Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

A Semantic Data Lake Framework for Autonomous Fault Management in SDN Environments

Fernando Benayas | Álvaro Carrera* | Manuel García-Amado | Carlos A. Iglesias

[1]Intelligent Systems Group, Universidad Politécnica de Madrid, Madrid, Spain, Email: f.benayas@upm.es, a.carrera@upm.es, manuel.garcia-amado.sancho@alumnos.upm.es, carlosangel.iglesias@upm.es

Correspondence
*Álvaro Carrera,
Av. Complutense, 30, 28040, Madrid, Spain
Email: a.carrera@upm.es

Fault Management is a vital issue for any network operator since the beginning of the telecommunications era. As networks have become more and more complex, their management systems are crucial for any operator company. In this ecosystem, the Software-Defined Networking (SDN) approach has appeared as a possible solution for different networking issues. The flexibility provided by SDN to the network management enables a great dynamism in the configuration of network devices. However, this feature introduces the cost of a potential increase in failures since every modification introduced on the control plane is a new possibility for failures to appear and cause a decrement of the quality for offered services. Because of the growing pace of the networks, the classical approach is not feasible to cope that dynamism. Increasing the number of human operators in charge of the fault management process would increase the operation cost dramatically. Thus, this paper presents an approach to apply machine learning over a big data framework for an autonomous fault management process in SDN networks. In this paper, we present a Semantic Data Lake framework for a self-diagnosis service which is deployed on top of an SDN management platform. Also, we have developed a prototype of the proposed service with different diagnosis models for SDN networks. Models and algorithms have been evaluated showing good results.

KEYWORDS:
SDN, Bayesian network, fault diagnosis, Machine Learning, Semantic, Data Lake

ACKNOWLEDGMENT OF ACCEPTANCE FOR PUBLICATION

This is the pre-peer reviewed version of the following article: Benayas F, Carrera Á, García-Amado M, Iglesias CA. A semantic data lake framework for autonomous fault management in SDN environments. Trans Emerging Tel Tech. 2019;e3629. https://doi.org/10.1002/ett.3629, which has been published in final form at https://doi.org/10.1002/ett.3629. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions.

1 | INTRODUCTION

Nowadays, one of the problems which have more impact on network operation is the traffic volume of the telecommunication networks[1]. The massive amount of data that networks have to transport is caused by the growing number of devices in our daily

life[2], by the adoption of new networking technologies and services[3], such as 5G or high-quality video streaming, or by the changes in the computational paradigm[4], i.e., cloud computing.

Flexible routing policies are needed to face the dilemma of unclogging computer networks in such a hostile environment traffic-wise[5]. To increase flexibility in computer networks, the use of Software-Defined Networking (SDN) technologies provides levels of flexibility never reached before[6], allowing the quickly modifying of routing policies according to such factors as traffic load or pre-defined rules.

Due to this advantage, the network industry is promoting the adoption of SDN. One of the main points in the strategic agenda of major companies heavily reliant on computer networks (such as telcos or Multiple Service Operators (MSO)) is the adoption and implementation of SDN technologies[7]. Indeed, more than 80% of Cloud Service Providers (CSP) network executives rank as "significant" the potential impact of the adoption of Network Function Virtualization (NFV) and SDN technologies in the operational model, and more than 50% of them are preparing investments for the migration to SDN in their data centers and mobile cores[8]. Furthermore, there is an expected growth at a Compound Annual Growth Rate (CAGR) of 71.4% in the 2017-2022 period in the SDN and NFV market, therefore reaching a market value of USD 54.41 billion[9].

Nevertheless, the advantages that SDN brings to the network environment, it also has some issues of resiliency. Since dynamic management involves constant changes in networking policies, there are numerous chances for the designing of faulty routing policies or failures regarding the implementation of such rules. Therefore, a system that audits these changes and diagnoses such failures is needed to avoid the possibility of inhibiting the benefits of adopting SDN technologies. This need is the motivation for the system proposed in this paper.

Since this system relies heavily on the collecting and processing of data, the use of semantic technology can significantly increase flexibility in the treatment of such data. It also allows us to ease interaction among different systems since two systems need to share their data models to adapt their data mining processes and be compatible with each other. Furthermore, the definition of ontologies for the data representation within common models facilitates the interaction with an ever-growing Semantic Web using standards as Resource Description Framework (RDF).

A previous work of the architecture presented in this paper was introduced at the 2018 Fifth International Conference on Software Defined Systems (SDS)[10]. In this paper, we include a semantic layer to enable the autonomous diagnosis process by an intelligent agent. We have developed several ontologies for the data description and the diagnosis process of our SDN scenario. We have also created an orchestration system that coordinates each process according to a diagnosis model for SDN networks.

This paper is structured as follows. Section 2, introduces related works in the field of failure management in SDN environments. Section 3 proposes an architecture for a Semantic Data Lake for Fault Management. Section 4 describes the semantic knowledge models used in the diagnosis process. In Section 5, we define a worked example to show the application of the semantic models. Later, Section 6 shows the results of the evaluation of the diagnosis module. Finally, Section 7 summarises some conclusions and explore possible paths for future work.

2 | RELATED WORK

Autonomic Networking is a crucial concept of the Future Internet[11], and different approaches have been explored for traditional telecommunication networks, both centralized[12] or distributed[13,14] approaches. However, in recent years, some reference models have been generated by ETSI and IRTF, such as Generic Autonomic Networking Architecture (GANA)[15] or RFC 7575[16] and RFC 7576[17], to build a common architecture for the next-generation networks. Moreover, SDN offers some attractive advantages to be used as a fault-aware element for some critical points for next-generation networks, such as Big Data[18] and smart grid infrastructures[19] or dynamic bandwidth management for data centers[20].

Software-Defined Networking (SDN) offers some key capabilities to enable autonomic network management, and the community has explored different approaches[21,22]. An extension of the OpenFlow protocol[23], named "OpenState"[24], introduces state machines into the switches for triggering transitions at the packet-level event which promotes a quick solution to node or link failures. This approach delegates some actions to network devices, but other approaches proposes recovery methods centralized in the network controller[25].

An agent-based solution is applied in SDN-RADAR[26]. Those agents act as probes when bad performance is detected in a specific service consumed by the client. Tang et al.[27] propose a system mapping by combining the network topology with the "Policy View" of each service. Using this combined view and an SDN reference model, a belief network is built for each service and is used to reason about the fault location.

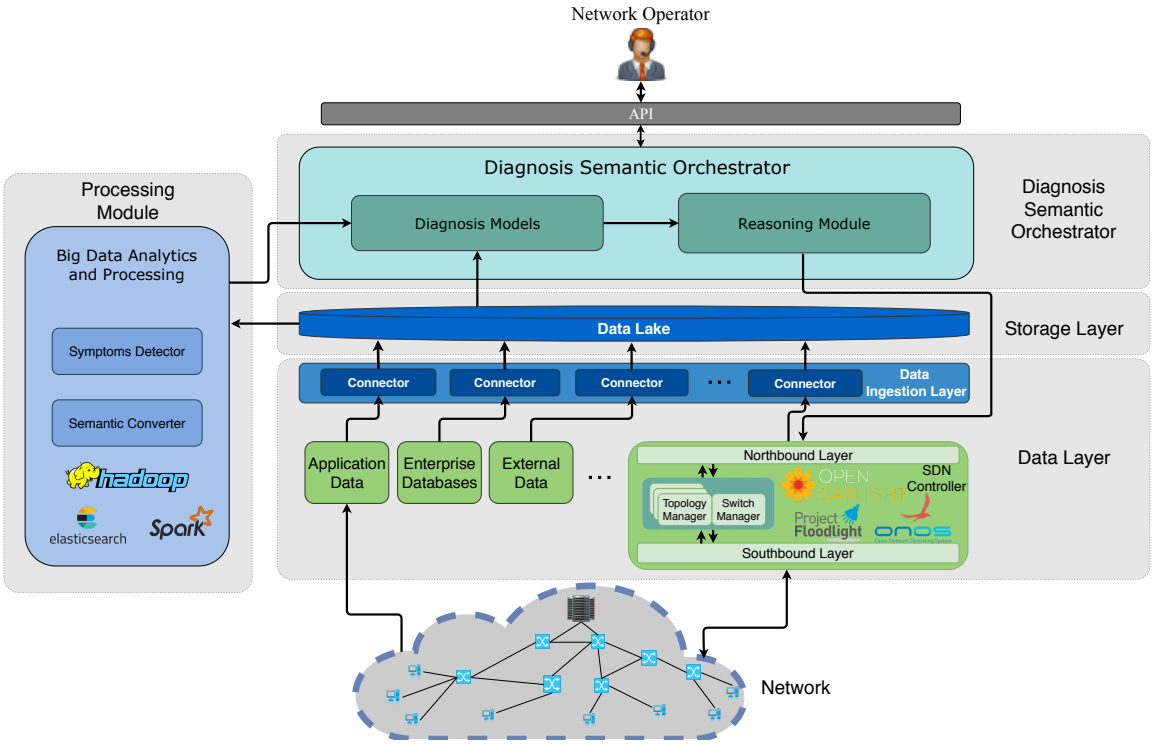

FIGURE 1 Overview of the proposed Fault Diagnosis Architecture.

A different method is proposed by Chandrasekaran and Benson[28], considering SDN application failures instead of link/nodes ones. To achieve fault tolerance for SDN applications, a new isolating layer, called "AppVisor", is defined between applications and SDN controller. It also defines a module which enables network transforming transactions between application and controller to be atomic and invertible.

Another approach which covers different network levels is the self-healing architecture proposed by Sánchez et al.[29]. This proposal includes monitorization of data level, control level, and service level, offering a global view of the problem. Moreover, this architecture proposes the application of self-modeling techniques for the generation of fault diagnosis models. Our proposal focuses more on the network behavior than in the topology itself to complement this approach, as described below. We extend a previous approach for Fault Diagnosis of non-SDN Telecommunication Networks[30] which combines the application of intelligent techniques with semantic modeling[31]. In this paper, we present an extension for the models used in the previous approach to extend them for an SDN environment.

3 | SEMANTIC DATA LAKE ARCHITECTURE FOR FAULT MANAGEMENT

This section presents a Semantic Data Lake Architecture for Fault Management based on Bayesian Reasoning for SDN Environments. Figure 1 shows an overview of the architecture which is based on a Data Lake Architecture which ingests data from different sources, being the SDN controller one of them. Section 3.1 presents the Data Layer, which contains all data sources. Section 3.2 introduces the Storage Layer, which raw data is collected and, later, processed in the Processing Module, shown in Section 3.3. Finally, Section 3.4 explains the features of the *Diagnosis Semantic Orchestrator*, which is the module responsible for carrying out the Fault Diagnosis process through its different stages.

3.1 | Data Layer

The proposed architecture is based on a standard Data Lake architecture[32] where data come from different sources. In our scenario, the most critical data source is the network controller which performs management tasks over the devices. Moreover, the SDN controller provides useful detailed information about the status of every network element.

Using the northbound Application Programming Interface (API) of the SDN controller, we can collect data via simple Representational State Transfer (REST) request. However, even inside this significant data source, i.e., the network controller, we find several essential data sources about different aspects of the network, such as topology or traffic.

3.2 | Storage Layer

This layer is responsible for ingesting all data sources in the data lake in raw format. Later, data are processed and enriched with semantic models. That ingestion is carried out by connectors. Every connector is responsible for collecting and pushing data from a specific data source in the data lake. Here, we have different strategies: streaming or periodic ingestion. If the data source generates valuable data for our system, real-time ingestion using streaming techniques in the connector implementation must be done. However, if the information from the data source is static, or at least, not real-time required, the connector can be implemented as a periodic data collector.

3.3 | Processing Module

This module is responsible for analyzing and processing all data coming from data sources, which is stored in Data Lake. For that purpose, the usage of Big Data platforms, such as ElasticSearch[33], is required for indexing and classifying high data volumes for further processing. That indexation is highly desirable for the conversion from raw data to enriched semantic data.

Thus, we enrich raw data in the *Semantic Converter* Module, which use semantic models, proposed in Section 4, to represent all relevant data for the Fault Management process.

After semantic models have been applied for representing both network and diagnosis, a module is created for tagging data according to these models. This mapping is implemented separately for each data source within the Data Lake. The annotated data resulting from such mapping can be stored in any RDF storage system, such as Apache Fuseki[34], which allows for the retrieval of RDF triples through semantic queries.

Summarising, we propose the following features for this module. Based on data collected from the SDN environment, we process and enrich those data with semantic models and select specific symptoms that can suggest a type of fault. Some of them need further processing, such as time series analysis, ontology-based reasoning or other variables can be directly discretized, assigning a class depending on the range that contains the value. Finally, processed data is converted to an adequate format to generate diagnosis models or to infer the most probable cause of fault using causal models.

3.4 | Diagnosis Semantic Orchestrator

In the designing of fault diagnosis task, we have followed the B2D2 Diagnosis Model[30] (further explained in Section 4.1), which proposes the division of diagnosis process into three phases: *Symptom Detection*, *Hypothesis Generation* and *Hypothesis Discrimination*.

Consequently, the execution of the diagnosis service starts in the processing module, where the task of *Symptom Detection* is performed. This task consists of comparing features in the data collected with expected values extracted from historical data if available. Depending on the result obtained from this comparison, the diagnosis service either saves the observation and continues the search for symptoms in new entries or appends the symptom detected.

Once a symptom has been detected, the reasoning process for hypothesis generation and discrimination is carried out in this module inferring with data provided by the processing module presented in Section 3.3 and the symptom appended. Multiple reasoning techniques can be used for this task. For instance, rule-based reasoning provides a straightforward method to express domain knowledge. That knowledge can be represented as rules expressing cause-effect relationships. However, it is incapable of dealing with situations uncovered by those rules. Another alternative is the application of probabilistic reasoning techniques. Specifically, causal models based on Bayesian reasoning are interesting for our uncertain and complex environment. They make a heuristic model that relates symptoms and cause of a fault, obtaining the probability of a failure based on those observed variables. Thus, we apply Bayesian networks as *causal models* for fault root cause inference models. Mainly, in the *Hypothesis*

Generation phase, Bayesian models are used to infer a set of possible faults according to the observations made and the symptoms detected.

Having both a set of hypotheses and symptoms detected, we reach the third and last phase of the diagnosis process as defined by the B2D2 Diagnosis Model: the *Hypothesis Discrimination* phase. To obtain a final diagnosis over the set of hypothesis used as input, we define a time window in which we perform additional *Symptom Detection* tasks. This allows us to obtain new sets of hypotheses with probabilities defined for each of the possible faults. Then, we recursively examine if each probability is over a defined threshold: if it is, we reach a final result for the diagnosis; otherwise, we execute test actions over the monitored network to update the probabilities of *hypothesis set*.

This process needs to be executed sequentially for each new entry of data. To achieve this, we create a *Diagnosis Semantic Orchestrator*, which uses the knowledge models explained in Section 4.

4 | SEMANTIC MODELLING FOR FAULT MANAGEMENT IN SDN

The proposed architecture is based on a previous work focused on Fault Management in traditional telecommunication networks[30]. That work proposed the BDI for Bayesian Diagnosis (B2D2) model as a knowledge model for the definition of domain knowledge and inference knowledge which allows an intelligent agent to carry out autonomously the fault diagnosis process. This B2D2 Knowledge Model is briefly presented in Section 4.1. The proposed description language for the SDN scenario is presented in Section 4.2, and a specific Diagnosis Model for our scenario is presented in Section 4.3.

4.1 | An overview of the B2D2 Knowledge Model

A knowledge model is composed of different models, each capturing a related group of knowledge structures. Those models describe different aspects of a specific problem to enable the development of a solution to solve it. This section presents an overview of the B2D2 Knowledge Model which relates a set of models which enables an agent to carry out an autonomous diagnosis process, as summarized in Figure 2. Those knowledge models are divided into two main areas: *Domain Model* and *Inference Model*. The *Domain Model* used to describe the domain knowledge, in our case, fault diagnosis of telecommunication networks. The *Inference Model* is used to define the tasks required to carry out a diagnosis process.

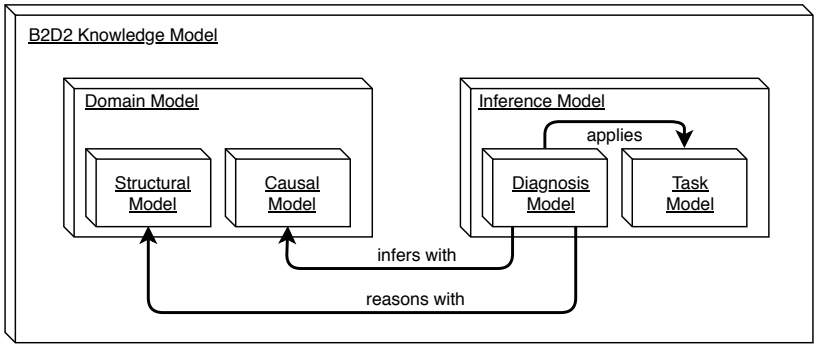

FIGURE 2 Overview of the B2D2 Knowledge Model.

The *Domain Model* describes the main static information and knowledge in an application domain, in our case, the fault diagnosis task for telecommunication networks. This section exposes two types of domain models which offer complementary views of the domain knowledge. Those models must be instantiated by agents with their knowledge bases which must contain information about specific networks they are diagnosing. The *Structural Model* contains knowledge about the network. In this paper, we use the Software-Defined Networking Description Language (SDNDL), presented in Section 4.2, as the language

to represent this type of knowledge. The *Causal Model* relates the symptoms with possible fault root causes while handling uncertainty using Bayesian networks. This model uses an existing language published as PR-OWL[†] ontology [35].

The *Inference Model* describes how domain knowledge can be applied to carry out the reasoning process. The *Task Model* defines the different phases of a diagnosis process and proposes several solving methods to perform it. This model is based on the analysis exposed by Benjamins [36]. The *Diagnosis Model* combines Structural and Causal Models to allow an agent to autonomously handle that knowledge during the different phases of the diagnosis process defined by the Task Model. In this paper, we propose an extension of the *Diagnosis Model* [30] for our SDN scenario in Section 4.3.

4.2 | SDN Description Language

In this section, a semantic model for the definition of SDN and OpenFlow environments, called Software-Defined Networking Description Language (SDNDL)[‡], is proposed to provide descriptions that are independent of the technologies or platforms used. In our work, this language is used to implement the *Structural Model* of the *B2D2 Knowledge Model* commented in the previous section.

The proposed semantic model extends Infrastructure and Network Description Language (INDL) [37] and Network Markup Language (NML) [38] for the description of network inventory. These models offer a set of core classes useful to describe any telecommunication network. Specifically, the main class proposed in NML is the *Network Object*, which is an abstract concept that includes any type of element present at the data plane of an SDN network. To specify a *Network Object* element, we can use a set of subclasses (such as *Port*, *Node* or *Service*) to refine these elements further. The NML also defines a *Topology* class to include the concept of a network domain. *Topology* class represents the grouping of multiple *Network Object* elements along with relations among them (*Links*).

This semantic model is well suited for mapping the data plane of an SDN network since it can represent all the elements which compose it. However, this model does not provide any class for the mapping of elements within the control plane; therefore, an extension for this model is needed to include the concepts and components present in the control plane. Therefore, we propose Software-Defined Networking Description Language (SDNDL), a semantic model to describe both the control and data planes of an SDN network. Moreover, the descriptions obtained from such language is technology-independent.

Since NML is a language well suited for describing the data plane of an SDN network, we have inherited most of its classes for the definition of a schema which describes the data plane in SDNDL. We have used the *Node* concept, defined by NML, as grounds for the representation of elements within the data plane of the SDN environment.

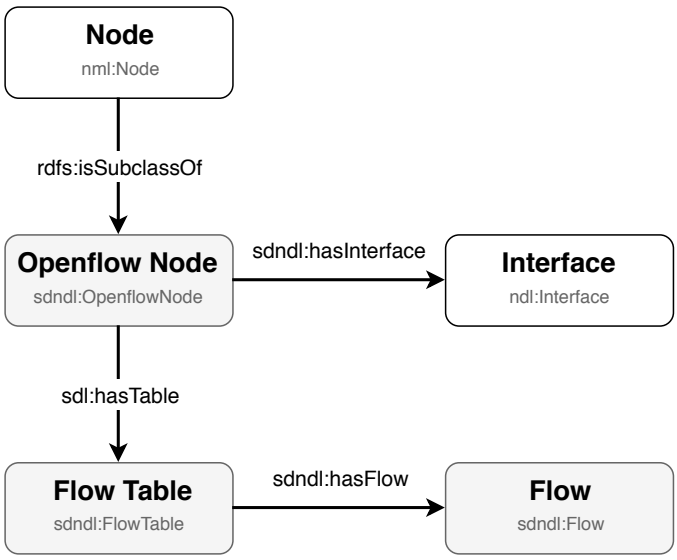

FIGURE 3 Data Plane Hierarchy

[†]PR-OWL Website: http://pr-owl.org/
[‡]SDNDL Website: http://www.gsi.upm.es/ontologies/sdndl/

As we can see in Figure 3, we create a class *OpenFlow Node*, which corresponds to a node which implements the OpenFlow protocol. Since routing in OpenFlow-based nodes is based on Flow Tables, we also add a *Flow Table* class. We also define a concept, *sdndl:hasTable*, and we link it as a property to the *OpenFlow Node*, to model routing policies within each node. Since an *OpenFlow Node* is a switch, it has interfaces; therefore, we add the *Interface* ontology from the Network Description Language (NDL) model.

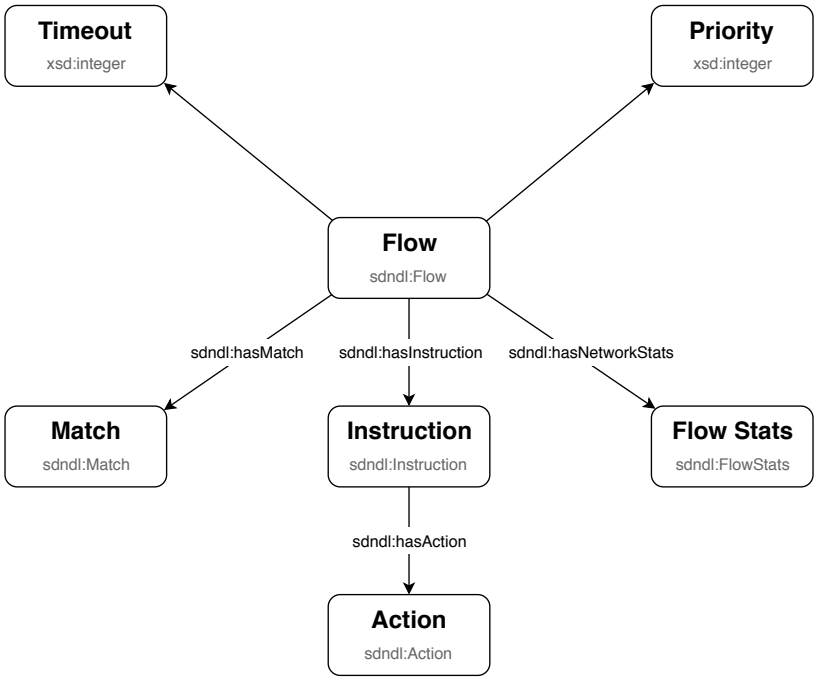

FIGURE 4 Flow Concept Hierarchy

The class *Flow* is further developed, as we can see in Figure 4. For each flow in a flow table, instructions for routing are injected. Therefore, we add an *Instruction* class to our model, which itself includes multiple fields represented with the *Action* class. To mimic real flows in the SDN network, we add the *Timeout*, *Priority*, *Match* and *Flow Stats* classes. These classes cover some initial concepts defined by the current versions of the OpenFlow protocol for the sake of generality of the model since covering all concepts defined by current OpenFlow versions creates a *Flow* model that could become easily obsolete by new OpenFlow releases.

The semantic hierarchies previously described allow us for the description of the data plane of SDN networks. However, our semantic model currently lacks classes and relations for the control plane description. Therefore, we create a control plane hierarchy that can be seen in Figure 5. While designing this hierarchy of classes, the initial approach was to define the interfaces of an SDN Controller. However, we are not interested in the inner workings of the controller, but in how it provides routing instructions to the data plane; therefore, this hierarchy is based on the *Controller* class. In this model, we represent the fact that the controller is composed of multiple services, tasked with implementing the controller functionalities.

In this model, we are specifically interested in the *Network Stats* class. These stats are generated by the *StatsManager* service. An inherited class of *Network Stats* can be seen in Figure 4 (the *Flow Stats* class).

4.3 | A Diagnosis Model for SDN scenarios

This section presents an extension for the generic *B2D2 Diagnosis Model*[§] presented in previous work[30]. The central concept of this model is the *Diagnosis* which is performed by *actors* that execute *actions* to collect *observations* from the monitored

[§]B2D2 Diagnosis Model Website: http://www.gsi.upm.es/ontologies/b2d2/diagnosis/

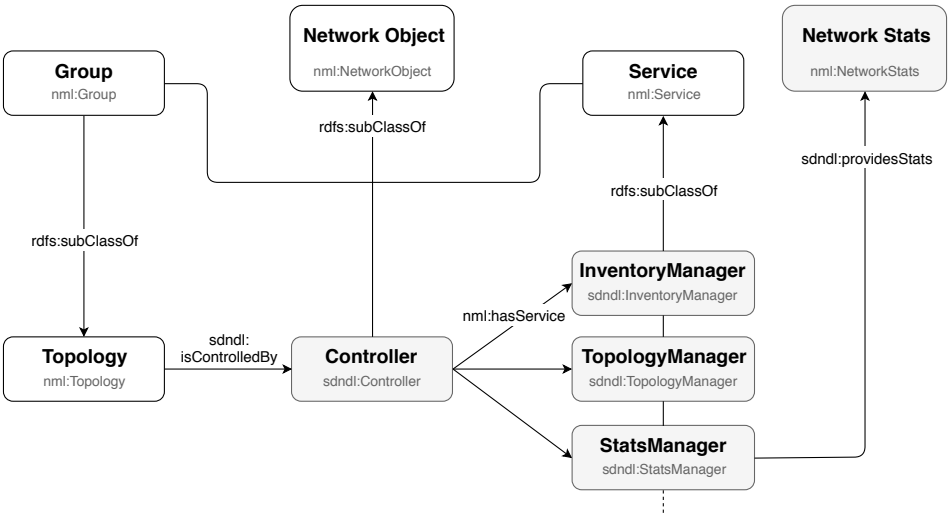

FIGURE 5 Control Plane Hierarchy

network. From these *observations*, a set of *hypotheses* is generated and discriminated until a *conclusion* with enough confidence is reached. The most important concepts in the model are simplified in Figure 6.

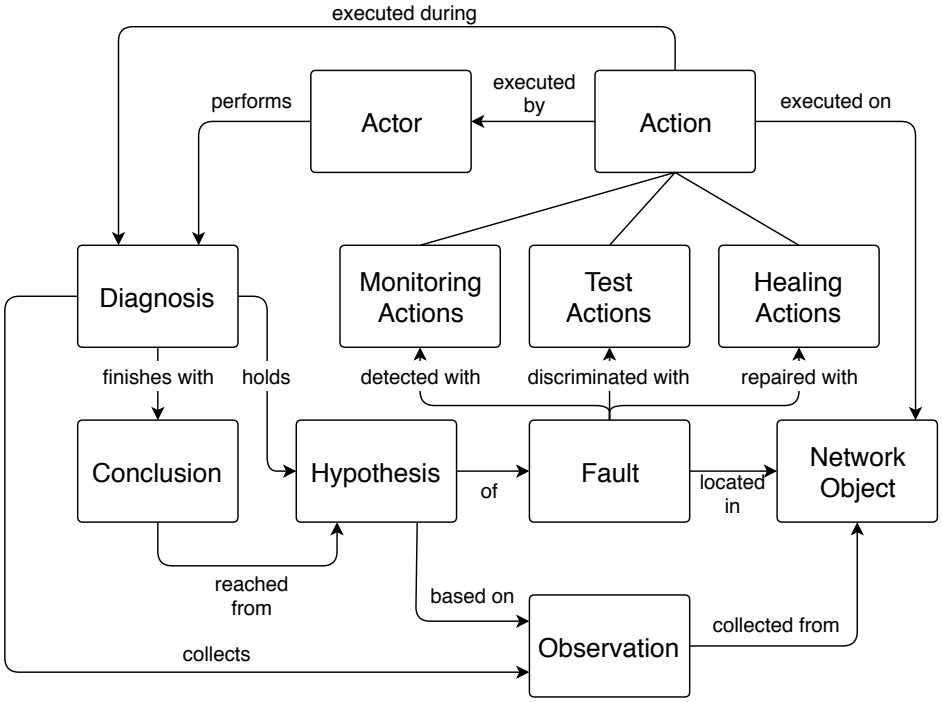

FIGURE 6 Main classes of B2D2 Diagnosis Model[30].

Extending that generic diagnosis model, we have defined *symptoms* in our SDN network that could interfere with the provision of specific services. Figure 7 shows the concept *Symptom*, defined in the *B2D2 Diagnosis Model*, with its corresponding defined sub-classes.

Moreover, we have defined the other need to complete the *Causal Model*, the possible faults. Figure 8 shows a fault hierarchy introduced inheriting from the class *Fault* in the diagnosis model.

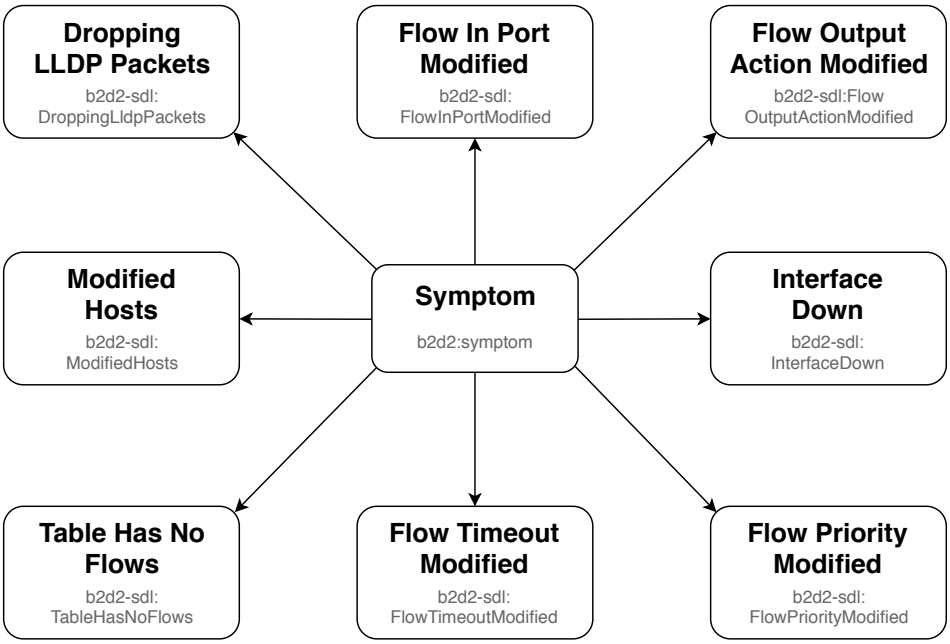

FIGURE 7 Symptoms for our SDN scenario.

We have defined symptoms as evidence variables in our SDN environment, and the possible fault root causes to complete our causal model. The causal relation among those variables is modeled with a Bayesian Network. It makes a heuristic model that relates symptoms and fault cause, inferring a hypothesis of the most probable status of the network. This status is predicted according to a set of Conditional Probability Tables (CPTs), where the conditional relations between pieces of evidence and the possible faults are modeled. Finally, Probabilistic OWL (PR-OWL) language is used for describing our Bayesian networks.

5 | A WORKED EXAMPLE

In this section, we have explained a detailed example to understand the ability of our system detecting failures. In this scenario, we deploy a streaming service, in which one of the hosts streams a sample video through one of its ports, and the other host connects to such streams. Then, we inject faults in the network to hamper the normal functioning of the streaming service. Specifically, we modify flow rules in switches within the network, to prevent the streaming flow from reaching the client.

Faults are injected through the SDN network controller API. Precisely, we force a switch to drop every package by setting the output node connector in every flow rule to the loopback interface. As a result, the streaming service is interrupted.

This section presents the semantic representation of network data using SDNDL language in Section 5.1, an overview of a faulty service in Section 5.2, a brief explanation about how faults are injected into the network in Section 5.3, and, finally, a simplified example of a diagnosis process modeled using *B2D2-SDN* extension in Section 5.4.

5.1 | Using SDNDL for Network Data Representation

The data extracted from the network-topology module is modeled using the classes shown in Figure 9. We have the concept of *Controller* (in our case Opendaylight) which has *Topology*. This topology is composed of *Nodes* (OpenFlowNodes in SDNDL) and *Links*. Example data presented in this section uses JSON-LD format[¶] for simplification.

Listing 1: Example of Topology Data using SDNDL.

```
{
"@context" : {...},
```

[¶]A brief view of JSON-LD format: https://json-ld.org/

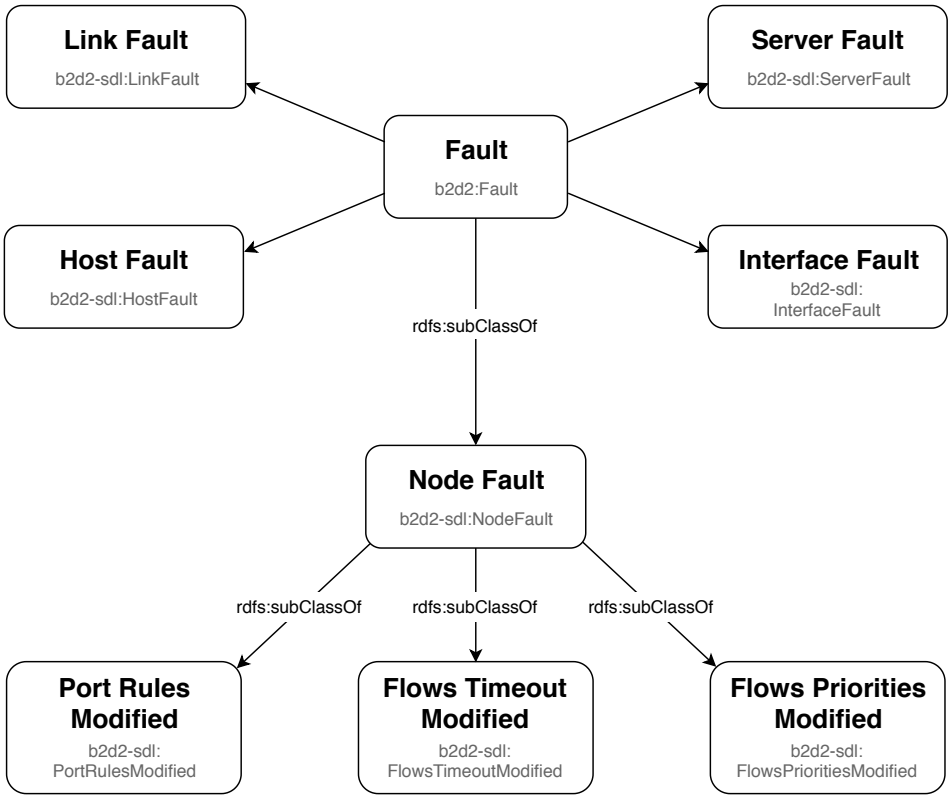

FIGURE 8 A Fault hierarchy in SDN

FIGURE 9 SDNDL Classes used for Topology Data.

```
"@id":"1518011749",
"@type":"nml:Topology",
"nml:hasNode":  [ {
        "@id": "openflow33",
        "@type": "sdndl:OpenFlowNode",
        "ndl:hasInterface": [...],
        "ndl:connectedTo": [...] } ...],
"nml:hasLink":  [{
        "@id": "openflow10_2",
        "@type": "nml:Link",
        "nml:isSource": {
          "@type": "Interface",
          "@id": "openflow10_2",
        }, ... {...}],
"sdndl:hasSnapshot":{...}
}
```

As an example, Figure 10 shows the hierarchy used for the status of nodes retrieved from the controller through the Opendaylight inventory module. Those data are modeled as *OpenFlowNode* RDF instances, each of them identified with a unique URI. An *OpenFlowNode* instance has *Flow Tables* and *Interfaces*. The *Flow Tables*, in turn, contain the *Flows* entries information.

Listing 2: Example of Inventory Data using SDNDL.

```
{
    "@context" : {...},
    "@id":"openflow33",
    "@type":"sdndl:OpenFlowNode",
    "sdndl:hasSnapshot":{},
    "sdndl:hasTable":[{ "@id":"table0",
                    "@type":"Table",
                    "sdndl:hasFlow":[{"@id":"L2switch-1","@type":"Flow",
                    "sdndl:hardTimeout":0,
                    "sdndl:iddleTimeout":0,
                    "sdndl:hasNetworkStats":{"@id":""L2switch-1#flowStats",
                                        "@type":"FlowStats",
                                        "sdndl:byteCount":25462,
                                        "sdndl:packetCount":5408},
                    "sdndl:priority":0,
                    "sdndl:hasMatch":[...],
                    "sdndl:hasInstruction":[...]
                }
    "ndl:hasInterface":[{"@id":"openflow33_2",
                        "@type":"Interface",
                        "sdndl:hasState":{},
                        "sdndl:hasNetworkStats":{},
                        "sdndl:stpStatus":"discarding"}]
}
```

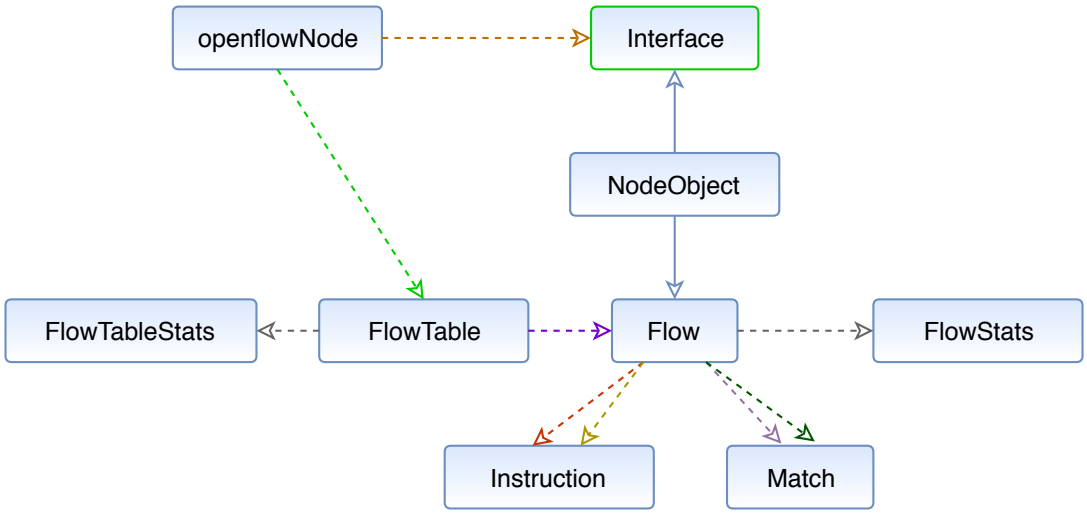

FIGURE 10 SDNDL Classes used for Inventory Data.

5.2 | Streaming Service

The streaming service uses Real-time Transport Protocol (RTP) as the protocol used at the transport layer. This protocol allows us to compensate for any jitter that could affect our transmission; it also provides detection of packet loss and out-of-order delivery, which is prevalent within real-time transmissions in an IP network.

Correctly, the RTP protocol works as follows: first, we establish a media session using the Real Time Session Protocol (RTSP) protocol. During the establishing of a media session, first, the client asks the streaming server which request types will the server accept. To do this, we send an "OPTIONS" message. Once we have this information, if the media server accepts "DESCRIBE" messages, we send a DESCRIBE message with the URL of the streaming server. Then, the server replies by sending a message with all media initialization information for the resource that it describes.

Once we have information on the server and the data is streamed, we ask the server to send us information regarding the transportation of the media stream. Specifically, we send a "SETUP" request in which we specify local ports for receiving RTP

and RTP Control Protocol (RTCP) message. The server reply confirms such parameters and includes the server's parameters. Finally, a "PLAY" message is sent by the client, and an RTP / RTCP stream is initiated.

Now the streaming session is fully established. Data is sent from the server to the client using RTP messages, and RTCP messages are sent periodically by both the client and the sender reporting statistics on the streaming session. Then, when the streaming is over, RTCP messages are used to terminate the RTP transmission, and a "TEARDOWN" RTSP message is sent by the client to terminate its session.

5.3 | Fault Generation

Once the streaming scenario is established, we insert faulty configurations into the switches that define the topology, to disable the provision of the streaming service. With this purpose, we access such configuration using the Opendaylight Northbound API. This API provides a point of access to the Opendaylight databases, where information on traffic policies being currently implemented in the network can be consulted and modified. Due to this feature, we are going to use the API as a channel through which we are going to manage traffic policies and inject faults.

Once we have chosen the method for injecting faults, we look for faults that could hinder the service provisioning. Specifically, by modifying fields in the rules associated with the streaming flow in each switch involved with the transmission, we can prevent packets from reaching their destination. Therefore, we modify flow rules in switches involved on the transmission to simulate the effect of faults within the network.

The Opendaylight database system is composed of two databases to manage flow rules: the *operational* database and the *config* database. The *operational* database holds data on the current status of the network, including traffic policies currently being implemented; however, this database does not allow for data modification. On the other hand, the *config* database doesn't hold any data: it accepts requests instead and pushes them into the *operational* database to introduce them into the network. Therefore, to inject failures, we obtain data from the *operational* database, modify it, and push it into the *config* database. When a flow is pushed into this database, the *Forwarding Rule Manager* module from the Opendaylight architecture is notified. This module sends an Remote Procedure Call (RPC) to the *OpenFlow Core Plugin* module, which then sends *FlowMod* OpenFlow protocol messages to the switches involved through the *OpenFlow Java Driver*.

Specifically, we are going to extract data representing flow rules of each switch, then we modify the fields that control the output ports of each flow, and finally, we push them into the *config* database. When this change comes into effect (following the process described in the previous paragraph) the packets of the stream are not able to reach the client of the service.

5.4 | Diagnosis Process

When a fault is generated in the previous scenario, the system detects a symptom. For example, at a certain time, a node has an unexpected change in flow from its table, and the port specified in an *entry port matches* rule changes to other port. If this port number is non-existent or the interface of that port is down, the node presents a fault and begins to discard packets. This fact triggers a diagnosis. This easy example serves as an introduction to this section where we explain how the semantic orchestrator module boots diagnoses. The symptoms detected use the B2D2-SDN model, shown in Section 4.3, carrying out the first necessary task for a diagnosis process. This diagnosis process performs a *Monitoring Action* to detect symptoms. If a symptom is detected, a diagnosis is started. In this worked example, this fact is implemented adding a "watcher" to the symptoms detector. For each symptom detected in the network data, a new diagnosis in the *Hypothesis Generation* phase starts. Listing 3 shows the diagnosis for the described phase.

Listing 3: Simplified Example of a Diagnosis Process.

```
@prefix b2d2-diag: <http://www.gsi.upm.es/ontologies/b2d2/diagnosis/ns#> .
@prefix b2d2-sdn: <http://www.gsi.upm.es/ontologies/b2d2/sdn/ns#> .
@prefix sdndl: <http://www.gsi.upm.es/ontologies/sdndl/ns#> .
@prefix pr-owl: <http://www.pr-owl.org/pr-owl.owl#> .
@prefix example: <http://testbed/simulation2018271454_0/> .

example:diagnosis-014 a b2d2-diag:Diagnosis ;
                b2d2-diag:hasCollectedInformation example:symptom-002 ;
                b2d2-diag:hasHypothesis example:hypotesis-03,
                    example:hypothesis-04 ;
                b2d2-diag:hasPerformedAction example:monitoringAction-002 ;
                b2d2-diag:isStartedBySymptom example:symptom-002 ;
```

```
                b2d2-diag:whenHasStarted 1518011943 ;
                b2d2-diag:whenHasFinished 1518011948 .

example:symptom-002 a b2d2-diag:Symptom ;
                b2d2-diag:collectedFrom example:openflow33 ;
                b2d2-diag:isCausalModelInput example:fd-input-001 ;
                b2d2-diag:gatheredWithAction example:monitoringAction-002 .

example:hypothesis-04 a b2d2-diag:Hypothesis ;
                b2d2-diag:hasConfidence example:prob-hyp-04 ;
                b2d2-diag:isCausalModelOutput example:bn-var-008 ;
                b2d2-diag:representsPossibleFault example:poss-fa-004 .

example:prob-hyp-04 a pr-owl:ProbAssign ;
                pr-owl:hasStateProb 0.57 .

example:poss-fa-004 a b2d2-sdn:FlowsPrioritiesModified,
                                b2d2-diag:Fault ;
                b2d2-diag:hasLocation example:openflow33 .
```

When receiving this initial set of hypothesis, the diagnosis commutes to the *Hypothesis Discrimination* phase. In this phase, the processed data is examined in search of more pieces of evidence, first in the switch, and then, events from the topology. For example, a diagnosis has been started for “openflow2", the symptom is introduced in the reasoning module and the semantic orchestrator response and release the initial report with the set of hypothesis. Then, the diagnosis module collects more symptoms from the node and the topology in a certain time window, and again, sends this data to the reasoning module.

Finally, the semantic orchestration uses this data to generate a complete report of the diagnosis process, including the final set of *hypothesis* whit an associated possible fault, the diagnosis *conclusion*, its beginning time (semantically *whenHasStarted*) and its end time (*whenHasFinished*). Also, this report includes information about all the monitoring actions collected to carry out it (in property *hasCollectedInformation*).

6 | EVALUATION

The prototype developed for experimentation purposes has been evaluated in the same testbed as the previous work[10]. We have included another machine learning algorithm which includes fast-model learning[39]. Then, we evaluate the quality of two different models for switch faults using two different algorithms.

Specifically, we have created a network simulation environment in which we use Mininet[40] to simulate an SDN network composed of OpenvSwitch nodes. As shown in Figure 11, the topology is defined by an interconnected core composed of thirteen nodes where three hosts representing servers of a datacenter are connected. Then, an access network is created at the edge of the core. This network, which is connected to the core by a set of three parallel switches, is composed of ten nodes and five hosts representing clients using services provided by servers from the datacenter.

Bayesian networks are used as causal models in the *Hypothesis Generation and Discrimination* processes. We feed the following information to the models shown in Figures 12-15: presence of flows (represented by *existence_of_flows*), changes in the number of hosts in the network (*modified_hosts*), changes in the output and in ports at any flow (*changed_output* and *changed_inport*), the detection of rules involving dropping packets with a 35020 Ethernet code (*not_dropping_lldp*), changes in any timeout (*modified_timeout*), changes in the priority order of the flows (*changed_priority*) and any change at any flow (*changed_flow*).

Two models have been designed, so-called Model 1 and Model 2. Model 1 includes all the variables related to flows in the node. In contrast, Model 2 includes only a binary variable to indicate if any flow of the node has been changed, reducing the number of variables.

Faults generated in the simulated network can be seen in Table 1. Most of these faults are injected into the network by pulling flows being implemented in a node in *XML* format from Opendaylight, modifying these *XML* files according to the fault being injected, and then pushing the *XML* file with a modified version of the flows initially pulled. We push these modified flows into the *operational* database, as mentioned in Section 5.3. Specifically, this is the procedure for injecting faults S1 and S3 to S9. In the case of the fault S2, the status of each link connecting a datacenter server to the network is switched to “down" using the Mininet Python API.

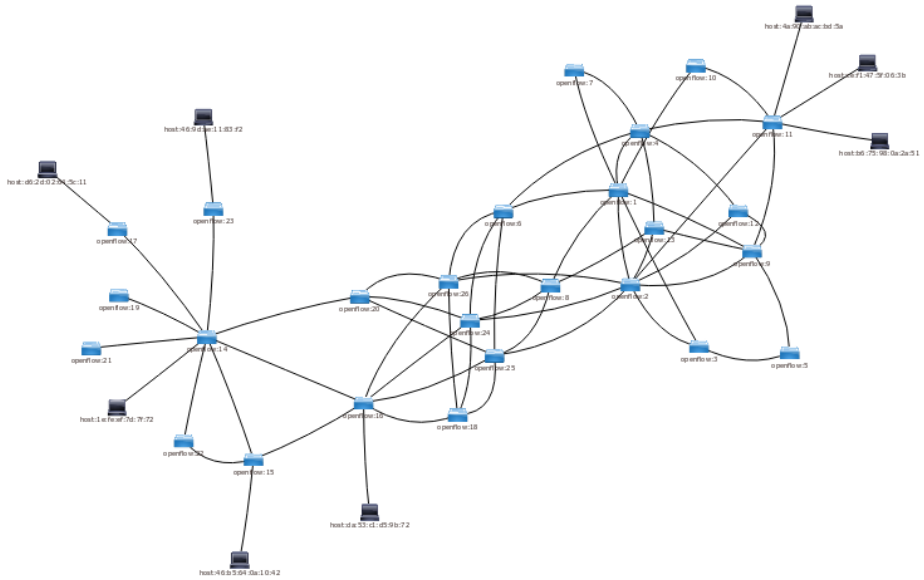

FIGURE 11 Topology of the simulated network.

Fault Type	Description
S0	No faults - OK status
S1	Shutting down a node
S2	Disconnecting a datacenter server from the network
S3	Modifying the out-port rules in a node
S4	Modifying the in-port rules in a node
S5	Adding idle-timeouts in a node
S6	Adding hard-timeouts in a node
S7	Changing flow priorities in a node
S8	Forcing a node to drop Link Layer Discovery Protocol (LLDP) packets
S9	Modifying both out-port and in-port rules in a node

TABLE 1 Fault types and descriptions.

To test and to improve the performance of the reasoning process, we have tested two different algorithms for the learning processes: Bayesian Search[41], and Chow-Liu[39] algorithms, using *Genie*[42] and *Pomegranate*[43] tools respectively. The Directed Acyclic Graphs (DAGs) generated with these algorithms are shown in Figures 12, 13, 14 and 15.

These models have been validated using a balanced dataset with 561 fault diagnosis cases from the simulated network mentioned previously. The results obtained from both models can be seen in Tables 2, 3, 4 and 5. The comparison among all these models can be seen in Table 6. The confusion matrices of each model can be seen in Tables 7, 8, 9 and 10.

Comparison between results of Model 1 (Tables 2 and 4) and Model 2 (Tables 3 and 5) shows there is a compromise between faster pre-processing and effectiveness, as can be seen in S3, S7, and S9 faults. Thus, we can conclude a more-specific model, as Model 1, shows better general results than Model 2, as shown in Table 6. Moreover, Model 1 significantly outstrips Model 2 and *Chow-Liu* algorithm slightly surpasses the *Bayesian Search* algorithm. However, the *Chow-Liu* algorithm is significantly faster; therefore, the advantage of using *Chow-Liu* over *Bayesian Search* is significant. A graphical representation of such results can be seen in Figure 16.

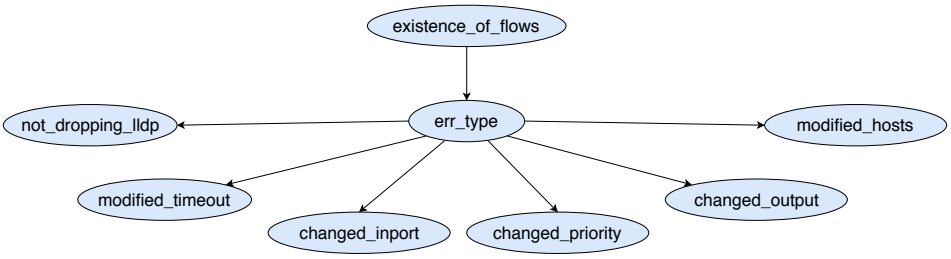

FIGURE 12 DAG for Switch Diagnosis - Model 1 using the Chow-Liu Algorithm[39].

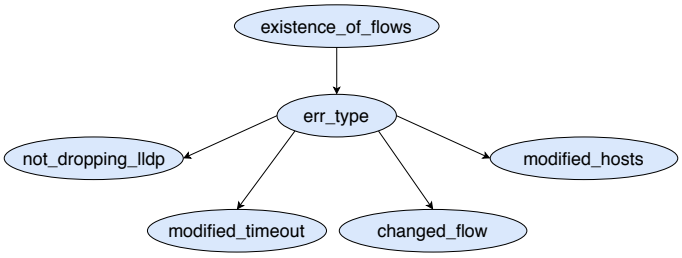

FIGURE 13 DAG for Switch Diagnosis - Model 2 using the Chow-Liu Algorithm[39].

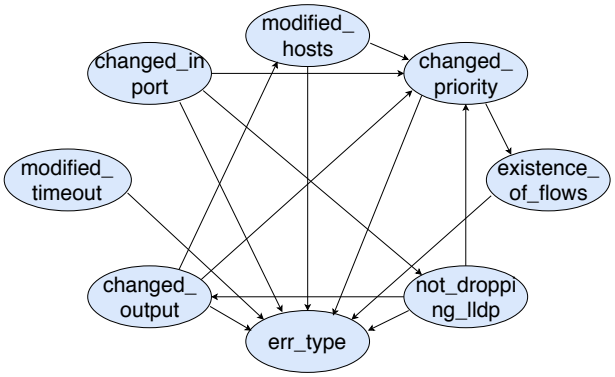

FIGURE 14 DAG for Switch Diagnosis - Model 1 using the Bayesian Search Algorithm[41].

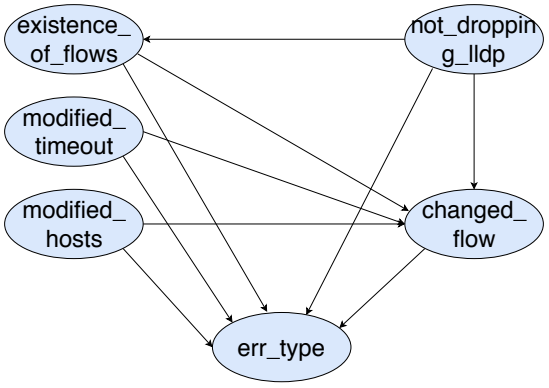

FIGURE 15 DAG for Switch Diagnosis - Model 2 using the Bayesian Search Algorithm[41].

Fault Type	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
F1-Score	0.91	0.95	0.99	0.94	0.97	0.92	0.91	0.92	0.93	0.96
Recall	0.99	0.91	1.00	0.89	0.94	0.85	1.00	0.84	0.89	0.92
Precision	0.85	1.00	0.99	1.00	1.00	1.00	0.83	1.00	0.97	1.00
Accuracy	0.95	0.99	0.99	0.99	0.99	0.99	0.98	0.99	0.99	0.99

TABLE 2 Metrics for Model 1 using the Chow-Liu Algorithm.

Fault Type	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
F1-Score	0.91	1.00	0.99	0.00	0.43	0.92	0.95	0.00	0.93	0.00
Recall	0.98	1.00	1.00	0.00	0.94	0.85	1.00	0.00	0.89	0.00
Precision	0.85	1.00	0.98	0.00	0.28	1.00	0.91	0.00	0.97	0.00
Accuracy	0.95	1.00	0.99	0.92	0.79	0.99	0.99	0.92	0.99	0.93

TABLE 3 Metrics for Model 2 using the Chow-Liu Algorithm.

Fault Type	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
F1-Score	0.71	0.84	1.00	0.88	0.93	1.00	0.87	0.94	0.92	0.85
Recall	0.59	1.00	1.00	0.89	1.00	1.00	0.90	1.00	1.00	0.86
Precision	0.89	0.73	1.00	0.88	0.87	1.00	0.85	0.88	0.84	0.84
Accuracy	0.88	0.96	1.00	0.98	0.99	1.00	0.98	0.99	0.99	0.98

TABLE 4 Metrics for Model 1 using the Bayesian Search Algorithm.

Fault Type	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
F1-Score	0.80	0.83	1.00	0.37	0.00	0.92	0.93	0.00	0.89	0.00
Recall	0.84	0.72	1.00	0.89	0.00	0.85	0.94	0.00	0.87	0.00
Precision	0.77	0.97	1.00	0.24	0.00	1.00	0.92	0.00	0.92	0.00
Accuracy	0.90	0.97	1.00	0.75	0.92	0.99	0.99	0.91	0.99	0.94

TABLE 5 Metrics for Model 2 using the Bayesian Search Algorithm.

Model	M1 - Chow-Liu	M2 - Chow-Liu	M1 - Bayesian Search	M2 - Bayesian Search
F1-Score	0.939	0.613	0.894	0.574
Recall	0.922	0.666	0.924	0.610
Precision	0.963	0.599	0.877	0.581
Accuracy	0.987	0.948	0.975	0.934

TABLE 6 Results Summary.

7 | CONCLUSIONS AND FUTURE WORK

The industry is adopting SDN technologies as a network management paradigm. SDN offers many benefits due to its agility and flexibility for monitoring and configuring telecommunication networks.

	S7	S8	S9	S1	S2	S3	S4	S5	S6	S0
S7	38	0	0	0	0	0	0	0	0	7
S8	0	34	0	0	0	0	0	0	0	4
S9	0	0	33	0	0	0	0	0	0	3
S1	0	0	0	48	0	0	0	0	5	0
S2	0	0	0	0	66	0	0	0	0	0
S3	0	0	0	0	0	42	0	0	0	5
S4	0	0	0	0	0	0	44	0	0	3
S5	0	0	0	0	0	0	0	39	5	2
S6	0	0	0	0	0	0	0	0	49	0
S0	0	1	0	0	1	0	0	0	0	132

TABLE 7 Confusion Matrix for Model 1 using the Chow-Liu Algorithm.

	S7	S8	S9	S1	S2	S3	S4	S5	S6	S0
S7	0	0	0	0	0	0	38	0	0	7
S8	0	34	0	0	0	0	0	0	0	4
S9	0	0	0	0	0	0	33	0	0	3
S1	0	0	0	53	0	0	0	0	0	0
S2	0	0	0	0	66	0	0	0	0	0
S3	0	0	0	0	0	0	42	0	0	5
S4	0	0	0	0	0	0	44	0	0	3
S5	0	0	0	0	0	0	0	39	5	2
S6	0	0	0	0	0	0	0	0	49	0
S0	0	1	0	0	1	0	0	0	0	132

TABLE 8 Confusion Matrix for Model 2 using the Chow-Liu Algorithm.

	S7	S8	S9	S1	S2	S3	S4	S5	S6	S0
S7	45	0	0	0	0	0	0	0	0	0
S8	0	38	0	0	0	0	0	0	0	0
S9	0	0	31	0	0	0	0	0	0	5
S1	0	0	0	53	0	0	0	0	0	0
S2	0	0	0	0	66	0	0	0	0	0
S3	0	0	0	0	0	42	0	0	0	5
S4	0	0	0	0	0	0	47	0	0	0
S5	0	0	0	0	0	0	0	46	0	0
S6	0	0	0	5	0	0	0	0	44	0
S0	6	7	6	15	0	6	7	0	8	79

TABLE 9 Confusion Matrix for Model 1 using the Bayesian Search Algorithm.

In this article, we propose a Semantic Data Lake architecture which combines the features of Big Data technologies and ontology-based data modeling, applying Bayesian reasoning as inference method for the Fault Diagnosis process. A semantic description language for SDN environments, called SDNDL, has been proposed. Moreover, an existing Fault Diagnosis model has been extended for our scenario, including symptoms and faults.

Finally, four causal models have been generated following different data processing criteria and learning algorithms. Their evaluation showed that the more specific model (Model 1) has better accuracy than the model in which the number of collected

	S7	S8	S9	S1	S2	S3	S4	S5	S6	S0
S7	0	0	0	0	0	38	0	0	0	7
S8	1	33	0	0	0	0	0	0	0	4
S9	0	0	0	0	0	33	0	0	0	3
S1	5	0	0	38	0	0	0	0	0	10
S2	0	0	0	0	66	0	0	0	0	0
S3	0	0	0	0	0	42	0	0	0	5
S4	0	0	0	0	0	44	0	0	0	3
S5	0	1	0	0	0	0	0	39	4	2
S6	0	2	0	1	0	0	0	0	46	0
S0	2	0	0	0	0	20	0	0	0	112

TABLE 10 Confusion Matrix for Model 2 using the Bayesian Search Algorithm.

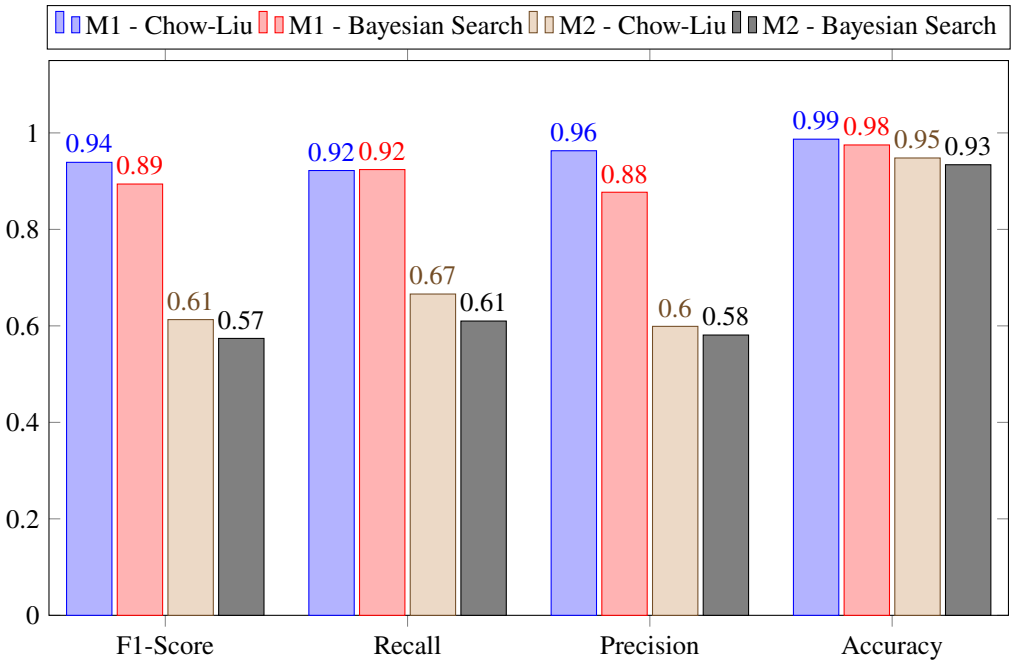

FIGURE 16 Bar diagram representing the results summary.

variables has been reduced to improve performance in the data processing phase (Model 2). It also displayed that the *Chow-Liu* algorithm shows better results in the diagnosis of network faults.

For future work, we propose to combine different diagnosis models to cope with more fault cases, not only focusing on switch faults. This can be done by further processing more data sources, such as final user applications for specific service monitoring, including probes in the network and servers or implementing testing agents which could execute specific tests when symptoms or anomalies are detected in the network. Moreover, a distributed approach can be explored to allow communication and dialogue between different network controllers. This is an interesting topic when data cannot be centralized in a unique data lake due to legal or technical restrictions.

ACKNOWLEDGMENTS

This research has been funded by Spanish Ministry of Industry, Energy and Tourism under the R&D project BayesianSDN (TSI-100102-2016-12) and the Spanish Ministry of Economy through the R&D project SEMOLA (TEC2015-68284-R). We also acknowledge the use of an academic license of the GeNIe bayesian modeler.

References

1. Alexander S. When networks hit the wall. *NetworkWorld* 2017.
2. Cisco VNI . Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021. *Cisco White Papers* 2016.
3. Cisco . The Zettabyte Era: Trends and Analysis. *Cisco White Papers* 2016.
4. Cisco . Cisco Global Cloud Index: Forecast and Methodology, 2015–2020. *Cisco White Papers* 2015.
5. Vidal A. Flexible networking hot trends at SIGCOMM 2016. *Ericsson Research Blog* 2016.
6. Baines S. Enterprises want SDN to build flexible networks. tech. rep., Orange; 2015.
7. Groene F, Isaac C, Nalinakshan H, Tagliaferro J. The software-defined carrier: How extending network virtualisation architecture into IT BSS/OSS architectures opens up transformational opportunities for telecom and cable operators. *Communications Review* 2016.
8. 2015 ACDCS. Network Transformation Survey 2015, Final Results. tech. rep., Accenture; 2015.
9. Markets and Markets . Software-Defined Networking and Network Function Virtualization Market by Component (Solution (Software (Controller, and Application Software), Physical Appliances), and Service), End-User, and Region - Global forecast to 2022. 2017.
10. Benayas F, Carrera A, Iglesias CA. Towards an autonomic Bayesian fault diagnosis service for SDN environments based on a big data infrastructure. In: IEEE. ; 2018: 7–13.
11. Tselentis G, Galis A. *Towards the Future Internet: Emerging Trends from European Research*. IOS press . 2010.
12. Song HH, Qiu L, Zhang Y. NetQuest: a flexible framework for large-scale network measurement. *IEEE/ACM Transactions on Networking (TON)* 2009; 17(1): 106–119.
13. Massie ML, Chun BN, Culler DE. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing* 2004; 30(7): 817–840.
14. Di Pietro A, Huici F, Costantini D, Niccolini S. DECON: Decentralized Coordination for Large-Scale Flow Monitoring. In: ; 2010: 1-5.
15. Laurent C, others . Autonomic network engineering for the self-managing Future Internet (AFI); Generic Autonomic Network Architecture (An Architectural Reference Model for Autonomic Networking, Cognitive Networking and Self-Management). 2013.
16. Behringer M, Pritikin M, Bjarnason S, et al. Autonomic Networking: Definitions and Design Goals. tech. rep., Internet Research Task Force; 2015.
17. Jiang S, Carpenter B, Behringer M. General Gap Analysis for Autonomic Networking. tech. rep., Internet Research Task Force; 2015.
18. Rezgui A, Alizadeh Noghani K, Taheri J. *SDN helps Big Data to become fault tolerant*: 319–336; The Institution of Engineering and Technology. 1 ed. 2018.

19. Lee C, Shin S. Fault Tolerance for Software Defined Networking in Smart Grid. In: IEEE Computer Society; 2018: 705–708.

20. Subedi TN, Nguyen KK, Cheriet M. SDN-based fault-tolerant on-demand and in-advance bandwidth reservation in data center interconnects. *International Journal of Communication Systems* 2018; 31(4): e3479.

21. Yu Y, Li X, Leng X, et al. Fault Management in Software Defined Networking: A Survey. *IEEE Communications Surveys and Tutorials* 2018.

22. Nayyer A, Sharma AK, Awasthi LK. Issues in Software-Defined Networking. In: Springer. ; 2019: 989–997.

23. McKeown N, Anderson T, Balakrishnan H, et al. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 2008; 38(2): 69–74.

24. Capone A, Cascone C, Nguyen AQ, Sanso B. Detour planning for fast and reliable failure recovery in SDN with OpenState. In: IEEE. ; 2015: 25–32.

25. Sharma S, Staessens D, Colle D, Pickavet M, Demeester P. Fast failure recovery for in-band OpenFlow networks. In: IEEE. ; 2013: 52–59.

26. Gheorghe G, Avanesov T, Palattella MR, Engel T, Popoviciu C. SDN-RADAR: Network troubleshooting combining user experience and SDN capabilities. In: IEEE. ; 2015: 1–5.

27. Tang Y, Cheng G, Xu Z, Chen F, Elmansor K, Wu Y. Automatic belief network modeling via policy inference for SDN fault localization. *Journal of Internet Services and Applications* 2016; 7(1): 1.

28. Chandrasekaran B, Benson T. Tolerating SDN application failures with LegoSDN. In: ACM. ; 2014: 22.

29. Sánchez JM, Yahia IGB, Crespi N. THESARD: On the road to resilience in software-defined networking through self-diagnosis. In: IEEE. ; 2016: 351–352.

30. Carrera Barroso Á. *Application of Agent Technology for Fault Diagnosis of Telecommunication Networks*. PhD thesis. Universidad Politécnica de Madrid, 2016.

31. Ghijsen M, Ham v. dJ, Grosso P, et al. A semantic-web approach for modeling computing infrastructures. *Computers Electrical Engineering* 2013; 39(8): 2553 - 2565. doi: https://doi.org/10.1016/j.compeleceng.2013.08.011

32. Inmon B. *Data Lake Architecture: Designing the Data Lake and Avoiding the Garbage Dump*. Technics publications . 2016.

33. Gormley C, Tong Z. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc." . 2015.

34. Jena A. Apache Jena Fuseki. *The Apache Software Foundation* 2014.

35. Costa P. *Bayesian semantics for the Semantic Web*. PhD thesis. George Mason University, 2005.

36. Benjamins R. Problem-Solving Methods for Diagnosis and their Role. *International Journal of Expert Systems: Research and Applications* 1995; 8(2): 93-120.

37. Ghijsen M, Ham v. dJ, Grosso P, Laat dC. Towards an Infrastructure Description Language for Modeling Computing Infrastructures. In: ; 2012: 207-214

38. Ham v. dJ, Dijkstra F, Łapacz R, Brown A. The network markup language (nml) a standardized network topology abstraction for inter-domain and cross-layer network applications. In: ; 2013.

39. Chow C, Liu C. Approximating discrete probability distributions with dependence trees. *IEEE transactions on Information Theory* 1968; 14(3): 462–467.

40. Kaur K, Singh J, Ghumman NS. Mininet as software defined networking testing platform. In: ; 2014: 139–42.

41. Heckerman D, Geiger D, Chickering DM. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine learning* 1995; 20(3): 197–243.

42. BayesFusion . GeNIe Modeler. Available at https://www.bayesfusion.com/; 2017.

43. Schreiber J. Pomegranate: fast and flexible probabilistic modeling in python. *Journal of Machine Learning Research* 2018; 18(164): 1–6.