ReasoNet

Citation for published version:
Rotsos, C, Farshad, A, King, D, Hutchison, D, Zhou, Q, Gray, AJG, Wang, C-X & McLaughlin, S 2018, ReasoNet: Inferring Network Policies Using Ontologies. in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft).* IEEE, pp. 217-221, 4th IEEE Conference on Network Softwarization and Workshops 2018, Montreal, Canada, 25/06/18. https://doi.org/10.1109/NETSOFT.2018.8460050

Digital Object Identifier (DOI):
10.1109/NETSOFT.2018.8460050

Link:
Link to publication record in Heriot-Watt Research Portal

Document Version:
Peer reviewed version

Published In:
2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)

ReasoNet: Inferring Network Policies Using Ontologies

Charalampos Rotsos, Arsham Farshad,
Daniel King, David Hutchison
Lancaster University

Qianru Zhou, Alasdair J. G. Gray,
Cheng-Xiang Wang, Stephen McLaughlin
Heriot-Watt University

***Abstract*—Modern SDN control stacks consist of multiple abstraction and virtualization layers to enable flexibility in the development of new control features. Rich data modeling frameworks are essential when sharing information across control layers. Unfortunately, existing NOS data modeling capabilities are limited to simple type-checking and code templating. We present an exploration of a more extreme point on SDN data modeling: ReasoNet. Developers can use semantic web technologies to enrich their data models with reasoning rules and integrity/consistency constraints, and automate state inference across layers. We demonstrate the ability of ReasoNet to automate state verification and cross-layer debugging, through the implementation of two popular control applications, a learning switch and a QoS policy engine.**

I. Introduction

Software defined networking (SDN) technologies have radically revised traditional network configuration and control practices. Many vendors offer production-ready SDN devices, while network operators explore SDN adoption strategies. A key challenge for SDN adoption is support for carrier-grade resilience guarantees in production systems.

SDN control planes follow a multi-layer approach – typically consisting of three layers [1] – to provide resource virtualization and logical centralization. In the lower layer of the architecture, network devices expose device configuration interfaces to external control entities using well-defined protocols, like OpenFlow. The middle layer of the SDN control stack, commonly known as the Network Operating System (NOS), uses these interfaces to synthesize new, high-level, centralized control abstractions. Finally, control applications form the upper control layer of the architecture. These applications use NOS APIs to realize novel control functionalities, like BGP routing and dynamic resource control.

In an SDN architecture, the NOS operates as a mediation layer between the network fabric and the control applications; similarly to an operating system, it virtualizes network resource access, abstracts technology heterogeneity and synchronizes resource access between control applications. Application developers can rapidly enable new control functionalities, while being agnostic to the underlying SDN technology and the state of running control applications. The specification of flexible data models is a key challenge for the NOS design, enabling control virtualization and abstraction.

Existing NOS implementations employ data modeling languages, notably YANG [2], for the specification of application interfaces and message wire formats. Dedicated compilers, integrated in the build toolchain of the controller, can generate language bindings and protocol parsers/serializers for YANG models, while a central service discovery mechanism allows late interface binding. Such modeling mechanisms simplify interface implementation and offer basic type checking capabilities. In parallel, verifiable NOS designs, like the Frenetic [3] project, offer bespoke interfaces with strong guarantees for various control properties, like policy consistency and resource allocation. Their extensibility is subject to the underlying formal model and extensions require model and proof updates, to avoid potential axiom violations.

NOS designs face significant challenges to improve support for network virtualization and abstraction, due to limitations in existing data modeling framework. For example, existing NOS interfaces rely on static information flows between interacting entities, which are prone to policy conflicts. Recent research efforts have demonstrated that controlled state visibility and flexible control interfaces can mitigate such non-deterministic behaviors [4], [5]. Furthermore, although the state of a control application depends on the state of the underlying network and co-located applications, existing platforms lack generic mechanisms capturing relevant state associations. The state of the NOS remains loosely associated with application state — typically relying on event-driven models to propagate state changes — and each control layer is responsible to implement custom state validation logic. Developers must analyze source code from the NOS and other control applications to ensure correct integration, while network managers must test extensively new or updated control applications to ensure policy correctness, prior to their deployment.

In this paper, we argue that better data modeling tools for SDN controllers can simplify the design of the NOS and control applications. Such a modeling framework can enable state association across control layers and improve the extensibility of the NOS, enable control state validation and simplify run-time debugging. Towards this goal, we explore the applicability of semantic web technologies as a data modeling and management framework for SDN control systems. Specifically, we present an OpenFlow ontology, modeling a NOS information model, and ReasoNet, a novel NOS architecture which allows control applications to exploit the modeling and data validation capabilities of semantic web technologies [1]. Specifically, our main contributions are:

- We present the first ontology for OpenFlow technologies, to the best of our knowledge (§ III).
- We elaborate on the design of a NOS architecture which provides support for network knowledge inference and integrity/consistency validation and present a straw-man implementation, based on the Ryu controller. (§ III).
- We demonstrate the abilities of ReasoNet and the designed ontology by developing two reference network control applications; a learning switch and a QoS policy (intent) engine (§ IV).

For the rest of this paper, we provide a short overview of semantic web technologies and their applicability in the development of control applications (§ II). Furthermore, we describe an ontology model for SDN control and elaborate on the design of ReasoNet, a novel NOS architecture using semantic web technologies to model the NOS and the application state (§ III). In addition, we demonstrate the modeling capabilities of ReasoNet using a set of popular control applications and evaluate the impact of semantic web technologies on the performance of an SDN network (§ IV). Finally, we present related research efforts (§ V) and conclude this work (§ VI).

II. Motivation

In this section, we identify some key challenges in implementing SDN control interfaces (§ II-A) and discuss the capabilities of semantic web technologies (§ II-B).

A. Control Application Development

SDN control functions are developed as single-purpose micro-applications. This development model is enabled by virtualizing resources and abstracting information in the NOS layer. The NOS platform aggregates all control channels from the underlying network infrastructure and synthesizes a high-level control API. Policy conflicts are resolved by the NOS, hence, application developers do not have to worry about the behavior of co-located applications.

The implementation of an SDN NOS or application extensively relies on data modeling, where an SDN control stack usually contains multiple data and information models. The NOS develops an information model to converge and abstract different southbound control protocols, like the ONF Core Information Model (CIM) [6]. The ONF CIM proposes a common control state model for NOSes, while specialized drivers enable control of different network technologies and allow extensions to the core model. Similarly, control applications use data models to organize their internal state and expose configuration parameters and monitoring information. Some of these models are standardized, like the ASPEN [7] QoS model, but the majority are implementation-specific.

In summary, SDN control layers transform network information to match different data models and data modeling is vital for the implementation of the control stack. Existing NOS platforms offer model specification capabilities to applications, however, their use is limited in automated parsing/marshaling and interface code generation. Developers are responsible to implement individual consistency validation functionalities in the control applications and the NOS.

This design approach enables rapid feature prototyping, however, it introduces new challenges for data consistency validation and debugging between control layers and applications. Firstly, resource virtualization in SDN is prone to frequent policy conflicts, since network events trigger control applications to modify common forwarding state. Using existing modeling mechanisms it is not possible to associate the impact of a policy update from an application on the resulting forwarding policy applied by the controller. Secondly, the generic conflict resolution strategies must address a fundamental trade-off between performance and guarantee violation. Data models offer semantics that assist application collaboration and improve conflict resolution efficiency [4]. Thirdly, although data models are strongly connected between layers, they remain weakly synchronized at run-time and state changes can lead to model inconsistencies. Finally, abstraction hides critical debugging information that limit the ability to localize bugs and associated states across layers [8], [9].

B. Semantic Web

The development of flexible data modeling frameworks is a research challenge for multiple computer science domains. Semantic web is a popular data modeling technology, which provides data semantics and enables common understanding of Web content between computers. Semantic web technologies use resource description frameworks (RDF) [10] and ontologies to organize data and their relations using labeled graphs. Using this data representation, multiple tools and frameworks have been developed enabling various data manipulation and modeling capabilities.

Data Modeling and Validation: A key component of a knowledge-base is the ontology, a conceptualization of the entities and relations in the application domain. An ontology is specified using an RDF, like the RDF Schema (RDFS) [11] and the Web Ontology Language (OWL) [12], which allows the developer to specify the entities comprising the application environment, their attributes and relationships. Furthermore, semantic web technologies support data manipulation capabilities, like the Simple Protocol and RDF Query Language (SPARQL) [13], which allow users to express complex data relationship and efficiently query the knowledge-base. Furthermore, semantic web technologies can accommodate advanced data validation mechanisms, well beyond simple type checking. For example, the Stardog [14] database offers an Integrity Constraint Validation (ICV) mechanism, which allows developers to inject SPARQL queries in a database, which are constantly validated against the data at run-time. Semantic databases rely on semantic reasoners to enforce ontology and integrity constraints. For instance, if the data violate the schema or an ICV rule, then the reasoner can either reject data mutations or highlight the invalid data to the user.

[1] https://github.com/SemanticSDN

```
of:Link       rdf:type owl:Class.

of:hasSrcPort rdf:type owl:ObjectProperty;
              rdfs:domain of:Link;
              rdfs:range of:Port.
of:hasDstPort rdf:type owl:ObjectProperty;
              rdfs:domain of:Link;
              rdfs:range of:Port.
of:isActive   rdf:type owl:DatatypeProperty;
              rdfs:domain of:Link;
              rdfs:range xsd:boolean.
of:hasLoad    rdf:type owl:DatatypeProperty;
              rdfs:domain of:Link;
              rdfs:range xsd:long.
```

Listing 1: A schema (partial) specifying the Link entity of our OpenFlow ontology.

```
[] a icv:Constraint; icv:query """
SELECT ?path
(GROUP_CONCAT(?state) as ?state_list)
WHERE {
  ?apath  a of:AvailPath;
        of:hasState "Active";
        of:realizes ?path.
  ?hop a of:PathHop;
        of:belongs ?apath;
        of:hasLink ?link.
  ?link a of:Link;
         of:isActive ?state.}
GROUP BY ?path
HAVING (CONTAINS(?state), "False") """.
```

Listing 2: An ICV rule evaluating if a Path uses a Link with aninActive status.

```
[] a r:SPARQLRule;
    r:content """
IF {
  ?port a of:Port;
        of:isUP ?state.
  ?link a ofLink;
        of:hasDstPort ?port.
}
THEN {
  ?link of:isActive ?state.
  } """.
```

Listing 3: A SPARQL rule for inferring the Link status from the Port status.

As an example, Listing 1 presents a part of the `Link` entity specification in our OpenFlow ontology. The ontology specifies two object properties `of:hasSrcPort`, `of:hasDstPort` which specify the relation between the `Link` and `Port` objects. Furthermore, the ontology specifies two data properties with specific data types, `of:isActive` and `of:hasLoad`, to represent the state of an entity. As we will discuss in Section III, a `Link` entity represents connectivity between two adjacent switches, while `Path` represents an established end-to-end connection. A `omprisesofatleastone{\ttof:PathHop}entitywhih` in turns is associated with exactly one `Link` entity, using the relationship `of:hasLink`.

Furthermore, Listing 2 depicts an ICV rule. The rule validates data consistency between the topology and established path state, ensuring that an active `Path` entity does not use an inactive `Link` entity. The schema and the ICVs are inserted in the database, prior to any data insertion. The semantic reasoner continuously validates the data and detects any potential data inconsistencies. The developer can choose either to forbid any data mutation that violates the schema or an ICV or to allow the data mutation and enforce consistency programmatically at a later time.

Reasoning: High-level knowledge in an SDN control stack is typically synthesized from low-level network information. For example, link discovery in an SDN network allows a controller to discover switch connectivity and treats the network as a directed graph. Semantic web technologies provide the ability to reason about new knowledge from data using simple logic rules. Semantic Web Rule Language (SWRL) is a low-level declarative programming language based on first-order logic. SWRL [15] rules contain a set of logically associated premises and a set of conclusions. When the data in the database compute the premise part of the rule to true, then the conclusions are added in the database. SWRL rules are low-level and complex validation policies can be quite verbose, while existing reasoners offer partial support of SWRL standards. SPARQLRule is an alternative approach to enable reasoning in a knowledge-base using the SPARQL language. This form of inference is not currently standardized and Stardog is the only semantic web software with integrated support.

Listing 3 contains a sample SPARQLRule, used in our OpenFlow ontology. This rule infers the state of a link entity based on the state of the underlying ports. The rule consists of two parts: *IF* and *THEN*, each containing a set of triple matches. The triples in the *IF* part of the rule are matched against the data of the knowledge base and the content of the *THEN* part of the rule is inferred on every query. SPARQLRule supports a subset of the SPARQL language and a rule can only infer new knowledge (knowledge updates are not currently supported in SPARQLRule).

Semantic web technologies offer a set of unique features for NOS in comparison to alternative data management frameworks. Graph databases, like Neo4j [16], provide rich graph processing and traversal capabilities, but lack versatile data validation and reasoning capabilities, available in knowledge-base, like Stardog. Moreover, semantic databases allow applications to relation can be queried and modified in runtime the same as for the data. Relational databases provide extensive data validation mechanisms and carrier-grade implementations. Nonetheless, their consistency model is ill-suited for a network control system, since the model forbids any data manipulation leading to data inconsistencies. Network systems are open world system and unobserved events (e.g. link failures) can violate policy. Reasoners in semantic databases decouple state from data validation; a semantic database allows a topology state update to violate internal data consistency and the application is responsible to identify a strategy to mitigate the inconsistency using the current state of the system.

III. ReasoNet Design

In this section, we elaborate on the proposed ontology for OpenFlow technologies (§ III-A) and present the architecture of ReasoNet (§ III-B).

A. Towards an OpenFlow Ontology

Ontologies are a key component of semantic web applications. The proposed system defines an extensible ontology to model all the layers of a control stack. The core ontology

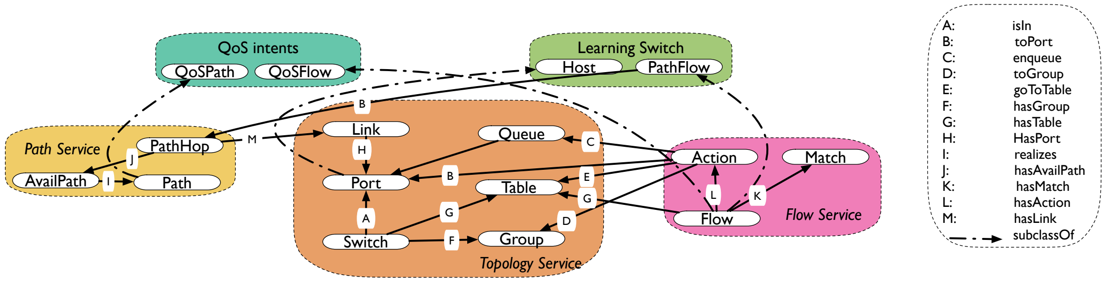

Fig. 1: The OpenFlow ontology model is extended to support the learning switch and the QoS intents usecases. The addition of the Path, AvailPath and PathHop concepts to model the connectivity requirements of control applications. Host and PathFlow extend the OpenFlow ontology for implementing the learning switch service.

entities captures the details of the OpenFlow protocol [2] and network connectivity and path information. Control applications can extend the ontology by introducing new concepts, relations and constraints at load-time and run-time.

Figure 1 presents the concepts of the proposed core ontology, grouped by the service responsible to maintain their consistency. The topology group contains five major concepts, Port, Group, Table, Queue and Switch which reflect the data of the PORT_DESCRIPTION, GROUP_DESC, TABLE_DESC, QUEUE_DESC and FEATURES_REPLY OpenFlow message types, respectively. Furthermore, the group contains a Link concept modeling the connectivity information inferred by the topology discovery service (LLDP packet injection) of the controller. In addition, the Link concept maintains load and capacity information. The flow group models the switch flow tables. A Flow concept models FLOW_MOD messages. Each Flow connects to one Match object, modeling the matching tuple, and multiple Action objects, modeling traffic manipulations. Our ontology does not currently support METER tables, since their support is limited.

In addition to elementary OpenFlow support, the ontology models network connectivity for control applications. The Path concept models end-to-end Port connectivity. Since, the SPARQL language standards (v1.1) has limited graph traversal capabilities, the ontology uses the AvailPath and PathHop concepts to model the individual hops of an end-to-end path. Specifically, AvailPath object represents alternative paths between the source and destination Ports of a Path concept. AvailPaths are computed using an external graph processing service and inserted in the database upon a new path request. Finally, the PathHop concept associates an AvailPath object with a Link object. An AvailPath object is connected with an ordered list of PathHops, reflecting the individual hops of the path.

Finally, the *Learning Switch* group is an example ontology extension. A learning switch application maintains a list of the MAC and IP addresses of all network hosts as well as the flows interconnecting them. We model this information using the Host and PathFlow concepts. Specifically, a Host extends the Port concept and adds IP address information, while the PathFlow concept extends the Flow and associates PathHop with installed flows.

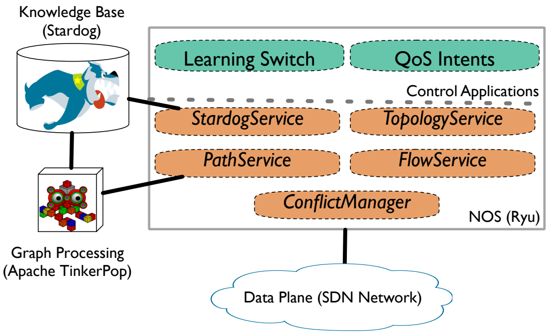

Fig. 2: ReasoNet architecture.

B. ReasoNet Architecture

The architecture of ReasoNet is depicted in Figure 2. ReasoNet comprises of three functional blocks: the NOS, the Applications and the Knowledge database.

The ReasoNet NOS offers five core services to Applications. The *TopologyService* offers topology discovery and populates the knowledge-base with network topology information. The *FlowService* module abstracts flow table access for applications and updates the knowledge-base with flow-level statistics in fixed intervals. In addition, it ensures consistency between the switch flow tables and the flow entities in the knowledge-base. The *PathService* module provides path computation capabilities to control applications. The *StargodService* module offers low-level knowledge-base access to other NOS services and applications and allows them to extend the data model and insert data validation rules at boot time.

Finally, the *ConflictManager* service allows applications to insert data validation rules and checks the integrity of the database during data updates. If a rule is invalidated, then the service notifies the appropriate Application, in order to resolve the data inconsistency. An example of an integrity check is presented in Listing 4. The query returns all the switch flows with overlapping match subspace and priorities. Any flow

[2] currently supporting version 1.3 [17], but can be easily adapted to support other protocol versions

```
SELECT  DISTINCT ?switch ?flow1 ?flow2  WHERE{
  ?switch a of:Switch; of:hasFlow ?flow1; of:hasFlow
  ↪  ?flow2.
  ?flow1 a of:PathFlow; of:hasPriority ?priority1.
  ?flow2 a of:PathFlow; of:hasPriority ?priority2.
  ?flow1 ?match_field1 ?value1.
  ?flow2 ?match_field2 ?value2.
  ?match_field1 rdfs:subPropertyOf
  ↪  of:flow_match_fields.
  ?match_field2 rdfs:subPropertyOf
  ↪  of:flow_match_fields.
  FILTER (?flow1 != ?flow2 && ?priority1 =
  ↪  ?priority2)
  {FILTER (NOT EXISTS {?flow1 ?match_field2 ?x})
  FILTER (NOT EXISTS {?flow2 ?match_field1 ?y})}
  UNION {FILTER (?match_field1 = ?match_field2 &&
  ↪  ?value1 = ?value2)}
} GROUP BY ?switch ?flow1 ?flow2
```

Listing 4: A SPARQL query reporting flow pairs with overlapping match subspaces. The query is used by the *ConflictService* to detect flow-level policy conflicts.

conflicts are propagated to the *FlowService* module, which contains code that synthesizes the individual flow actions of the overlapping flows, based on the network manager's policy.

ReasoNet uses the Apache Tinkerpop [18] framework to enhance its graph processing capabilities. The Stardog developers provide a Tinkerpop client driver to enable seamless integration. The Tinkerpop service allows ReasoNet to overcome the limited graph processing capabilities of the Stardog SPARQL engine.

The ReasoNet architecture can be integrated with any available NOS. For our straw-man implementation we used the event engine and the OpenFlow parsing capabilities of the Ryu controller. Ryu offers a simple and extensible architecture, in comparison to similar production NOSes, while the adoption of Python as the implementation language allowed rapid prototyping for our framework. ReasoNet service and applications are implemented as *RyuApplication* modules.

IV. ReasoNet In Action

To demonstrate the expressiveness of ReasoNet, we present and evaluate in this section the implementation of two advanced network control functions, namely a learning switch application (§ IV-A) and a QoS-oriented declarative policy engine (§ IV-B). We evaluate ReasoNet using a quad-core Xeon Server (E5-1603) with 16 GB of memory and running Ubuntu 14.04. We use the Mininet platform (v2.2) and the Open vSwitch software switch (v2.5) to emulate various topologies and traffic conditions. ReasoNet uses the OpenFlow parsing capabilities and event engine of the Ryu controller (v4.17). The ontology is stored in a Stardog knowledge base instance (v5.02) connected to a Tinkerpop graph processing service (v3.0.2).

A. Learning switch

The learning switch functionality is a reference OpenFlow control function, implemented by all OpenFlow controllers. The application passively monitors all unmatched traffic on edge switches and records triples of source IP, MAC addresses and receive switch port. For each unmatched Ethernet packet, the application looks up the destination MAC address in its MAC-port table, and, upon a successful match, the application computes a path and transmit relevant `FLOW_MOD` messages to establish connectivity. To avoid loops, the implementation in ReasoNet runs an ARP proxy, which responds to ARP requests if the application know the MAC-IP address association.

```
SELECT  DISTINCT ?flow
WHERE {
  ?switch a of:Switch; of:hasFlow ?flow.
  ?flow a of:PathFlow; of:in_port ?in_p;
    of:hasAction ?action.
  ?action of:toPort ?to_p.
  ?in_p a of:Port; of:isUP ?isUp1.
  ?to_p a of:Port; of:isUP ?isUp2.
FILTER (?isUp1 = "false"^^xsd:boolean ||
        ?isUp2 = "false"^^xsd:boolean)
}
```

Listing 5: A SPARQL query detecting invalid flows due to link failures. A non-empty result denotes rule violation.

Our learning switch implementation re-use the OpenFlow protocol parsing capabilities of the Ryu platform and the application relies on `PACKET_IN` events from the main Ryu event thread. Furthermore, the application extends the core data model with two additional concepts: `Host` and `PathFlow`. The `Host` concept is a subclass of the `Port` concept and stores MAC-IP associations. The `PathFlow` stores associations between installed `Flow` objects and `Path` objects.

Upon a `PACKET_IN` reception, the application extracts and stores any MAC-IP association. If the packet is an ARP request and the application has an association in the knowledge base, it injects an ARP response using a `PACKET_OUT` message. In case of an IP packet with a known destination address, a path establishment process is initiated. The application requests from the *PathService* to populate the database with all available paths between the two end-points. Subsequently, it selects the path with the lowest path count and requests from the *FlowService* to install appropriate flow entries in the switches' flow tables.

Path-based learning switch application ensure path connectivity during the occurrence of low-level topological events, like link loss or host mobility. The topology discovery module monitors and stores such events in the knowledge base and individual applications are responsible to detect potential state inconsistencies and resolve them. ReasoNet applications use ICV rules, through the *ConflictManager* module, to automate this process. Listing 5 depicts the query of an ICV rule which retrieves all `PathFlow` objects and inspect the state of the input and output ports. If any of the ports is offline, then the query will raise an inconsistency in the ontology. The learning switch application will remove all invalid flows from the network and select the next best `AvailPath` to realize the path.

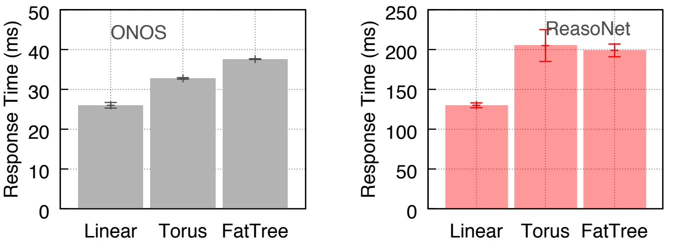

Fig. 3: Path reconfiguration latency (5th, 50th and 95th percentile) for ReasoNet (right) and ONOS (left) during a link failure. The percentiles for some topology-controller combinations are extremely close and unnoticed in the graph.

Discussion: Scalability is a key challenge for semantic web technologies. The scalability limitations of semantic web databases are defined by multiple factors, including: the database implementation, the ontology design, the query optimizer, the number of installed tuples, and inference/validation rules. To evaluate the scalability of ReasoNet, we compare the path reconfiguration performance of the ReasoNet learning switch application and the ONOS path intent subsystem [19], a mature SDN controller maintained by the Open Network Foundation (ONF). The ONOS intent subsystem supports automated path establishment, when possible, between MAC address pairs (`add-host-intent`), similar to our learning switch application. In our experiment, we generate ICMP traffic between two random end-hosts of the topology and disable a random switch port, which is part of the established path. We measure the latency to re-establish path connectivity, which we define as the time between the transmission of the `PORT_STATUS` message by the Open vSwitch instance that manages the disabled port and the time the controller sent the required `FLOW_MOD` messages to configure the new path between the two hosts. In order to ensure measurement accuracy, we use the monitor option of the `ovs-vsctl` command to extract OpenFlow message timestamps.

Our evaluation uses three topologies: a 4-port fat-tree (FT) topology, a 4x5 Torus (TR) topology and the custom topology from Figure 4. We repeat the experiment 20 times for each controller-topology combination and report the 5th, 50th and 95th percentiles of path reconfiguration latency in Figure 3. From the results we note that ReasoNet has an inflated path reconfiguration latency in comparison to the ONOS platform, however, the latency difference is upper-bound at 150 msec. These latency differences can be attributed to the serialization delays when transmitting information between a ReasoNet instance and the Stardog database. ONOS uses an in-memory data storage service, running as a thread within the ONOS runtime. In contrast, ReasoNet must use the OS network stack to communicate with the Stardog database. Furthermore, such latency differences can be further minimized by improving the integration of the knowledge-base with our controller.

B. QoS policy engine

A key SDN goal is the realization of new high-level policy abstractions by synthesizing low-level control interfaces. Declarative policies is a policy paradigm that has recently gained significant interest in the SDN community. Unlike Event-Control-Action (ECA) policies, which explicitly define the required transformation in the network configuration, declarative policies describe the criteria to choose an acceptable network configuration state (intent) and delegate the responsibility to define the required transformation to the controller. Advanced NOSes, like ONOS [19] and OpenDayLight [20], provide built-in support for similar policy paradigms.

ReasoNet implements an application supporting a policy engine for QoS intents. The application exposes a simple restful API, which allows external applications to inject path requests with QoS requirements. An intent request describes a flow match, an ingress and an egress switch and port and a specific bandwidth goal. The application is responsible to instantiate an end-to-end path that fulfill these requirements at run-time.

The intent application extends the ontology with two new concepts: `QoSPath` and `QoSFlow`. The `QoSPath` concept is a subclass of the `Path` concept and adds a status, match and bandwidth property. The status property is an enum type which reflects if a QoS path is *pending* (the application evaluates if the request can be fulfilled), *installed* (the application has installed the required flows), or *failed* (the application cannot fulfill the path request). The `QoSFlow` concept associate a `QoSPath` object with the respective `Flow` objects.

On a path request the application requests from the *PathService* to populate the database with all the alternative `AvailPath` objects realizing the end-to-end path. In addition, the application uses the query in Listing 6, to select an `AvailPath` object which (i) has sufficient capacity to accommodate the path request and (ii) uses links with low utilization. The application registers similar integrity checks, as Listing 5, to validate the liveness and resources of all established paths.

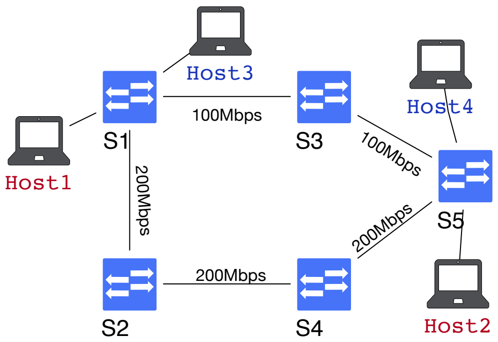

Fig. 4: The random topology used for the evaluation of ReasoNet.

Discussion: To demonstrate the functionality of our QoS policy engine application, we recreate in Mininet the topology

```
SELECT ?availPath (COUNT(?hop) AS ?hop)
       (MIN(?capacity - ?load) AS ?pathbw){
of:pathRequest a of:QosPath;
                 of:requiresBW ?bw.
?availPath a of:AvailPath;
             of:realizes of:pathRequest.
?hop a of:PathHop;
       of:belongs ?availPath;
       of:hasLink ?link.
?link a of:Link;
        of:hasLoad ?load;
        of:hasCapacity ?capacity.
} GROUP BY ?availPath HAVING(?pathbw > ?bw)
ORDER BY DESC(?pathbw) ?hop
LIMIT 1
```

Listing 6: A SPARQL query selecting an `AvailPath` object that realize the `QoSPath` object with label `of:pathRequest`. The query selects a path with the lowest average link load over its links.

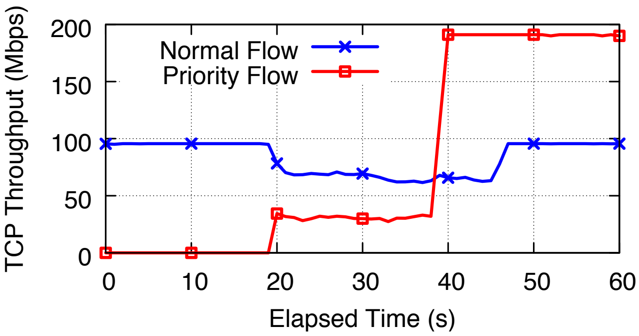

Fig. 5: TCP throughput (1 min monitoring window) during a QoS intent update scenario. The priority flow starts transmission at **t=20** sec and at **t=40** sec the manager updates its bandwidth goal, triggering the control application to migrate the flow over a path with higher capacity (200Mbps).

depicted in Figure 4. We generate two steady-state TCP flows: a *priority* flow between hosts 1 and 2 and a *normal* traffic flow between hosts 3 and 4. We generate TCP traffic using the Iperf traffic generation tool. The QoS policy engine is initialized with a set of `QoSPath` rules with a zero minimum bandwidth guarantee. The rules effectively enable best-effort connectivity between hosts. Because both paths have no bandwidth guarantees, they both use the Path s1-s3-s5, since the path computation query favors shorter paths. The *normal* flow starts at time t=0 sec, while the *priority* flow starts data transmission at time t=20 sec. At time t=40 sec we update the `QoSPath` rules for the *priority* flow and increase the minimum bandwidth guarantee to 120 Mbps.

Updating the minimum bandwidth of the `QoSPath` entity, violates initially one of the ICVs of the QoS application; the existing path capacity is 100 Mbps, insufficient to fulfill the 120 Mbps request, and the NOS will report this violation to the application using a registered callback. Using the query in Listing 6, the QoS application will force the *priority* flow to follow the longer path s1-s2-s4-s5, providing higher available bandwidth. As seen in Figure 5, the transition to the new path incurs no performance degradation on both flows. The new path allows the *priority* flow to fully utilize the available capacity of the new path (200 Mbps).

V. Related Work

In this section, we review the recent related works looking into the different aspects including: data models for NOSes, network modelling standardization and application development APIs efforts.

SDN Data Models: ONOS and OpenDayLight (ODL), the most mature open source NOS frameworks, follow significantly different data modeling approaches. ONOS relies on the JAVA language to describe core NOS interfaces and objects. ODL uses the YANG data modeling language to specify service objects and interfaces. A YANG compiler generates the appropriate JAVA objects and interface files and marshaling code during compilation, while the interfaces can be exposed seamlessly through the ODL Restfull API. Furthermore, ODL uses a Model-Driven Service Adaptation Layer [21] (MD-SAL) to aggregate all active models and interfaces and allow entities to bind to interfaces at run-time. In comparison, ReasoNet offers richer modeling capabilities, supporting data integrity validation and run-time data model adaptation.

Recently, NOS implementations actively explore the applicability of declarative policy languages in SDN networks. Declarative policies define high-level policy goals, which must be fulfilled by the NOS at run-time. ONOS provides low-level intent support as a core service and applications can define end-to-end path intents between hosts or ports. ODL provides three different declarative policy frameworks. Firstly, ODL has built-in support for NEMO [22], a declarative policy language developed by Huawei. NEMO defines three primitive type: Node, Connection and Flow. Using these primitives, an application can describe its connectivity requirements, as well as, define flexible monitoring and notification functionalities. The ODL Network Intent Composition (NIC) [23] framework defines a set of intents to control path establishment. A user can describe whether different end-points or services can communicate over the network using NIC. ODL Group-Based Policy (GBP) [24] extends the NIC access control model in order to match the requirements of a cloud environment. GBP policies describe how (contracts) different network end-points (*e.g.,* application, tenant) can connect. In comparison to existing intent frameworks, ReasoNet offers greater flexibility since it allows control applications to extend the primitives of the intent framework.

Table I compares the different data modeling mechanisms in modern NOS. Our analysis focuses on the aspects of extensibility, conflict management and integrity constraint support. From the table entries, we identify that the ODL MD-SAL, the NEMO language and our system provide the best feature set. Nonetheless, the ODL MD-SAL modeling framework lacks support for high-level integrity constraints across models, while the NEMO language is constraint by the set of primitive language types.

NOS	Language Support	Extensibility	Conflict Management	Constraints
ONOS Intent	CLI and API	Low	None	Type, Resources
ODL MD-SAL	Yang model	High	None	Type
ODL+NEMO	NEMO SDL	High	None	Type, Resources
ODL+NIC	Yang Model	Low	None	None
ODL+GBP [24]	UI and API	Low	None	None
ReasoNet	SPARQL	High	Programmable	Type, State, Resource

TABLE I: Feature comparing of data modeling mechanisms in open-source NOS.

Network Modeling and Ontologies: Multiple standardization bodies have developed information and data models for network infrastructures. The Internet Engineering Task Force (IETF) runs multiple working groups developing network data models. The IETF NETCONF data modeling language (netmod) WG develops YANG models [2] to control network properties from the physical layer (*e.g.,* Flexi-Grid optical networks [25]) up to the application layer (*e.g.,* firewall [26]). The Open Network Foundation (ONF) is actively developing data models and protocols for all SDN network control layers. Of interest to our work is the Core Information Model (CIM) [6], a generic and extensible information model aiming to converge control across southbound interfaces.

Semantic web technologies have been applied previously to model networks and many specialized network ontologies exist. The Network Markup Language (NML) [27] is an ontology converging network control and configuration for different physical and network layer technologies. The Infrastructure and Network Description Language (INDL) [28] extends NML with support for computation resources. The NOVI information model [29] adopted INML to model computational and network testbed resources for the service deployment.

NML and its extensions have been used to represent network maps and to exchange topology informations between different administrative domains. Effectively, NML is an information model, abstracting functional capabilities across technologies from technology-specific details. Unfortunately, lack of technology-specific data models and drivers limits its applicability. Pantuza *et al.* [30] integrate the NML language with the OpenFlow protocol, using the neo4j graph database. Neo4j provides excellent support for graph processing, but it lacks reasoning and consistency validation automation capabilities. [31] uses semantic web technologies to model physical substrate resources, service function chain and deployment constraints for virtual function embedding.

Application Development APIs: Pyretic is an SDN programming language using high-level services to abstract low-level SDN functionality [32]. A Pyretic control application effectively describes how to combine core control services in order to realize the desired functionality. The modules and the composition mechanisms are designed to guarantee verifiable synthesis. However, the introduction of a new core module must ensure compatibility with the API semantics and language theory. Extensions may be required to accommodate new capabilities. PANE [4] enables automated and conflict-free network policy synthesis from the output of control applications. Conflict-resolution uses hierarchical policy trees, which are explicitly defined by the network operator. FRESCO [33] is a network controller providing secure composition of control applications in an SDN network. The system uses a security enforcement kernel, through which a network manager can define application priorities to resolve policy conflicts. ReasoNet follows a similar approach and enables a flexible conflict resolution mechanism, where network managers can define conflict policies as SPARQLRule rules and specialized functions can be implemented to resolve policy conflicts.

VI. Discussion and Conclusion

In this paper, we have discussed the data modeling requirements of modern SDN control systems. We have identified a number of limitations in the current solutions with respect to consistency, extensibility and debugging. To address these challenges, we advocated the adoption of semantic web technologies as a data modeling and management mechanism. Towards this goal, we presented the first OpenFlow ontology and ReasoNet, an SDN controller with support for automated consistency validation and knowledge reasoning. We demonstrated the ability of ReasoNet to support flexible control application development, by elaborating on the design of two control applications, namely a learning switch and a QoS policy engine.

We believe that ReasoNet is a step towards the adoption of extensible and feature-rich data modeling mechanisms by SDN platforms. In our study, we limited our exploration on wired Ethernet network technologies. Nonetheless, a number of standardization bodies have developed detailed ontologies of additional network technologies, like LTE [34] and sensor networks [35]. Existing production SDN technologies provide limited control below the data link layer. Semantic web technologies can seamlessly introduce support for different network technologies by using relevant ontologies. In addition, semantic web technologies allow applications to delegate subquery execution to other databases. This functional can be exploited to provide the required information abstraction when developing control application for multi-administrative domains.

Our choice of semantic web technologies has highlighted a number of limitations in existing semantic web tools. Firstly, although the Stardog software can efficiently extract knowledge by parsing the graph of the underlying database, it offers limited graph processing capabilities. ReasoNet employs a third-party graph processing framework to support its path computation requirements. The Stardog developers currently explore new extensions for their SPARQL query engine to support compound graph traversal operations[3]. Secondly, during

[3] https://www.stardog.com/blog/a-path-of-our-own/

ing the development of our straw-man implementation we identified a number of limitations in the specification of validation rules using SPARQLRules, including the lack of support for non-pure functions and nested queries in the IF statement. Support for such features can significantly improve the modeling capabilities for SDN control.

Acknowledgments

The authors are grateful to the UK Engineering and Physical Sciences Research Council (EPSRC) for funding the TOUCAN project (EP/L020009/1), which supported much of the work presented in this paper; and to the EPSRC and BT for funding Daniel King's Industrial CASE PhD research. We would also like to specially thank Dr Vatsala Nundloll for sharing her pearls of wisdom with us during the development of the ontology.

References

[1] Open Network Foundation, "ONF TR-521: SDN Architecture 1.1," 2016.

[2] M. Bjorklund, "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)," RFC 6020, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc6020.txt

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *ICFP '11*. ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2034773.2034812

[4] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory Networking: An API for Application Control of SDNs," in *SIGCOMM 2013*, 2013.

[5] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul, "Democratic resolution of resource conflicts between sdn control programs," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 391–402. [Online]. Available: http://doi.acm.org/10.1145/2674005.2674992

[6] Open Network Foundation, "Core Information Model (CoreModel)," https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/ONF-CIM_Core_Model_base_document_1.1.pdf, 2015.

[7] Open Networking Foundation, "Project ASPEN: real time media interface specification," http://opensourcesdn.org/projects/project-aspen-real-time-media-interface-specification/.

[8] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences," in *SIGCOMM 2014*. ACM, 2014.

[9] R. Durairajan, J. Sommers, and P. Barford, "Controller-agnostic SDN Debugging," in *CoNEXT 2014*. ACM, 2014.

[10] "Resource Description Framework," 2014. [Online]. Available: https://www.w3.org/RDF/

[11] "RDF Schema," 2014. [Online]. Available: https://www.w3.org/TR/rdf-schema/

[12] "OWL 2 Web Ontology Language," 2012. [Online]. Available: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/#Documentation_Roadmap

[13] "SPARQL 1.1 Query Language for RDF," 2013. [Online]. Available: https://www.w3.org/TR/sparql11-overview/

[14] "Stardog: The Knowledge Graph Platform," 2018. [Online]. Available: http://www.stardog.com

[15] "Semantic Web Rule Language," 2004. [Online]. Available: https://www.w3.org/Submission/SWRL/

[16] "neo4j graph database," https://neo4j.com/, 2015.

[17] Open Network Foundation, "OpenFlow Switch Specifications - version 1.3.0," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.0.pdf, 2015.

[18] "Apache TinkerPop," https://tinkerpop.apache.com/.

[19] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *HotSDN*. ACM, 2014.

[20] "The OpenDaylight Platform," https://www.opendaylight.org/.

[21] OpenDayLight, "Open Day Light Model Driven Service Adaptation Layer Architecture," 2017. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Explained

[22] S. Hares, "Intent-Based Nemo Overview," Internet Draft, RFC, Oct. 2015. [Online]. Available: https://tools.ietf.org/pdf/draft-hares-ibnemo-overview-01.pdf

[23] "Open Day Light Network Intent Composition (NIC) project," 2015. [Online]. Available: https://wiki.opendaylight.org/view/Network_Intent_Composition:Main

[24] "Open Day Light GBP project," 2015. [Online]. Available: https://wiki.opendaylight.org/view/Group_Policy:Main

[25] J. E. L. de Vergara, D. Perdices, V. Lopez, O. G. de Dios, D. King, Y. Lee, and G. Galimberti, "YANG data model for Flexi-Grid Optical Networks," Internet Engineering Task Force, Internet-Draft, Mar. 2017, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-vergara-ccamp-flexigrid-yang-04

[26] J. L. de Vergara, D. Perdices, V. Lopez, O. G. de Dios, D. King, Y. Lee, and G. Galimberti, "Network Access Control List (ACL) YANG Data Model," Internet Engineering Task Force, Internet-Draft draft-ietf-netmod-acl-model, Mar. 2017, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-netmod-acl-model

[27] J. Van Der Ham, F. Dijkstra, R. Kapacz, and A. Brown, "The Network Markup Language (NML) A Standardized Network Topology Abstraction for Inter-domain and Cross-layer Network Applications," in *TERENA Networking Conference*, 2013.

[28] M. Ghijsen, J. Van Der Ham, P. Grosso, C. Dumitru, H. Zhu, Z. Zhao, and C. De Laat, "A semantic-web approach for modeling computing infrastructures," in *Computers and Electrical Engineering*, vol. 39, no. 8. Elsevier Ltd, 2013, pp. 2553–2565. [Online]. Available: http://dx.doi.org/10.1016/j.compeleceng.2013.08.011

[29] J. Van Der Ham, J. Stéger, S. Laki, Y. Kryftis, V. Maglaris, and C. De Laat, "The NOVI information models," *Future Generation Computer Systems*, vol. 42, 2014.

[30] G. Pantuza, F. Sampaio, L. F. M. Vieira, D. Guedes, and M. A. M. Vieira, "Network management through graphs in Software Defined Networks," *10th International Conference on Network and Service Management (CNSM) and Workshop*, pp. 400–405, 2014.

[31] N. Bouten, M. Claeys, R. Mijumbi, J. Famaey, S. LatrÃľ, and J. Serrat, "Semantic validation of affinity constrained service function chain requests," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 202–210.

[32] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482629

[33] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 121–126.

[34] Q. Zhou, C. X. Wang, S. McLaughlin, and X. Zhou, "Network virtualization and resource description in software-defined wireless networks," *IEEE Communications Magazine*, vol. 53, no. 11, pp. 110–117, November 2015.

[35] W3C Semantic Sensor Network Incubator Group, "Semantic Sensor Network Ontology," https://www.w3.org/TR/vocab-ssn/, 2017.