Journal of Network and Computer Applications 90 (2017) 1–16

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

# ARKHAM: An Advanced Refinement toolkit for Handling Service Level Agreements 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, Av. Bento Gonçalves, 9500 – Porto Alegre, RS, Brazil*

ARTICLE INFO

*Keywords:*
Policy-based network management
Policy refinement
Software-Defined Networking
Service level agreement

ABSTRACT

Software-Defined Networking (SDN) provides a more sophisticated and flexible architecture for managing and monitoring network traffic. SDN moves part of the decision-making logic (i.e., flow processing and packet routing) from network devices into a logically centralized controller. However, the expected behavior and configuration of network devices are often defined directly in the controller as static rules for specific situations. This approach becomes an issue when associated with an increasing number of network elements, links, and services, resulting in a large amount of rules and a high overhead related to network configuration. As an alternative, techniques such as Policy-Based Network Management (PBNM) and more specifically policy refinement can be used by operators to write Service Level Agreements (SLAs) in a user-friendly interface without the need to manually reconfigure each network device. To address these issues, we specifically introduce ARKHAM: an Advanced Refinement Toolkit for Handling SLAs in SDN. In this article, we present (*i*) a policy authoring framework that uses logical reasoning for the specification of business-level goals and to automate their refinement; (*ii*) an OpenFlow controller which performs information gathering and configuration deployment; (*iii*) a policy repository that stores information about the behavior of the infrastructure, which is obtained by the OpenFlow Controller, and policy authoring operations; and (*iv*) a formal representation using *event calculus* that describes our solution. The main contributions of this work are (*i*) the capacity to deploy 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. The experimental results demonstrate that the refinement toolkit achieves the expected results within acceptable performance bounds, even with the increasing complexity and size of SLAs, network topologies, and repositories.

## 1. Introduction

The emergence of Software-Defined Networking (SDN) offers a more flexible architecture for managing computer networks (Nunes et al., 2014; Wickboldt et al., 2015; Machado et al., 2014). SDN provides an extensible platform to accommodate network services able to quickly adapt to dynamic changes of service requirements. SDN presents a clear separation between data/forwarding and control planes by moving part of the decision-making logic from switching physical devices to logically centralized entities often refereed to as controllers. Decisions are then taken at the control plane, by controllers, while switching devices become ordinary packet forwarding nodes. Controllers have a global view of the network infrastructure and ultimately define its behavior. Switching devices (or just switches), in turn, use flow tables (structures that contain a set of packet fields to match traffic flows with their specific actions) that are remotely configured by controllers through an open interface, such as the OpenFlow protocol (McKeown et al., 2008); OpenFlow itself is completely independent of the underlying switch hardware specificities (Bakshi, 2013). As a result, SDN realizes a more flexible architecture for deploying network services, including for example, access control, Quality of Service (QoS), and load balance (Sezer et al., 2013; Hakiri et al., 2014).

Despite the benefits of SDN, the expected behavior of network switches is often defined by static rules written directly at controllers (Sezer et al., 2013; Monsanto et al., 2013; Kim and Feamster, 2013). As a consequence, SDN becomes susceptible to some traditional network management problems, including (*i*) human work overload due to the need of writing a large set of rules; (*ii*) limited deployment of new services and resources that were not anticipated because static rules and

* Corresponding author.
*E-mail addresses:* ccmachado@inf.ufrgs.br (C.C. Machado), jwickboldt@inf.ufrgs.br (J.A. Wickboldt), granville@inf.ufrgs.br (L.Z. Granville), alberto@inf.ufrgs.br (A. Schaeffer-Filho).

http://dx.doi.org/10.1016/j.jnca.2017.04.009
Received 27 July 2016; Received in revised form 2 December 2016; Accepted 15 April 2017
Available online 21 April 2017

configurations use fixed values, for example, TCP/UDP port number, destination IP address, becoming not adaptable to other services and resources; and (*iii*) too low-level rules that do not faithfully fulfill high-level goals, as network programmers may not be aware of business goals.

A possible approach to address these problems is the employment of Policy-Based Network Management (PBNM) (Han and Lei, 2012; Moffett and Sloman, 1993). In PBNM, network administrators define the infrastructure's goals and restrictions in the form of policies that then guide the behavior of the managed system (Aib and Boutaba, 2007; Pueschel et al., 2012). Policy refinement techniques are used to automatically translate high-level policies – e.g., specified as a Service Level Agreement (SLA) – into a set of low-level rules (Moffett and Sloman, 1993; Bandara et al., 2004). A PBNM solution for managing SDN can offer three main benefits. First, due to the homogeneous switch flow tables, a single policy is created to any switching device without the need for adaptations, decreasing human work overload. Second, due to the centralized decision-making logic in SDN controllers, it is more flexible to deploy or change policies dynamically at runtime when new services emerge, resulting in minimal service disruptions. Third, thanks to the controller's overall view of the network, it is easier to analyze policies in order to guarantee consistency, i.e., that policies will be fulfilled (Verma, 2002; Koch et al., 1996; Carey and Wade, 2008).

The use of PBNM and policy refinement for the management of traditional networks has been extensively investigated (Bandara et al., 2004, 2005; Rubio-Loyola et al., 2006; Craven et al., 2011). However, we argue that policy refinement in the new research area of SDN has been a neglected topic. When using SLAs, their translation to low-level rules, *e. g.*, rules for configuring switching elements, is not straightforward. If this translation is not performed properly, the system elements may not be able to meet the implicit requirements specified in the SLA. For this reason, when applying policy refinement, specific problems must be addressed, such as (*i*) determining which resources are needed to fulfill the policy requirements; (*ii*) translating high-level policies into operational rules that the system can enforce; and (*iii*) checking whether or not the low-level rules are accurate and faithfully satisfy the requirements specified by the high-level policy.

In this article, we introduce ARKHAM: an **A**dvanced **R**efinement tool**K**it for **H**andling service level **A**gree**M**ents in SDN. ARKHAM is a solution for the refinement of high-level policies (expressed in a controlled natural language) into low-level rules to be enforced by the network controller. This solution comprises (*i*) a policy authoring framework that uses *logical reasoning* (Kowalski and Sergot, 1986) for the specification of business-level goals and to automate their refinement; (*ii*) an OpenFlow controller that performs information gathering and configuration deployment; (*iii*) a policy repository that stores information about the behavior of the infrastructure, which is obtained by the OpenFlow Controller, and policy authoring operations; and (*iv*) a formal representation using *event calculus* (Shanahan, 2000) that describes our solution. As a proof-of-concept, we have applied ARKHAM for QoS management in SDN. We developed a system prototype and performed several experiments considering different SLAs, scenarios, and the amount of information in the repository to be processed. Furthermore, to evaluate the use of logical reasoning and event calculus, we measured the number of iterations required by our solution to identify QoS classes that can faithfully fulfill the requirements of different SLAs.

The remainder of this article is organized as follows. In Section 2, we discuss related work. In Section 3, we introduce the ARKHAM refinement toolkit, together with the elements, techniques, and concepts that are part of it. In Section 4 we present a formal representation to model both the system behavior and the policy refinement process for SDN management. In Section 5, we describe the prototype, experiments, and the results achieved. In Section 6, we discuss the lessons learned during the development of this work. Finally, in Section 7, we conclude this article with final remarks and perspectives for future work.

## 2. Related work

In this work, we advocate that policy refinement techniques can be used to reduce the manual work needed for SDN configurations. In this section, we describe research efforts that either use SDN to improve network resource management or propose frameworks for PBNM and policy refinement.

SDN features have been employed to enhance the monitoring and management of network traffic. Foster et al. (2011) introduced a new language, called Frenetic, for network programming supporting OpenFlow. Frenetic has a set of operators for handling network traffic flows, and a runtime engine that abstracts the details related to installing and uninstalling low-level rules in switches. Monsanto et al. (2013) introduced the Pyretic language. Pyretic introduced programming abstractions that simplifies the creation of modular management programs. Nonetheless, Frenetic and Pyretic are languages to model low-level rules and do not employ any policy refinement technique to translate high-level policies into a set of low-level rules.

Rubio-Loyola et al. (2011) investigated the sharing of virtualized network resources in SDN. The authors proposed an autonomic management system capable of separating control and data planes, providing greater isolation in the execution of applications. In addition, an Orchestration Plane was presented, which aims to manage the system behavior in response to context changes, and in accordance with business goals and policies. However, the authors do not specify a refinement approach to automate the translation of policies in different levels of abstraction. Kim and Feamster (2013) presented a discussion about three problems with the current state-of-the-art in network management: the need to support modifications to network behavior, the lack of a high-level language for network configuration, and the need to exert control over tasks related to network analysis and troubleshooting. Further, the authors described the use of the Procera language to assist operators to express network policies that react to various types of events using a high-level functional programming language. Despite presenting a broad discussion on network management problems, that work does not investigate aspects concerning policy refinement and its implementation issues.

The use of PBNM and policy refinement in computer networks has been investigated for over two decades. Bandara et al. (2005) presented the use of goal design and abductive reasoning to derive strategies that attain a specific high-level goal. Policies can be refined by combining strategies with events and restrictions. The authors provided tool support for the refinement process and use examples of DiffServ (Blake et al., 1998) QoS management. Craven et al. (2011) presented a refinement process for authorization and obligation that involves stages of decomposition, operationalization, re-refinement, and deployment. The authors discussed how concrete low-level rules can be produced from a high-level policy, decomposition rules, and a description of system objects (described in UML). Rubio-Loyola et al. (2006) presented a goal-oriented approach for goal decomposition using KaOS (Canos et al., 1998). The refinement approach makes use of linear temporal logic and reactive systems analysis techniques, thus generating deployable policies in Ponder (Lupu et al., 2000).

Although many efforts related to PBNM have achieved important results, they were also limited by the characteristics imposed by traditional IP networks, such as best-effort packet delivery and

distributed control state. In addition, they did not use information about network guarantees to accommodate service requirements. We distinguish our policy refinement solution from the existing approaches by exploring the characteristics of the SDN architecture, such as the logically centralized control plane, overall view of the network infrastructure, and the existence of standard communication protocols, such as OpenFlow (McKeown et al., 2008) and ForCES (Liu et al., 2009). The work presented in this article uses policy refinement to model the behavior of SDN components without the need to recode them or interrupt the network operation. Although some of the ideas in our toolkit are inspired by the research efforts mentioned before, to the best of our knowledge, this is the first time a policy refinement toolkit for SDN management is presented.

## 3. ARKHAM refinement toolkit

In this section, we describe ARKHAM: an **A**dvanced **R**efinement tool**K**it for **H**andling service level **A**gree**M**ents in SDN. ARKHAM introduces the use of PBNM to tackle issues of SDN such as static rules and configurations often written for specific situations directly in the controller. PBNM and policy refinement techniques can be used to reduce the amount of static rules deployed in the network. These techniques also assist network operators in writing more generic, high-level configurations for the management of specific network devices.

Our solution is inspired by previous investigations in the area of PBNM and policy refinement (Bandara et al., 2005; Craven et al., 2011), which have been limited due to the characteristics of traditional IP networks, such as best-effort packet delivery and distributed control state in forwarding devices. In SDN, these limiting factors can be overcome, since in implementations such as OpenFlow the control plane is logically centralized in SDN controllers. The controller receives information from all network elements becoming aware of network-wide events and traffic flows. This centralization of network traffic information simplifies the process of gathering information from the network infrastructure. Thus, ARKHAM relies on the characteristics of SDN to enhance the policy refinement process.

We present an overview of the ARKHAM refinement toolkit in Section 3.1. In Section 3.2, we describe the main components of our solution. We introduce a controlled natural language (CNL) created to establish restrictions and requirements for writing SLAs in Section 3.3. In Section 3.4, we explain the stages to collect network information and refine SLAs into a set of low-level rules.

### 3.1. Policy refinement toolkit: an overview

A PBNM system is composed of four basic entities (Waller et al., 2011): (*i*) *Policy Repository (PR)*, which stores policy-related information; (*ii*) *Policy Management Tool (PMT)*, which allows the operator to manage policies that are inserted into the PR; (*iii*) *Policy Decision Point (PDP)*, which performs event monitoring on the system to verify and validate the necessary conditions for policies; and (*iv*) *Policy Enforcement Point (PEP)*, which executes and monitors policies, and provides feedback on relevant information during runtime. In ARKHAM, these entities are mapped as follows: a Policy Authoring Framework implements the PMT; a MySQL database implements the PR; an OpenFlow controller implements the PDP; and OpenFlow switches implement PEPs.

Fig. 1 shows an overall view of our conceptual solution. On the right side, an infrastructure-level programmer inserts service specifications and system configurations into the PR. On the left side, a business-level operator inserts SLAs descriptions into the system. A policy analyzer component finds expressions in order to parse and match an SLA to some QoS class stored in the repository. Periodically, the controller collects network information and stores it in the policy repository. Subsequently, it gets configuration parameters, such as service port numbers (e.g., HTTP =80, SMTP = 25), SLA parameters (e.g., delay <2 ms, bandwidth >128 kbps) from the policy repository. These parameters are used to perform the requirements formulation, in other words, to verify how to accommodate each SLA in the network infrastructure based on the network characteristics, e.g., number of hops, available bandwidth. Finally, the controller encodes deployable rules, such as forward packet to specific ports and installs low-level rules in forwarding devices.

### 3.2. Main components

Our solution comprises the following fundamental components:

*OpenFlow Controller* – Its operation is divided into three phases: (*i*) Startup Phase: discovers network elements, possible paths, and performs information gathering such as number of hops, delay, and available bandwidth for each path. In addition, in this phase, a set of forwarding rules, which we call *standard rules*, are initially deployed to guarantee best-effort delivery of traffic; (*ii*) Events Phase: identifies network services and analyzes the duration of each flow to propose modifications to the network configuration triggering the *Analysis Phase*; and (*iii*) Analysis Phase: determines the best path based on the characteristics of the network and service requirements. Additionally, this phase implements the rules and waits for the *Events Phase* in order to identify possible enhancements to the active flows and reconfigure the infrastructure. For further details about the OpenFlow Controller, we refer the reader to Machado et al. (2014).

*Policy Authoring Framework* – It has two independent Graphical User Interfaces (GUIs): (*i*) *Policy Authoring GUI*, and (*ii*) *Configurator GUI*. The *Policy Authoring GUI* enables the writing of SLAs in a Controlled Natural Language (CNL) and automates the translation of SLAs into the low-level controller configuration. It analyses the SLA requirements that match the *regexes* (regular expressions) stored in the *Policy Repository*, and uses abductive reasoning to suggest the more appropriate QoS classes for the SLA. In addition, if the network cannot accommodate the SLA requirements, it uses inductive reasoning to suggest classes or the creation of a new class that can fulfill the SLA requirements. The *Configurator GUI* allows the specification of regular expressions and technical characteristics of services and their parameters such as TCP/UDP port numbers and service name. The policy authoring operations were described in more details in Machado et al. (2015b).

*Policy Repository* – It stores information about the behavior of the infrastructure, which is obtained by the OpenFlow Controller and policy authoring operations. For example, the repository stores information about the network topology, available bandwidth, delay, and jitter. Additionally, the repository maintains a list of all services and their parameters, as well as some QoS classes. The information in this repository will be used by the OpenFlow Controller during the *Events Phase* and the *Analysis Phase* and for policy authoring. Finally, the repository stores a wide range of regular expressions separated by type. The regular expressions are presented in Section 3.3.

### 3.3. Controlled natural language (CNL)

We created a CNL to establish restrictions and requirements for writing SLAs. Our CNL aims to improve the translation between what humans (business-level operators and infrastructure programmers) intend and what the ARKHAM Refinement Toolkit needs to do. In addition, our CNL is used as the basis for creating natural and intuitive representations for formal notations. Furthermore, it can be adapted as needed for new applications such as monitoring, firewalls, and access control. The grammar of the controlled natural language is defined below:

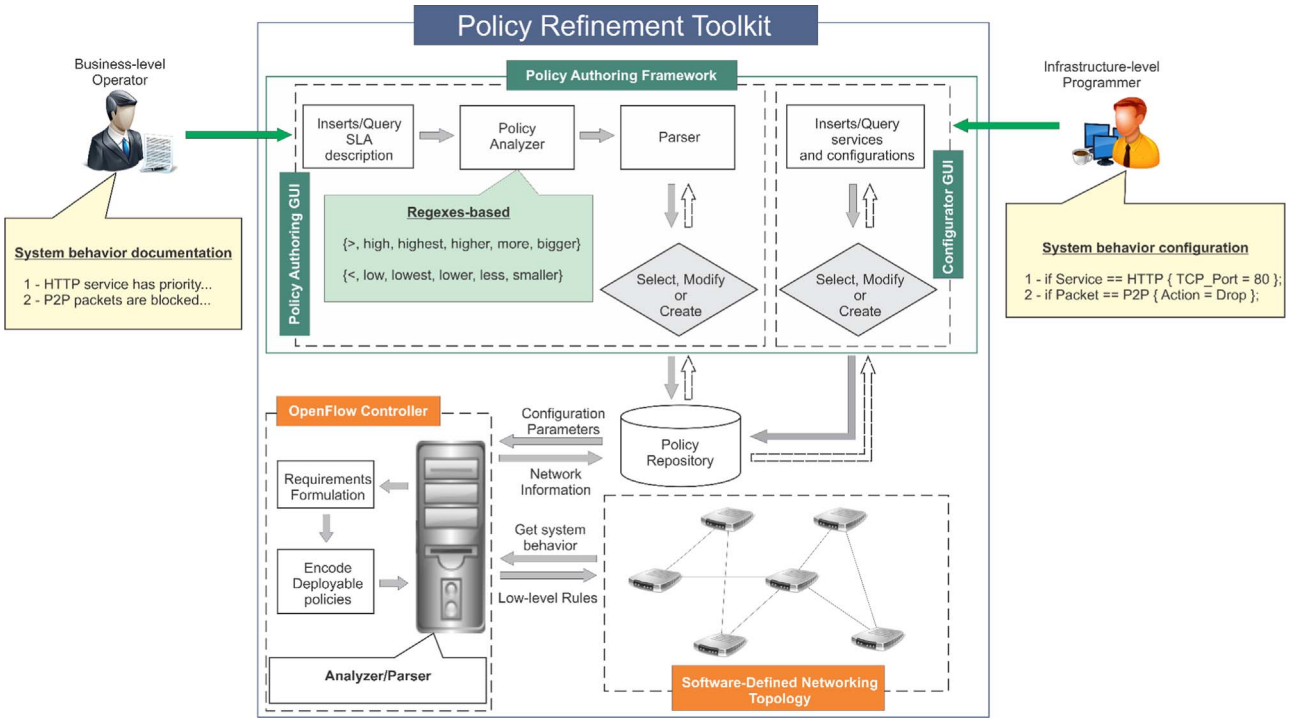



**Fig. 1.** Overall conceptual solution.

**Lstlisting 1.**

```
Language :→ (<QoS-Class> | <Service>) <ModalVerb> <Expression>
QoS-Class :→ qos-regexes
Service :→ service-regexes
ModalVerb :→ should | should not
Expression :→ <Term> | <Term> <Connective> <Expression> | <QoS-Class>
Term :→ <Parameter> <Operator> <Value>
Parameter :→ requirements-regexes
Operator :→ adjective-regexes
Value :→ v
Connective :→ and | or
```

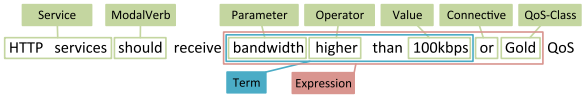



**Fig. 2.** Example of occurrences of grammar constructs in an SLA.

Our toolkit uses *regexes* as a concise and flexible way of identifying strings of interest such as particular characters (e.g., >, < , =) or terms (e.g., high, low, HTTP, gold). We defined the following types of *regexes*: (*i*) *qos-regexes* – regular expressions to identify QoS classes; (*ii*) *service-regexes* – regular expressions to identify services; (*iii*) *requirements-regexes* – regular expressions to identify service requirements; and (*iv*) *adjective-regexes* – regular expressions to identify adjectives in service requirements. Table 1 shows examples of regular expressions that can be contained in an SLA. Fig. 2 shows an example of SLA containing regular expressions.

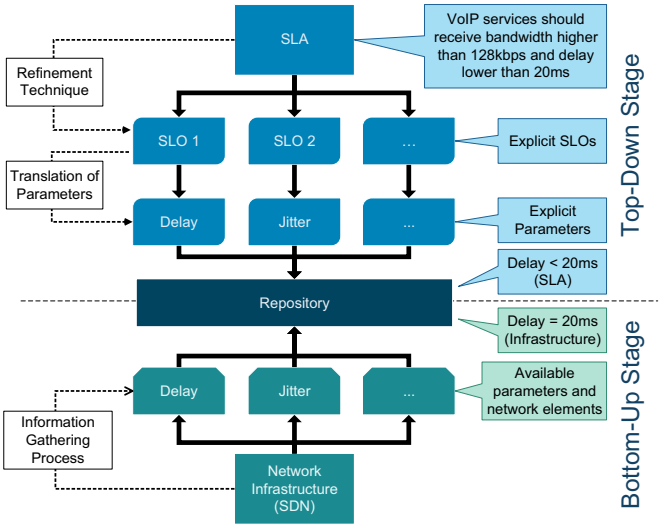



**Fig. 3.** Deriving SLOs/parameters from goal and gathering network information.

**Table 1**
Examples of regular expression.

| Type | Expression | |
|---|---|---|
| *qos-regexes* | Bronze, Silver, Gold,... | N/A |
| *service-regexes* | VoIP, Streaming, HTTP, FTP,... | N/A |
| *requirements-regexes* | Priority, Bandwidth, Delay, and Jitter | N/A |
| *adjective-regexes* | more, high, higher, highest,... | > |
| | equal, like, even, uniform,... | = |
| | less, low, lower, lowest,... | < |

### 3.4. Bottom-up and Top-down stages

Policy refinement encompasses the process of derivation of SLOs that must meet the SLA (Fig. 3). These SLOs are considered QoS requirements, e.g., priority, bandwidth.

As mentioned previously, we identify business-level objectives and high-level policies as Service Level Agreements (SLAs). Therefore, our refinement process consists in a technique that extracts *regexes* from an SLA and decomposes them into meta-goals or Service Level Objectives (SLOs), i.e.,

$SLA_1 \rightarrow SLO_{1-1}$, $SLA_1 \rightarrow SLO_{1-2}$, $SLA_1 \rightarrow SLO_{1-3}$, $SLA_1 \rightarrow SLO_{1-n}$.

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, according to Table 1.Thus, our refinement process comprises a restricted SLO mapping, where $SLO_{1-1} \rightarrow$ D, $SLO_{1-2} \rightarrow$ J, $SLO_{1-3} \rightarrow$ B, $SLO_{1-4} \rightarrow$ P or $SLO_{1-1} \rightarrow$ qos-regexes or $SLO_{1-1} \rightarrow$ service-regexes.

Our refinement approach is split into two stages: The first stage, called *bottom-up*, consists of gathering network information (e.g., bandwidth, delay). A key element of this stage is the OpenFlow Controller, which performs the data collection process. The information gathered is stored in a Policy Repository. Using this information, the Policy Authoring framework uses *abductive reasoning* to indicate to the business-level operator what are the possible configurations that can accommodate a given SLA. These indications are provided through settings performed previously – other SLAs or policies created manually by the business-level operator or by the infrastructure-level programmer – along with the characteristics that the network can support.

The second stage, called *top-down*, is a technique for the refinement of high-level goals, extracted from SLAs and translated into achievable goals (SLOs). An operator writes SLAs and creates (if necessary) the QoS classes needed to fulfill them. As mentioned previously, the *bottom-up* stage tries to indicate using *abductive reasoning* what are the best configurations for the SLA that is being written. Thus, multiple policy configuration options will be offered to the operator, who can select or customize an existing configuration, or even create a new configuration.

After any selection or customization performed by the operator, the low-level rules for switch configuration that satisfy the given SLA are stored in the Policy Repository. Subsequently, the OpenFlow Controller reads from the Policy Repository these new policies, starting the *Analysis Phase* for setting up the appropriate rules in the network.

## 4. An EC-Based formalism for policy refinement

In this section we present a formalism for representing SLAs using Event Calculus (EC) Kowalski and Sergot (1986), and apply logical reasoning to model both the system behavior and the policy refinement process. We aim with this formal representation to assist infrastructure-level programmers to develop refinement tools and configuration approaches to achieve more robust SDN deployments.

EC is a formalism that permits representing and reasoning about dynamic systems. It consists of axioms and predicates that are independent of application or domain. We use the form described by Bandara et al., (2005, 2003), consisting of (*i*) a set of event types, (*ii*) a set of properties (called fluents) that can vary over the system lifetime, and (*iii*) a set of time points. In order to achieve our goals we extended it with new constants, variables, operations, and predicates as follows:

*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 ($Obj_{Dst}$), a link object ($Obj_{Link}$), or even a route ($Obj_{Route}$).

*Variables* – define $V_o$ to represent the attributes of objects and $V_\rho$ to represent the parameters for the operations supported by objects.

*Predicates* – specify what objects represent in the system, what is declared about them or relationships between objects. Table 2 presents the new predicates.

*Operations* – specify actions used with predicates. For example, a query in a repository or the triggering of a phase.

In our solution we first describe the SLAs in EC, and use logical reasoning to derive the SLOs (*requirements-regexes*) 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 SLOs. SLOs are used to search for matching QoS classes in the repository. In the following, we demonstrate how our formal model can be used for matching an SLA with QoS Classes.

$$
\begin{aligned}
&\lambda_1 : \text{isMemberParameter}(SLA_n, Priority, V_o) \\
&\quad \wedge \text{isMemberParameter}(SLA_n, Bandwidth, V_o) \\
&\quad \wedge \text{isMemberParameter}(SLA_n, Delay, V_o) \\
&\quad \wedge \text{isMemberParameter}(SLA_n, Jitter, V_o) \\
&\lambda_2 : \text{isMemberParameter}(Class_n, Priority, V_o) \\
&\quad \wedge \text{isMemberParameter}(Class_n, Bandwidth, V_o) \\
&\quad \wedge \text{isMemberParameter}(Class_n, Delay, V_o) \\
&\quad \wedge \text{isMemberParameter}(Class_n, Jitter, V_o) \\
&\lambda_3 : \text{isMemberParameter}(Class_n, Priority, V_o) \\
&\quad \wedge \text{isMemberParameter}(Class_n, Bandwidth, V_o) \\
&\quad \wedge \text{isMemberParameter}(Class_n, Delay, V_o) \\
&\lambda_4 : \text{isMemberParameter}(Class_n, Bandwidth, V_o) \\
&\quad \wedge \text{isMemberParameter}(Class_n, Priority, V_o) \\
&\lambda_5 : \text{isMemberParameter}(Class_n, Priority, V_o) \\
&\quad \wedge \text{isMemberParameter}(Class_n, Delay, V_o) \\
&\lambda_6 : \text{isMemberParameter}(Class_n, Bandwidth, V_o) \\
&\quad \wedge \text{isMemberParameter}(Class_n, Delay, V_o) \\
&\lambda_7 : \text{isMemberParameter}(Class_n, Priority, V_o) \\
&\quad \vee \text{isMemberParameter}(Class_n, Bandwidth, V_o) \\
&\quad \vee \text{isMemberParameter}(Class_n, Delay, V_o) \\
&\quad \vee \text{isMemberParameter}(Class_n, Jitter, V_o) \\
&\phi_1 : \lambda_1 \leftrightarrow \lambda_2 \\
&\phi_2 : (\lambda_1 \leftrightarrow \lambda_3) \leftarrow \neg\ \phi_1 \\
&\phi_3 : (\lambda_1 \leftrightarrow \lambda_4) \leftarrow \neg\ \phi_2 \\
&\phi_4 : (\lambda_1 \leftrightarrow \lambda_5) \leftarrow \neg\ \phi_3 \\
&\phi_5 : (\lambda_1 \leftrightarrow \lambda_6) \leftarrow \neg\ \phi_4 \\
&\phi_6 : (\lambda_1 \leftrightarrow \lambda_7) \leftarrow \neg\ \phi_5
\end{aligned}
$$

$SLA_n$ *is an SLA*

$Class_n$ *is a QoS Class*
$V_o$ *is a value of a parameter*
$\lambda_n$ *is a predicate*

The set of $\lambda$ rules are used to retrieve the QoS classes that satisfy the largest amount of requirements. Thus, $\lambda_2$ will retrieve QoS classes that satisfy all requirements (priority, bandwidth, delay, and jitter), whereas $\lambda_7$ will retrieve QoS classes that satisfy at least one of the requirements. The set of $\phi$ rules specifies an order of matches that happens until an occurrence of $\phi$ matches the desired result. Thus, $\phi_1$ represents an ideal match where all requirements are satisfied while $\phi_6$ is a match where at least one requirement is satisfied.

We also use EC and the predicates in Table 2 to specify all the processes performed by the refinement toolkit. For a better understanding, we use friendly names to indicate to the reader where each process occurs. In the following, we use the topology discovery performed by the controller as a case-study to demonstrate how the formal model can be used to specify a system process.

$$
\begin{aligned}
&\lambda_m : \text{initiates}(\text{doAction}(\text{Controller}, \\
&\quad \text{operation}(\text{Controller}, \text{startupPhase}(\text{packets})), \\
&\quad \text{state}(\text{Controller}, \text{status}, \text{on}), T)). \\
&\lambda_{m+1} \leftarrow (\text{happens}(\text{operation}(\text{Controller}, \\
&\quad \text{discoveryTopology}(\text{LLDP})), T+1) \\
&\wedge\ \text{happens}(\text{operation}(\text{Controller}, \\
&\quad \text{discoveryLinks}(\text{LLDP})), T+1)) \\
&\leftarrow \text{happens}(\text{doAction}(\text{Switch}, \\
&\quad \text{operation}(\text{Controller}, \\
&\quad \text{sendPacket}(\text{LLDP})), T+1)) \\
&\leftarrow \lambda_m \\
&\lambda_{m+2} \leftarrow (\text{happens}(\text{doAction}(\text{Controller}, \\
&\quad \text{operation}(\text{Repository}, \\
&\quad \text{registerSwitchId}(\text{idSwitch})), T+2)) \\
&\wedge\ (\text{happens}(\text{doAction}(\text{Controller}, \\
&\quad \text{operation}(\text{Repository}, \\
&\quad \text{registerSwitchLink}(\text{linkSwitch})), T+2)) \\
&\leftarrow \lambda_{m+1}.
\end{aligned}
$$

$\lambda_m$ *is a predicate.*
$T$ *is a time point.*

As can be observed, when starting the controller, it monitors

**Table 2**
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. |
| method($Obj_n$, Action($V_\rho$)) | Defines an action supported by an object. |
| 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 QoS 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 $Obj_n$. |

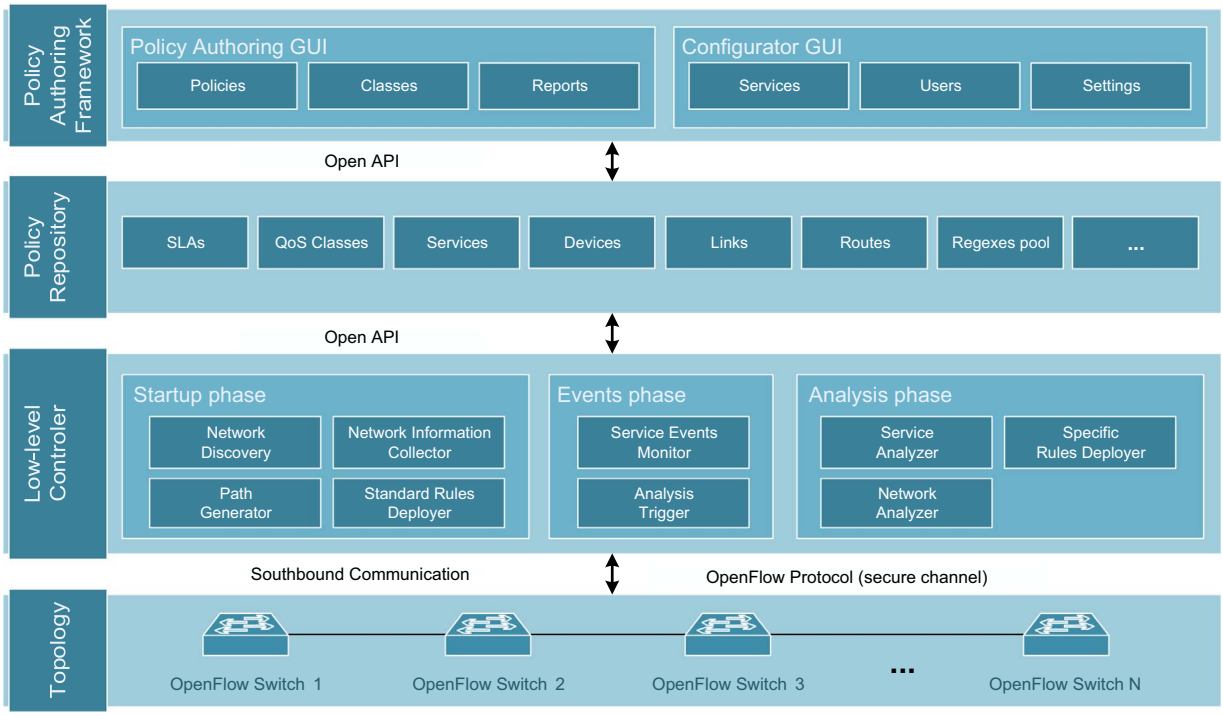



**Fig. 4.** Overall policy refinement toolkit.

(StartUp phase) the network in order to discover the network elements and their links. Switches are instructed to send LLDP packets to report their location in the topology. The controller collects these packets and calculates all possible links between every element. For each result of this calculation a spanning tree 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 is stored in the repository, and will be subsequently used to find a matching QoS class. For further details about the formalism we refer the reader to Machado et al. (2015a).

## 5. Prototype and experimental evaluation

In this section we present the prototype developed (Section 5.1) and the experiments performed (Section 5.2) in order to validate ARKHAM.

### 5.1. Prototype

The overall implementation of ARKHAM and its main modules are depicted in Fig. 4. To implement our solution we customized a POX[1] OpenFlow controller based on the *l2_multi* module. This module has some important presets for our solution such as support for the *discovery library*, used to discover network elements and the *spanning tree library*, used to calculate and deploy different paths between source and destination elements. The customization enables the gathering of additional information about the network infrastructure, which is used, for example, to calculate optimal routes. This customization is based on SDN native features, thus it can be applied to any controller implementation. For example, topology discovery, which is

[1] http://www.noxrepo.org/pox/about-pox/

available in the POX controller from the discovery module, is a feature that can be achieved by other controllers in different implementations by collecting Link Layer Discovery Protocol (LLDP) packets that are sent from network devices. We used the OpenFlow Protocol 1.0 (Machado et al., 2015) for southbound communication because it is the most widely supported version of the protocol. We are aware that newer versions of the protocol introduced a more flexible pipeline with multiple tables and the use of groups to organize flow rules. However, so far version 1.0 has presented satisfactory results, although in the future we might consider newer versions of the OpenFlow protocol for optimization or organization purposes.

We developed a Policy Authoring Framework to enable operators to express business goals, e.g., Service Level Agreements (SLAs), without having to specify in detail what elements in the network infrastructure should receive the configurations and how they should be configured. This authoring framework is based on the Django web framework.[2] We chose Django due to its support to the Python[3] language and the support it provides to create web applications. For the interface design we used the Bootstrap front-end framework.[4] Fig. 5 illustrates the home screen of the Policy Authoring Framework. It presents statistics about the number of policies, classes, services, and users registered. In addition, it shows charts about the top 5 services that most appear in policies and the top 5 QoS classes that most have linked policies. Finally, the dashboard comprises items to manage policies, classes, services, and users; view reports; and determine system configurations.

We created a Policy Repository to store information about the behavior of the infrastructure, which is obtained by the Controller, and information inserted by the Policy Authoring Framework. The Policy Repository uses a MySQL database 5.5.[5] We chose MySQL because it supports the MyISAM (My Indexed Sequential Access Method) storage engine, which provides fast response times in queries. The Repository is composed of several tables, such as SLAs, QoSClasses, and RegexesPool, used by the Policy Authoring Framework; and Services, Devices, Links, and Routes, used by the Controller.

### 5.2. Experimental evaluation

In this section we present and discuss the experimental evaluation of ARKHAM. It is important to emphasize that ARKHAM is generic enough and can be used to perform configuration and manage services and network functions such as monitoring, access control, load balance, and firewall. However, in order to validate the principles employed in our solution, we have limited the scope for QoS management, providing a more specific case-study. QoS mechanisms allow network administrators to use existing resources efficiently and ensure the required level of service without the need of expanding or over provisioning their networks.

#### 5.2.1. Controller experiments

This section describes our test environment and some initial results with the controller prototype. The experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory. Each experiment was run thirty times, in order to obtain a 95% confidence level. The scenarios were created using the Mininet emulator.

Five scenarios were created in our experiments. The scenarios that we built had increasing numbers of switches and redundant links, thus increasing path diversity between any two hosts. Table 3 shows the number of hosts, switches, and links in each scenario.

The topologies used in the experiments were based on Fat-Tree topologies Leiserson (1985). An overview of scenarios V, W, and X is shown in Fig. 6. Due to the large number of elements, scenarios Y and Z could not be clearly presented in the figure. By increasing the number of switches and redundant links, these topologies can be used to, for example, maintain system availability in the presence of link disruptions and to reduce traffic congestion in the infrastructure.

The first set of experiments aim to evaluate the reduction of the amount of time necessary for switch configuration when our customized controller is used. Thus, we compare the time spent by our solution that initially recognizes the topology (Fig. 7(a)) and deploys specific rules using the known topology (Fig. 7(b)) against the time spent by an approach that deploys specific rules without previous knowledge of the topology (Fig. 7(c)).

As can be observed in Fig. 7(a), with the increase in the number of elements and paths in the topology, the time to calculate all paths between any network element also increases, reaching 78.17 s in scenario Z. Despite this, we advocate this pre-calculation and the installation of the initial standard rules because this process will speed up the processing of new flows when they arrive. This happens because later, in the Analysis Phase, all possible flow paths will be already pre-calculated, making the calculation and installation of specific rules as fast as 0.841 s in scenario Z (as can be seen in Fig. 7(b)). The increase in the number of elements in each topology reflects in a small increase in the time to deploy specific rules as can be observed in Fig. 7(b). Thus, there is a minimum time necessary for deploying the specific rules, but this does not delay service flow processing due to the pre-established standard rules.

For comparison purposes, Fig. 7(c) shows an alternative approach in which standard rules are not pre-calculated. As can be observed, when a new flow arrives the time for rule calculation and installation in scenario Z is 8.23 s, which is nearly 10 times lower than the time taken by our solution for pre-calculation and deployment of standard rules (78.14 s according to Fig. 7(a)). During runtime, however, our approach based on the pre-calculation of standard rules can reduce the delay in deploying specific rules, thereby increasing the performance of the network as can be observed in Fig. 7(b)). Certainly there is an overhead related to the startup phase of the controller. Nevertheless, this is justifiable as it improves the performance of the network during runtime. For example, in scenario Z our approach takes 0.841 s to calculate and install specific rules when a new flow arrives (Fig. 7(b)), which is approximately 10 times lower than the 8.23 s required for rule calculation and installation for the same scenario using the alternative approach without pre-calculation (Fig. 7(c)). As mentioned previously, these results show that due to the calculations performed initially, our proposal is able to perform a quick reconfiguration of specific rules, since we already have a populated list of the best path based on best-effort delivery between network elements.

It is important to mention that the path chosen in the experiments (Fig. 7(c)) is the first one found between a source and a destination. Thus, there is no checking if this path is the best one to a particular flow, i.e., if it complies with the QoS requirements of a particular flow. Nevertheless, rules deployed by our solution intend to choose the best routes to fulfill the service requirements among several possible paths.

#### 5.2.2. End-to-end process experiments

We present in this section experiments obtained with the implemented toolkit. Our goal is to measure the response time of the end-to-end process, in other words, from policy authoring to deployment of low-level rules in the switches. Each experiment was run thirty times for the same reason as explained in the previous experiment. The scenarios were also created using the Mininet emulator. The experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory.

In order to perform the experiments, we created three SLAs (Table 4) with an increasing 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.

[2] http://www.djangoproject.com/
[3] http://www.python.org/
[4] http://getbootstrap.com/
[5] http://www.mysql.com/

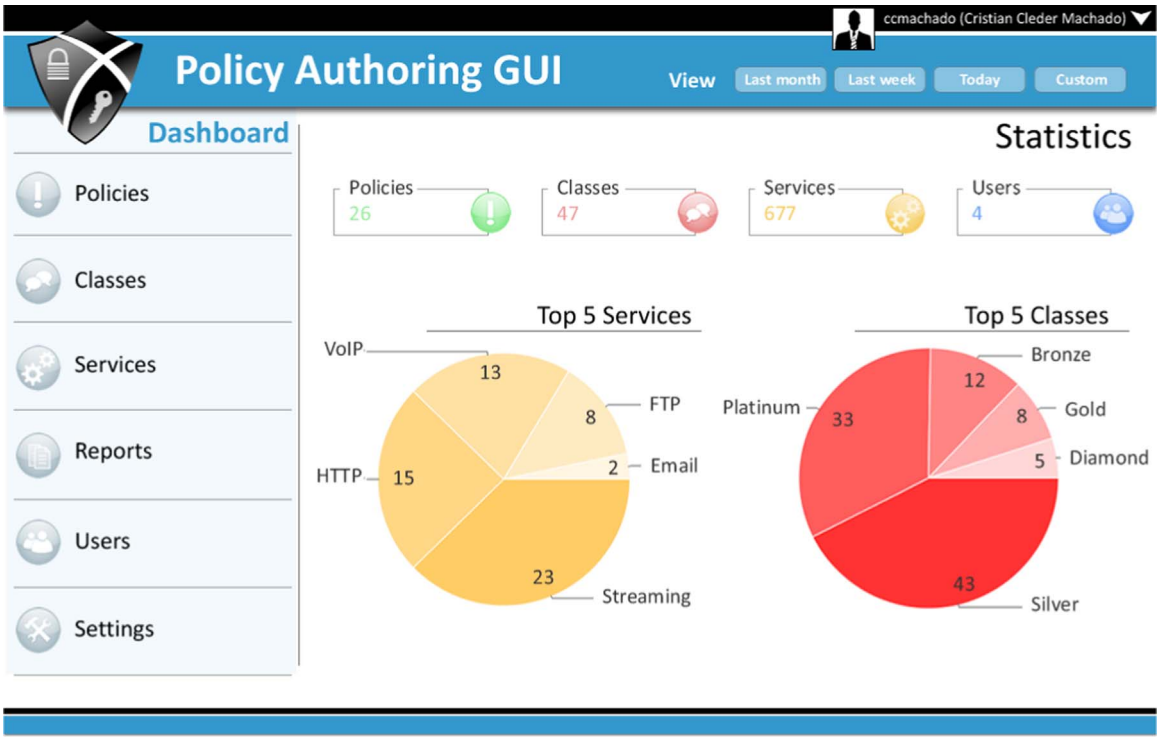



**Fig. 5.** Policy authoring graphical user interface.

**Table 3**
Number of hosts, switches, and links in each scenario.

| Scenario | SwL0 | SwL1 | SwL2 | SwL3 | Host | Link |
|---|---|---|---|---|---|---|
| V | 2 | 2 | 0 | 0 | 4 | 4 |
| W | 4 | 4 | 2 | 0 | 8 | 12 |
| X | 8 | 8 | 4 | 0 | 16 | 32 |
| Y | 16 | 16 | 8 | 4 | 32 | 80 |
| Z | 32 | 32 | 16 | 8 | 64 | 96 |

We also created three scenarios (Table 5) by varying the number of network devices and adding redundant links between some network devices. The scenarios used in the experiments were based on Fat-Tree topologies (Leiserson, 1985). Our goal was to demonstrate the ability of the framework to operate in increasingly large topologies.

We applied the three SLAs to five different repositories and populated each repository according to Table 6. We performed experiments on all variations of SLAs, repositories, and scenarios.

In particular, the experiments described in this section intend to evaluate our prototype in terms of average execution time and percentage of the total time taken by each stage of our policy refinement toolkit.

Fig. 8 shows the average response time for each SLA 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 rule deployment (i.e., install the flow rules in the switches). 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 experiments performed with SLAs 1, 2, and 3. This behavior is expected, since the number of classes has influence on the time spent in queries to obtain the ideal matches between SLAs and QoS classes.

In Fig. 9 the y-axis shows the percentage of the total time taken by each process in the experiments performed with SLAs 1, 2, and 3 in each scenario. From these results we can observe 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.

Fig. 10(a) shows the total number of rules generated by refining each SLA *separately* in each scenario. The total number of rules only includes the amount of specific rules for each SLA, not including any standard rule. 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 fewer 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 created on all network devices. As our approach is based on routing, it creates a spanning tree to find all routes between sources and destinations. Thus, some routes may share common elements 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 increase 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 (Fig. 10(b)). Our framework attempts to fulfill the requirements of each SLA. In order to achieve this, it identifies the possibility of routing 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, Fig. 10(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 (Fig. 8(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 require-

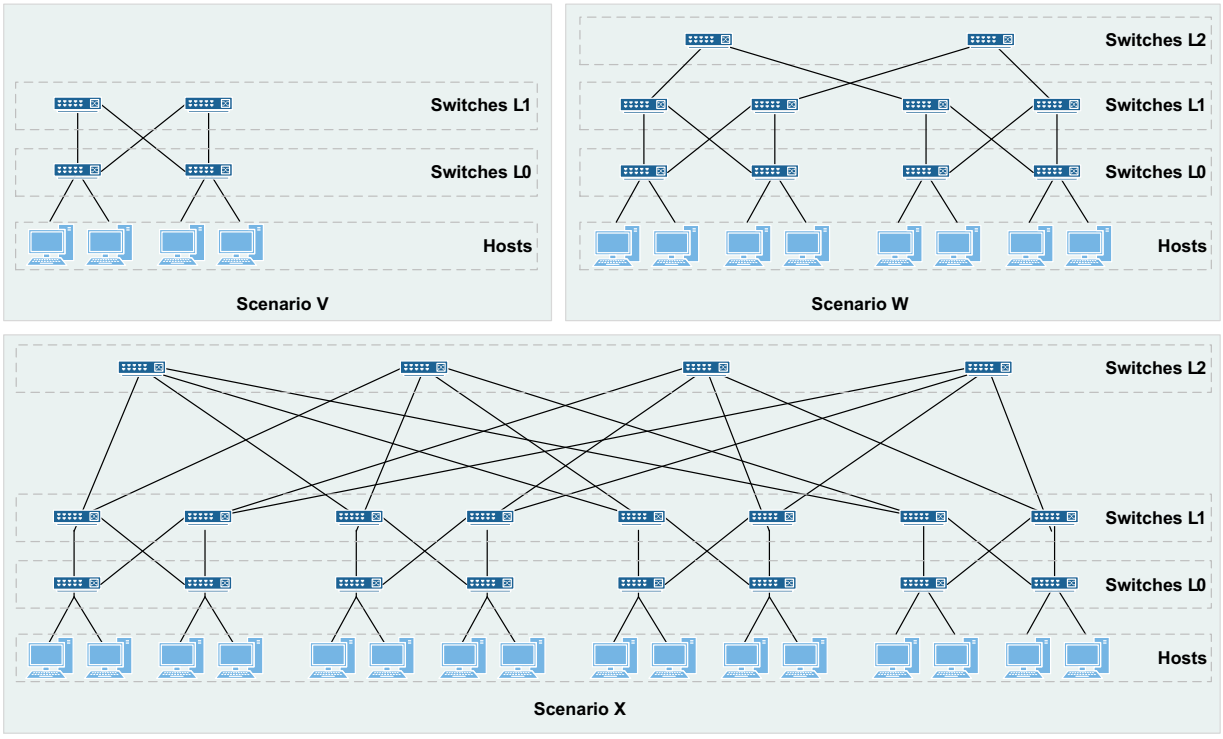
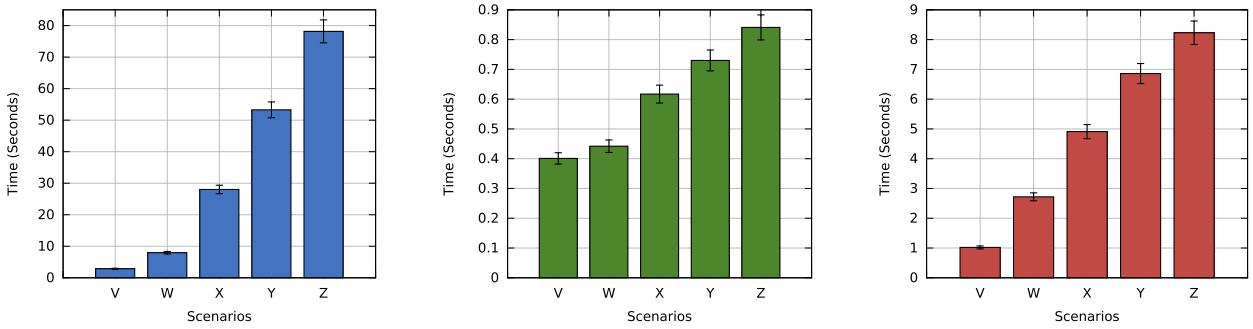



**Fig. 6.** Scenarios for the experiments with increasing number of switches and link redundancy.




(a) Time for calculation and installation of standard rules in the switches in the startup phase and recognition of topology in each experiment.

(b) Time for calculation and installation of specific rules in the switches in each experiment.

(c) Time for recognition of the first path for best-effort delivery between source and destination and calculation and installation of specific rules in the switches in each experiment.

**Fig. 7.** Time for recognition of paths, calculation, and installation of rules in the switches, (a) Time for calculation and installation of standard rules in the switches in the startup phase and recognition of topology in each experiment, (b) Time for calculation and installation of specific rules in the switches in each experiment, (c) Time for recognition of the first path for best-effort delivery between source and destination and calculation and installation of specific rules in the switches in each experiment.

**Table 4**
Description of the SLAs used in the experiments.

| SLA | Description of SLAs |
|---|---|
| $SLA_1$ | HTTP traffic should receive lowest priority. |
| $SLA_2$ | Streaming traffic should receive highest priority, lowest delay, and bandwidth higher than 512kbps. |
| $SLA_3$ | VoIP traffic should receive highest priority, delay lower than 200 ms, lowest jitter, and bandwidth higher than 128kbps. |

**Table 5**
Number of switches and links in each scenario.

| Scenario | SwL0 | SwL1 | SwL2 | SwL3 | SwL4 | Link |
|---|---|---|---|---|---|---|
| X | 16 | 8 | 8 | 4 | 0 | 88 |
| Y | 32 | 16 | 16 | 8 | 4 | 176 |
| Z | 64 | 32 | 32 | 16 | 8 | 210 |

**Table 6**
Number of classes registered in the repository.

| Repository | Number of Classes |
|---|---|
| Repository A | 10 |
| Repository B | 100 |
| Repository C | 1000 |
| Repository D | 10,000 |
| Repository E | 100,000 |

ments of each SLA and according to each scenario.

### 5.2.3. EC-based formalism experiments

In this section, our goal is to evaluate the EC-based formalism and measure the amount of iterations and rules required to find QoS classes that fulfill the requirements of different SLAs. The experimental evaluation was performed in Prolog 6.6.4. Each experiment was run ten times since we were able to obtain a confidence level of 95% with a reduced number of executions. The experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory.

We created six SLAs by changing the number of expressions

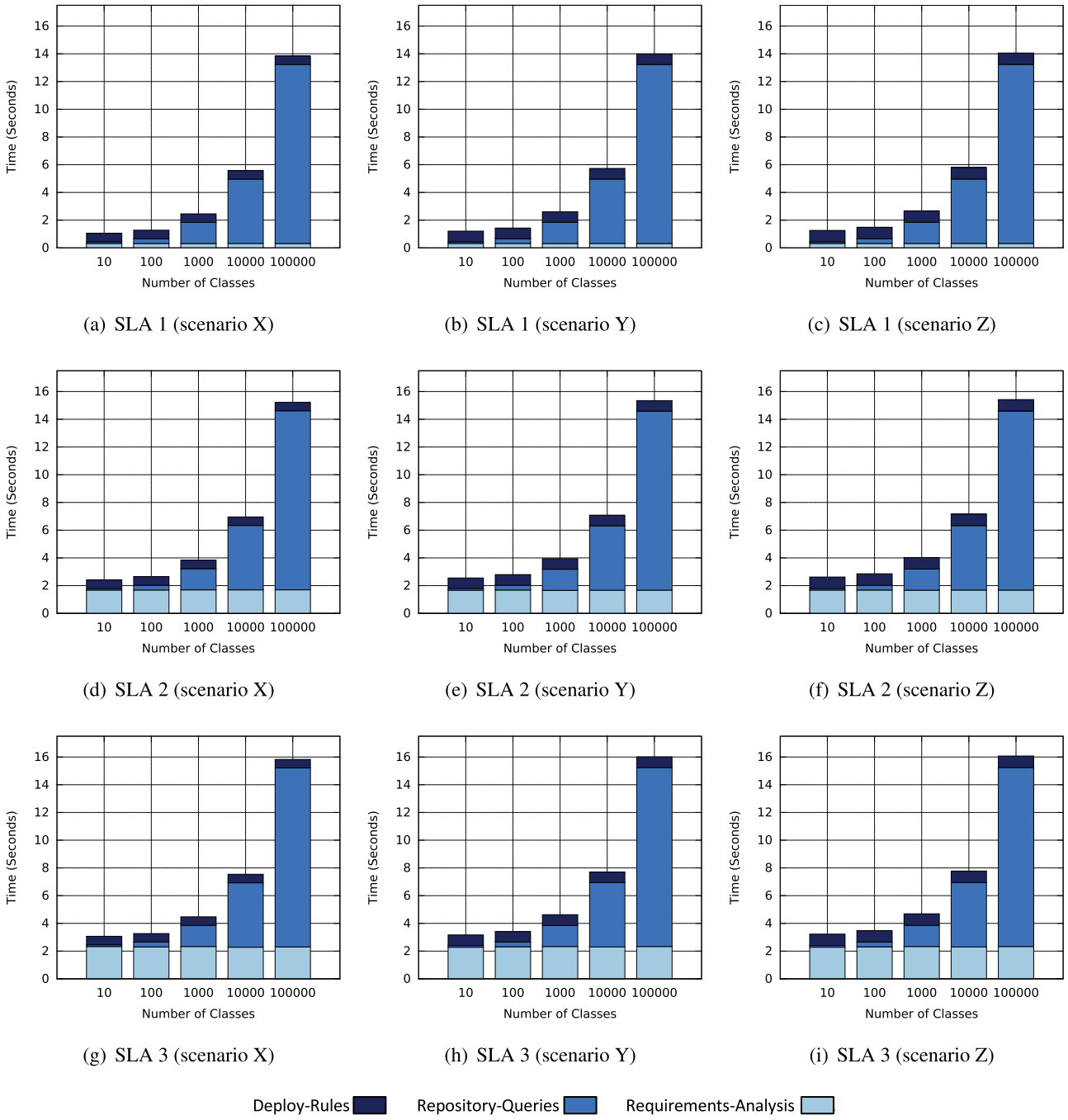



**Fig. 8.** Average response time for SLAs 1, 2, and 3 performed in scenarios X, Y, and Z, (a) SLA 1 (scenario X), (b) SLA 1 (scenario Y), (c) SLA 1 (scenario Z), (d) SLA 2 (scenario X), (e) SLA 2 (scenario Y), (f) SLA 2 (scenario Z), (g) SLA 3 (scenario X), (h) SLA 3 (scenario Y), (i) SLA 3 (scenario Z).

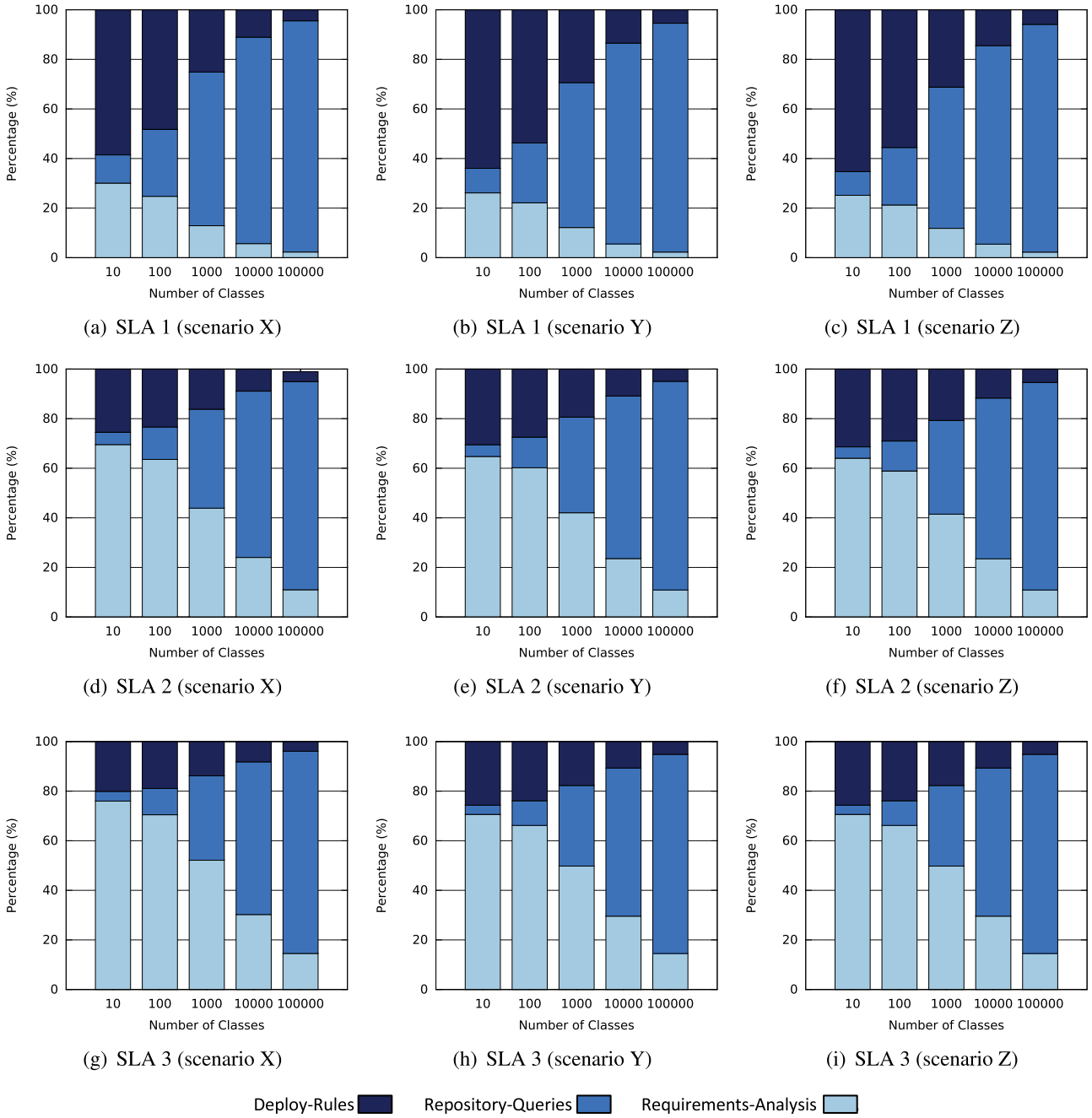



**Fig. 9.** Percentage of total time for SLAs 1, 2, and 3 performed in scenarios X, Y, and Z, (a) SLA 1 (scenario X), (b) SLA 1 (scenario Y), (c) SLA 1 (scenario Z), (d) SLA 2 (scenario X), (e) SLA 2 (scenario Y), (f) SLA 2 (scenario Z), (g)SLA 3 (scenario X), (h) SLA 3 (scenario Y), (i) SLA 3 (scenario Z).

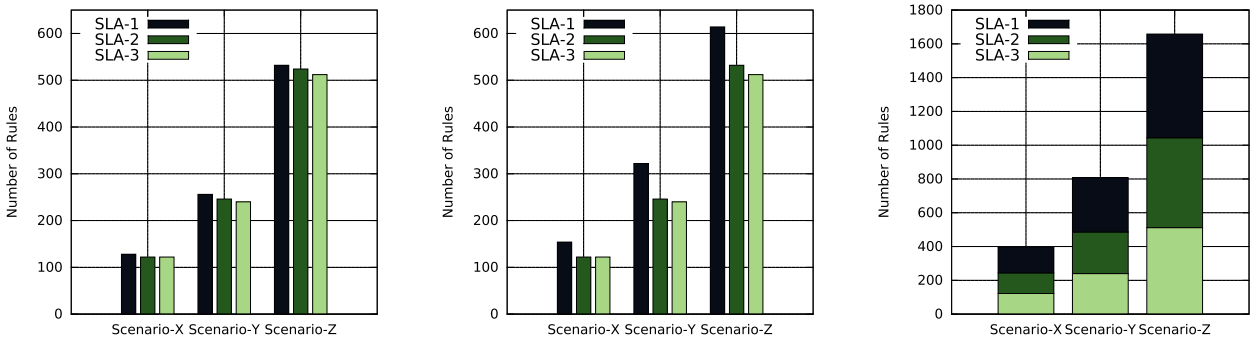
according to Table 7. We applied the six SLAs to five repositories containing different amounts of classes. We populated each repository according to Table 8. Each QoS class considers all QoS requirements, i.e., priority, bandwidth, delay, and jitter. Each QoS requirement has different values.[6]

In this experiment our goal is to measure the number of iterations for the identification of QoS requirements and for querying all suggestions of QoS classes. Each SLA (Table 7) generated a set of prolog rules to find at least one occurrence of a QoS class. Each rule is composed of a set of prolog operations. Each prolog operation is connected by connectives. Connectives can be *"and"* (represented by a comma []) or *"or"* (represented by a semicolon [;]). The prolog rules generated for each SLA are described in Table 9. The prolog operations used in the prolog rules are described in Table 10.

As can be seen in Table 9 some SLAs have generated more rules such as SLA 1(4), SLA 2(3), SLA 4(2), and SLA 6(3). This happens when the rule for an SLA does not match the information stored for QoS classes. Thus, our technique applies a first rule that includes all parameters found in an SLA. If there is not a match, new rules are generated until a match is found. In the worst case, the rule generated will fetch the parameters using the connective *"or"* (represented by a semicolon), for example, $parameter_1$ *or* $parameter_2$ *or* $parameter_3$ *or* $parameter_4$.

Table 11 shows the relationship between the number of iterations generated by each rule and the number of classes found. As can be

[6] 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 1ms and 999ms for delay, and 10% of the delay value for jitter.




(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. 10.** Number of rules deployed in each scenario, (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.

**Table 7**
Description of the SLAs used in the experiments.

| SLA | Description of SLAs |
|---|---|
| $SLA_1$ | Streaming traffic should receive highest priority, lowest delay, lowest jitter, and highest bandwidth. |
| $SLA_2$ | Peer-to-peer traffic should receive lowest priority, highest delay, lowest jitter, and highest bandwidth. |
| $SLA_3$ | FTP traffic should receive highest bandwidth. |
| $SLA_4$ | VoIP traffic should receive delay lower than 20 ms and bandwidth higher than 128kbps. |
| $SLA_5$ | SNMP traffic should receive priority higher than 500 and bandwidth lower than 16kbps. |
| $SLA_6$ | SSH traffic should receive bandwidth higher than 1024kbps, delay lower than 50 ms, and lowest jitter. |

**Table 8**
Number of classes registered in the repository.

| Repository | Number of Classes |
|---|---|
| Repository A | 5 |
| Repository B | 10 |
| Repository C | 50 |
| Repository D | 100 |
| Repository E | 250 |

seen, in all cases at least one class has been found by a rule. In addition, some rules have found multiple classes such as S5R1. This happens because each rule aims to find an ideal match of parameters ignoring the other parameters registered in the class. Thus, a rule requiring priority=12 and bandwidth=128 kbps can identify classes with the following information: Class1: priority=12, bandwidth=128 kbps; Class2: priority=12, bandwidth=12 8kbps, and delay=10 ms; Class3: priority=12, bandwidth=128 kbps, delay=10 ms, and jitter=1 ms; Class4: priority=12, bandwidth=128 kbps, and delay=200 ms.

Fig. 11 shows the number of iterations for each SLA in each repository. Each color indicates a different prolog rule. For example, in Fig. 11(a) the repository with 50 classes performed 4 prolog rules to find at least one class. As previously mentioned, our solution generates prolog rules until a match is found. Thus, in cases where the repository has a smaller number of QoS classes, the tendency is that a larger number of prolog rules will be generated. This happens because the first prolog rule comprises all QoS requirements to find a more precise match, while each next prolog rule generated removes or toggles QoS requirements in order to find usable matches, i.e., QoS classes that fulfill some QoS requirements. As can be observed in Fig. 11, 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 has influence on 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

**Table 9**
Rules performed for each SLA in each scenario.

| SLA | Name[a] | Rule |
|---|---|---|
| 1 | S1R1 | lowestValue(Class,priority,Priority), lowestValue(Class,delay,Delay), lowestValue(Class,jitter,Jitter), highestValue(Class,bandwidth,Bandwidth). |
| | S1R2 | lowestValue(Class,priority,Priority), lowestValue(Class,delay,Delay), highestValue(Class,bandwidth,Bandwidth). |
| | S1R3 | lowestValue(Class,priority,Priority), highestValue(Class,bandwidth,Bandwidth). |
| | S1R4 | lowestValue(Class,priority,Priority), lowestValue(Class,delay,Delay). |
| 2 | S2R1 | highestValue(Class,priority,Priority), highestValue(Class,delay,Delay), highestValue(Class,jitter,Jitter), lowestValue(Class,bandwidth,Bandwidth). |
| | S2R2 | highestValue(Class,priority,Priority), highestValue(Class,delay,Delay), lowestValue(Class,bandwidth,Bandwidth). |
| | S2R3 | lowestValue(Class,bandwidth,Bandwidth), highestValue(Class,priority,Priority). |
| 3 | S3R1 | highestValue(Class,bandwidth,Bandwidth). |
| 4 | S4R1 | findParLower(Class,delay,20), findParHigher(Class,bandwidth,128). |
| | S4R2 | findParLower(Class,delay,20); findParHigher(Class,bandwidth,128). |
| 5 | S5R1 | findParHigher(Class,priority,500), findParLower(Class,bandwidth,16). |
| 6 | S6R1 | findParHigher(Class,bandwidth,1024), findParLower(Class,delay,50), lowestValue(Class,jitter,Jitter). |
| | S6R2 | findParHigher(Class,bandwidth,512), findParLower(Class,delay,20). |
| | S6R3 | findParHigher(Class,bandwidth,1024); findParLower(Class,delay,50); lowestValue(Class,jitter,Jitter). |

[a] Name is the short representation of a rule where S SLA and R Rule. Thus, S1R1 is the first rule for SLA1.

**Table 10**
Description of Operations used in the rules.

| Prolog Operation | Description |
|---|---|
| findParLower(Class,Parameter,Value): – isMemberParameter(Class,Parameter,Value2), Value2 < Value. | Used to find a parameter *lower* than a specific value. |
| findParHigher(Class,Parameter,Value): – isMemberParameter(Class,Parameter,Value2), Value2 > Value. | Used to find a parameter *higher* than a specific value. |
| findParIdentical(Class,Parameter,Value): – isMemberParameter(Class,Parameter,Value2), Value2=Value. | Used to find a parameter *identical* the specific value. |
| lowestValue(Class,Parameter,Value): – isMemberParameter(Class,Parameter,Value), (isMemberParameter(Class2,Parameter2,Value2), Parameter=Parameter2, Value2 < Value, Class != Class2). | Used to find the QoS Class which has the *lowest* parameter among the registered QoS Classes. |
| highestValue(Class,Parameter,Value): – isMemberParameter(Class,Parameter,Value), (isMemberParameter(Class2,Parameter2,Value2), Parameter=Parameter2, Value2 > Value, Class != Class2). | Used to find the QoS Class which has the *highest* parameter among the registered QoS Classes. |

**Table 11**
Results of the iterations and classes found for each scenario.

| | | Number of Classes | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 5 | | 10 | | 50 | | 100 | | 250 | |
| SLA | Rule[a] | Iterat. | Found | Iterat. | Found | Iterat. | Found | Iterat. | Found | Iterat. | Found |
| 1 | S1R1 | 158 | 1 | 332 | 0 | 2617 | 0 | 4747 | 0 | 11,573 | 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 | 77 | 0 | 137 | 0 | 1783 | 1 | 1352 | 0 | 10,763 | 0 |
| | S2R2 | 77 | 0 | 137 | 0 | N/A | N/A | 1352 | 0 | 10,763 | 0 |
| | S2R3 | 118 | 1 | 270 | 1 | N/A | N/A | 12,218 | 1 | 40,923 | 1 |
| 3 | S3R1 | 68 | 1 | 289 | 2 | 1743 | 1 | 2137 | 2 | 13,563 | 6 |
| 4 | S4R1 | 19 | 0 | 12 | 0 | 58 | 2 | 114 | 4 | 290 | 13 |
| | S4R2 | 14 | 1 | 24 | 1 | N/A | N/A | N/A | N/A | N/A | N/A |
| 5 | S5R1 | 14 | 2 | 33 | 5 | 125 | 20 | 293 | 45 | 718 | 107 |
| 6 | S6R1 | 7 | 0 | 12 | 0 | 564 | 3 | 2126 | 4 | 16,579 | 13 |
| | S6R2 | 7 | 0 | 12 | 0 | N/A | N/A | N/A | N/A | N/A | N/A |
| | S6R3 | 100 | 1 | 314 | 2 | N/A | N/A | N/A | N/A | N/A | N/A |

[a] Rules described in Table 9. The occurrence of an N/A means that a previous rule found a class.

of iterations because they require finding lowest or highest values among all possible values stored in the repository.

## 6. Open issues

Several points that were not part of the proposed goals in this work call for our attention. As mentioned previously, we defined a formal representation of high-level policies in the form of SLAs using Event Calculus and used logical reasoning to model both the system behavior and the policy refinement process in SDN. Although this formalism is not integrated with the refinement toolkit, it can be incorporated for validation of the toolkit operations and of the properties of each predicate using logical inferences achieved with the aid of an interpreter, such as a Prolog engine. The integration with an interpreter can be used to increase the reliability of the refinement process, since all SLAs could be modeled as logical predicates and the mapping to low-level rules could be automatically performed via logical inferences. However, by incorporating an interpreter, there is a possible increase in the response time of the processes, due to the set of inferences to be performed. Thus, it is important to analyze and select the set of processes and predicates requiring logical inferences, so that the time spent with an interpreter has the lowest operating cost.

It is noteworthy that many tools, techniques, and algorithms can be exploited to improve the toolkit processes. An example the replacement of the repository. When observing the toolkit performance experiments, we found that in cases where the amount of information in the repository increases, there is an exponential growth in response time for the queries. Replacing the repository for databases that support semantic metadata can dramatically improve the response time for the queries. In addition, we are aware of the limitations of our database schema to organize and store information about network devices, services, and links. We chose to present a simple and understandable organization, maintaining the focus only on our policy refinement solution. However, by using a network management schema such as DMTFs Common Information Model (CIM) there is a possible improvement to represent and define how to manage the set of network elements and their relationships.

Regarding the controller phases, each phase comprises a set of modules responsible for specific actions. In the events phase, we only observed network services and analyzed the duration of each flow. However, several modules can be implemented to assist the decisions taken upon the arrival of new flows and even concerning the occurrence of events in network elements. An example is the creation of a module that identifies new network elements. Through the identification of a new element, a reformulation of the routes can be performed. Another example is the development of a module that monitors the processing overhead on switches. Through this monitoring, a load balancing process can be activated, for example, to reorganize routes between network elements.

We also noted some questions regarding the toolkit scalability. When we observe the experimental results related to the sum of low-level rules deployed in each switch, it is evident that a large set of classes leads to an increase in the number of rules produced. It is important to emphasize that the toolkit induces the operator to register each SLA with existing QoS classes, aiming to restrict the growth of the number of classes, which results in a reduction on the number of rules produced. Despite this, the flexibility provided by the toolkit for creation and modification of QoS classes does not limit the maximum number of classes that can be created. Thus, a study of possible toolkit configurations/characteristics can reduce the number of rules pro-

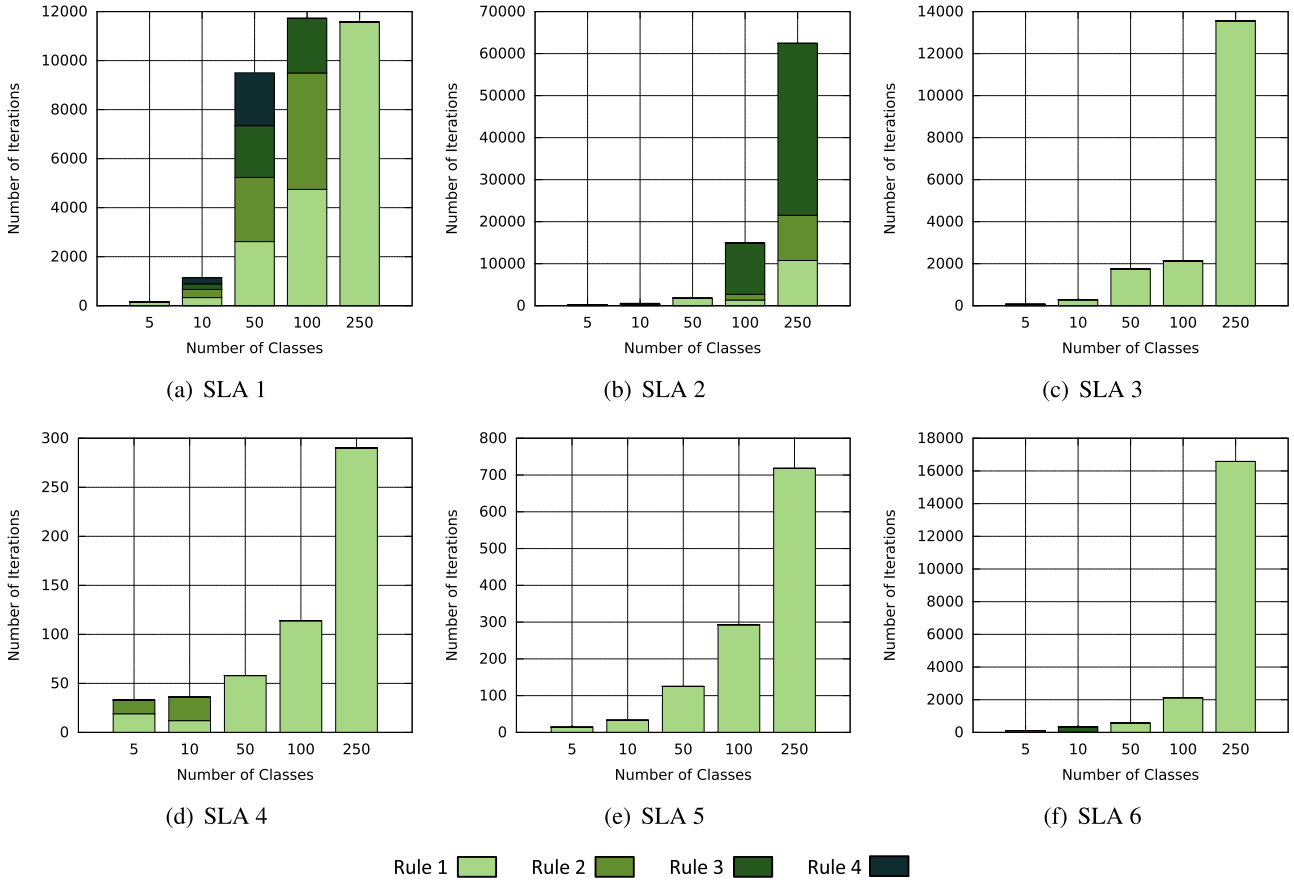



**Fig. 11.** Average number of iterations for each SLA performed in each scenario, (a) SLA 1, (b) SLA 2, (c) SLA 3, (d) SLA 4, (e) SLA 5, (f) SLA 6.

duced, optimizing the use of computing and network resources. For example, limiting the creation of QoS classes or inducting the accommodation of an SLA with a QoS class based on service behavior.

Another issue that may be better explored is the way network information is gathered. We have used ICMP packets and applied a cross multiplication to calculate information/parameters/weights of links between network elements. In this context, other approaches, techniques, and tools can be explored, for example, entropy-based classification algorithms (Giotis et al., 2014) used to estimate and observe the baseline distribution related to network traffic or Iperf[7] tool used to measure network throughput.

## 7. Concluding remarks

Despite the benefits of SDN, the expected behavior of network elements is still defined by static rules written to handle specific circumstances. This approach presents several problems such as human work overload to write, analyze, and manage a large set of hard-coded rules. It also limits or prohibits the development and deployment of new services and resources that were not anticipated when rules were written in the controller. Finally, low-level rules are difficult to analyze and do not provide guarantees for compliance with high-level goals.

In this work, we presented a high-level policy refinement toolkit for SDN management. We aim to remove much of the manual workload of administrators in the configuration of network elements. In particular, we focused on the refinement of QoS requirements for different applications and services (specified in SLAs) into the configuration of controllers and switches. As a result of our toolkit, we identified the resources that need to be configured in accordance with the SLAs, and successfully executed reactive dynamic actions for the reconfiguration of the infrastructure.

Our experiments have shown that the initial recognition of topology and collection of network performance metrics performed during the controller's startup phase improves the deployment time of specific rules for each service. Also, our policy authoring performs well even with the increase in the number of QoS classes and in the complexity of the SLAs. Furthermore, we performed experiments to analyze the amount of iterations and suggestion of classes for the rules created by our solution, showing that the refinement technique found suggestions of QoS classes to match different SLAs.

As part of our future work, we intend to analyze the broader applicability of the toolkit when managing other types of services and functions. Thus, with some adaptations, the toolkit can be used to perform configuration and manage services and network functions such as access control and firewall. In addition, we intend to investigate techniques for detection and resolution of policy conflicts. One common source of policy conflicts is the refinement process itself, during the translation of high-level goals into implementable low-level policies (Lupu and Sloman, 1999). Moreover, we intend to extend the policy authoring framework to support more terms, expressions, and rules. Finally, 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 within a given time frame.

[7] http://iperf.fr/

## References


Aib, I., Boutaba, R., 2007. On leveraging policy-based management for maximizing business profit. IEEE Trans. Netw. Serv. Manag. 4 (3), 25–39. http://dx.doi.org/10.1109/TNSM.2007.021104.

Bakshi, K., 2013. Considerations for software defined networking (SDN): Approaches and use cases. In: Aerospace Conference, 2013 IEEE, pp. 1–9.

Bandara, A., Lupu, E., Moffett, J., Russo, A., 2004. A goal-based approach to policy refinement. In: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY '04, IEEE Computer Society, pp. 229–239.

Bandara, A., Lupu, E., Russo, A., Dulay, N., Sloman, M., Flegkas, P., Charalambides, M., Pavlou, G., 2005. Policy refinement for diffserv quality of service management. In: Integrated Network Management, 2005. IM 2005. 2005 In: Proceedings of the 9th IFIP/IEEE International Symposium on, pp. 469–482. ⟨http://dx.doi.org/10.1109/INM.2005.1440817⟩.

Bandara, A.K., Lupu, E.C., 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, IEEE Computer Society, pp. 26–39.

Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., Weiss, W., 1998. An architecture for differentiated services, Tech. rep., Internet Engineering Task Force (IETF), RFC 2475 (December).

Canos, J.H., Lozano, M., Ramos, I., 1998. Kaos: an object-oriented software tool for the objects definition, updating, querying and programming in an object-oriented environment. In: Electrical and Computer Engineering, 1998. IEEE Canadian Conference on, Vol. 2, pp. 711–714 vol.2. ⟨http://dx.doi.org/10.1109/CCECE.1998.685596⟩.

Carey, K., Wade, V., 2008. Using automated policy refinement to manage adaptive composite services. In: Network Operations and Management Symposium Workshops, 2008. NOMS Workshops 2008. IEEE, pp. 239–247. ⟨http://dx.doi.org/10.1109/NOMSW.2007.40⟩.

Craven, R., Lobo, J., Lupu, E., Russo, A., Sloman, M., 2011. Policy refinement: Decomposition and operationalization for dynamic domains. In: Network and Service Management (CNSM), 2011 Proceedings of the 7th International Conference on, pp. 1–9.

Foster, N., Harrison, R., Freedman, M.J., Monsanto, C., Rexford, J., Story, A., Walker, D., 2011. Frenetic: a network programming language. SIGPLAN Not. 46 (9), 279–291. http://dx.doi.org/10.1145/2034574.2034812.

Giotis, K., Argyropoulos, C., Androulidakis, G., Kalogeras, D., Maglaris, V., 2014. Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments. Comput. Netw. 62 (0), 122–136.

Hakiri, A., Gokhale, A., Berthou, P., Schmidt, D.C., Gayraud, T., 2014. Software-defined networking: challenges and research opportunities for future internet. Comput. Netw. 75 (Part A), 453–471. http://dx.doi.org/10.1016/j.comnet.2014.10.015.

Han, W., Lei, C., 2012. A survey on policy languages in network and security management. Comput. Netw. 56 (1), 477–489. http://dx.doi.org/10.1016/j.comnet.2011.09.014.

Kim, H., Feamster, N., 2013. Improving network management with software defined networking. IEEE Commun. Mag. 51 (2), 114–119. http://dx.doi.org/10.1109/MCOM.2013.6461195.

Koch, T., Krell, C., Kraemer, B., 1996. Policy definition language for automated management of distributed systems. In: Proceedings of the 2nd IEEE International Workshop on Systems Management (SMW'96), SMW '96, IEEE Computer Society, Washington, DC, USA, pp. 55.

Kowalski, R., Sergot, M., 1986. A logic-based calculus of events. New Gen. Comput. 4 (1), 67–95. http://dx.doi.org/10.1007/BF03037383.

Leiserson, C., 1985. Fat-trees: universal networks for hardware-efficient supercomputing. IEEE Trans. C. Comput. 34 (10), 892–901. http://dx.doi.org/10.1109/TC.1985.6312192.

Liu, K., ZhuGe, B., Wang, W., 2009. A design method of synthetic network management system based on ForCES protocol. In: Computer Sciences and Convergence Information Technology, 2009. ICCIT '09. In: Proceedings of the Fourth International Conference on, pp. 737–741. ⟨http://dx.doi.org/10.1109/ICCIT.2009.184⟩.

Lupu, E.C., Sloman, M., 1999. Conflicts in policy-based distributed systems management. IEEE Trans. Softw. Eng. 25 (6), 852–869. http://dx.doi.org/10.1109/32.824414.

Lupu, E., Sloman, M., Dulay, N., Damianou, N., 2000. Ponder: realising enterprise viewpoint concepts. In: Enterprise Distributed Object Computing Conference, 2000. EDOC 2000. Proceedings. Fourth International, pp. 66–75. ⟨http://dx.doi.org/10.1109/EDOC.2000.882345⟩.

Machado, C.C., Granville, L.Z., Schaeffer-Filho, A., Wickboldt, J.A., 2014. Towards SLA policy refinement for QoS management in software-defined networking. In: Advanced Information Networking and Applications (AINA), 2014 IEEE Proceedings of the 28th International Conference on, pp. 397–404. ⟨http://dx.doi.org/10.1109/AINA.2014.148⟩.

Machado, C.C., Wickboldt, J.A., Granville, L.Z., Schaeffer-Filho, A., 2015a. An EC-based formalism for policy refinement in software-defined networking. In: Proceedings of the 20th IEEE Symposium on Computers and Communication (ISCC) (ISCC2015), pp. 464–469.

Machado, C.C., Wickboldt, J.A., Granville, L.Z., Schaeffer-Filho, A., 2015b. Policy authoring for software-defined networking management. In: Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on, pp. 216–224. ⟨http://dx.doi.org/10.1109/INM.2015.7140295⟩.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J., 2008. Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38 (2), 69–74. http://dx.doi.org/10.1145/1355734.1355746.

Moffett, J., Sloman, M., 1993. Policy hierarchies for distributed systems management. IEEE J. Sel. Areas Commun. 11 (9), 1404–1414. http://dx.doi.org/10.1109/49.257932.

Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D., 2013. Composing software-defined networks. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, nsdi'13, USENIX Association, Berkeley, CA, USA, pp. 1–14.

Nunes, B., Mendonca, M., Nguyen, X.-N., Obraczka, K., Turletti, T., 2014. A survey of software-defined networking: past, present, and future of programmable networks. IEEE Commun. Surv. Tutor. 16 (3), 1617–1634. http://dx.doi.org/10.1109/SURV.2014.012214.00180.

OpenFlow, OpenFlow Switch Specification - Version 1.0.0 (Wire Protocol 0×01), Tech. rep., Open Networking Foundation (ONF), available at: . (accessed January 2015) (December 2009). ⟨https://www.opennetworking.org/sdn-resources/technical-library⟩.

Pueschel, T., Putzke, F., Neumann, D., 2012. Revenue management for cloud providers–a policy-based approach under stochastic demand. In: System Science (HICSS), 2012 Proceedings of the 45th Hawaii International Conference on, IEEE, pp. 1583–1592. ⟨http://dx.doi.org/10.1109/HICSS.2012.505⟩.

Rubio-Loyola, J., Galis, A., Astorga, A., Serrat, J., Lefevre, L., Fischer, A., Paler, A., Meer, H., 2011. Scalable service deployment on software-defined networks. IEEE Commun. Mag. 49 (12), 84–93. http://dx.doi.org/10.1109/MCOM.2011.6094010.

Rubio-Loyola, J., Serrat, J., Charalambides, M., Flegkas, P., Pavlou, G., 2006. A functional solution for goal-oriented policy refinement. In: Policies for Distributed Systems and Networks, 2006. Policy 2006. In: Proceedings of the Seventh IEEE International Workshop on, pp. 133–144. ⟨http://dx.doi.org/10.1109/POLICY.2006.5⟩.

Sezer, S., Scott-Hayward, S., Chouhan, P., Fraser, B., Lake, D., Finnegan, J., Viljoen, N., Miller, M., Rao, N., 2013. Are we ready for SDN? Implementation challenges for software-defined networks. IEEE Commun. Mag. 51 (7), 36–43. http://dx.doi.org/10.1109/MCOM.2013.6553676.

Shanahan, M., 2000. An abductive event calculus planner. J. Log. Program. 44 (1), 207–240.

Verma, D., 2002. Simplifying network administration using policy-based management. IEEE Netw. 16 (2), 20–26. http://dx.doi.org/10.1109/65.993219.

Waller, A., Sandy, I., Power, E., Aivaloglou, E., Skianis, C., Muñoz, A., Maña, A., 2011. Policy based management for security in cloud computing. In: Secure and Trust Computing, Data Management, and Applications, Springer, pp. 130–137.

Wickboldt, J., de Jesus, W., Isolani, P., Both, C., Rochol, J., Granville, L., 2015. Software-defined networking: management requirements and challenges. IEEE Commun. Mag. 53 (1), 278–285. http://dx.doi.org/10.1109/MCOM.2015.7010546.


**Cristian Cleder Machado** is a Ph.D. student in computer science at the Federal University of Rio Grande do Sul (UFRGS), in Brazil. He achieved his B. Sc. degree in computer science at the Integrated Regional University (URI) - Campus of Frederico Westphalen, in 2006. He received his M. Sc. degree in computer science from UFRGS in 2015. He also is a professor at the Integrated Regional University (URI) since 2010. His research interests include Policy-Based Network Management (PBNM), Network Functions Virtualization (NFV), Software-Defined Networking (SDN), and Network Resilience.

**Juliano Araujo Wickboldt** is a Postdoctoral researcher at the Federal University of Rio Grande do Sul (UFRGS) in Brazil. He achieved his B.Sc. degree in computer science at Pontifical Catholic University of Rio Grande do Sul in 2006. He also holds an M.Sc. degree from UFRGS conducted in a joint project with HP Labs Bristol and Palo Alto. He received his Ph.D. degree in computer science from UFRGS in 2015. Juliano was an intern at NEC Labs Europe in Heidelberg, Germany for one year between 2011 and 2012. Between 2013 and 2014, Juliano was a substitute professor at UFRGS. His current research interests include cloud resource management and software-defined networking.

**Lisandro Zambenedetti Granville** is associate professor at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS), Brazil. He received his M. Sc. and Ph.D. degrees, both in computer science, from UFRGS in 1998 and 2001, respectively. Lisandro has served as a TPC co-chair of IFIP/IEEE DSOM 2007, IFIP/IEEE NOMS 2010, TPC vice-chair of CNSM 2010, and general co-chair of CNSM 2014. He is also chair of the IEEE Communications Society Committee on Network Operations and Management (CNOM), co-chair of the Network Management Research Group (NMRG) of the Internet Research Task Force (IRTF), and president of the Brazilian Computer Society (SBC). His areas of interest include management of virtualization for the Future Internet, P2Pbased services and applications, Software-Defined Networking (SDN), and Network Functions Virtualization (NFV).

**Alberto Schaeffer-Filho** is an Associate Professor at the Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS). He obtained his Ph.D. in Computing from Imperial College London, UK, in 2009. Between 2009 and 2012, he worked as a Research Associate at the School of Computing and Communications (SCC), Lancaster University, UK. His research interests include network management, security and resilience, Network Functions Virtualization (NFV), Software- Defined Networking (SDN) and Policy-Based Network Management (PBNM). See http://www.inf.ufrgs.br/~alberto for selected papers.