PonderFlow: A New Policy Specification Language to SDN OpenFlow-based Networks

Bruno Lopes Alcantara Batista Marcial Porto Fernandez
Universidade Estadual do Ceará (UECE)
Av Silas Munguba 1700 - Fortaleza/CE - Brazil
{bruno,marcial}@larces.uece.br

***Abstract*—The SDN/OpenFlow architecture is a proposal from the Clean Slate initiative to define a new Internet architecture where network devices are simple, and the control plane and management are performed on a centralized controller, called Openflow controller. Each Openflow controller provides an Application Programming Interface (API) that allows a researcher or a network administrator to define the desired treatment to each flow inside controller. However, each Openflow controller has its own standard API, requiring users to define the behavior of each flow in a programming or scripting language. It also makes difficult for the migration from one controller to another one, due to the different APIs. This paper proposes the PonderFlow, an extension of Ponder language to OpenFlow network policy specification. The PonderFlow extends the original Ponder specification language allowing to define an Openflow flow rule abstractly, independent of Openflow controller used. Some examples of OpenFlow policy will be evaluated showing its syntax and the grammar validation.**

Keywords–Openflow; OpenFlow Controller; Policy-based Network Management; Policy Definition Language

I. INTRODUCTION

This paper is an extended version of the paper presented in The Thirteenth International Conference on Networks (ICN 2014) [1]. Comparing to the original paper, this one shows more description examples and evaluates the parser implementation to validate the proposal.

The Software Defined Network (SDN) architecture with OpenFlow protocol is a proposal of the Clean Slate initiative to define an open protocol to sets up forward tables in switches [2]. It is the basis of the SDN architecture, where the network can be modified dynamically by the user, and the control-plane are decoupled from the data-plane. The OpenFlow proposal tries to use the most basic abstraction layer of the switch to achieve better performance. The OpenFlow protocol can set a condition-action tuple on switches like forward, filter and also, count packets from a specific flow that match a condition.

The network management is performed by the OpenFlow Controller maintaining the switches simple, only with the packet forwarding function. This architecture provides several benefits: (1) OpenFlow controller can manage all flow decisions reducing the switch complexity; (2) A central controller can see all networks and flows, giving global and optimal management of network provisioning; (3) OpenFlow switches are relatively simple and reliable, since forward decisions are defined by a controller, rather than by a switch firmware. However, as the number of switches increases in a computer network and it becomes more complex to manage the switches flows, it is necessary to use a tool to help the network administrator to manage the flows in order to modify dynamically the system behavior.

A policy-based tool can reduce the complexity inherent to this kind of problem. It is a way to manage a large network environment, where the behavior of the network assets may change over time.

Policy-Based Network Manager (PBNM) is the technology that provides the tools for automated network using policies to describe environment behavior abstractly. The PBNM can help network administrators to manage OpenFlow networks simply defining policies, where a policy is a set of rules to govern all the system.

This paper presents the PonderFlow, an extension of Ponder policy specification language. Ponder is a declarative, object-oriented language for specifying management and security policy proposed by Damianou et al. [3]. The PonderFlow provides the necessary resources to define or remove flows, grant privileges to a user, add or remove flows (authorization policy) and force a user or system to execute an action before a particular event (obligation policy).

The rest of the paper is structured as follows. In Section II, we present some related work about OpenFlow policy specification languages. Section III introduces the OpenFlow, the Policy-Based Openflow Network Manager (PBONM) architecture and introduces the Ponder specification language. In Section IV, we present the PonderFlow language, and its respective grammar and validation. In Section VI, we conclude the paper and present some future works.

II. RELATED WORK

Foster et al. [4] designed and implemented the Frenetic, a set of Python's libraries for network programming to provide several high-level features for OpenFlow/NOX [5] programming issues. Frenetic is based on functional reactive programming, a paradigm in which programs manipulate streams of values, delivering the need to write event-driven programs leading a unified architecture where programs "see every packet" rather than processing traffic indirectly by manipulating switch-level rules. However, the network administrator needs to use a programming language, Python [6] in this case, to define the behavior of OpenFlow network.

Mattos et al. [7] propose an OpenFlow Management Infrastructure (OMNI) for controlling and managing OpenFlow networks and also for allowing the development of autonomous applications for these networks. OMNI provides a web interface with set of tools to manage and control the network, and the network administrators interact through this interface. The outputs of all OMNI applications are eXtensible Markup Language (XML), simplifying the data interpretation by other applications, agents or human operators. However, the network administrator needs to use a programming language to call any OMNI function using a web Application Programming

Interface (API) or access the web interface and proceed manually.

Voellmy et al. proposed Procera [8], a controller architecture and high-level network control language that allow to express policies in the OpenFlow controllers. Procera applies the principles of functional reactive programming to provide an expressive, declarative and extensible language. Users can extend the language by adding new constructors.

The PonderFlow has similarities with Procera and Frenetic, but our main goal is to create a policy specification language decoupled from the conventional programming languages, and also, regardless of the OpenFlow controller used. The PonderFlow language is an extension of Ponder language and can be easily ported to another OpenFlow controller. As Ponder is a well-known policy language, the validations is not necessary. In this work, we used the Java language to implement the parser and lexical analyses in Floodlight OpenFlow controller [9]. We want to achieve a level of independence from the programming language and of the OpenFlow controllers. This paper presents the PonderFlow, an extensible, declarative language for policy's definition in an OpenFlow network.

III. OpenFlow Policy Architecture

In this section, we introduce SDN architecture and the Policy-based network management concepts. We also show the application of Policy-based management architecture in OpenFlow environment.

A. OpenFlow

The SDN architecture has several components: the OpenFlow controller, one or many OpenFlow devices (switch), and the OpenFlow protocol. This approach considers a centralized controller that configures all devices. Devices should be kept simple in order to reach better forward performance and leave the network control to the controller.

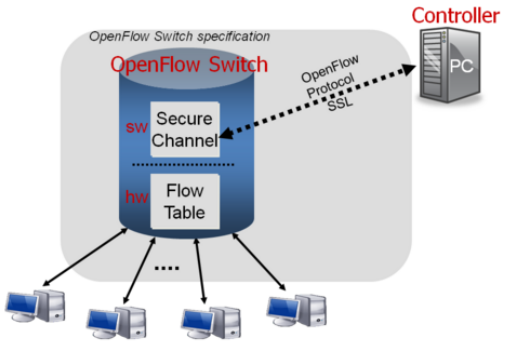

Figure 1. The OpenFlow architecture [2]

The OpenFlow Controller is the centralized controller of an OpenFlow network. It sets up all OpenFlow devices, maintains topology information, and monitors the overall status of entire network. The device is any capable OpenFlow device on a network such as a switch, router or access point. Each device maintains a Flow Table that indicates the processing applied to any packet of a certain flow. There are several OpenFlow controllers available, e.g., NOX [5], FloodLight [9], Beacon [10], POX [11], and Trema [12].

The OpenFlow Protocol works as an interface between the controller and the OpenFlow devices setting up the Flow Table. The protocol should use a secure channel based on Transport Layer Security (TLS). The controller updates the *Flow Table* by adding and removing Flow Entries using the OpenFlow Protocol. The Flow Table is a database that contains Flow Entries associated with actions to command the switch to apply some actions on a certain flow. Some possible actions are: forward, drop, and encapsulate.

The Openflow Controller presents two behaviors: reactive and proactive. In the **Reactive** approach, the first packet of flow received by switch triggers the controller to insert flow entries in each OpenFlow switch of network. This approach presents the most efficient use of existing flow table memory, but every new flow incurs in a small additional setup time. Finally, with hard dependency of the controller, if a switch lost the connection, it has limited utility.

In the **Proactive** approach, the controller pre-populates flow table in each switch. This approach has zero additional flow setup time because the forward rule is defined. Now, if the switch loss the connection with controller it does not disrupt traffic. However, the network operation requires a hard management, e.g., requires to aggregate (wildcard) rules to cover all routes.

Each device has a Flow Table with flow entries as shown in Figure 2. A Flow Entry has three parts: rule match fields, an action and statistics fields and byte counters. The *rule match fields* is used to define the match condition to a specific flow. *action* defines the action to be applied to an exact flow, and *statistical fields* are used to count the rule occurrence for management purposes.

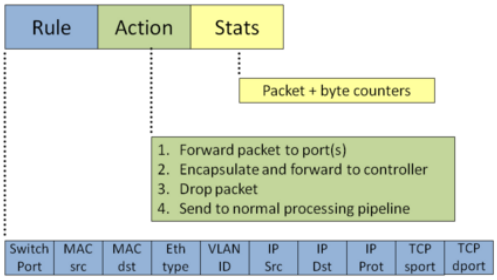

Figure 2. The OpenFlow Flow Entry [13]

When a packet arrives to the OpenFlow Switch, it is matched against *flow entries* in the *flow table*, and the action will be triggered if the header field is matched and then updates the counter.

If the packet does not match any entry in the *flow table*, the packet will be sent to the controller over a secure channel. Packets are matched against all *flow entries* based on a prioritization, where each *flow entry* on *flow table* has a priority associated. Higher numbers have higher priorities.

The OpenFlow Protocol uses the TCP protocol and port 6633. Optionally, the communication can use a secure channel based on TLS.

The OpenFlow Protocol supports three types of messages [13]:

1) Controller-to-Switch Messages: These messages are sent only by the controller to the switches, and they perform the functions of switch configuration, exchange information on the switch capabilities and also manage the Flow Table.

2) Symmetric Messages: These messages are sent in both directions reporting on switch-controller connection problems.

3) Asynchronous Messages: These messages are sent by the switch to the controller to announce changes in the network and switch state.

All packets received by the switch are compared against the Flow Table. If the packet matches any Flow Entry, the action for that entry is performed on the packet, e.g., forward a packet to a specified port. If there is no match, the packet is forwarded to the controller that is responsible for determining how to handle packets without valid Flow Entries [13].

It is important to note that when the OpenFlow switch receives a packet to a nonexistent destination in the Flow Table, it requires an interaction with the controller to define the treatment of this new flow. At least, the switch will need to send a message to the controller with regards to the unknown packet received (message Packet-In). The controller needs to configure all switches along the path from source to destination (message Modify-State). In a network with N switches, we can estimate the need of $6 \times (N + 1)$ messages for each new flow, considering the start and end of TCP connection, the Packet-in and Modify Flow Entry Messages [13]. If the path is already pre-defined (there is an entry in Flow Table), this procedure is not necessary, reducing the amount of messages exchanged through the network and reducing the processing at the controller.

Furthermore, the maintenance of old Flow Entries in the switch Flow Tables gives waste of fast Ternary Content-Addressable Memory (TCAM) memory. Therefore, it is required to remove unused flows using a time-out mechanism. If a flow previously excluded by time-out restarts, it is required to reconfigure all switches on the end-to-end path.

An OpenFlow device is basically an Ethernet switch with OpenFlow protocol. However, there are different implementation approaches: OpenFlow-enable switch and OpenFlow-compliant switch.

The OpenFlow-enable switch uses off-the-shelf hardware, i.e., traditional switches with OpenFlow protocol that translate the rule according to the hardware chipset implementation. The OpenFlow enable-switch re-use existing TCAM, that in a conventional switch has no more than only few thousands of entries for IP routing and MAC table. Considering that we need at least one TCAM entry per flow, in a current hardware would be not enough for production environments. The Broadcom chipset switches based on Indigo Firmware [14], e.g., Netgear 73xxSO, Pronto Switch and many other, are an example of this approach.

The OpenFlow-compliant switch uses a specific network chipset, designed to provide better performance to OpenFlow devices. OpenFlow philosophy relies on matching packets against multiple tables in the forwarding pipeline, where the output of one pipeline stage being able to modify the contents of the table of next stage. Some examples are devices based on the EZChip NP-4 Network Processor [15] and Intel FM-6000 [16]. But, nowadays, there are few commercial OpenFlow-compliant switches; one example is the NoviFlow Switch 1.1 [17].

B. Policy Based Network Management

In traditional network management, policies are hard coded and require manual intervention o be modified. The costs of configuration the network results in a manpower intensive task, and it can result a significant portion of network operations because [18]:

- There are many network elements to be configured;
- Network problems require manual intervention;
- Dynamic user demands or network conditions require repeated reconfiguration;
- Manually maintaining consistency and coherence across and between systems is error prone.

PBNM has emerged as a promising paradigm for network operation and management. PBNM has the advantage of being able to change dynamically the behavior of a managed system according to the changing context requirements without having to modify the implementation of managed system [19].

The general PBNM can be considered an adaptation of the IETF policy framework to apply to the area of network provisioning and configuration. The IETF/DMTF policy framework is shown in Figure 3 and consist of four elements:

- *Policy Management Tool* (PMT): Graphical tool to define which policies will be applied in the network.
- *Policy Repository* (PR): It is used for the storage of policies, and it is typically a relational database or a directory.
- *Policy Decision Point* (PDP): It parses the policy, checks the authorization and validity before communicating them to the PEP.
- *Policy Enforcement Point* (PEP): It can apply and execute the different policies into network devices.

With the PBNM, we can simplify the management process through of centralization and business-logic abstractions [19].

Centralization refers to the process of defining all the devices provisioning and configuration at a single point (the PMT) rather than provisioning and configuring each device itself. The benefits of centralization in reducing manual tedium can easily be seen. In a network of 500 machines requires from 10 to 15 minutes of configuration per machine, the network administrator would need to work around 10 a 15 days (on a journey of eight hours of daily work).

Business-level abstractions make the job of the policy administrator simpler by defining the policies in terms of a language closer to the business needs of an organization rather than in terms of the specific technology need to deploy it. The network administrator needs not to be very conversant with the details of the technology that supports the desired need.

C. Policies

Policies [20] are one aspect of information, which influences the behavior of objects within the system. They are often used as a means of implementing flexible and adaptive systems for management of Internet services, distributed systems, and security systems [21].

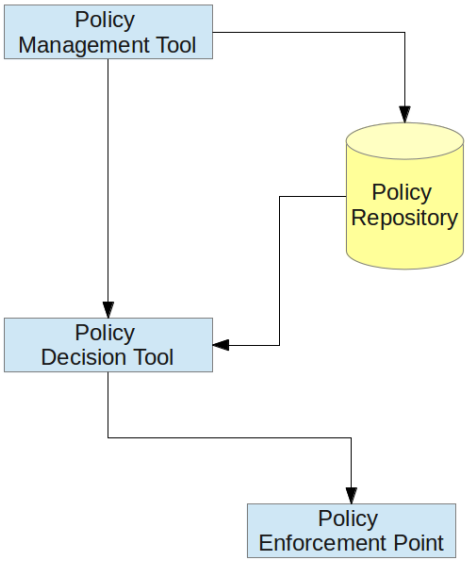

Figure 3. The IETF/DTMF policy framework [19]

Policies are classified into two categories:

- **Authorization Policies**: Define what a manager *is permitted or not permitted to do*. They limit the information made available to managers and the operations they are permitted to perform on managed objects.
- **Obligation Policies**: Define what a manager *must or must not* do and hence guide the decision-making process.

Authorization policies are specified to protect target objects and are usually implemented using security mechanisms when subjects cannot be trusted to enforce them. Obligation policies are event-triggered condition-action rules that can be used to define adaptable management actions [21].

Large-scale systems may contain millions of users and resources, and it is not practical to specify policies relating to individual's entities. It instead must be possible to specify policies relating groups of entities and also to nested groups such as sections within departments, within sites in different countries in an international organization. Domains can be used for this case.

A **Domain** is a collection of managed objects, which have been explicitly grouped together, based on geographical boundaries, object type, responsibility and authority or for covenience of human managers, for the purposes of management [20] [21].

More specifically a domain is a managed object which maintains a list of references to its member managed objects. If a domain holds a reference to an object, the object is said to be a *direct member* of the domain, and the domain are said to be its *parent* [20].

D. Policy Conflicts

In large distributed systems, there will be multiple human administrators specifying policies. Policy conflicts can arise due to omissions, errors or conflicting requirements of the administrators specifying the policies [22].

For example, an obligation policy may define an activity a manage must perform but there is no authorization policy to permit the manager to perform the activity. Obvious conflicts occur if both a positive and negative authorization or obligation policy with the same subjects, targets and actions.

The problem of detecting conflicts is extremely difficult. Analysis of the policy objects without any knowledge of the application or activities may detect positive-negative conflicts of modalities and conflicts between obligation and authorization policies, and so it may be possible to automate this [20].

E. Policy-Based Openflow Network Manager

The behavior of an OpenFlow network is defined by flow table entries of the devices (e.g., switch) comprising the network. These entries determine the action to be taken by the device, which may authorize the entry of a package in the device so that, it can be forwarded to another device or host or deny the packet in the device. However, some questions arise naturally about: (1) How to create or manage OpenFlow network with controllers currently present? (2) How to delegate or revoke network permissions to a particular user? (3) How to manage the switches flows as the number of hosts and switches increases?

Policy-Based Network Manager (PBNM) has emerged as a promising paradigm for network operation and management, and has the advantage to dynamically change the behavior of a managed system according to the context requirements without the need to modify the implementation of managed system [19]. The general PBNM can be considered an adaptation of the Internet Engineering Task Force (IETF) policy framework to apply to the area of network provisioning and configuration.

With PBNM the management network process can be simplified through of centralization and business-logic abstractions [19]. Centralization refers to the process of configuring all devices in a single-point (Policy Management Tool (PMT)) instead of reconfiguring the device individually.

In a previous work [23], we propose to use the PBNM concepts in OpenFlow networks. PBONM was proposed, a framework based on the IETF policy framework. Ponder language was chosen as the standard policy specification language in the PBONM. The PBONM is depicted on Figure 4. The architecture is divided in the following layers:

Policy Management Tool (PMT): it is a software layer that manages the network users, switches and OpenFlow layers providing the User Interface to enable these features. The Ponder is used to specify the policies in this layer. The Policy Repository (database) will store the policies and other information about of the network.

Policy Decision Point (PDP): it is responsible to interpreting the policies stored in the repository, checks the users' authorization (if the user has permission to add or remove a flow in specific switch), check policy conflicts on database and release the policies to Policy Enforcement Point.

Policy Enforcement Point (PEP): it is responsible to execute the configuration of OpenFlow controller. When the policies are interpreted, OpenFlow flows are generated and forwarded to the OpenFlow controller. So, the OpenFlow controller can enforce these flows on the network.

OpenFlow Network Devices: they are OpenFlow switches controlled by an OpenFlow controller and configured by PEP.

The PBONM is depicted on Figure 4. The architecture is divided in the following layers:

- *PMT*: it is a software layer to manage the network users, switches and OpenFlow layers providing the User Interface to enable these features. The Ponder is used to specify the policies in this layer. The database (LDAP or RDBMS) will store the policies, and other informations needed the network.
- *Policy Decision Point(PDP)*: it is responsible to interpreting the policies stored in the repository, checks the users' authorization (if the user has permission to add or remove a flow in specific switch), check policy conflicts on database and release the policies to Policy Enforcement Point.
- *Policy Enforcement Point(PEP)*: it is responsible to execute the configuration of OpenFlow controller. When the policies are interpreted, OpenFlow flows are generated and forwarded to the OpenFlow controller. So the OpenFlow controller can enforce these flows on the network.
- *Network Components(NC)*: t are switches and routers subordinate to OpenFlow controller. These network components are configured by flows sent by PEP.

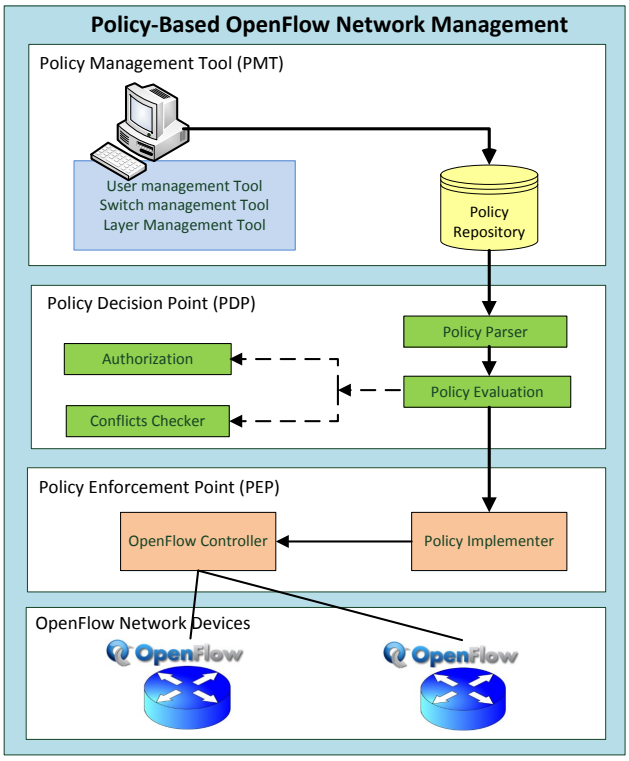

Figure 4. The Policy-Based OpenFlow Network Manager architecture

Thus, the network administrator can specify network flows and the users' permission through of a graphical tool using a policy specification language. These policies will be translated to OpenFlow controller API calls and will be applied to the network devices.

F. Ponder: Policy Specification Language

Ponder is a declarative, object-oriented language for specifying security and management policy for distributed object systems proposed by Damianou et al. [3]. The language is flexible, expressive and extensible to cover the wide range of requirements implied by the current distributed systems requirement and allows for the specification of security policies (role-based access control) and management policies (management obligations) [20].

There are four building blocks supported on Ponder, which are: (1) *authorizations*: what activities the subject can perform on the set of target objects; (2) *obligations*: what activities a manager or agent must perform on target objects; (3) *refrains*: what actions a subject must not execute on target objects; (4) *delegation*: granting privileges to grantees.

However, the Ponder language does not support the network flows abstraction. In contrast, OpenFlow architecture works over the network flows concept. To use Ponder in PBONM, an extension to the language is needed, to support the requirement inherent in the new environment. Thus, a network administrator can define flows in a network switch OpenFlow clearly and concisely.

The advantage of using a policy language is to permit the network administrator only needs to think in an abstract form, how the OpenFlow network should work, without worrying about the implementation details of a specific controller. Unlike other flow language's definition, that requires the administrator to use a programming language [4], [7], [8].

Ponder2 is a re-design of Ponder language and toolkit, maintaining the concepts and the basic constructs [24]. In contrast to the original Ponder, which was designed for general network and systems management; Ponder2 was designed as an extensible framework to configure more complex services. It uses the PonderTalk, a high-level configuration and control language, and it permits user-extensible Java objects. In our proposal, we prefer to use the original Ponder language because the new functionality of Ponder2 is not necessary. We believe that the concise description of Ponder is easier for a network administrator, unlike the more extensible and complex PonderTalk description.

IV. PONDERFLOW: OPENFLOW POLICY SPECIFICATION LANGUAGE

Ponder is the policy language used to manage security policies and access control. However, the Ponder language is too vague to cover all types of manageable environments [25]. PonderFlow is a policy definition language for OpenFlow networks where your main objective is to specify flows transparently, independent of OpenFlow controller used in the network. The PonderFlow extends the Ponder language [3] to suit the flow definition paradigm of OpenFlow environment.

Some of the Ponder's building blocks were kept and others were not used in favor of simplicity. Nevertheless, even keeping some building blocks from the original Ponder language;

the philosophy behind these blocks was changed to suit the paradigm of OpenFlow networks. Furthermore, it was added a way to specify flows through policies, making PonderFlow a declarative scripting language. In this way, the new keyword **flow** is included to specify the flow's characteristics. In the following subsections, the building blocks will be explained, and we will show some examples to manage network flows.

ANTLR framework [26] was used to generate the lexical analyzer and parser grammar in the Java programming language, as well as to generate the images of Abstract Syntax Tree (AST) tree of the building blocks defined in the PonderFlow.

A. Authorization Policies

The authorization policies define what the members within a group (subject) may or may not do in the target objects. Essentially, these policies define the level of access the users possess to use an OpenFlow switches network.

A positive authorization policy defines the actions that subjects are permitted to do on target objects. A negative authorization policy specifies the actions that subjects are not allowed to do on target objects.

This building block is very similar to the original language Ponder, but the focus of this building block in PonderFlow context is in the access by the users in the switches that comprise the OpenFlow network and OpenFlow controller itself.

Listing 1. PonderFlow Authorization Policy Sintax

```
1 inst ( auth+ | auth- ) policyName {
    subject [<type_def>]  domain-scope-expression;
3   target  [<type_def>]  domain-scope-expression;
    [ flow  [<type_def>]  flow-expression; ]
5   action        action-list;
    [ when constraint-expression |
      constraint-flow-expression ];
7 }
```

The syntax of an authorization policy is shown in Listing 1. Everything in bold is language keywords. Choices are enclosed within round brackets **( )** separated by **|**. Names and variables are represented within **< >**. Optional elements are specified with square brackets **[ ]**. The policy body is specified between braces **{ }**.

Constraints are optional in authorization policies and can be specified to limit applicability of policies based on time or attribute values to the objects on which the policy refers.

The elements of an authorization policy can be specified in any order, and the policy name must begin with a letter and contain letters, numbers and underscore in the rest of your name.

The specification of the subject and target may be optionally specified using an Uniform Resource Identifier (URI) to represent the domain of the subject or of the target. Moreover, we can specify the subject type or the target type in the policy definition.

Listing 2. Positive authorization policy example

```
1 inst auth+ switchPolicyOps {
    subject <User>     /NetworkAdmin;
3   target  <OFSwitch>  /Nregion/switches;
    action addFlow(), removeFlow(), enable(), disable();
5 }
```

Listing 2 shows an example of a positive authorization policy allowing all network administrators to perform the actions of adding flows, remove flows, enable and disable all switches in Nregion. Note, this policy is applied to any flow, and it is similar to conventional Ponder authorization policy. In Figure 5, we show the AST tree of a positive authorization policy from Listing 2.

A snippet of ANTLR grammar defined for PonderFlow authorization policies is described in Listing 3.

Listing 3. Authorization policy grammar

```
1 auth_policy:
      INST ( AUTH_POSITIVE | AUTH_NEGATIVE )
3       policy_name
        OPEN_BODY
5         auth_policy_options*
        CLOSE_BODY
7     | INST ( AUTH_POSITIVE | AUTH_NEGATIVE )
        policy_name SET policy_name
9       OPEN_PARENTESIS
          variable ( COMMA variable)*
11      CLOSE_PARENTESIS;
```

In Figure 5, we have the AST tree generated by ANTLR to move the policy of positive authorization of Listing 2 to the interpreter and Figure 6 for the same negative authorization policy of Listing 4.

Listing 4. Negative authorization policy example

```
1 inst auth- researcherOps {
    subject <User>     /Researchers;
3   target  <OFSwitch>  /Nregion/switches;
    action enable(), disable();
5 }
```

In Listing 4, we define a negative authorization policy restricting users from the researchers group does not perform the actions to enable or disable the switches in a Nregion.

The language also provides the ability to define policy types, enabling the reuse of policies by passing formal parameters in its definition. Several instances of the same type can be created and adapted to the identical environment through real values as arguments.

Listing 5. Type definition policy sintax

```
1 type ( auth+ | auth- ) policyType ( formalParameters) {
    authorization-policy-parts
3 }
  inst ( auth+ | auth- ) policyName = policyType(
       actualParameters )
```

The authorization policy switchPolicyOps (from Listing 2) can be specified as a type of the subject and target given as parameters as shown in Listing 6.

Listing 6. Type policy definition example

```
  type auth+ PolOpsT(subject s, target <OFSwitch> t) {
2     action load(), remove(), enable(), disable() ;
  }
4 inst auth+ admPolyOps=PolOpsT(/NetworkAdmins,
       /NregionA/switches);
  inst auth+ rsrPolOps=PolOpsT( /Researchers,
       /NregionB/switches);
```

Furthermore, we can use the PonderFlow Authorization Policies to define a flow in the OpenFlow network. A flow

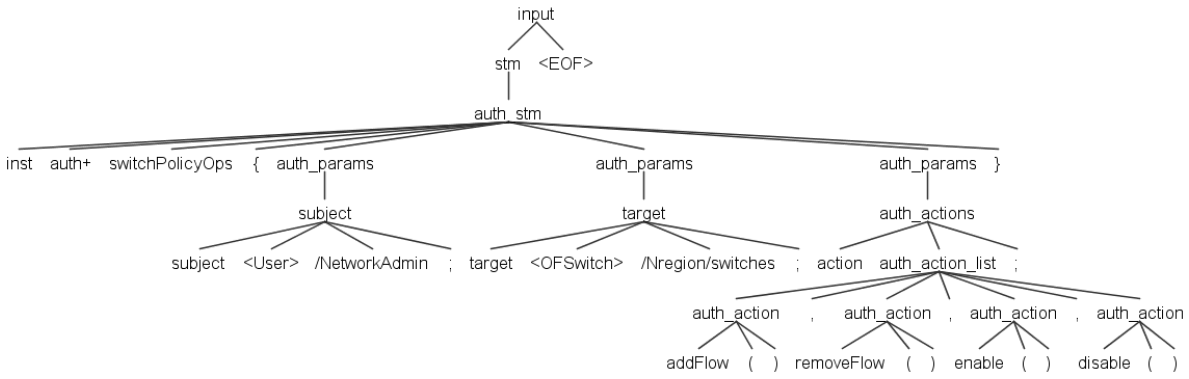

Figure 5. The AST tree for Listing 2 example

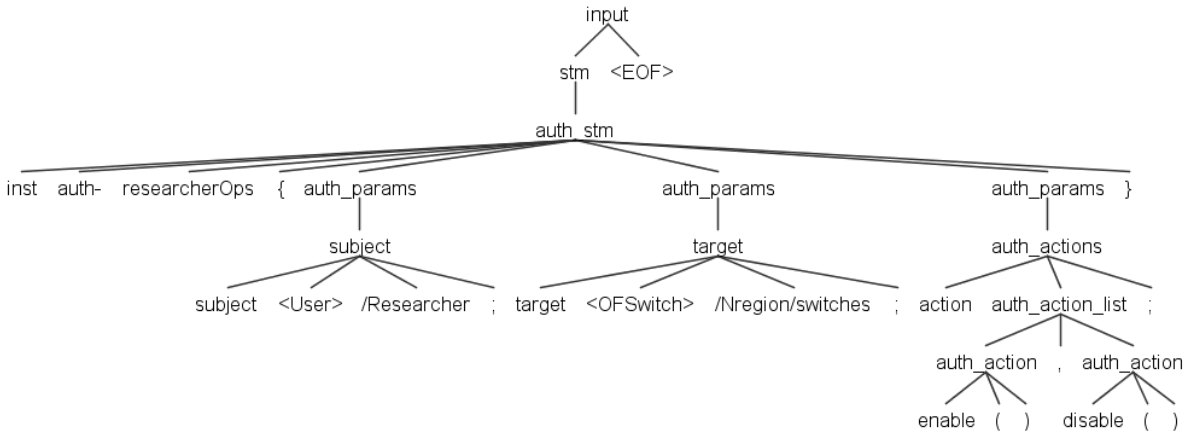

Figure 6. The AST tree for Listing 3 example

is an OpenFlow network path between hosts, independent of the switch quantity.

Thus, network administrator does not need to use a programming language like Java, Python or C++, in order to manipulate directly behavior through of the OpenFlow controller.

Listing 7. Type policy definition example

```
1 flow-expression = on = <DPID> ,
          | src = <DPID>/<switch_port> ,
3         | src = <IP-ADDRESS> ,
          | src = <MAC_ADDRESS> ,
5         | dst = <DPID>/<switch_port> ,
          | dst = <IP-ADDRESS> ,
7         | dst = <MAC_ADDRESS> ,
          | by = <DPID> ;
```

To define a flow, we need to use the keyword **flow** in the authorization policy statement. With this keyword, we can define the characteristic of the flow. Furthermore, it is possible define a path restriction where the network administrator can define where the flow must pass.

Listing 7 shows the grammar of *flow-expression*, where: *DPID* is the switch identification, *src* and *dst* are respectively the source device and destination device, *switch_port* is the incoming packet switch port, *IP-ADDRESS* is a valid IP address and *MAC-ADDRESS* is a valid MAC address.

TABLE I. OPENFLOW POLICY WILDCARDS

ingress-port	The switch port on which the packet is received
src-mac	The source mac address value
dst-mac	The destination mac address value
vlan-id	The VLAN identification value
vlan-priority	The VLAN priority value
ether-type	The ethernet type value
tos-bits	The ToS bits value
protocol	The IP protocol number used in the protocol field
src-ip	The source IP address value
dst-ip	The destination IP address value
src-port	The source protocol port value
dst-port	The destination protocol port value

The example in Listing 8 authorizes a flow to user /User/Students/John (**subject**), on the switches of domain */Uece/Macc/Larces/Switches*, set flows (**action**) on the network to establish a path starting from the switch with Datapath ID (DPID) 00:00:00:2C:AB:7C:07:2A on the port 2 (**src**) and ending in the switch with DPID 00:00:00:47:5B:DD:3F:1B on port 5 (**dst**), passing by the switches with DPID 00:00:00:C5:FF:21:7F:3B and 00:00:00:33:45:AF:1C:8A (**by**) when the source IP address of the flow is 192.168.0.21, the destination IP address 192.168.0.57 and the protocol destina-

tion port is 80.

Listing 8. A PonderFlow authorization policy

```
inst auth+ flow01{
  subject <User>   /Users/Students/John;
  target  <Switch> /Uece/Macc/Larces/Switches;
  flow <Flow> src=00:00:00:2C:AB:7C:07:2A/2  ,
              dst=00:00:00:47:5B:DD:3F:1B/5  ,
              by =00:00:00:C5:FF:21:7F:3B  ,
                  00:00:00:33:45:AF:1C:8A  ;
  action setFlow();
  when src-ip=192.168.0.21,
       dst-ip=192.168.0.57,
       dst-port=80;
}
```

PonderFlow specifies a set of default actions for flow definition, but the developers are free to add more actions to the language. The default actions are listed in Table II. Listing 9 defines a policy which user Alice can set a flow action to change the source IP address of the packet to 10.23.45.65 when the destination IP address is 10.23.45.123 on the switch with DPID 00:00:00:4F:32:1D:56:9C.

Listing 9. The flow definition to change the source ip address

```
inst auth+ flow02{
  subject <User>   Alice;
  target  <Switch> 00:00:00:4F:32:1D:56:9C;
  action setSrcIP('10.23.45.65');
  when dst-ip=10.23.45.123;
}
```

Furthermore, it is possible to define a policy to be applied in a specific switch and not a path. This is desirable when the network administrator wishes to add or remove a particular flow in a specific switch, in this way, the network administrator changes the network behavior in a single point on the network.

In Listing 10 is shown a policy example authorizing user Bob adds a flow (**action**) for the (**subject**) in the switch 00:00:00:4F:32:1D:56:9C (**target**) when the destination IP address is 172.24.5.17, and the destination port is 5432.

Listing 10. Authorizing add a specific flow in the switch

```
inst auth+ flow3{
  subject <User>   Bob;
  target  <Switch> 00:00:00:4F:32:1D:56:9C;
  action setFlow();
  when dst-ip=172.24.5.17,
       dst-port=5432;
}
```

TABLE II. OPENFLOW ACTION FIELD

setFlow()	Set the flow(s) in a specified path
delFlow()	Delete the flow(s) in a specified path
setSrcIp(ip-address)	Set the source IP address of the packet
setDstIp(ip-address)	Set the destination IP address of the packet
setSrcMac(mac-address)	Set the source MAC address of the packet
setDstMac(mac-address)	Set the destination MAC address of the packet
setSrcPort(port)	Set the source port of the packet
setDstPort(port)	Set the destination port of the packet
setVlanId(integer)	Set the VLAN of the packet
setVlanPriority(integer)	Set the VLAN priority of the packet

We can restrict the actions of the network users with the authorization policies. For example, we cannot permit a certain kind of flows through the OpenFlow network with the authorization policies. In Listing 11 we define an authorization policy not allowing Alice (**subject**) add flows (**action**) in the switches with DPID 00:00:00:4F:32:1D:56:9C, 00:00:00:47:5B:DD:3F:1B and 00:00:00:33:45:AF:1C:8A.

Listing 11. Negative authorization policy for deny add flow in the switch

```
inst auth- flow4{
  subject <User>   Alice;
  target  <Switch> /Uece/Macc/Larces/Switches;
  flow by=00:00:00:4F:32:1D:56:9C,
          00:00:00:47:5B:DD:3F:1B,
          and 00:00:00:33:45:AF:1C:8A
  action setFlow();
}
```

Another kind of usage is not allows the user to change a packet values in the network. Listing 12 shows an example of this policy.

Listing 12. Negative authorization policy for deny add flow in the switch

```
inst auth- flow5{
  subject <User>   Alice;
  target  <Switch> /Uece/Macc/Larces/Switches;
  action setSrcIP(),
         setDstIp(),
         setSrcMac(),
         setDstMac(),
         setSrcPort(),
         setDstPort();
}
```

The previous example the user Alice (**subject**) cannot change the source ip address, destination ip address, source mac address, destination mac address, source port and destination port (**action**) in any switch of */Uece/Macc/Larces/Switches* domain.

Moreover, it is possible define action which specific user can perform in the network. Listing 13 shows a policy which allows the user of domain */Uece/Macc/Larces/Admin* (**subject**) remove flows (**action**) of in any switch of */Uece/-Macc/Larces/Switches* domain (**target**).

Listing 13. Authorization policy for allow remove flow

```
inst auth- flow5{
  subject <User>   /Uece/Macc/Larces/Admin;
  target  <Switch> /Uece/Macc/Larces/Switches;
  action delFlow();
}
```

B. Obligation Policies

Obligation policies allow to specify actions to be performed by the network administrator or by the OpenFlow controller when certain events occur in an OpenFlow network and provide the ability to respond any change in circumstances.

These policies are event-triggered and define the activities subjects (network administrator or OpenFlow controller) must perform on objects within the target domain. Events can be simple, e.g., an internal timer, or more complex, starting by reading some kind of sensor, e.g., a network card stopped.

This building block is very similar to the original language Ponder, but in the context of PonderFlow, including flow definition. This block sets an obligation for the network administrator or the OpenFlow controller performs some action, or simply is notified, when a particular event occurs.

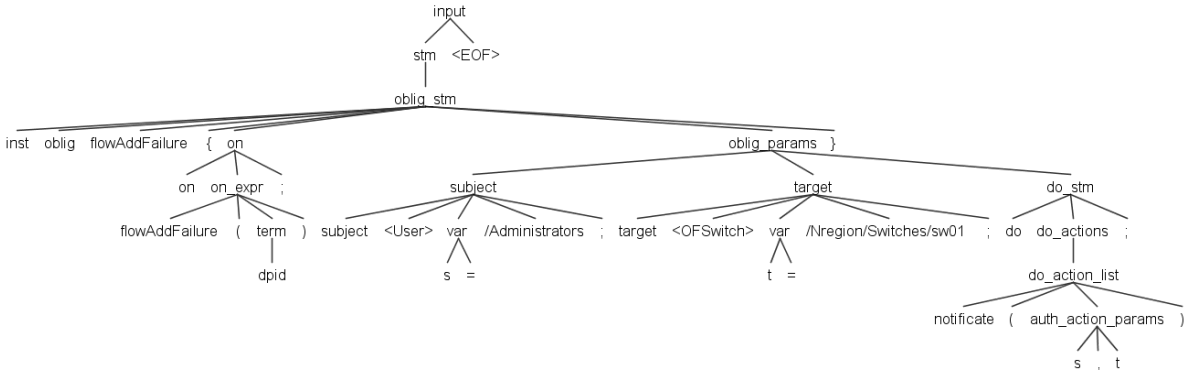

Figure 7. The AST tree for Listing 16 example

Listing 14. Obligation policy sintax

```
1 inst oblig policyName {
    on event-specification ;
3   subject [<type_def>] domain-Scope-Expression ;
    [ target [<type_def>] domain-Scope-Expression ; ]
5   do obligation-action-list ;
    [ catch exception-specification ; ]
7   [ when constraint-Expression ; ]
  }
```

The syntax of obligation policies is shown in Listing 14. The required event specification follows the **on** keyword. The target element is optional in obligation policies. The optional catch-clause specifies an exception to be performed if the actions fail to execute, for some reason.

A snippet of ANTLR grammar defined for PonderFlow obligation policies is described in Listing 15.

Listing 15. Obligation policy grammar

```
  oblig_policy:
2   INST OBLIG policy_name
      OPEN_BODY
4       oblig_policy_options*
      CLOSE_BODY;
```

In Listing 16, the obligation policy is triggered when a failure on adding a flow occurs. Network administrator will be notified when this event occurs, and he will receive the switch ID where it happened. Figure 7 shows the AST tree of Listing 16.

Listing 16. Obligation policy sintax

```
1 inst oblig flowAddFailure {
    on flowAddFailure(dpid) ;
3   subject <User> s=/Administrators ;
    target <OFSwitch> t = /Nregion/Switches/sw01 ;
5   do notificate(s, t) ;
  }
```

To perform an obligation policy, it is required the user has an authorization over the target. This can be specified with an authorization policy. If there is no authorization policy specifying who can perform a particular action, the obligation policy will produce an exception error (depends on the implementation), and the policy will not be applied in the system.

V. PonderFlow Parser

A PonderFlow parser was developed in Java using ANTLR framework [26]. The ANTLR framework is a flexible and powerful tool for parsing formal languages like PonderFlow. The parser translates the PonderFlow statements describe in Section IV.

ANTLR provides support for two tree-walking mechanisms; the parse-tree listeners and the parse-tree visitors. By default, ANTLR generates a parse-tree listener interface to respond to events executed by the built-in tree walker. The PonderFlow parser uses the parse-tree listeners to parse its grammar.

The parser consists of some Java classes, generated automatically by ANTLR. The PonderFlow parser is basically a Java class implementing a parse-tree listener interface, and this interface requires some Java methods to be implemented.

In these methods, we add Java code describing what must be done for each PonderFlow statement read. It is possible to use other tools such as a DBMS or other framework to assist in analyzing the statements OpenFlow process.

Some performance test was executed over PonderFlow parser. The instances of the performance testing vary between 1 and 10000 statements. Figure 8 shows the elapsed time to parse instances of 1, 5, 10, 20, 50, 100, 200, 500, 1000, 2500, 5000 and 10000 PonderFlow statements, respectively.

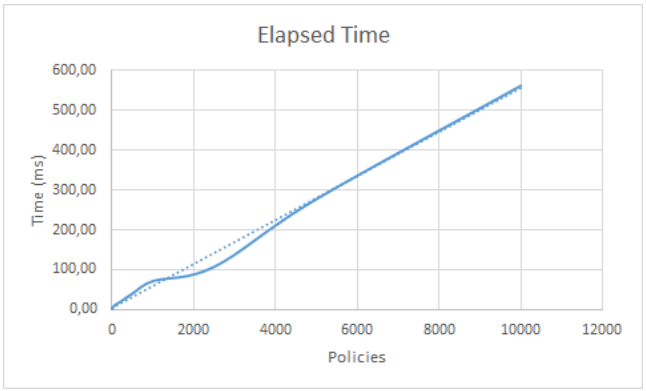

Figure 8. Time required to parse several instances.

As shown in Figure 8 can be seen the analyzer is easily scalable and can analyze tens of thousands of statements in a

second. Use other tools, such as a DBMS, the time of analysis may increase, hampering the analysis process.

However, since some compilers of programming languages may take several seconds to compile the source code, even if it is added to other tools and frameworks on the PonderFlow parser, the compile time will still be at a tolerable level of acceptability.

VI. Conclusion and Future Works

This paper described the PonderFlow language, a new policy specification language for OpenFlow networks. With this language, the network administrator does not need to be an expert in a programming language, like Java, Python or C++, to specify the policy of an OpenFlow network. The language statements are simple and concise to define policies.

The PonderFlow grammar was presented as well as some examples of usage and their AST tree representation. The grammar was tested using the ANTLR framework, which generates the parser and the lexical analyzer for the Java programming language.

Some tests were performed and it was observed that this solution had scalability, and is easily integrated into an OpenFlow controller. However, the PonderFlow works merely with OpenFlow Switch Specification version 1.0, because most of the commercial switches only support this version.

It shall extend the Ponder language to use the OpenFlow Switch Specification version 1.3. Another point that should be studied is the treatment of policy's conflicts, where a network administrator can, by accident or malpractice, declare two or more conflicting policies. It is necessary to perform an assessment on all policies before applying them on OpenFlow controller.

References

[1] B. Batista and M. Fernandez, "Ponderflow: A policy specification language for openflow networks," in ICN 2014, The Thirteenth International Conference on Networks, Nice, France, Feb. 2014, pp. 204–209.

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

[3] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY '01). London, UK, UK: Springer-Verlag, 2001, pp. 18–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=646962.712108

[4] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," SIGPLAN Not., vol. 46, no. 9, Sep. 2011, pp. 279–291. [Online]. Available: http://doi.acm.org/10.1145/2034574.2034812

[5] NOXRepo.org, "NOX Openflow Controller," Last accessed, Aug. 2014. [Online]. Available: http://www.noxrepo.org/nox/about-nox/

[6] G. VanRossum and F. L. Drake, The Python Language Reference. Python Software Foundation, 2010.

[7] D. M. F. Mattos, N. C. Fern, V. T. D. Costa, L. P. Cardoso, M. Elias, M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "Omni: Openflow management infrastructure," Paris, France, 2011.

[8] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in Proceedings of the first workshop on Hot topics in software defined networks, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 43–48. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342451

[9] D. Erickson, "Floodlight Java based OpenFlow Controller," Last accessed, Aug. 2014. [Online]. Available: http://floodlight.openflowhub.org/

[10] ——, "Beacon," Last accessed, Jun. 2014. [Online]. Available: https://openflow.stanford.edu/display/Beacon/Home

[11] NOXRepo.org, "POX Openflow Controller," Last accessed, Aug. 2014. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[12] NEC Corporation, "Trema Openflow Controller," Last accessed, Aug. 2014. [Online]. Available: http://trema.github.com/trema/

[13] B. Heller, "Openflow switch specification, version 1.0.0," Dec. 2009. [Online]. Available: www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf

[14] OpenFlow Hub, "Indigo OpenFlow Switching Software Package," Last accessed, Jun. 2013. [Online]. Available: http://www.openflowswitch.org/wk/index.php/IndigoReleaseNotes

[15] O. Ferkouss, I. Snaiki, O. Mounaouar, H. Dahmouni, R. Ben Ali, Y. Lemieux, and O. Cherkaoui, "A 100gig network processor platform for openflow," in Network and Service Management (CNSM), 2011 7th International Conference on. IEEE, 2011, pp. 1–4.

[16] R. Ozdag, "Intel ethernet switch fm6000: SDN with openflow," Intel Corporation, Tech. Rep., 2012.

[17] NoviFlow Inc, "NoviFlow Switch 1.1," Last accessed, Sep. 2013. [Online]. Available: http://www.noviflow.com/

[18] R. Bertó-Monleón, E. Casini, R. van Engelshoven, R. Goode, K.-D. Tuchs, and T. Halmai, "Specification of a policy based network management architecture," Military Communication Conference, 2011, pp. 1393–1398.

[19] D. C. Verma, "Simplify network administration using policy-based management," IEEE Network, vol. 16, no. 2, March/April 2002, pp. 20–26.

[20] M. Sloman, "Policy driven management for distributed systems," Journal of Network and Systems Management, vol. Vol.2, no. No 4, 1994.

[21] N. C. Damianou, A. K. Bandara, M. S. Sloman, and E. C. Lupu, "A survey of policy specification approaches," April 2002.

[22] E. C. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," IEEE Trans. Softw. Eng., vol. 25, no. 6, Nov. 1999, pp. 852–869. [Online]. Available: http://dx.doi.org/10.1109/32.824414

[23] B. L. A. Batista, G. A. L. de Campos, and M. P. Fernandez, "A proposal of policy based OpenFlow network management," in 20th International Conference on Telecommunications (ICT 2013), Casablanca, Morocco, May 2013.

[24] K. Twidle, E. Lupu, N. Dulay, and M. Sloman, "Ponder2-a policy environment for autonomous pervasive systems," in Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on. IEEE, 2008, pp. 245–246.

[25] T. Phan, J. Han, J.-G. Schneider, T. Ebringer, and T. Rogers, "A survey of policy-based management approaches for service oriented system," 19th Australian Conference on Software Engineering, 2008.

[26] T. Parr, "ANTLR: ANother Tool for Language Recognition," Last accessed, Aug. 2013. [Online]. Available: http://www.antlr.org/