes of logical reasoning: on the one hand, inductive reasoning indicates the SLA requirements that should be extracted and fulfilled at lower-levels of abstraction. On the other hand, abductive reasoning compares these requirements with the network characteristics obtained from an SDN controller to determine whether the network infrastructure can accommodate such requirements.

B. Low-level Controller Configuration

In order to support our policy refinement approach it was necessary to customize some functionality in the OpenFlow controller. This was required for collecting information about the network infrastructure, which is later used, for example, to calculate optimal routes. This customization was based on SDN native features only, and thus can be applied to any controller implementation. Therefore, our solution is not tied to any specific controller design or language. For example, topology discovery, which is available in POX [20] is a native feature of SDN offered by all controllers in different implementations. We emphasize that even though the commands supported by the forwarding elements are standardized, the controllers require different programming languages and/or support different features. This difference between controllers can reflect in the effort for customization that must be employed by an infrastructure-level programmer.

Thus, the behavior of the customized OpenFlow controller is divided into three phases: *(i) Startup Phase*: discovers services and possible paths between network elements and writes rules (which we call *standard rules*) in the flow table of the switches that are in the shortest path between each of the network elements; *(ii) Events Phase*: stays in a loop during the operation of the infrastructure to identify service events and determine the shortest path based on the characteristics of the network and service requirements; and *(iii) Analysis Phase*: implements the rules and monitors the network in order to identify possible enhancements for the active flows.

Ultimately, the policy authoring framework (described in detail in Section III) can be used to derive QoS class requirements from business-level SLAs. Policy authoring performs a process of computing low-level objectives/rules (SLOs Service Level Objectives) that must meet the high-level goals/policies. Then, the customized controller is capable of assigning the specific type of network traffic described by the SLA to its optimal route, given the set of requirements derived from the SLA. This QoS management strategy is based on routing (using the best path between network devices). The calculations for the best path are carried out using as weights/requirements the bandwidth, delay, jitter, and number of hops in each path of the physical topology. Each weight/requirement is presented in order of importance. If only one occurrence of a

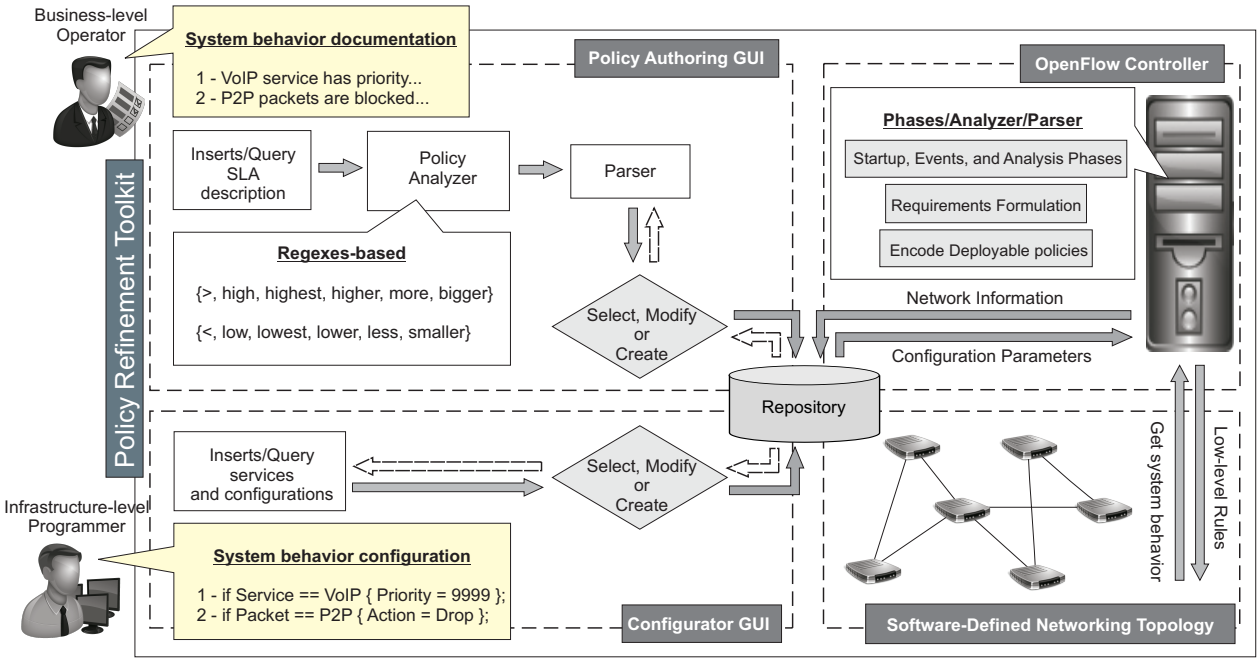


Fig. 1: Overall Policy Refinement Toolkit.

weight/requirement is found, it will be chosen. Otherwise, the path that satisfies the largest number of weights/requirements – compared with the QoS class requirements identified in the policy authoring process – is chosen as the best path. The best path is part of the rule that will be configured by the controller into the flow table of each switch later. Besides informing which one is the best path, each rule receives the established priority in each QoS class. This is what establishes fairness and the distinction between traffic deserving high and low priority in the network. In summary, our strategy uses a requirements-based path and a priority for deciding routes.

Each rule is deployed in the flow table of the network device at runtime, aiming to minimize disruption of the network. Periodically, the controller checks if the configured paths remain the best choices, aiming to reduce processing overhead in network devices. In our experiments, each rule is configured with a timeout. We also set the checking intervals to, for example, *30s*. If at any time the controller identifies that there is a better alternative path, new rules are sent to the switches. If the current path remains the best, the controller only increases the value of the timeout for the rules on each network device. For further details we refer the reader to Machado *et al.* [12].

C. Policy Repository

The *Policy Repository* stores information about the behavior of the infrastructure, which is obtained during the network configuration process. For example, the repository stores all the possible links between elements, number of elements, bandwidth, delay and jitter.

Additionally, the repository maintains a list of all services and their parameters (*e.g.*, the packet identifier of the HTTP protocol), all QoS classes and services associated with them. We use standard TCP/IP information from packet headers to register a given service in the repository.

III. Policy Authoring for SDN

This paper extends our previous work on a refinement toolkit for high-level policies in SDN. Details on the low-level controller operation have been described in Machado *et al.* [12], and in this paper we focus on the policy authoring aspects only. In this section we describe in detail our Policy Authoring framework for SDN management. The main goal is to enable operators to express business goals, *e.g.*, Service Level Agreements (SLAs), without having to specify in detail what elements of the network infrastructure should receive the configurations and how they should be configured.

To provide a more targeted case-study, we concentrated our efforts in the support of policy configurations for QoS classes. The result obtained from the refinement of high-level policies are QoS-class requirements. Thus, the interpretation of an SLA is used for extracting the Service Level Objectives (SLOs). These SLOs are considered QoS-class requirements (*e.g.*, priority, bandwidth) by the Policy Authoring framework.

A. Controlled Natural Language

In this paper, we identify the business-level goals and high-level policies as SLAs. We introduce a *controlled natural language* (CNL) [11] to establish restrictions and requirements for writing business-level goals. The grammar of this language is defined below:

Listing 1: Grammar of the controlled natural language.

```
1 Language : →(<QoS>|<Service>)<Preposition><Expression>
2 QoS : →qos−regexes
3 Service : →service−regexes
4 Preposition : →should receive|should not receive
5 Expression : →<Term>|<Term><Connective><Expression>
6 Term : →<Parameter><Operator><Value>
7 Parameter : →requirements−regexes
8 Connective : →And|Or
9 Operator : →adjective−regexes
```

Our Policy Authoring framework uses *regexes* as a concise and flexible way of identifying strings of interest such as particular characters (*e.g.*, $>$, $<$, $=$, $\neq$, $\leq$, $\geq$) or words (*e.g.*, high, low, http, ftp, gold, silver). We defined the following types of *regexes*: `qos-regexes`: regular expression to identify QoS classes; `service-regexes`: regular expression to identify services; `requirements-regexes`: regular expression to identify service requirements; `adjective-regexes`: regular expression to identify adjectives in service requirements. Table I shows examples of regular expressions that can be contained in an SLA.

TABLE I: Examples of regular expression.

Type	*Expression*	*Operator*
qos-regexes	Bronze, Silver, Gold, Platinum...	N/A
service-regexes	VoIP, Streaming, HTTP, FTP, SMTP, POP, P2P...	N/A
requirements-regexes	Priority, Bandwidth, Delay, and Jitter	N/A
adjective-regexes	more, high, higher, up, over...	$>$
	equal, like, even, same, similar...	$=$
	less, low, lower, down, below...	$<$

B. Bottom-up and Top-down Phases

We specifically introduce in this paper a policy authoring framework where infrastructure-level programmers specify technical characteristics of services, and business-level operators write SLAs in a controlled natural language. This framework and a customized controller [12] compose a refinement toolkit of high-level policies for SDN management. All aspects of the refinement process both in the framework as in the controller are automatically performed. The results generated by the refinement process are a set of rules to be deployed by the controller for the network infrastructure configurations. This toolkit is integrated with a formal representation based on Event Calculus (EC) and applies logical reasoning to model both the system behavior and the policy refinement process in SDN. This formalism assists infrastructure-level programmers to develop refinement tools and configuration approaches to achieve more robust SDN deployments. The EC-based formalism is described in [15].

The refinement process is split into two phases (Figure 2): The first phase, called bottom-up, consists of the network information (*e.g.*, bandwidth, delay) gathering process. A key element of this phase is the OpenFlow controller, which performs the data collection process. Using this information, the Policy Authoring framework uses abductive reasoning to indicate to the business-level operator what are the possible configurations. These indications are provided through settings performed previously – other SLAs or policies created manually by the operator – along with the characteristics that the network can support. More details about Policy Authoring are described in Section III-C.

The second phase, called top-down, refines high-level goals extracted from SLAs and translate them into achievable goals (SLOs). An operator writes the SLAs and creates – if necessary – the QoS classes needed to fulfill them. As mentioned

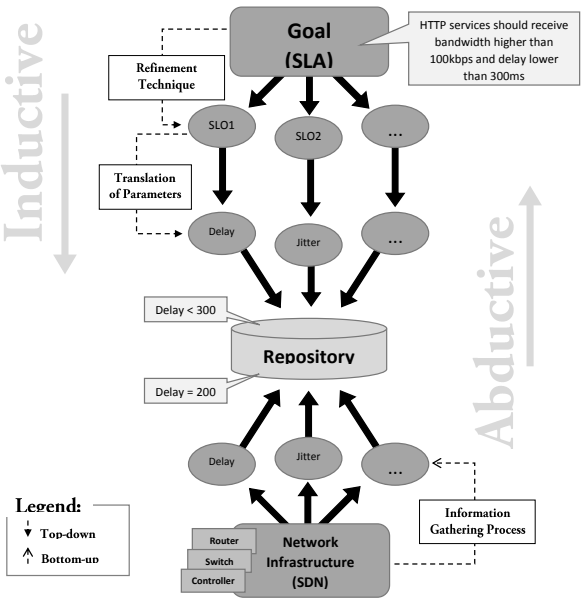


Fig. 2: Deriving SLOs/parameters from goal and gathering network information.

previously, the bottom-up phase will try to indicate using abductive reasoning which are the best configurations for the SLA that is being written. Thus, multiple configuration options will be offered to the operator, who can select or customize an existing configuration, or even create a new configuration.

C. Policy Authoring Framework

Regarding the Policy Authoring framework, the operator inserts an SLA that defines explicitly or implicitly business-level goals. When inserting each policy, the *Policy Analyzer* component (Figure 1) uses *regexes* (regular expressions) – previously stored in the *Policy Repository* – to match the expressions written in natural language, and suggests the more appropriate QoS class/classes to the SLA. The operator can create a set of QoS classes beforehand. Each class may have a number of QoS requirements. For example, Gold QoS class may contain priority = 20, bandwidth = 512kbps, delay = 2ms, and jitter = 1ms, while Bronze QoS class may contain bandwidth = 2kbps.

This Policy Authoring framework relies on abductive reasoning to suggest QoS classes. As mentioned previously, in abductive reasoning, starting from a *conclusion* and a known *rule*, it is possible to explain a particular *premise*. We use the following SLA to illustrate how logical reasoning works and, subsequently, we use the same SLA to explain how the Policy Authoring operates:

"HTTP services should receive bandwidth higher than 100kbps and delay lower than 300ms".

The *conclusion* of this SLA is *"HTTP services should receive"* certain characteristics. The *rules* for reaching this conclusion are *"bandwidth $>$ 100kbps"* and *"delay $<$ 300ms"*. Thus, we present the *premise* (QoS class in the repository) that has this rule and which can *possibly* arrive at this conclusion.

We define a query that assigns weights to results based on the importance of the *regexes* contained in the SLA.

These expressions are compared to the information stored in the repository to sort the results and display them. The ordering thus follows: *(i)* expressions related to QoS classes; *(ii)* expressions related to services, and *(iii)* expressions related to service requirements. The steps to query and display the information to the business-level operator are the following:

Step1 – Check if there is any `qos-regexes` expression in the SLA indicating a class, *e.g.*, QoS Gold, Silver. If there are occurrences of these expressions, the Policy Authoring framework returns the QoS class values, based on the identified `qos-regexes`. For the SLA presented in the example, we have no expressions of this type.

Step2 – Check if there is any `service-regexes` expression in the SLA relating to services, *e.g.*, FTP, VoIP. If there are occurrences of these expressions, the Policy Authoring framework returns the QoS class values to which the services are associated. For the SLA in the example, there is a `service-regexes` (HTTP), which may be associated with a QoS class in the repository.

Step3 – Analyze the expressions indicating service requirements, *e.g.*, priority, bandwidth. If there are occurrences of these expressions, the Policy Authoring framework performs the following operations: *(i)* identify and count the `requirements-regexes` found, and *(ii)* identify and count the `adjective-regexes` that come before and after any `requirements-regexes`.

We also developed a technique for identifying and counting the `requirements-regexes`, which allows the operator to optimally match the `adjective-regexes` found with their respective requirements. In the SLA above, we can observe the `adjective-regexes` *higher* and *lower*, which are related to the `requirements-regexes` *bandwidth* and *delay*, respectively. The *Policy Analyzer* identifies any `adjective-regexes` and examines the SLA, identifying the proximity of the `adjective-regexes` referring to `requirements-regexes`. This is performed by checking if `adjective-regexes` are located before or after `requirements-regexes`. At the end, the result is presented to the business-level operator.

The Policy Authoring framework uses abductive reasoning to show what are the best configurations for the SLA. Thus, the *Policy Analyzer* can identify, for example, that there is already a QoS class configured with low delay, or that the throughput for the specified network path already exceeds the network configuration, indicating that the policy should be reformulated. Also, the operator can be warned of potential conflicts or even non-compliance with policies. If the operator chooses one of the suggested QoS classes, the Policy Authoring framework will store the information extracted from the SLA, *e.g.*, the service, with the selected class.

Suggestions provided through abductive reasoning are not mandatory. If after analyzing them the operator decides they do not meet the high-level goals, the suggestions can be ignored. At this point, the operator can analyze the information presented by the Policy Authoring framework and rely on inductive reasoning to perform the following actions:

- *Modify existing policy/class* – This action allows the operator to change a predetermined parameter, *e.g., priority = 100* to *priority = 101*, or add a parameter that does not yet exist, *e.g., delay* $\leq$ *120ms*. This modification may impact other policies, and the *Analyzer* uses inductive reasoning to identify the classes in the repository that may be impacted. Thus, the operator has the opportunity to analyze policy-by-policy and decide if the change is viable or not.
- *Create policy/class based on existing class* – This action is an alternative to modifying an existing class. Through this action, a new class created by the operator inherits the parameters of an existing class, which can be customized as needed. The *Policy Analyzer* uses inductive reasoning to automatically check if the parameter values of this new class are not identical to the ones in an existing class in the repository. If so, the existing class is returned instead.
- *Create a new policy/class* – The creation of new classes can be conducted *(i)* if a class that meets the objectives of the SLA does not exist, or *(ii)* if the parameters of other classes retrieved via abductive or inductive reasoning are not related to the objectives of the new SLA. Thus, the operator can set the new class parameter-by-parameter to meet the SLA objectives.

After any of the actions above is executed, the *Parser* component (Figure 1) will be executed and the policies/classes will be stored in the *Policy Repository*. It is based on this information that the Policy Authoring framework estimates the amount of allocated traffic per class and warns if the infrastructure can support or not new policies. Further, the policy repository contains a list of services associated with their TCP/UDP port (*e.g.*, HTTP = [80,8080], SSH = [22], SMTP = [25,587]). This list was created using RFC 1700 [21]. Subsequently, the OpenFlow controller reads from the repository these new policies starting the *Analysis Phase* for setting up the appropriate rules in forwarding devices, as explained in Section II-B.

IV. Prototype and Evaluation

In this section we describe a prototype implementation and evaluation of our Policy Authoring framework. For details about the low-level controller implementation we refer the reader to our previous work [12].

A. Prototype Implementation

We developed the prototype using the Django web framework[1]. We chose Django due to its support to the Python language and the support it provides to create web applications. For the interface design we used the Bootstrap front-end framework[2]. The prototype is split into two modules, Policy Authoring GUI and Configuration GUI, described as follows.

1) Policy Authoring GUI: We developed a user-friendly interface for Policy Authoring in order to allow the configuration of the network through business goals. Thus, a business-level operator can use the Policy Authoring GUI to express high-level goals and receive feedback from his/her requests.

Figure 3 illustrates the home screen of the Policy Authoring GUI. It presents statistics about the number of policies, classes,

[1] http://www.djangoproject.com/
[2] http://getbootstrap.com/

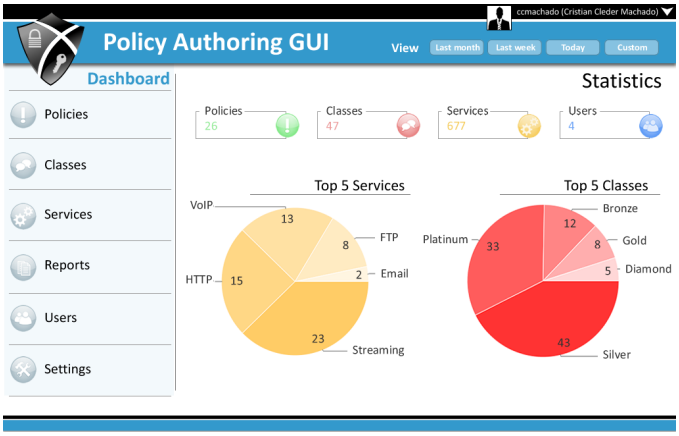


Fig. 3: Policy Authoring GUI Dashboard.

services, and users registered. In addition, it shows two charts about the top 5 services that most appear in policies and the top 5 QoS classes that most have linked policies. The *dashboard* is composed of the following items:

- *Policies* – Used by business-level operators to create, search, edit, remove, enable or disable policies. Operators can also associate a high-level SLA with the QoS class that best meets the SLA requirements.
- *Classes* – Used to specify QoS classes. Infrastructure-level programmers and business-level operators can perform the necessary parameter settings for each class.
- *Services* – Used by infrastructure-level programmers to record, edit, and delete services. Also, through this interface a service can be associated with a QoS class.
- *Reports* – Used by business-level operators and infrastructure-level programmers to view reports of policies, services, and classes. For example, classes that contain most policies or services that appear less frequently in policies. Additionally, some reports can be filtered by specific parameters, *e.g.,* priority, delay.
- *Users* – Used to create, search, edit, remove, enable or disable system users.
- *Settings* – Used to configure system settings, such as database connection information.

2) Configurator GUI: Our aim is to facilitate not only the description of business objectives but also the configuration of the infrastructure. The *Configurator GUI* is designed to manage the registration of services and parameters. An infrastructure-level programmer inserts service information, such as *ServiceName* and *Port* (as used in TCP/IP). Subsequently, the infrastructure-level programmer may create QoS classes with parameters and their respective values. The fields that may be informed are *ClassName*, Priority, Bandwidth, Delay, and Jitter.

We decided to group services by class, thus after QoS classes have been defined, each service is associated with a QoS class. This step is important because if services are previously associated with some class, the toolkit will have a better performance since there will be an entry in the repository for a group of services as opposed to one entry for each service. Thus, services with similar requirements can be grouped into a single class while maintaining fairness among competing in the same link.

B. Evaluation

We present in this section experiments and initial results obtained with the implemented toolkit. Our goal is to measure the response time of the end-to-end process, *i.e.*, from policy authoring to deployment of low-level rules in the controller device. In order to perform the experiments, we created three SLAs (Table II) by changing the number of expressions, where SLA 2 has more expressions than SLA 1 and SLA 3 has more expressions than SLA 2. Our goal is to show the robustness and efficiency of the refinement process when we increase the number of expressions that should be compared. We also created three scenarios (Table III) by varying the number of network devices and adding redundant links between some network devices. The scenarios used in the experiments were based on mesh topologies. Our goal was to demonstrate the ability of the framework to operate in increasingly large topologies. These scenarios were created using the Mininet emulator and experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory.

TABLE II: Description of SLAs used in the experiments.

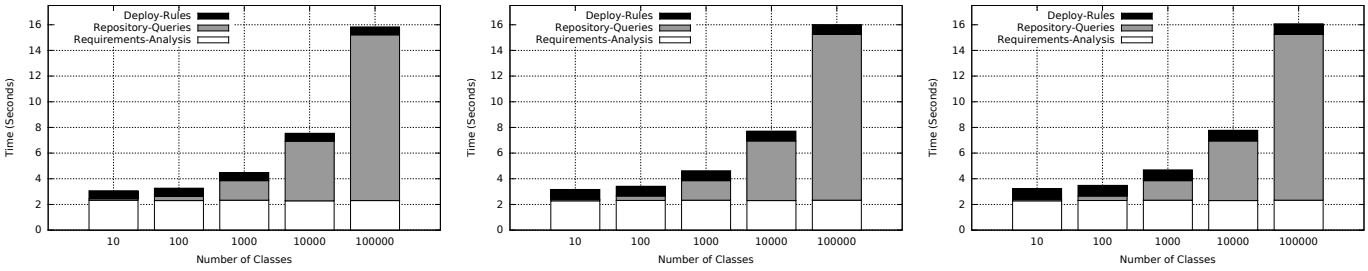
SLA	Description of SLAs
SLA_1	HTTP traffic should receive lower Quality of Service and low priority compared with other services.
SLA_2	Streaming traffic should receive higher priority, low delay and bandwidth higher than 512kbps.
SLA_3	VoIP traffic should receive higher priority, delay less than 200ms, low jitter, and bandwidth higher than 128kbps.

TABLE III: Number of switches and links in each scenario.

Scen.	SwL0	SwL1	SwL2	SwL3	SwL4	Hosts	Links
X	16	8	8	4	0	32	88
Y	32	16	16	8	4	64	176
Z	64	32	32	16	8	128	210

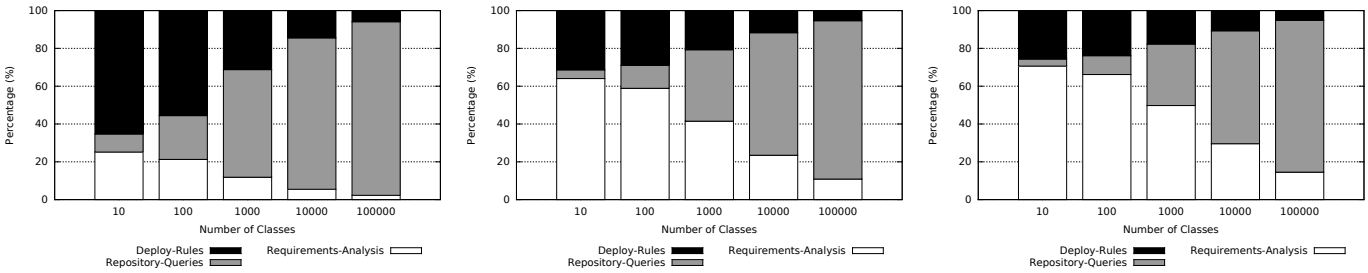
We applied the three SLAs to five different repositories A-E and populated each repository according to the number of classes, where $A = 10$, $B = 100$, $C = 1,000$, $D = 10,000$, and $E = 100,000$ classes. In addition, each experiment was executed thirty times. We performed experiments on all variations of SLAs, repositories, and scenarios. Due to space limitations, we present the most relevant results only. In particular, the experiments described in this sections intend to evaluate our prototype in terms of average execution time and percentage of the total time occupied by each stage of the policy authoring process.

Figure 4 shows the average response time for SLA 3 in each scenario. We break the total execution time down in three categories, namely requirements analysis (*i.e.*, parse the SLAs and their regexes), repository queries (*i.e.*, search for the best matching QoS class), and deploy rules (*i.e.*, install the flow rules in the controller). By increasing the number of classes, it is possible to observe that the average time spent performing repository queries also grows. This increase is visible in all



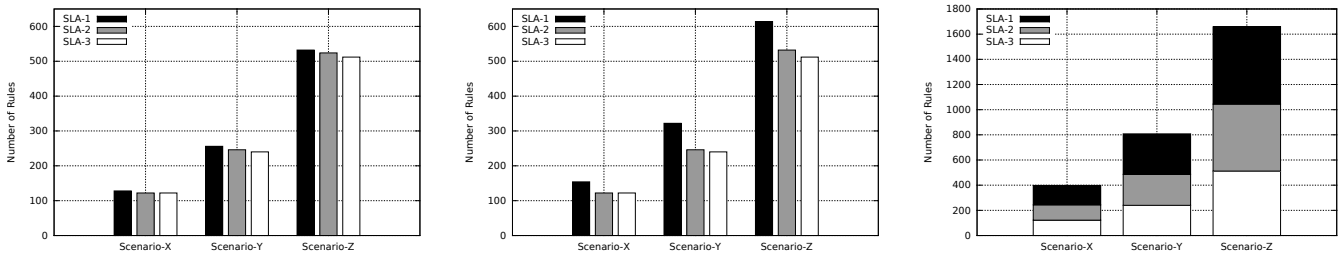
(a) SLA 3 (scenario X) (b) SLA 3 (scenario Y) (c) SLA 3 (scenario Z)

Fig. 4: Average response time for SLA 3 performed in scenarios X, Y, and Z.



(a) SLA 1 (scenario Z) (b) SLA 2 (scenario Z) (c) SLA 3 (scenario Z)

Fig. 5: Percentage of total time for each experiment performed in scenario Z.



(a) Total number of rules deployed by each SLA performed separately in each scenario. (b) Total number of rules deployed by each SLA performed simultaneously in each scenario. (c) Total amount of rules deployed by all SLAs in each scenario.

Fig. 6: Number of rules deployed in each scenario.

experiments performed with SLAs 1, 2, and 3. This behavior is expected, since the number of classes has influence on the number of queries to obtain the ideal matches between SLAs and QoS classes.

In Figure 5 the y-axis shows the percentage of the total time occupied by each process in the experiments performed with SLAs 1, 2, and 3 in scenario Z. From these results it is possible to note that, according to the level of complexity of each SLA, the percentage of time for analyzing requirements also increases. This happens due to the increase in the number of occurrences of regular expressions found in each SLA.

Figure 6(a) shows the total number of rules generated by refining each SLA *separately* in each scenario. As can be observed, each SLA generates practically the same number of rules in each scenario. SLA 1 shows a small difference in the number of rules deployed compared to SLAs 2 and 3. This occurs because SLA 1 has lower QoS requirements (*i.e.,* low priority) compared to other services, which causes the choice of routes with more hops and consequently causes rules to be deployed in more devices. It is worth mentioning that the total number of rules generated by the policy authoring framework is smaller than the total number of rules that would have to be manually created on all network devices. This is because, as our approach is based on routing, it creates a spanning tree to find all routes between sources and destinations. Thus, some routes may be common between different sources and destinations. As a result, a number of switches do not need to be configured, thus reducing the total number of rules required in each scenario.

The growth in the total number of rules in SLA 1 appears more clearly when we performed *simultaneously* the refinement of the three SLAs in each scenario (Figure 6(b)). Our framework attempts to fulfill the requirements of each SLA. In order to achieve this, it identifies the possibility of routing (balancing) each SLA by alternative routes without failing to fulfill their requirements. Thus, SLA 1 receives routes with more hops in order not to compete with SLAs 2 and 3 which have higher priority requirement.

Finally, Figure 6(c) shows the total amount of rules generated by all SLAs in each scenario. This illustrates the benefits of our policy authoring and refinement approach, in which the infrastructure-level programmer does not need to

be concerned with the number of low-level configuration rules to be deployed in the network. Our results suggest that the prototype is able to support the refinement of SLAs and the installation of flow rules in large-scale deployments. Even if we consider the scenario with the largest number of switches and links (Figure 4(c)), and the largest number of QoS classes, the total measured time remains within acceptable bounds. Moreover, as mentioned previously, the framework optimizes the deployment of rules according to the requirements of each SLA and according to each scenario.

V. Related Work

Policy Authoring approaches to facilitate the writing, analysis, and implementation of high-level policies have been proposed in the past. Brodie *et al.* [22] present a platform-independent framework to specify, analyze, and deploy security and networking policies. A portal prototype for policy authoring, based on natural language and structured lists, allows the management of policies from their specification to enforcement. The policy authoring portal enables web users to write policies, using a high-level language, which are translated and mapped to specific low-level configurations. Johnson *et al.* [23] present a template-based framework for policy authoring. The work describes the relationship between general templates and specific policies, and the skills required from users to produce high-quality policies. Although these research efforts investigate important issues regarding policy authoring, none of them presents a formal language for authoring, the use of logical reasoning to assist the refinement process, or experimental results.

Zhao *et al.* [24] describe the design and implementation of an end-to-end framework for the management of cloud-hosted databases from a consumer's perspective. The approach is based on the interpretation of SLAs to assist the dynamic provisioning of databases. The framework checks if SLAs have changed and automatically performs corrective actions to enforce the new specifications. Villegas *et al.* [25] present a framework for the analysis of provisioning and allocation policies for Infrastructure-as-a-Service clouds, *i.e.*, policies to dynamically allocate resources which remain largely under-utilized over time. Oriol Fito *et al.* [26] introduce a Business-Driven ICT Management (BDIM) model to satisfy the business strategies of cloud providers. The objective is to evaluate the impact of events related to ICT using business-level metrics. A Policy-Based Management system analyzes these events and is able to determine automatically the ICT management actions that are most appropriate. Craven *et al.* [17] introduced a refinement process for obligation and authorization policies that addresses policy translation, operationalization, re-refinement, and deployment. The work describes in details how a UML information-based formalism of system elements, a high-level policy, and translation rules that relate actions can produce concrete low-level policies. Bandara *et al.* [16] presented a tool support for the refinement process, and used case-studies based on DiffServ QoS management. The refinement process introduced the use of goal design and applied abductive reasoning as a strategy to generate low-level policies that aim to achieve a specific high-level goal.

Despite the above research efforts have achieved satisfactory results, they were also limited by the characteristics imposed by traditional IP networks, such as best-effort packet delivery and distributed control state. We distinguish our policy authoring framework from other existing approaches by exploring the characteristics of SDN architectures, such as centralized control plane and overall view of the network infrastructure to enhance the policy refinement process. To the best of our knowledge, this is the first time that policy authoring and refinement techniques have been applied to SDN management.

VI. Concluding Remarks

In this paper we presented a policy authoring framework to facilitate the configuration of SDN architectures based on the interpretation of high-level policies. The proposed policy authoring framework assists business-level operators to more easily specify overall service requirements, which can then be automatically translated into the configuration of an SDN infrastructure. An important aspect to be emphasized is that our approach is flexible, and allows the business-level operator to decide whether to accept or not the suggestions given. Thus, the operator can fully or partially accept the suggestion, or create his/her own configuration. Also, our experiments have showed that the toolkit performs well even with the increase in the number of QoS classes and in the complexity of the SLAs.

Different from past research efforts ([22], [23], [16], [17]), our policy authoring process is based on a policy refinement technique that analyzes the infrastructure ability to fulfill the requirements of high-level policies using the information obtained from an SDN controller. As a result, policies are refined with minimal human intervention, as the framework analyzes regexes in each SLA and applies logical reasoning based on network conditions that can fulfill the requirements of these SLAs. Thus, manual workload related to SDN management can be reduced because the flow rules are automatically generated and installed, instead of requiring the operator to directly write and deploy rules. Further, SLAs are specified and rules are deployed through a user-friendly policy authoring framework with minimal disruption to the network.

While we relied on the use of SDN architectures to improve the refinement process, by using the PBNM paradigm we also indirectly addressed problems typically found in SDN, *e.g.*, the issue of having static rules and configurations that are often written for specific situations directly in the controller. From the viewpoint of the network operation, the use of PBNM aims to reduce the complexity of the network management tasks allowing the system to gain a certain level of autonomy [7]. Thus, by using PBNM we reduced the amount of static rules and configurations. This was achieved by writing reusable code that deploys specific rules obtained from a repository.

As a part of our future work, we intend to extend the policy authoring framework to support more terms, expressions, prescriptions, and rules. In addition, our approach is limited to rules triggered by the occurrence of an event, *i.e.*, a flow receives a specific action. We intend to extend our grammar to support temporal logic. This will allow the specification of policies defined by an interval of time. Moreover, we also intend to investigate techniques for detection and resolution of policy conflicts in different levels of abstraction. Further, we intend to identify ongoing standardization efforts related to policy-based management in order to improve the prototype. Finally, we intend to analyze the toolkit behavior when managing other resources and types of services.

REFERENCES

[1] Open Networking Foundation, "Software-defined networking: The new norm for networks," Open Networking Foundation ONF, Tech. Rep., April 2012.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, mar 2008.

[3] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? implementation challenges for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 7, pp. 36–43, July 2013.

[4] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," *NSDI, Apr*, 2013.

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

[6] R. Neisse, E. Pereira, L. Granville, M. Almeida, and L. Rockenbach Tarouco, "An hierarchical policy-based architecture for integrated management of grids and networks," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, June 2004, pp. 103–106.

[7] 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.

[8] A. Bandara, E. Lupu, J. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, 2004, pp. 229–239.

[9] 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.

[10] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A software defined internet exchange," *SIGCOMM Comput. Commun. Rev.*, 2014, (to appear).

[11] T. Kuhn, "A survey and classification of controlled natural languages," *Computational Linguistics*, vol. 40, no. 1, pp. 121–170, 2013.

[12] 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*, 2014, pp. 397–404.

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

[14] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New Gen. Comput.*, vol. 4, no. 1, pp. 67–95, jan 1986. [Online]. Available: http://dx.doi.org/10.1007/BF03037383

[15] C. C. Machado, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho, "An EC-based formalism for policy refinement in software-defined networking." 2015, submitted to ISCC 2015.

[16] A. Bandara, E. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, and G. Pavlou, "Policy refinement for diffserv quality of service management," in *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, May 2005, pp. 469–482.

[17] 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.

[18] K. Bakshi, "Considerations for software defined networking (SDN): Approaches and use cases," in *Aerospace Conference, 2013 IEEE*, March 2013, pp. 1–9.

[19] 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.

[20] POX, "Pox openflow controller," 2013, Accessed: Sept. 2013. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[21] J. Postel and J. K. Reynolds, "Rfc 1700 assigned numbers," *Network Working Group*, 1994.

[22] C. Brodie, D. George, C.-M. Karat, J. Karat, J. Lobo, M. Beigi, X. Wang, S. Calo, D. Verma, A. Schaeffer-Filho, E. Lupu, and M. Sloman, "The coalition policy management portal for policy authoring, verification, and deployment," in *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, June 2008, pp. 247–249.

[23] M. Johnson, J. Karat, C.-M. Karat, and K. Grueneberg, "Optimizing a policy authoring framework for security and privacy policies," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, ser. SOUPS '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:9.

[24] L. Zhao, S. Sakr, and A. Liu, "A framework for consumer-centric SLA management of cloud-hosted databases," *Services Computing, IEEE Transactions on*, vol. PP, no. 99, 2013.

[25] D. Villegas, A. Antoniou, S. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, May 2012, pp. 612–619.

[26] J. Oriol Fito, M. Macias, F. Julia, and J. Guitart, "Business-driven it management for cloud computing providers," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, Dec 2012, pp. 193–200.