UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Haikunet: An Intent Programming Language for the Software Defined Networking Paradigm

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Andrés Laurito

Directores: Hernán Melgratti y Rodrigo Castro

Buenos Aires, 2017

HAIKUNET: AN INTENT PROGRAMMING LANGUAGE FOR THE SOFTWARE DEFINED NETWORKING PARADIGM

Los lenguajes de programación orientados a *intents* están tomando un rol cada vez más importante en el paradigma de *Software Defined Networking* (SDN). A pesar de ello, la gran mayoría de estos lenguajes de programación son soluciones acopladas rígidamente a controladores SDN específicos, dificultando su reuso y portabilidad. Además, los lenguajes existentes no suelen implementar herramientas de verficación para detectar errores en los intents antes de que estos sean aplicados a la red que se pretende manipular en forma programática. En esta tesis presentamos Haikunet, un lenguaje de programación orientado a intents que implementa capacidades básicas de verificación y es agnóstico frente a controladores SDN. Esta última propiedad es lograda gracias al uso de TopoGen, una nueva herramienta genérica para traducir y generar topologías de redes desarrollada en el marco de esta tesis. TopoGen es usado para generar modelos de grafos y de simulación para la topología analizada. Sobre el modelo de grafos realizamos verificaciones de propiedades estáticas, mientras que con los modelos de simulación abrimos el camino para la verificación de propiedades dinámicas. La semántica operacional de Haikunet es presentada siguiendo el estilo de Structural Operational Semantics (SOS), esto es, un conjunto de reglas de inferencia que define de manera inductiva la relación de reducción. Estas reglas de inferencia son dadas en dos niveles: 1. En un conjunto de reglas gramáticales de grafos, que describen como cada operación del lenguaje afecta a la topología de la red y 2. Mediante reglas de inferencia, que describen el cómputo de un programa en Haikunet. En este trabajo presentamos distintos escenarios, en donde mostramos como Haikunet puede detectar errores en intents antes de que estos sean aplicados en la red. Todos los escenarios considerados fueron probados contra los controladores actualmente soportados por Haikunet: ONOS y OpenDayLight.

Keywords: Haikunet, TopologyGenerator, SDN, Programming languages, Intent-oriented programming languages, Verification system, Graph Grammars, Inference Rules, Type Systems, Simulation, DEVS.

HAIKUNET: AN INTENT PROGRAMMING LANGUAGE FOR THE SOFTWARE DEFINED NETWORKING PARADIGM

Intent-oriented programming languages are taking an important role in the Software Defined Networking (SDN) paradigm. Yet, most of these programming languages are rigidly coupled to specific SDN controllers, hindering the re-usability and portability properties of *intents*. Besides, existent intent-oriented programming languages usually don't implement a verification system that can detect errors in *intents* before these are applied to the network that is being manipulated programatically. In this thesis we present Haikunet, an intent-oriented programming language that implements basic verification capabilities and is agnostic to SDN controllers. This last property is achieved thanks to TopoGen, a new tool developed in this thesis to translate and generate network topologies programatically. TopoGen is used for generating graph and simulation models of the network being analyzed. The graph model is used for static verification of properties, while the simulation models open the path for the dynamic verification of properties. Haikunet operational semantics is formally defined in the Structural Operational Semantics (SOS) style, i.e., a set of inference rules inductively defines the reduction relation. Inference rules are given in two levels: 1. A set of graph grammars rules describe how each operation of the language changes the topology of a network, and 2. By inference rules that describe how a *Haikunet* program compute. In this work, we present different scenarios in which we show how Haikunet can detect errors in intents before these are applied to the network. Every scenario presented was tested against the supported controllers by Haikunet: ONOS and OpenDayLight.

Keywords: Haikunet, TopologyGenerator, SDN, Programming languages, Intent-oriented programming languages, Verification system, Graph Grammars, Inference Rules, Type Systems, Simulation, DEVS.

AGRADECIMIENTOS

A mamá, papá, la abu y aldi, porque me bancaron en toda la carrera universitaria, en los mejores y peores momentos, siempre incondicionales a mi lado, y porque me dieron una de las cosas que más me define: la curiosidad por el saber.

A mis amigos del colegio y la vida, Martín, George, el Negro, Lucas, Guti, Javi Vanna, Javi, Niko Cance, Niko, porque estuvieron siempre para bancarme, ayudarme, escucharme y darme fuerzas para haber podido llegar hasta acá.

A mis amigos de la facu, el Gallego, Lion, Santi Rulos, Santi Dr, Santi, Nico, Gaby, Juanma, porque hicieron de muchas materias un gran recuerdo solamente por haber cursado con ustedes, y sin duda alguna me ayudaron a llegar a este lugar.

A Rodri y Hernán, porque me siguieron enseñando luego de haber terminado de cursar, me enseñaron lo lindo que es la investigación, me bancaron en el arduo camino de lo que implica hacer una tesis, y porque simplemente sin ustedes todo este trabajo no hubiese sido posible.

A Mati, porque gran parte de este trabajo fue gracias a él, me banco en Europa en todo momento, me dio grandes consejos y excelentes momentos vividos.

A mis profesores del colegio, porque son uno de los pilares más importantes de mi educación académica.

A los profes y ayudantes de la facu, porque su gran laburo inspiró, fomentó y ayudó a que hoy este acá.

A aquellos con los que hoy ya no comparto tantos momentos, pero fueron muy importantes para que pueda llegar a este lugar.

A todos ustedes, muchas gracias por haber ayudado a este momento tan importante en mi vida.

A mi familia y amigos, que me dieron la inspiración para que este trabajo sea hoy una realidad.

SUMMARY

1. INTRODUCTION

Several requirements of distributed systems usually relate to properties of the underlying network infrastructure. The growth of cloud computing has magnified such connection. Since software requirements change, networks may be forced to do the same. Remarkably, changing the network infrastructure may be difficult and time-consuming. Software Defined Networking (SDN) aims at facilitating this task. The main idea behind SDN is to move the logic from the *switching devices* to software running on centralized computers, which are called *Controllers*. In the SDN paradigm, *switching devices* (that live in the so called *SDN Data Plane*) become simple forwarding elements while *controllers* become the orchestrators of the network. *Controllers* help with the task of reconfiguring a network infrastructure by providing a software-oriented programmable interface. Developing software in *controllers* endows opportunities to software developers to implement custom solutions for known network requirements (such as Border Gateway Protocol (BGP) [24], Spanning Tree Protocol (STP) [35] or load balancing [20]), and to provide new network services (such as topology discovery or specification of *intents* on the network).

Intents are declarative descriptions of **network requirements** that state *what* is required from a network instead of *how* that requirement is achieved. For example, an intent may describe a requirement for connecting a user to a database cluster. In this case, the *controller* has the responsibility of accomplishing such requirement, with the flexibility of deciding on how to achieve it. When *intents* are applied to the network by a *controller* they may, e.g.,: 1. force the *controller* to load forwarding rules into the *switching devices*, 2. create monitor applications to detect salient conditions specified in the requirements, 3. create or delete *switching devices* (when working in a virtual network).
Intents can be embedded as components of *controllers*. In this way, intents and *controllers* are tightly coupled and changes in an intent force the re-compilation of the controller's source code, which affects availability. As an alternative, intent-oriented programming languages (IOPL) disassociates *intents* from *controllers*, making possible to create new *intents* without affecting *controllers* availability. Besides, the separation of *intents* from *controllers* makes them portable and facilitates their distribution. Several IOPLs with different capabilities exist nowadays. Some examples are NEMO [48], Frenetic [17] and NetKAT [3].

1.1. Motivations

Despite IOPL approach has advantages, it also presents some limitations. IOPL are usually developed to run in specific vendor *controllers*, which introduces unnecessary coupling between *intents* and *controllers*. This problem arises in part because there is still no standard for *controller's* API. Consequently, *intents* need to be rewritten when using different *controllers*. For instance, NEMO relies on OpenDayLight [28], Frenetic relies on NOX [18] and NetKAT relies on the OpenFlow *controller* [27].

Another important shortcoming is that existing IOPLs do not provide mechanisms for detecting errors before *intents* are applied to the network. As a consequence, *intents* are first tested in simulated networks such as Mininet [15], before they are applied to the real network. Testing *intents* over simulated networks forces developers to maintain the simulated environment, which is a time-consuming and error prone task.

Finally, IOPL are usually implemented without mathematical formalism, e.g. NEMO, Nettle [45] (a functional reactive programming language for OpenFlow Networks), Procera [46] (a language for high-level reactive network control), FatTire [36] (a declarative fault tolerance language) and Flog [21] (a logic programming language for SDNs). Few are the IOPL that present such mathematical formalism. An example of such a language is NetKAT [3]. Giving mathematical foundations create opportunities for known software-engineering solutions to be applied to *intents*, for instance applying static/dynamic analysis techniques.

1.2. Contributions

This thesis introduces *Haikunet*, which is an intent-oriented programming language (developed in Ruby), that is agnostic to a specific vendor controller and which provides a basic semantic checker. The semantic checker is applied to the *intent* before the *intent* is applied to the underlying *controller*. When an error is detected, the application of the *intent* is stopped and an error is raised. The different actions involved when programming an *intent* in *Haikunet* are detailed in Figure 1.1.

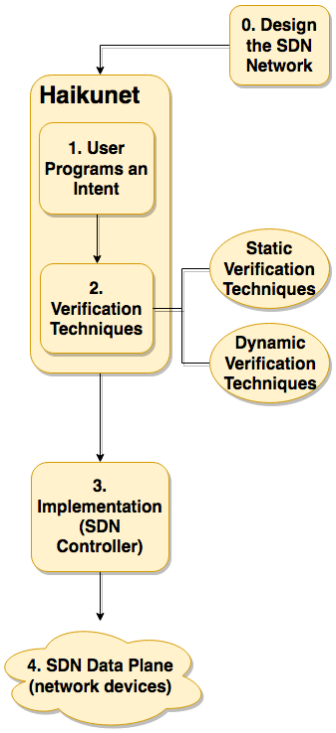


Fig. 1.1: Actions Involved When Programming an Intent with Haikunet

Figure 1.1 depicts the flow of designing an *intent* in *Haikunet*. Initially, a *SDN Network* (where the *intent* will be applied) needs to be designed. Given the *SDN Network*, the user develops an *intent* in *Haikunet*. This *intent* is verified by the semantic checker by applying verification techniques. The verification techniques applied to the *intent* concern *static* and *dynamic* verifications. These are further explained later in this section. If the *intent* is error-free, it is received and implemented by the *SDN Controller*, which applies the *intent* to the *SDN Data Plane*. Otherwise, an error is raised and the execution is stopped. *Haikunet* implements a common framework for intent-oriented programming languages proposed in the Internet Engineering Task Force (IETF) [14].

A key component of *Haikunet* is *TopoGen* [26], which is an SDN-oriented topology generator tool designed to automatically create network models from serializable network descriptions. *TopoGen* can obtain serialize network descriptions from different *controllers* APIs and use that information to build a graph that represents the network topology. This graph can be serialized by *TopoGen* to different output formats. *TopoGen* comes equipped with its own object-oriented network topology description language, which is called NTM and it is based on Ruby. The semantic checker of *Haikunet* uses *TopoGen* to generate a description of the network over which the intent is going to be applied. This description of the network is later used by the semantic checker for performing verification techniques.

TopoGen is independent from *Haikunet* and can be used in different scenarios to retrieve a network model and generates different network representations. For instance, in this thesis we present the use of *TopoGen* in a real-world project in the context of the Trigger and Data Acquisition (TDAQ) network of the ATLAS Experiment at CERN [5].

Haikunet semantic checker was designed for detecting errors of two kinds: static and dynamic.

Static errors refer to problems related with the topology (i.e., the graph) of the network over which the intent will be applied. Examples of such errors are references to inexistent nodes or paths between nodes that do not exist on the network. Dynamic errors are related instead with the actual traffic over the network. An example of this kind of error is when the buffers of some specific routers drop more that 40 % of packets, and a maximum threshold of 30 % was set.

We handle static errors by providing a semantic checker of *Haikunet intents*. In order to do that, we formally defined the operational semantics of *Haikunet* following the Structural Operational Semantics (SOS) [33] style,, i.e., a set of inference rules inductively define the reduction relation. Inference rules are given in two levels: 1. a set of graph grammars rules describes how each operation of the language changes the topology of a network, and 2. a set of inference rules describe how a *Haikunet* program computes.

The detection of dynamic errors uses *TopoGen* to automatically generate DEVS simulation models from the network topology. These simulation models are then run in PowerDEVS [7], and the results of the simulations are rendered by Scilab [10].

We illustrate the error detection capabilities of *Haikunet* with several scenarios. Chosen scenarios were tested in the currently supported targeted *controllers* of *Haikunet*: *ONOS* ([6]) and *OpenDayLight* ([28]).

The relation among the different components of the architecture are summarized in Figure 1.2

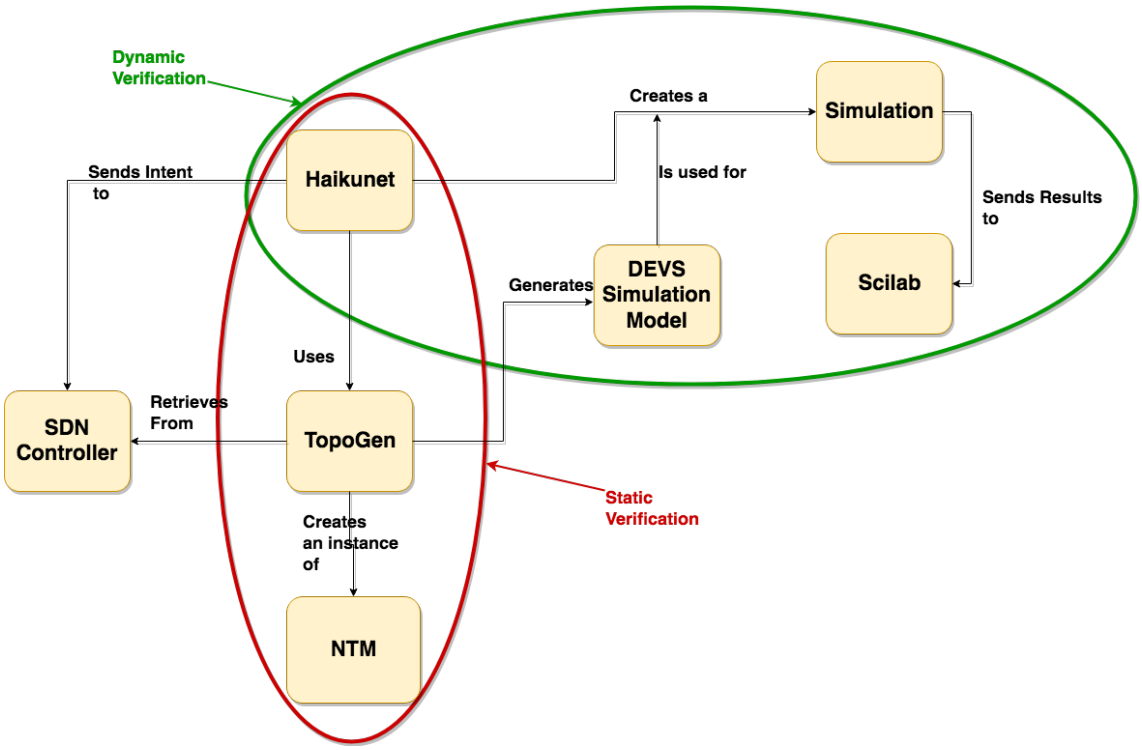


Fig. 1.2: Relation Among the Different Components of the Architecture.

1.3. Organization

This work is organized as follows: Firstly, concepts used along the thesis are presented in Chapter 2. Then, in Chapter 3 we provide a summary of main related works on network simulators, intents-oriented programming languages and standards for description languages of network topologies. Chapter 4 introduces *TopoGen* and Chapter 5 presents *Haikunet*. In Chapter 6 we draw some final conclusion and discuss about possible future works.

2. BACKGROUND

2.1. Software Defined Networking

Software Defined Networking (SDN) is an emerging architectural approach for computer networks where control logic is taken away from *switching devices* and moved up to centralized software running in *controller devices* [25].

In this new architecture, *switching devices* are lumped into much simpler packet forwarding elements operating at the so-called *Data Plane*. Controllers decide on and set up forwarding rules for each connected *switching device*, in an effort to comply with the overall quality of service requirements for the entire network. Controllers communicate with *switching devices* by using protocols that allow the loading of forwarding rules. The nowadays most common used protocol is OpenFlow [27]. OpenFlow is an Ethernet-based protocol that provides a standardized interface for adding and removing forwarding rules in *switching devices*.

Controllers operate at the so-called *Control Plane* and concentrate most of the network service logic (e.g. monitoring, packet forwarding decisions, network topology discovery, *intents* manipulation, rule conflicts) Implementations of centralized SDN controllers (e.g. ONOS [6] , OpenDayLight [28], Nox [18], Floodlight [29]) provide different functionalities through APIs. The interfaces provided by controllers that are used by devices living in the *Data Plane* layer are called the *Southbound Interfaces*.

In turn, controllers provide different networking services through an API. Typical services include monitoring, the definition of packet forwarding policies, and topology discovery. Such API is consumed by applications and services that live in the so called *Application Plane* ([19]). Generally, such applications address problems that do not specifically fall within the networking domain, but require networking services to provide other ones, concerning, e.g., databases, web-browsers and instant messaging services. These interfaces provided by controllers constitute the *Northbound Interfaces*. Applications require certain functionalities and capabilities of the underlying network but do not define the way in which those requirements are satisfied. *Intents* and *intents* programming languages has been proposed to address such problematic (more details in 2.1.1). Figure 2.1 depicts the different layers and interfaces that were mentioned above.

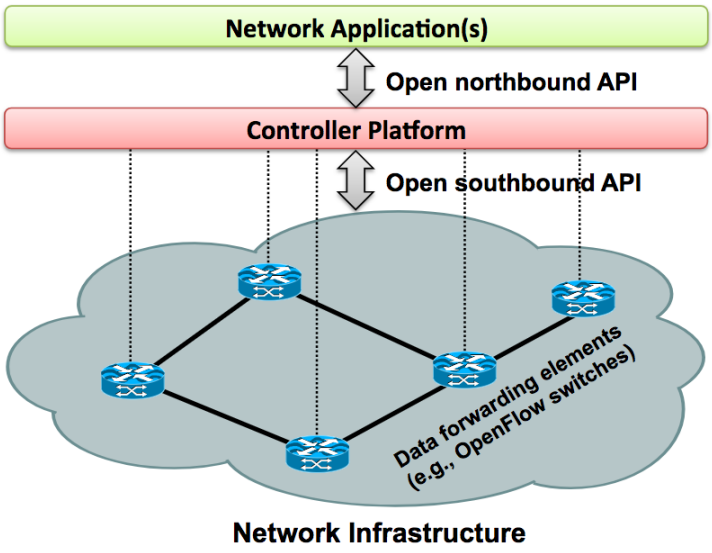


Fig. 2.1: SDN Layers (borrowed from [25])

2.1.1. Intents

Intents define high-level requirements for a network, and describe *what* is required from the network instead of *how* that requirement is achieved. The *controllers* transform *intents* into operations that affect the behaviour of the network, such as loading forwarding rules in *switching devices* or creating monitor applications. *Intents* are meant to be used by network users instead of network operators. As a consequence, *intents* are usually developed in the *Application plane* and *controllers northbound interfaces* commonly implement *intents* services, such as *intent* creation/deletion and conflict resolution. These services are generally used by applications and Intent-Oriented Programming Languages (IOPL).

Some examples of use case scenarios for *intents* are: 1. applying load balancing policies over the network traffic, 2. denying traffic coming from a user to a specific cluster (notice that *user* is a high-level abstraction from a network perspective), 3. allowing a link to be used only when some particular condition is fulfilled, e.g., to allow traffic over an AWS Direct Connect link ([44]) only when the buffers of some specific routers drop more that 40 % (usage of this kind of links usually implies huge costs).

Intents can be programmed inside a controller as part of its core, or defined in a specific language outside the controller. In order to accommodate changes in the intents, the first approach may require to stop the execution of the controller to deploy the required changes in the source-code of the controller. This approach creates undesirable down-time and a coupling between *intents* and controllers. As a consequence, IOPL have arose as an alternative approach to deal with intents. IOPLs use the services provided by the *Northbound Interface* of the *controllers* for supporting the development of *intents* outside controllers. Developing *intents* outside controllers gives important additional characteristics to them. Some worth mentioning features are: 1. declarativity, which allows intent to be used as as Software Level Agreements (SLA); 2 portability, because *intents* are not coupled to a specific technology; 3. compositionality, since intents can be used with other *intents* to create more complex requirements; 4. explicit documentation of network requirements. Different *intent* programming languages are presented and discussed in Chapter 3.

2.2. Graph Grammars

Graph Grammars ([38]) provide a system in which transformations on graphs can be modeled in a mathematically way. The main component of a graph grammar is a finite set of productions; a production is, most commonly, a triple (M,D,E) where M and D are graphs and E is some embedding mechanism. Such a production can be applied to a ("host") graph H whenever there is an occurrence of M in H . It is applied by deleting this occurrence of M from H , replacing it by (an isomorphic copy of) D, and finally using the embedding mechanism E to attach D to the remainder H^- of H. There are two approaches to graph grammars: the gluing approach (or algebraic approach, used in this thesis) and the connecting approach.

The algebraic approach to graph transformation is based on the concept of gluing of graphs, modeled by pushouts in suitable categories of graphs and graph morphisms. This allows one not only to give an explicit algebraic or set theoretical description of the constructions, but also to use concepts and results from category theory in order to build up a rich theory and to give elegant proofs even in complex situations. There exists two possible algebraic approaches: The double-pushout (DPO) approach and the single- pushout (SPO) approach. In this thesis, we use the DPO approach extended with negative application conditions.

We define a double-pushout transformation rule ([16]) in a category C as a pair of injective morphisms $P = (L \leftarrow_l K \rightarrow_r R)$ in C, and it can be applied to an object G when there exists a morphism $m : L \rightarrow G$ and a diagram:

$$\begin{array}{ccccc} L & \xleftarrow[l]{} & K & \xrightarrow{r} & R \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle g'} \\ G & \xleftarrow[l']{} & D & \xrightarrow{r'} & H \end{array}$$

such that both squares are pushout squares. In other words, when there exists a, pushout complement D for l and m (that is, an object D and two morphisms $g : K \to D$ and $l' : D \to G$ such that G, together with m and l', is the pushout of l and g), and then a pushout of r and g. In this case, the object H is said to be derived from G by the application of rule P through morphism m.

An example of a double-pushout transformation rule is illustrated in Figure 2.2. This rule models the different states of a SDN service when a process is being request. For this case example, we assume that a service can process one request at a time, and this request is processed by executing a new thread.

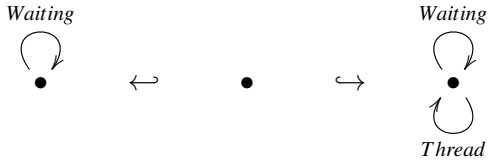


Fig. 2.2: SDN Service States when Receiving a Request

In the previous rule, the service is represented by a bullet. Initially, the service is in a *Waiting* state (this is depicted by the left-hand side graph). When a request is received, the service does not accept any new request until the current one has been processed. This is denoted by removing the *Waiting* arrow from the service (depicted by the graph in the middle). When the service process the received request, its state changes to accept new requests. This is illustrated by adding the *Waiting* arrow again, and by adding a *Thread* arrow (right-hand side graph).

The previous rule can be used in the example given in Figure 2.3. This figure illustrates two running services in a controller.

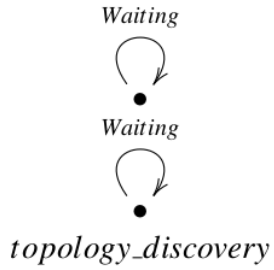


Fig. 2.3: Two Services in a SDN Controller

The scenario illustrated in Figure 2.4 shows how the previous rule can be applied in the mentioned controller. In this example, service *topology_discovery* receives a request, and the graph grammar rule is applied over the service state.

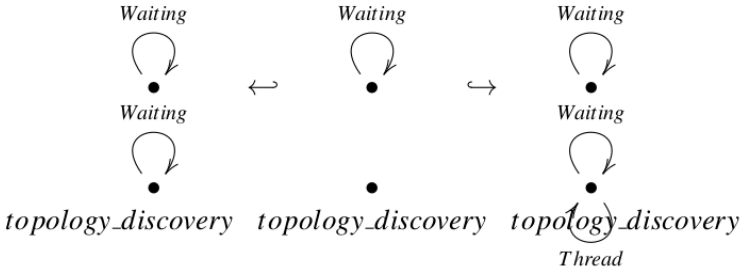


Fig. 2.4: States of Topology Discovery Service While Managing a Request

In the example above we can see how a double-pushout rule is applied to a showcase scenario. First, both services in the controller are waiting for requests (left-hand side graph). When a request is received in service *topology_discovery*, the *Waiting* arrow is removed indicating that no longer requests can be managed (middle side graph). When service *topology_discovery* executes a new thread for managing the request, it can receives new requests. This last state is illustrated with the right-hand side graph.

Another graph grammar property used in this thesis concerns negative application conditions. The idea behind negative application conditions is to have constraints in the left-hand side graph of a rule. These constraints represent forbidden structures. We introduce below an example where negative application conditions are used for describing a state in a controller.

We want to illustrate that new services can be created in a controller, but we want to specify that we can only have a unique service. Graph grammar rule depicted in figure 2.5 illustrates how this example can be handled with negative application conditions.

Fig. 2.5: Creating duplicated services is not allowed

Figure 2.5 shows a rule that states that a SDN controller can only have a unique *topology discovery* service. The negative application conditions in Figure 2.5 is represented by crossing-out the forbidden structure (in this case, the service *topology discovery*). This rule states that, if a service named *topology_discovery* does not exist in the graph, then this service can be created.

2.3. Formal Model-Based Simulation

DEVS (Discrete EVent System specification [49]) is a formal approach for the modular and hierarchical modelling of discrete systems. There are two types of models in DEVS formalism: 1. *Atomic models*, which represent indivisible building blocks of a model and 2. *Coupled models*, which can be coupled with other submodels (either atomic or coupled).

Models in a DEVS system interact by means of events sent through input and output ports. Models posses implementation behaviours for the different types of events. When an event is received by a model, the associated behaviour is triggered. These triggered behaviours can alter the internal state of a submodel, and/or create new events that are later sent to models connected to this one.

Events can be categorized as internal, whenever the event is originated from an internal behaviour of the submodel, or external, when the event is originated from an external source (can be another submodel, a timer, etc).

In formal terms, an *atomic submodel* is defined a by the tuple: $A = \{S, X, Y, \delta_{int}, \delta_{ext}, ta, \lambda\}$, where S is the set of internal states, X is the set of accepted external events, and Y is the set of available outputs. The behaviour of an atomic submodel is defined by the following functions:

$ta(s) : S \rightarrow R_0^+$ is the lifetime of each state $s \in S$. After $ta(s)$ units of time an internal state transition $\delta_{int} : S \rightarrow S$ is triggered (assuming that no external input event has arrived in the meantime).

δ_{int} models the transition from one state to the next sequential state. δ_{int} is the transition function of a deterministic finite state automaton.

$\delta_{ext}(s, e, x) : S \times X \times R_0^+ \rightarrow S$ models state transitions caused by external event, i.e., transitions that are triggered when an input event e arrives at time t and $0 \leq e < ta(s)$.

$\lambda(s) : S \rightarrow Y$ is the output function which defines the output events to be sent when an internal transition is triggered.

An external transition is triggered every time an input in X is received. This change is performed instantaneously. On the other hand, the output function is executed when ta has elapsed since last event. Simultaneously, an internal transition is produced.

A *coupled model* interconnects components together through their input and output ports and is described as a tuple: $C = \{X, Y, D, EIC, EOC, IC, Select\}$, where: X and Y are the sets of input and output events respectively, D is the set of components names, IC is the set of internal couplings among members of D, EIC is the coupling relation among external events (set of couplings between external input ports and internal components) and EOC is the external output coupling relation. Select is a tie-breaking function to assign execution priorities when several internal or external transition functions are scheduled for the same simulation time. DEVS formalism is closed under coupling, i.e., any hierarchical coupling of DEVS models defines an equivalent atomic DEVS model.

Being a formal specification, DEVS offers the capability of defining simulation models in an unambiguous, systematic and programmatic way. We shall profit from this when querying an SDN controller (through its topology discovery service) to retrieve a network description and then produce an equivalent simulation model.

2.3.1. DEVS–Based Network Simulation With PowerDEVS

PowerDEVS [7] is a discrete event simulator that implements DEVS mathematical formalism [49] capable of representing any type of discrete system and approximating continuous systems with controlled accuracy. PowerDEVS provides a graphical interface to compose DEVS models via hierarchical block diagrams. Besides, it provides a model library specific to computer network simulation [12].

In PowerDEVS systems can be built by composing graphically pre-developed units of behaviour (*atomic models*) and structures (*coupled models*) from a model library (e.g. routers, switches, links, generators, etc.) and interconnecting them through input/output ports. In DEVS, structure and behaviour are kept under strict separation. The interconnection of several atomic and/or coupled models creates the *coupling information*.

For this thesis we adopt the DEVS formalism and the PowerDEVS tool to deal with the systematic building and execution of simulation models.

3. RELATED WORK

In this section we present different studies performed over intent-oriented programming languages (see Section 3.1), languages adopted for network topologies (detailed in Section 3.2) and network simulators (further explained in Section 3.3). Each section gives a state-of-the-art in the topic, explains different known problems, and propose different solutions by using the tools introduced in this thesis.

3.1. Intent-Oriented Programming Languages

Intent-Oriented programming languages (IOPL) are either frameworks or domain specific languages (DSL). These languages are usually applied to controllers through an API. An example of a programming language implemented by using the *Northbound interface* ([32]) is NEMO [48]. NEMO is an IOPL developed for OpenDayLight controllers. NEMO allows a user to specify an intent in a DSL, and then uses features available in OpenDayLight to create an intent into a controller. As a consequence, intents written in NEMO only work with OpenDayLight controller.

Another example of an IOPL is Frenetic [17]. Frenetic can be used in both OCaml and Python for specifying actions over a network form a high-level perspective, e.g., create a repeater, perform a topology discovery on the given network, make the switching devices to drop all packets received. Frenetic is implemented to be used together with OpenFlow controllers, and therefore is attached to the latest one. Frenetic provides a rigorous mathematic foundation that documents the language, and implements an API that can be used by either applications or other programming languages via JSON messages.

NetKAT [3] is an IOPL targeted to the creation of ad-hoc languages in the SDN domain. NetKAT presents solid mathematical foundations to solve SDN problems scenarios, and is thought to be used in collaboration with an OpenFlow controller.

A mathematical study that has not been performed yet over IOPL is the definition of the operational semantics of a language using SOS. Defining the reduction relation of a language by giving a set of inference rules may allow new software-engineering techniques to be applied to IOPL. For example, static/dynamic verification techniques can use the mentioned inference rules to validate intents before these are applied to the network.

Unfortunately in the nowadays SDN realm, there is no standardization for controller's API. Consequently controllers implement different ways of exposing intent services, and therefore intents need to be redefined when applied to different controllers.

This thesis introduces Haikunet, which is an intent-oriented programming language (developed in Ruby), decoupled from controllers and oriented to be extended for supporting new targeted controllers. The extensibility property of targeted controllers is achieved thanks to TopoGen. Currently supported targeted controllers are ONOS and OpenDayLight. Haikunet is presented in Chapter 5.

3.2. Languages and standards for network topologies

Description languages for network topologies are used in different scenarios, e.g., for simulation purposes (as explained in Chapter 4), for describing complex properties as remarked in [23] or for defining virtual networks as in [15].

Describing network topologies in Domain Specific Languages (DSL) that do not depend on specific applications is an important feature. Having DSL allows the description of a network topology to be decoupled of a specific technology. This feature grants multiple applications to access to the same topology without the need of replicating information. There are several description languages proposed in the literature for describing networks, for example NED [43], VXDL [23] and NDL [40]. A problem arises when an application wants to retrieve information from different

topologies sources, e.g. Haikunet wants to retrieve the network topology from both ONOS and OpenDayLight controllers. The problem is that languages formats are different. This provokes to implement specific solutions to retrieve topology and to translate it to a convenient format. This process is then repeated every time a new description language is used or when the application consumer is changed. The mentioned approach is time-consuming and is not code-reuse oriented.

In this work we propose a solution for this problem by using TopoGen. The proposed solution is described in Chapter 4. Besides, we introduce a new framework, called NTM, for the description of network topologies. NTM is an object-oriented description language (based on Ruby). NTM was implemented for describing specific entities in the network topology, such as flows or paths.

3.3. Network Simulators

There are several network simulators available for both commercial and academic use [47]. They vary in several aspects: the adopted discrete-event techniques and principles (sequential or parallel, replication- or decomposition-based, CPU- or GPU-based) [30]; the provided library of reusable models, and the software interfaces that assist the modeling activity (e.g. to define a network topology).

In some simulation packages network model behaviour and model topology are defined intermingled in the code (e.g. NS-3 [11]). While this allows for great flexibility, the code can soon become too complex to understand, debug and maintain. A number of simulation tools (e.g. OPNET [13], OMNET++ [41]) provide graphical editors, which allow for an easy and compact understanding of the network topology, separating topology from model behaviour.

Nevertheless, defining a topology graphically can soon become inflexible for mid- to large-sized topologies (adding thousands of nodes with drag and drop methods can be very tedious and time-consuming). To address this issue some tools combine graphical editors with domain-specific languages (e.g. OMNET++) making it possible to parametrize the number of nodes and use programming-like structures to describe regular interconnect structures. This approach is efficient to describe large, mostly regular, topologies, but presents some limitations: 1. The modeler learns a description language that is specific to a single simulation tool, 2. A new topology always needs to be created from scratch, and 3. When dealing with an existing network, there is no guarantee that a network description accurately represents the real system.

In this thesis we present *TopoGen* to mitigate these problems by accommodating all the aforementioned methods under a common architecture to define/transform network topologies: either using a graphical editor (when available), programming code (when desired), or using automatic data retrieval (e.g. from SDN controllers, if needed).

4. TOPOGEN

4.1. Introduction

In this section we present TopoGen, a general purpose tool for topology serialization in the SDN paradigm.

This chapter is organized as follows: First, motivations for developing TopoGen are presented. Then, implementation details are given. Afterwards, the Network Topology Model, a Ruby framework for describing network topologies within the context of TopoGen is introduced. A brief tutorial on how to install and use TopoGen is provided, and then details on how TopoGen was used in a real scenario are given.

This section concludes by listing left-out features that would be useful to have in TopoGen.

4.2. Motivations

SDN technology offers unprecedented capabilities to reconfigure large network topologies automatically and programmatically without the intervention of human operators. These topologies can then be retrieved from many SDN Controllers by use of the Controller's API. To obtain programatically a serialized definition of a network topology is a major advantage. For instance, a graph representation can be built upon a serialized network. This graph can then be used for studying properties on the network such as assortative vs. disassortative, coefficient of clustering, K-Core decomposition, among others.

Automatic graph construction to represent a network topology allows other disciplines to use the network in a programatically way. For instance, simulation models for network systems can be built based on real, even changing topologies, where modifications on the real system imply the need for updating the simulation model accordingly. The standard practice is to upgrade topology descriptions manually for a given modeling and simulation tool of choice. Such manual changes can get considerably time-consuming and error-prone, in particular for medium to large-sized networks.

On another topic, a graph representation of the network topology structure can enable the use of software engineering validation and verification techniques. For instance, an operational Graph Grammar semantics can be built upon the graph to study network behaviours. Having an operational semantics over a network allows the implementation of semantic checkers. A semantic checker can be used in the context of an intent-oriented programming language to detect possible undesired behaviours over the network during the execution of an intent.

Unfortunately, there is no current standardization neither for the SDN controllers' API calls nor network serialization structure. This implies that when a network topology is retrieved from a different controller, new network serialization and API calls must be adopted. This can involve understanding new APIs, developing/modifying software and troubleshooting an unknown controller, among other tasks.

In this section we introduce TopoGen, an SDN-oriented topology generator tool designed to automatically create models of network topologies based on parseable network descriptions. Such descriptions rely on an intermediate topology abstraction that can be a) generated automatically, b) programmed manually from scratch or c) a mix of both options, i.e. translated automatically first, and tailored via programming later. TopoGen can retrieve network topologies from different sources, in particular from SDN controllers such as ONOS [6] and OpenDayLight [28] implementations, and can generate different targeted outputs, notably Ruby-based network topology structures used by Haikunet and DEVS-based simulators used by PowerDEVS [7].

We show an application of TopoGen to a real-world network design project in the context of the Trigger and Data Acquisition (TDAQ) network of the ATLAS Experiment at CERN [5]. A new network layer is added to a preexisting infrastructure, comprising approximately 120 nodes and

240 high speed links. TopoGen is first used to retrieve a candidate network topology prototyped by network engineers in Mininet [15] connected to an ONOS SDN controller. Then, the topology is augmented with additional nodes to provide a more exhaustive representation of a future version of the network whose performance is studied.

4.2.1. Motivating case study: Designing the new FELIX network at CERN

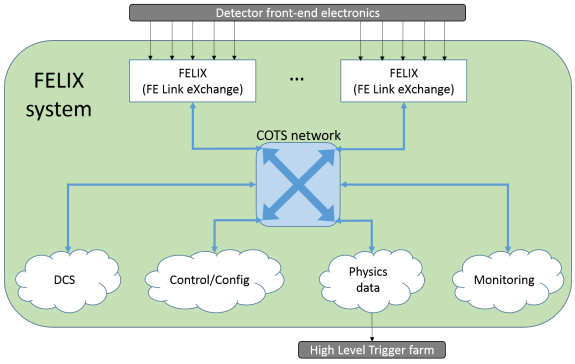


Fig. 4.1: FELIX system components.

The ATLAS experiment at CERN hosts one of the four detectors at the Large Hadron Collider (LHC) where bunches of particles collide every 25 ns. Currently, the ATLAS detector generates information at about 80 TB/s which needs to be filtered an then permanently stored for offline analysis. The TDAQ layered system reduces a 40 MHz collision event rate down to 1 kHz of useful information by analyzing events in real time. A first-level trigger (L1) uses custom electronics, filtering events down to roughly 100 kHz. L1-accepted events are temporarily transfered over custom optical point- to-point fibers to 100 Read-Out System (ROS) server nodes. The High Level Trigger (HLT) accesses events stored in the ROS to further filter the data by running selection algorithms on approximately 2000 server nodes interconnected with 1 Gbps and 10 Gbps Ethernet links.

For 2025 the ATLAS experiment is planning a full deployment of the new Front-End LInk eXchange (FELIX) system (Anderson et al. 2015), shown in Figure 1, that aims at interfacing between detector electronics and the TDAQ system.

FELIX is meant to replace the custom point-to-point connections with a Commercial-Off-The-Shelf (COTS) network technology (e.g. Ethernet, Infiniband, Omnipath). FELIX servers will act as routers between 24/48 detector's serial links and 2/4 standard 40/100 Gbps links. FELIX servers will communicate with a smaller set of commercial servers, known as Software ReadOut Drivers (SW ROD) used for data collection and processing of physics data. In addition, different components need to connect to the FELIX servers. For example, the Detector Control System (DCS) monitors and controls the detector front-end electronics while the Control & Configuration system sets up and manages data acquisition applications. The FELIX project is planned to be implemented in two phases. In 2018-2019 some detector hardware will be moved to this new schema (approx. 68 FELIX and 44 SW ROD servers will be installed). A complete migration of the remaining hardware is planned for 2025.

Part of the efforts described above consists of designing and implementing a network that can meet the future demands of the system (more stringent high-availability, high-throughput, low-latency, redundancy, etc.)

In this context, Dataflow modeling and simulation methodology can support the design of the future network, aiding the decision process to select technologies, topologies, node distributions, etc. Yet, the generation of many possible scenarios to be simulated and evaluated is currently a manual process which is time-consuming, error-prone and does not provide an automated update procedure.

4.3. Implementation

In this section we describe the overall architecture of TopoGen. Illustrative examples are given by using the ONOS SDN controller as a sample source for topology information, and the PowerDEVS toolkit as a sample target for simulation.

4.3.1. Architecture

Figure 4.2 details the architectural module viewtype of TopoGen. Its main components are:

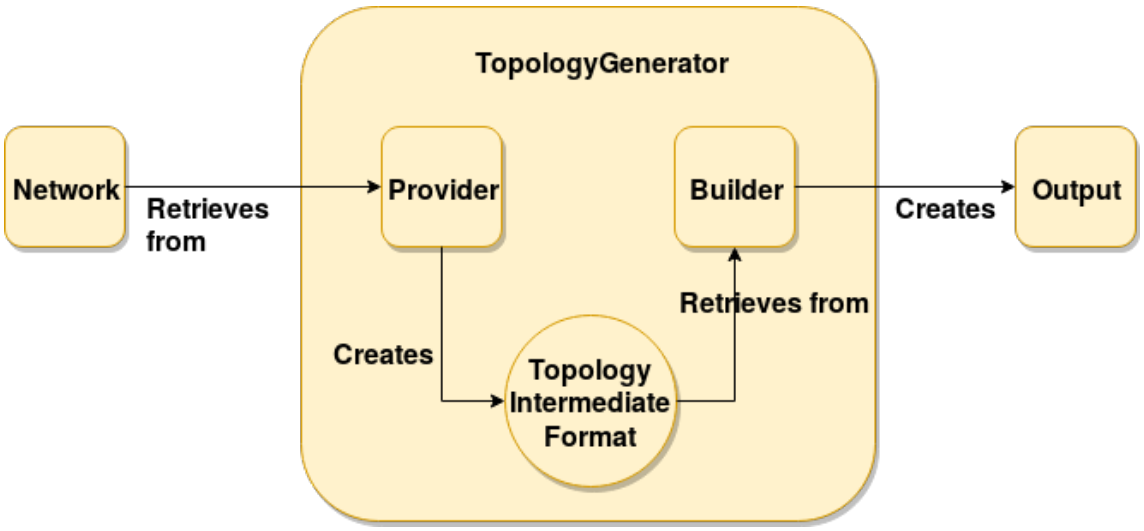


Fig. 4.2: TopoGen abstract architectural view

- **Network**: A network description to be loaded, modified or translated. It can be either a real network (e.g. one described by an SDN controller) or a virtual one described by some description language.
- **Provider**: A component that handles the interaction with the *Network* component. A *Provider* retrieves (parses) network information (such as nodes, resources, connections) and translates it into a common *Topology Intermediate Format*. Every *Provider* component is specialized to the *Network* component to be accessed. Built-in providers developed in this work are discussed in section 4.3.2.
- **Topology Intermediate Format (TIF)**: The internal in-memory representation of a network topology, written by any *Provider* and read by any *Builder*. The main goal of a *TIF* is to serve as an internal abstraction layer that permits orchestration of different *Builders* and *Providers* in a flexible way. For implementation purposes, *TIF* is created as an instance of the *Topology* class described in 4.4.
- **Builder**: A component that parses a TIF and serializes it according to a desired output. A *Builder* component is specialized to the *Output* component to be generated. Built-in builders developed for this work are discussed in section 4.3.2.
- **Output**: An output format that some *Builder* must comply with in order to perform a translation from the TIF format. The *Output* can consist of a single file or a set of files, depending on the requirements of the software tool that will ultimately consume them (in the scope of this work it will consist of a simulation software tool)

4.3.2. Built-in Providers and Builders

Figure 4.3 shows a more detailed component architecture of TopoGen developed for this work. It shows different *Builder* and *Provider* implementations according to the requirements of the

expected *Network* and *Output*. The implemented components are:

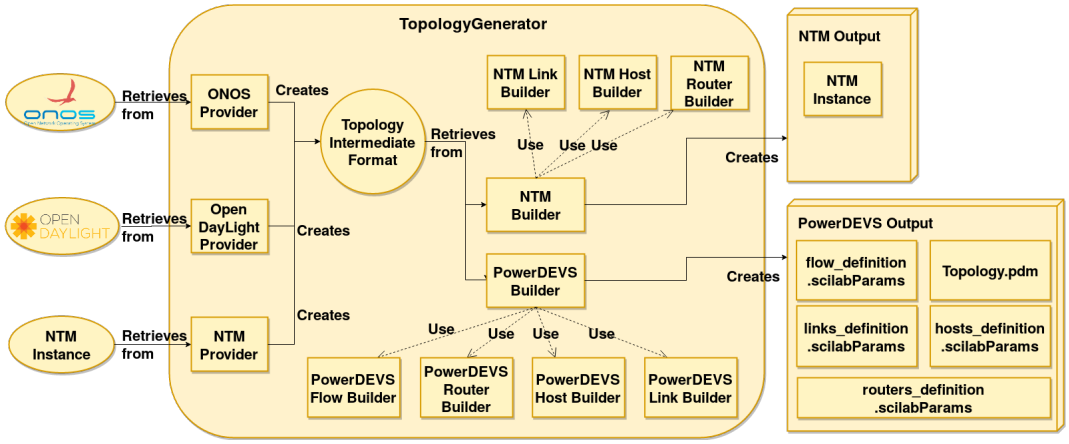


Fig. 4.3: Providers and Builders currently implemented

- **Providers**
  - **Network Topology Model (NTM)**: The NTM provider serializes topologies described with NTM, a new object-oriented network topology description language developed in this work (NTM is based on Ruby, and is detailed in section 4.3.6). NTM can be used to model existing networks or to model new draft designs.
  - **ONOS and OpenDayLight**: Providers for the SDN Controllers ONOS and OpenDayLight retrieve topology information by accessing their respective exposed APIs. Each SDN controller implements different API calls, besides different serialization responses. ONOS and OpenDayLight Providers encapsulate the logic to interact with their respective controller API, and build upon the received responses a TIF instance. These providers are examples for parsing existing, operative networks.
- **Builders**
  - **NTM**: This builder serializes a network described in the TIF format, creating a new NTM instance. This instance is typically used to programmatically customize a topology retrieved from different sources before generating a final target *Output*. An example of topology augmentation is presented in section 4.4.
  - **PowerDEVS**: This builder creates a PowerDEVS models structure for simulation. It relies on parameterizable DEVS atomic models that provide basic behavioural building blocks. The builder interconnect DEVS atomic (basic) models in a modular and hierarchical way to create DEVS coupled (more complex) models. Typical DEVS atomic models in the context of networks are queues, links, etc. For instance, in order to compose a network switch, several atomic models of input/output queues are composed together and parameterized as prioritized queues (with the Quality of Service (QoS) flag activated). Meanwhile, in order to compose a regular host (e.g. a PC with one network interface) only one input/output queue is needed, parameterized as a standard non-prioritized NIC queue.

Providers and *Builders* are decoupled by means of the Topology Intermediate Format. Any provider implementation can be used with any builder implementation. TopoGen can be extended by defining new providers and builders at will. An introduction to extending the currently supported implementations of Providers and Builders is given in section 4.3.5. Provider and Builder classes implement a Strategy Pattern, where each strategy is implemented outside TopoGen. This allows TopoGen to be adapted to different scenarios while implementing each Provider and Builder only once.

4.3.3. Class Diagram

Figure 4.4 presents a class diagram for the TopoGen implementation in the Ruby language.

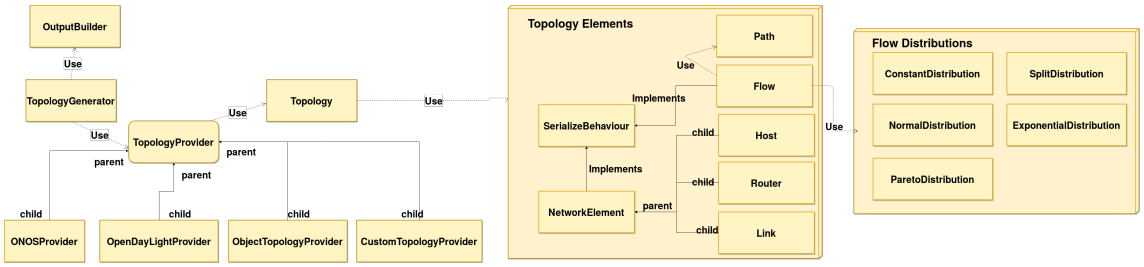


Fig. 4.4: TopoGen class diagram.

When TopoGen is used, an instance of *TopologyGenerator* class is created and then the initialize method is invoked with parameters indicating a provider, a builder (where the output will be stored into) and a URI (where to retrieve the topology from).

The *TopologyGenerator* class has one *TopologyProvider* and one *OutputBuilder* instance. The *TopologyProvider* class can be mapped to the *Provider* component showed in Figure 4.2. This class has four children:

- *OnosTopologyProvider* and *OpenDaylightProvider* classes encapsulate the logic for retrieving information from SDN controllers' APIs.
- *ObjectTopologyProvider* class retrieves a topology from a *Topology* instance.
- *CustomTopologyProvider* class retrieves information from a NTM instance.

All the mentioned classes have one *Topology* instance created at initialization time.

The *Topology* class in Figure 4.4 plays the role of the TIF in the architecture (Figure 4.2). It uses TopologyElements classes to represent the elements in the network. A *Topology* can have multiple TopologyElements, however every TopologyElement belongs to a unique *Topology*. The TopologyElements box contains all the classes that can be used to create elements in a *Topology* instance. The *NetworkElements* class represents an abstraction of the physical elements of the network (in this case *Host*, *Link* and *Router*). The *Flow* and *NetworkElement* classes implement a *SerializeBehaviour*, which is a module for serializing classes. The *Flow* class represents a flow of packets between hosts. When creating a *Flow* instance, a packet rate distribution and a packet size distribution are needed. Distribution classes are shown in the Flow Distribution box.

Finally, the *OutputBuilder* class represents the component *Builder* in Figure 4.2. This class looks for built-in builders of each TopologyElements classes in the received URI as it is detailed in the following section.

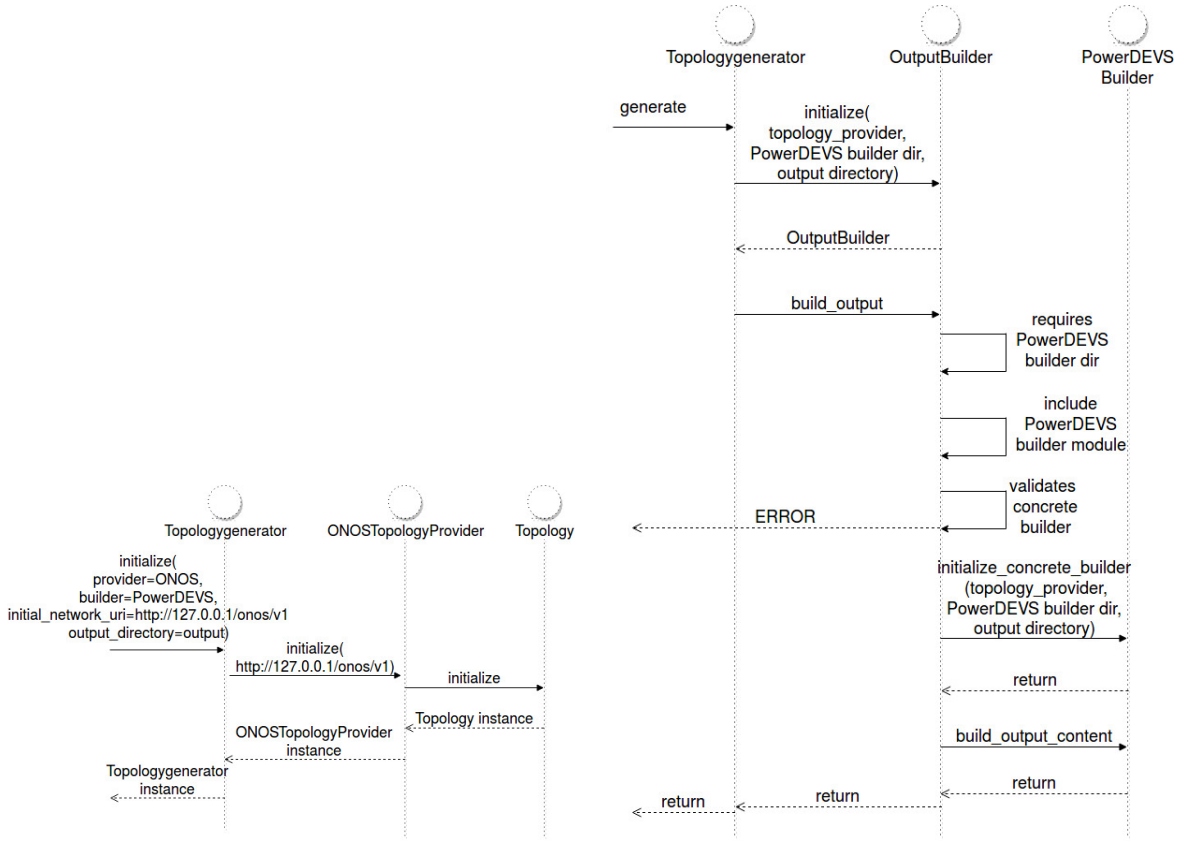
4.3.4. Sample Sequence Diagrams

We describe two sequence diagrams showing examples of interactions among classes depicted in Figure 4.4. We use a sample case having an ONOS controller as a source for the Provider, and PowerDEVS as a target for the Builder. This use case assumes an existing ONOS local installation.

Figure 4.5 (a) illustrates TopologyGenerator initialization sequence. The initialize method is invoked with four arguments: Provider ONOS, Builder PowerDEVS, the URI where to retrieve the initial network from (in this case a local ONOS controller), and an output folder where to store results. The initialize method creates an OnosTopologyProvider instance by providing the received URI as an argument. OnosTopologyProvider instance creates a Topology instance. Finally a OnosTopologyProvider instance is returned.

The topology serialization is made by calling the generate method in Topologygenerator. The flow produced by calling this method is shown in Figure 4.5 (b).

The generate method creates an instance of OutputBuilder using arguments previously received in Figure 4.5 (a). Then build_output method of OutputBuilder is invoked. This method requires



(a) Topologygenerator initialization sequence *(b)* Topologygenerator serialization sequence

Fig. 4.5: Sequence diagrams for initialization and serialization methods.

and includes all ruby files in Power DEVS builder directory after validating them. Methods implemented in these files will run in the OutputBuilder context. In case validation fails, an error is thrown and the execution stops. Otherwise, Builder PowerDEVS class is included and instantiated with three arguments: ONOS topology provider, Builder PowerDEVS folder and the output folder. Finally, *build_output_content* method is called where the expected outputs are generated.

4.3.5. How to Augment Built-in Providers and Builders

This section is focused on describing how to augment currently supported providers and builders in TopoGen.

In order to support a new source provider, a new class must be implemented. The new provider class has to extend the *ITopologyProvider* class (see 4.4), and it has to define two methods, the *get_topology* method which was already discussed in section 4.3.6 when NTM was introduced, and the *get_path_between(source,destination)* method, which has to implement an algorithm for retrieving a path between the given source and destination, and return it by returning a class Path instance. After defining this class, a new case must be added in the current implemented switch case method in classes *Topologygenerator* and *CommandLineArguments*. The new case is expected to create a new instance of the recently implemented class. The current implementation in Provider class should be changed to follow the same pattern as Builder classes. Because a lack of time, this work has been left out as future work.

In order to support a new targeted output, a directory path were the builder is has to be provided by argument, as it was shown in Figure 4.5 (b). In this directory, TopoGen will try to load:

- Files called after the TopologyElements classes (see Figure 4.4) having appended *concrete_builder.rb* at the end of them. For instance, *host_concrete_builder.rb*, *link_concrete_builder.rb*,

router_concrete_builder.rb and *flow_concrete_builder.rb*. Each file must contain a module named after the Topology Element class it represents with *ConcreteBuilder* appended at the end of it. This module has to implement the serialization logic of the respective topology element in the *build_output_representation* method. For instance *HostConcreteBuilder* will encapsulate the logic on how to serialize a Host in *build_output_representation* method.

- A file called *output_concrete_builder.rb*. This file has to contain a ruby module called **OutputConcreteBuilder**, with at least two methods, the *initialize_concrete_builder* method, which will receive an instance of the topology provider being used in the execution, the path of the directory where the builders are and the path of the output directory, that is where the desired output should be stored, and the *build_output_content* method, which will be called in the process of generation, and it's expected to generate the output in the specified folder. An example of code execution process of a serialization sequence in TopoGen can be seen in Figure 4.5 (b).

Previous explanation implies that in order to implement a new builder, all what is needed is to define these modules with their corresponding methods.

4.3.6. The Network Topology Model

The Network Topology Model (NTM) is an object oriented approach to represent networks topologies in Ruby. NTM makes it possible to describe all the elements in a network: physical elements (e.g. hosts, routers, links, etc.), and logical elements (e.g. data flows, routing paths, etc). NTM is currently dependent on TopoGen as it was designed to be used by the CustomTopologyProvider class (see Figure 4.4).

NTM and TIF are two different pieces of the architecture, and none is intended to replace existing network description languages (in fact, such existing languages can take the role of input/output formats consumed/produced by TopoGen). We opted for defining our custom TIF to act as an in-memory relay format, independent of any third party existing language to describe networks. Regarding NTM, it belongs to the category of network specification languages, but has the salient feature that multi-hop flows can be described explicitly. Besides, NTM is "native to TIF" making the "NTM Builder" a trivial one-to-one object mapper. This sets an operational baseline that is agnostic of any third party language. Yet, the architecture leaves the door open for new Builders to be developed to accommodate known network description languages.

4.3.6.1. Describing a Network Topology with NTM

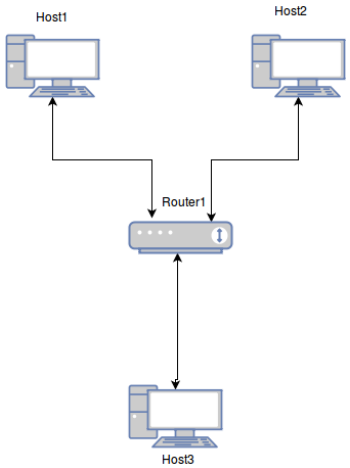


Fig. 4.6: Simple network topology to be described with NTM.

The following NTM Ruby code describes the network shown in Figure 4.6 including the communication flow between Host1 and Host3:

```
1  module NetworkTopology
2   def get_topology
3    return @topology.topology_elements unless @topology.topology_elements.size == 0
4    hosts = []
5    router = @topology.add_router "Router1"
6    for i in 0..2
7      host = @topology.add_host "Host#{i}"
8      hosts.push host
9    end
10   bwidth = 500*1000*1000 # 500 Mbps
11   @topology.add_full_duplex_link "Link1",hosts[0],0,router,0,bwidth
12   @topology.add_full_duplex_link "Link2",hosts[1],0,router,1,bwidth
13   @topology.add_full_duplex_link "Link3",hosts[2],0,router,2,bwidth
14   link1 = @topology.get_element_by_id "Link1_up"
15   link2 = @topology.get_element_by_id "Link3_down"
16   flow_1_path = Path.new hosts[0], hosts[2]
17   flow_1_path.add_link link1
18   flow_1_path.add_link link2
19   @topology.add_flow "Flow1",10,[flow_1_path],(ExpDistrib.new 1.0/6875),(ConstDistrib.new
          1000*8)
20   @topology.topology_elements
21  end
22 end
```

Each NTM instance must define a *NetworkTopology* module (line 1) and a *get_topology* method (line 2) which returns the elements added in the Topology instance (variable *@topology*). To create the topology, the router is first added in line 5 (*add_router* method). The hosts are defined in lines 6-9 (*add_host* method) with a unique identifier for each host. In lines 11-13 links are added using *add_full_duplex_link*. This method expects an ID, a source element (an instance of Router or Host), a source port number, a destination element, a destination port number and the bandwidth (in bps), and creates a source and destination links.

The first link goes from source to destination (its ID is the concatenation of strings *up* and the ID received). The second link goes from destination to source (its ID is the concatenation of strings *down* and the ID received). The method returns the second link only. In lines 14 and 15 the links that define a path between Host1 and Host3 are retrieved. In lines 16 to 18 a new path is created between Host1 and Host3 using the retrieved links. In lines 19 to 20 a new flow is added with the *add_flow* method (it expects an ID, a flow priority, an array of possible paths for the flow, and stochastic distributions for packet rate and size). The NTM description ends by returning the new elements with the *topology_elements* method.

4.3.7. TopoGen Installation and Usage

TopoGen can be installed in a Linux environment in two different ways:

- It can be installed as a Ruby Gem, and integrated in an existing Ruby project. This can be done by adding the following line to an application's Gemfile:

  ```
  gem 'topologygenerator'
  ```

  And then executing:

  ```
  $ bundle
  ```

  Or it can be installed by executing the following command:

  ```
  $ gem install topologygenerator
  ```

- It can be installed as a binary, by running the following command:

  ```
  \curl -sSL https://raw.githubusercontent.com/andyLaurito92/
      topologygenerator/master/install_topologygenerator.sh | bash
  ```

TopoGen can be used both as a Ruby gem or a command line tool. The following Ruby code excerpt shows an example using the ONOS provider and the NTM builder. The TopoGen object then needs to be specified with a desired output directory, and an URI where the ONOS SDN Controller API accepts requests.

```
my_topology_generator = TopoGen.new({
  "source" => "ONOS",
  "directory_concrete_builders" => "ruby_builder",
  "output_directory" => "output_example",
  "uri_resource" => "http://192.168.24.3/onos/v1/" })
my_topology_generator.generate
```

The same results are obtained with the following TopoGen command line version (a –help option is available):

```
TopoGen source -n ONOS -o output_directory -u http://127.0.0.1/onos/v1
    / -d ntm_builders
```

These parameters can be changed to retrieve topologies from different providers or to generate topology files for different tools, allowing the use of TopoGen in contexts where different technologies are available.

4.4. Support for designing the FELIX network at the ATLAS Data Acquisition System

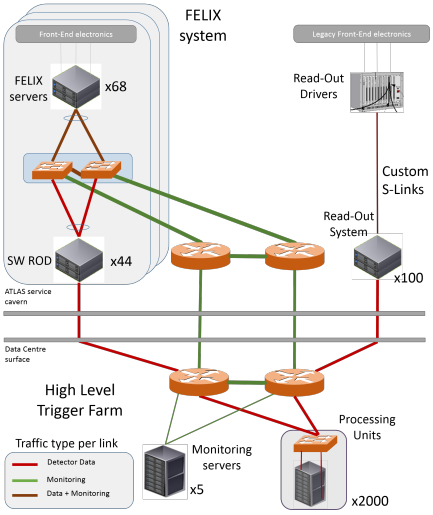
In this section we describe TopoGen as applied in a real world scenario. The case study builds upon a modeling and simulation-driven engineering process [9] developed for the ATLAS TDAQ network at CERN [34]. We show how TopoGen can assist the design phase for the network to be implemented 2019-2020 in the ATLAS FELIX project [4].

4.4.1. The FELIX Network Requirements

The FELIX network will provide connectivity between different components of the FELIX system (see Figure 4.1) and will handle various types of traffic which differ in their throughput, latency, priority and availability requirements. For example, the Detector Control System (DCS) that monitors and controls the detector's front-end electronics requires the highest priority and low latency to react fast, but is expected to require low throughput. Meanwhile, the detector's data will use most of the network bandwidth so it can have less priority to avoid saturation. Table 4.1 summarizes the different traffic types and their requirements.

The communication patterns are also different for each type of traffic. While DCS traffic follows a many-to-one pattern (all FELIX servers communicate with a single DCS server), Control and Monitoring traffic require a many-to-few pattern. Detector data, on the other hand, uses a simple one-to-one or two-to-one pattern from a FELIX server to SW RODs.

To provide confidence about the coexistence of these traffic types while meeting performance requirements we adopt a modeling and simulation approach to study expected throughput and latency for each traffic type, and anticipate possible bottlenecks. Although the high level requirements are defined, each subsystem's specification is updated often during the design process. Specific subsystem parameters (throughput, processing times, etc.) will not be known until the final system is in place. Yet, simulation can provide guidelines for realistic ranges of candidate parameter values (parameter sweeping).



Tab. 4.1: FELIX traffic types and requirements.

Traffic type	Throughput	Latency	Priority	Comm. Pattern
DCS	Low	Low	High	Many to one
Control and Config.	Med.	-	High	Many to few
Detector Data	High	-	Med.	One to one
Monitoring	High	-	Low	Many to few

Fig. 4.7: Topology of the FELIX system.

Other investigations are being directed to define the underlying high-throughput technology that will be used. The NetIO library [39] provides an abstraction layer that allows FELIX applications to transparently operate on Ethernet, Infiniband and similar interconnect technologies. NetIO supports three different high-level programming models to support the different FELIX traffic types: low latency, high throughput and publish/subscribe. Each of these impose different buffering delays and that also need to be considered in the simulation model as they directly affect dataflow patterns.

Technologies rely on different protocols, congestion control algorithms, and routing schemes which also need to be considered in simulation studies. In particular, different restrictions are imposed over candidate topologies depending on selected technologies (e.g. Ethernet allows for heterogenous link speeds, Infiniband does not. Infiniband efficiently supports mesh and leaf-spine topologies, Ethernet supports topologies with cycles but using algorithms that perform poorly). The simulation platform needs to be able to define all these types of topologies in a flexible way to support agile design iterations.

4.4.2. Using TopoGen to Support the Modeling and Simulation Process

Figure 4.8 shows the workflow used while applying TopoGen to aid the modeling and simulation process for the FELIX network. The workflow consists of three phases: first, the topology under design by the networking team is automatically retrieved and serialized into an NTM instance; second, the NTM topology is augmented programmatically with extra resources (nodes and data flows), and third the new topology is serialized into a PowerDEVS simulation model. Hence a simulation model is automatically created from an existing specification originally meant for other purposes. This workflow deals with topology changes at design-time. Run-time adaptation of simulations to topology changes remains a subject of future work.

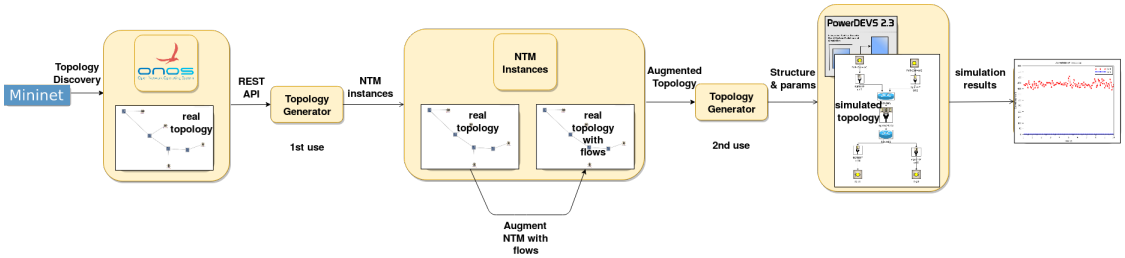
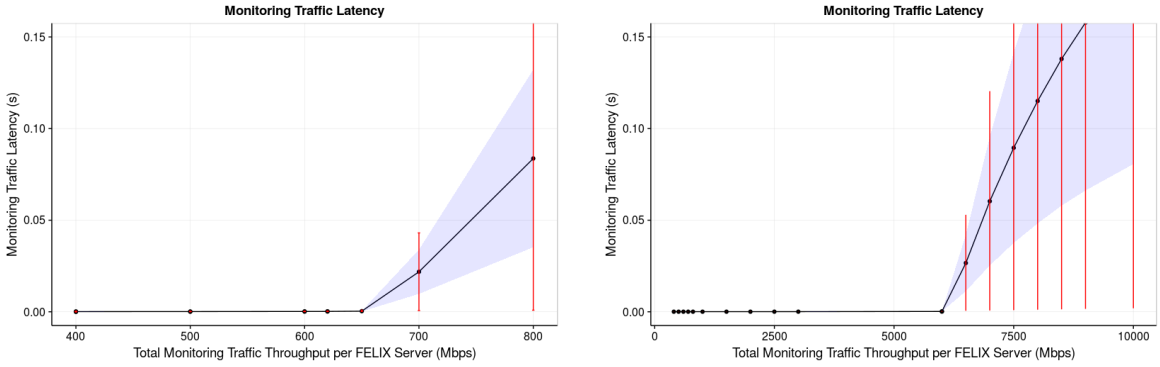


Fig. 4.8: Modeling and simulation workflow using TopoGen.

In the **first phase**, the networking team provided a Mininet emulated environment used to test the connectivity of a topology, including nodes only from the FELIX network. The ONOS SDN



(a) *(b)*

Fig. 4.9: Simulated mean packet latency seen by the Traffic Monitoring servers. Link capacity allocated to monitoring traffic: (a) 1 Gbps (b) 10 Gbps. Blue area: standard deviation. Red lines: min-max range.

controller was installed within the emulated environment to provide network discovery services. Then, the TopoGen ONOS Provider was configured to connect with the REST API exposed by ONOS to query the topology. Once the topology is retrieved, the TopoGen NTM Builder serialized it into an NTM instance for later use. Each time the networking team updates their emulated topology, it can be retrieved again to keep the NTM and simulation models up-to-date. The fact that the original topology was specified in an emulated environment is transparent for TopoGen.

In the **second phase**, additional nodes are added to the NTM topology to also represent the HLT network (see Figure 4.7). To generate a meaningful simulation model extra information is needed about the traffic generated by different servers. Nodes and data flows, along their respective parameters, were added programmatically into the original NTM instance guided by the network engineers. For this case study, only the Detector Data traffic type and the Monitoring traffic type were considered (see Table 4.1).

In the **third phase**, the augmented NTM instance is loaded by the TopoGen NTM Provider and used by the PowerDEVS Builder to generate all necessary files for simulation.

The recently explained phases were automatize by the development of a unique execution bash script.

The network actually simulated with PowerDEVS is the one presented in Figure 4.7. It includes the FELIX network nodes (automatically retrieved from the SDN controller) and the HLT network nodes, the Detector Data and Monitoring traffic flows (added programmatically with NTM). Exact parameters for traffic generation will remain unknown until the final setup, so parameter sweeping is used in simulations to scan possible ranges of values. This case study focused on network behaviour under different intensities of Monitoring traffic rates, matching an engineering requirement.

4.4.3. Simulation Results

We studied the potential effects on the average latency of FELIX Monitoring traffic in the case of an upgrade of the bottlenecked links from 1 Gbps to 10 Gbps. Figure 4.9 (a) shows the average packet latency for FELIX Monitoring flows in different scenarios with increasing monitoring throughput for all servers. As monitoring traffic grows the latency slightly increases until a transition is observed at the point when each server generates 650 Mbps of monitoring data. After that point the latency increases rapidly denoting the presence of congestion. The buffer sizes and link utilization at the switches (not included in this report) indicate that the source of congestion are the 1 Gbps links of monitoring servers. We then updated the topology in the NTM instance, now with a link capacity of 10 Gbps for monitoring nodes. The PowerDEVS simulation model was regenerated with TopoGen, and new experiments were run. Figure 4.9 (b) shows how the saturation point moves up to 6500 Mbps of monitoring traffic. The congestion point in the topology remains at the

links directly connecting the monitoring servers.

4.5. Conclusions

We presented TopoGen, a reference architecture and tool for systematic translation and generation of network topologies. We also introduced NTM, a network topology model to describe networks programmatically using the Ruby language. Decoupled Providers, Builders, and a Topology Intermediate Format allow the creation of flexible topology transformation workflows that can suit diverse needs. We focused on the generation of DEVS simulation models starting from network information available at SDN Controllers.

We applied TopoGen successfully in the context of a real-world network design process: the FELIX system of the ATLAS TDAQ network at CERN. TopoGen proved effective to retrieve a large topology from an ONOS controller, export it to a programmable network model, augment the model manually according to particular simulation needs, and generate a fully operational DEVS simulation for the PowerDEVS tool.

When compared with previous experiences in our team achieving the same results and in the same context, TopoGen strongly reduced the time to completion, complexity and error-proneness. In the next section we details future steps for TopoGen.

4.6. Future work

In this section we introduce features which were left-out from development process because of a lack of time. These features are a good starting point for contributions to TopoGen.

4.6.1. Augment supported Providers and Builders

Many information sources exist for retrieving networks topologies that are yet not supported. Some examples of these sources are: Infiniband ([31]) , The Internet Topology Zoo ([22]), OMNeT++ ([42]), Frenetic [17]. On the other hand, and because of their extended uses, it would be useful to implement built-in Builders for: YANG (([8]), NS3 ([37]) and DEVS ([12]) (besides the current PowerDEVS builder implemented). Augmenting providers and builders can extend TopoGen current reachability scenarios of work, and make of TopoGen a more general-purpose tool.

4.6.2. Extend TopoGen to become a Serialization Graph Tool

Nowadays TopoGen is a general-purpose serialization graph tool in the SDN environment. By implementation of new Providers and extension of the current Topology Elements classes, TopoGen can become a generic serialization graph tool. The key point in this task is to increment reachability of TopoGen over new scenarios, including the existing ones. Regarding the study of the current thesis, this point is of interest for enabling the use of operational graph semantics libraries from TopoGen.

4.6.3. Improve NTM to become a Network Topology Language for PowerDEVS

The Network Topology Model presented in this chapter is a Ruby framework within TopoGen used to describe network topologies. Extending NTM to become a Network Topology Language for PowerDEVS can be of great use for the Simulator. Nowadays network models are either program in C++ or built in using the UI provided by PowerDEVS. As it was explained in section 4.2, this is an error-prone and time-consuming task when large-size networks are targeted. Extending NTM to a PowerDEVS network topology language can help to decrease complexity of this task.

4.6.4. Run-Time Adaptations of Network Topologies

Network topologies tend to change because of different factors. Every time a topology changes, TopoGen must be re-run in order to maintain up to-date the topology model. Implementing run-time adaptations of network topologies is an important feature to keep up to date targeted generated models. A possible starting point of this work concerns the study of monitor implementation in controllers.

5. HAIKUNET

5.1. Introduction

In this section we introduce Haikunet, which is a Domain Specific Language for programming intents in the SDN architecture. This section is organized as follows: We start by introducing the motivations for developing Haikunet. Then, representative program examples are given, and attainable errors are discussed within the context of the different programs. Subsequently, standard terminology is introduced and the notion of errors is given in terms of graph grammars. Finally, a design and implementation perspective of Haikunet is presented, and brief installation and tutorial guidelines are given.

5.2. Why Haikunet?

The name Haikunet comes from a combination of Haiku and Network. A Haiku is a Japanese poem which usually describes the nature from the point of view of an observer. Haikunet aimed at being an easy-to-use programming language for describing functional requirements of the underlying network. Haikunet is though to be used by non-expert network users, i.e. Haikunet programs are intended to be written from the viewpoint of a network user, which is the purpose of an intent programming language.

5.3. Motivations

Despite intent-oriented programming languages (IOPL) approach has advantages, it also presents some limitations. IOPL are usually developed to run in specific vendor *controllers*, which introduces unnecessary coupling between *intents* and *controllers*. This problem arises in part because there is still no standard for *controller's* API. Consequently, *intents* need to be rewritten when using different *controllers*. For instance, NEMO relies on OpenDayLight [28], Frenetic relies on NOX [18] and NetKAT relies on the OpenFlow *controller* [27].

Another important shortcoming is that existing IOPLs do not provide mechanisms for detecting errors before *intents* are applied to the network. As a consequence, *intents* are first tested in simulated networks such as Mininet [15], before they are applied to the real network. Testing *intents* over simulated networks forces developers to maintain the simulated environment, which is a time-consuming and error prone task.

Finally, IOPL are usually implemented without mathematical formalism, e.g. NEMO, Nettle [45] (a functional reactive programming language for OpenFlow Networks), Procera [46] (a language for high-level reactive network control), FatTire [36] (a declarative fault tolerance language) and Flog [21] (a logic programming language for SDNs). Few are the IOPL that present such mathematical formalism. An example of such a language is NetKAT [3]. Giving mathematical foundations create opportunities for known software-engineering solutions to be applied to *intents*, for instance applying static/dynamic analysis techniques.

In this chapter we introduce Haikunet, an interpreted Intent programming language agnostic to a specific controller that implements a semantic checker. Haikunet semantic checker can detect a group of errors, which will be introduced in the following sections. Haikunet was developed to act as a client consumer of an API. It currently implements consumers for the OpenDayLight and ONOS controllers API but, as will be explained later on, the Haikunet architecture can be extended to consider other options.

5.4. Program Examples

In this section we introduce Haikunet by example, by given the program and the network in which it will be executed. Haikunet implements the following features:

- Variables, which context is the entirely program.
- Explicit named parameters for referring to network elements properties, such as *mac* or *ip*.
- Pre-defined keywords to referred to specific network elements, such as: Host, Flow or Intent.
- Basic type expressions, such as Strings and Arrays.

Features detailed above are used in the following program examples:

5.4.1. First Example: A Simple Network

We start by showing a program that defines a flow between two hosts in the network depicted in Figure 5.1.

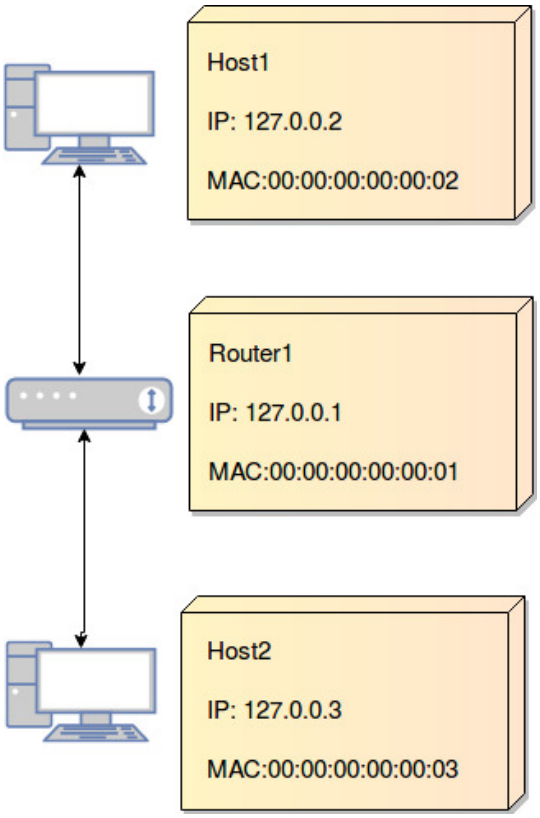


Fig. 5.1: The Simple Network of Example 1

The following program achieves the desired behaviour.

```
1
2  first_host := Host(mac="00:00:00:00:00:02")
3
4  second_host := Host(mac="00:00:00:00:00:03")
5
6  my_flow := Flow(src=first_host, dst=second_host, priority="55")
7
8  Intent newFlow
9  Select my_flow
```

The expression *Host(mac="00:00:00:00:00:02")* in line 2 denotes a host associated with the *mac* address *"00:00:00:00:00:02"*. When this intent is run over the network in Figure 5.1, the expression *Host(mac="00:00:00:00:00:02"* refers to *Host1*. The assignment bounds the local variable *first_host* to *Host1*. A host can be denoted either by its *ip* or its *mac* address. Variable names are non-empty sequences of alphanumeric values not ending with a period. Line 4 shows the definition of *second_host* to denote *Host2* of figure 5.1.

Line 6 creates a new flow with *Host1* as its source and *Host2* as its target. The priority number represents a quality attribute assigned to the packets of this flow. The new flow is then bound to the local variable *my_flow*. A flow is defined by giving its source and its target. Source and target are respectively specified by providing *src* and *dst* parameters. In this case, the values for those properties can be either an array of elements denoting hosts or an element denoting a host. Examples of elements that denote hosts are variables bound to a host, such as *first_host*, or a string containing a valid *IP* or *MAC* address.

Finally, line 8 and 9 denotes an intent called *newFlow*. This intent creates the previous flow by using the Select keyword. Intent and Select are both keywords used for denoting an intent and a flow respectively. Intent keyword must always be followed by the intent name, meanwhile Select must always be followed by an expression denoting a flow.

5.4.2. Second Example: Connecting Multiple Hosts

The second program example concerns the network depicted in Figure 5.2. The goal is to program an intent that creates a flow having *Host5* as source and *Host2*, *Host3* and *Host4* simultaneously as target.

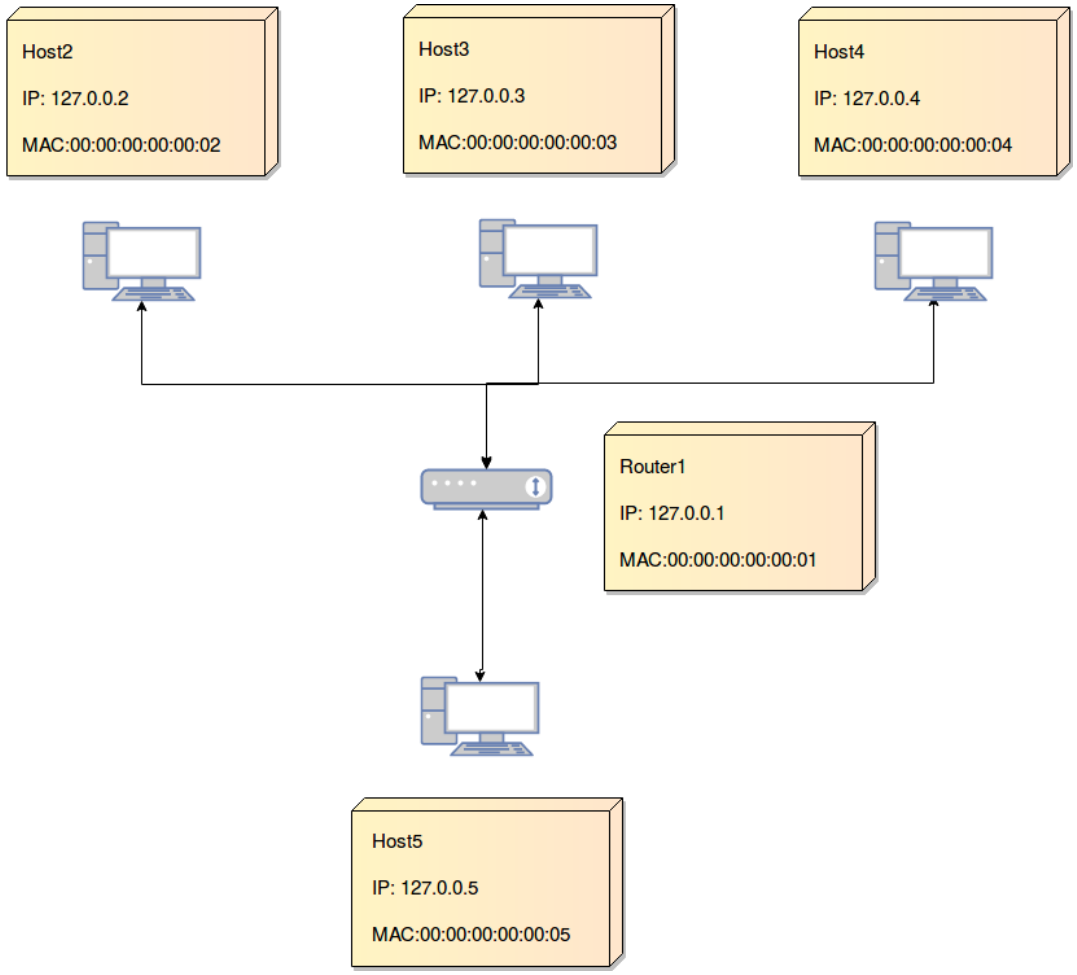


Fig. 5.2: Connecting Multiple Hosts Network Example 2.

The following Haikunet program achieves the problem commented above:

```
1
2 myHost5 := Host(mac="00:00:00:00:00:05")
3
4 myNewFlows := Flow(src=myHost5, dst=["127.0.0.2", "127.0.0.3", "
    127.0.0.4"])
5
6 Intent newFlows
7 Select myNewFlows
```

Line 2 shows the definition of *myHost5* to denote the host with *mac "00:00:00:00:00:05"*. In line 4 a new flow is created where *myHost5* variable denotes the source host, and the array of *Ips ["127.0.0.2", "127.0.0.3", "127.0.0.4"]* denotes the target. When this intent is run over the network in Figure 5.2, expression *["127.0.0.2", "127.0.0.3", "127.0.0.4"]* refers to *Host2*, *Host3* and *Host4* respectively. The new flow is then assigned to the variable *myNewFlows*. In this example, we avoid defining each host with a variable (as it was done in the previous example), and use instead an array where the *IP* of each host is provided.

Finally, line 6 and 7 denotes an intent called *newFlows* which creates the previous flow.

5.5. Attainable Errors

When presenting program examples in the previous section, several mistakes could have been made. For instance: 1. Denoting a *Host* with a property that is misspelled; 2. Creating a *Flow* with an unexisting property (for example by providing an invalid source *ip*); 3. Creating a *Flow* between hosts which have no physical connection between them.

Previous mistakes made in a program lead to execution errors in the targeted controller or errors in the resultant network. Sometimes these types of errors are difficult to find, making troubleshooting considerable time-consuming. This section explores situations in which errors are introduced when developing a Haikunet program.

Following errors are presented with a name, a network in which they occur, a Haikunet program which introduces the issue and finally a detailed description of the problem.

5.5.1. Flow Property Definition Error

First Haikunet program error concerns the network depicted in figure 5.3.

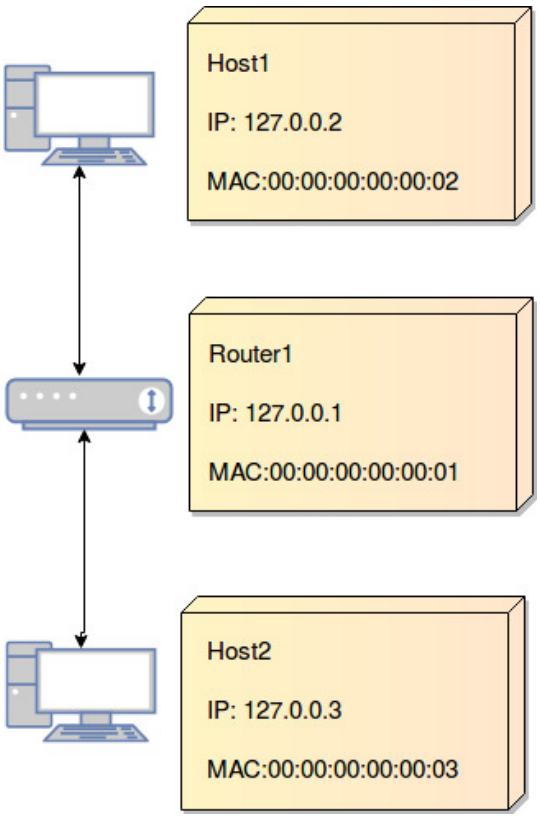


Fig. 5.3: Flow Property Definition Error Network Example

The following is an example of a program which contains an error in the depicted network.

```
1
2 source_host := Host(mac="00:00:00:00:00:02")
3
4 my_flow := Flow (src=source_host, dst="127.0.0.10", priority="55")
5
6 Intent firstExample
7 Select my_flow
```

The problem here is that the named property *dst = "127.0.0.10"* is not a valid *IP* in the given network (i.e. The *IP* does not denote a host in the network depicted in Figure 5.3). Because of this, the property cannot be neither inferred nor found from the context.

Previous program has an error because the mistake is made in Flow definition (line 4). If the misspelled would have been made in Host definition (line 2) as it is shown in the following code:

```
1
2 source_host := Host(ip="127.0.0.10")
3
4 my_flow := Flow (src=source_host, dst="127.0.0.3", priority="55")
5
6 Intent firstExample
7 Select my_flow
```

The program would not be considered as having an error. This is because the semantic of the program above is: 1) Create a host with *ip "127.0.0.10"*, 2) Create a flow between the new *Host* and *Host* with *ip 127.0.0.3*. When a *Host* is created without *mac*, the underlying controller is in charge of setting up a new *mac* address if possible. If this is not possible, and exception is raised.

5.5.2. No Path Error

This error concerns the creation of a flow between two network elements that do not have physical connection between them. Figure 5.4 depicts the initial network topology.

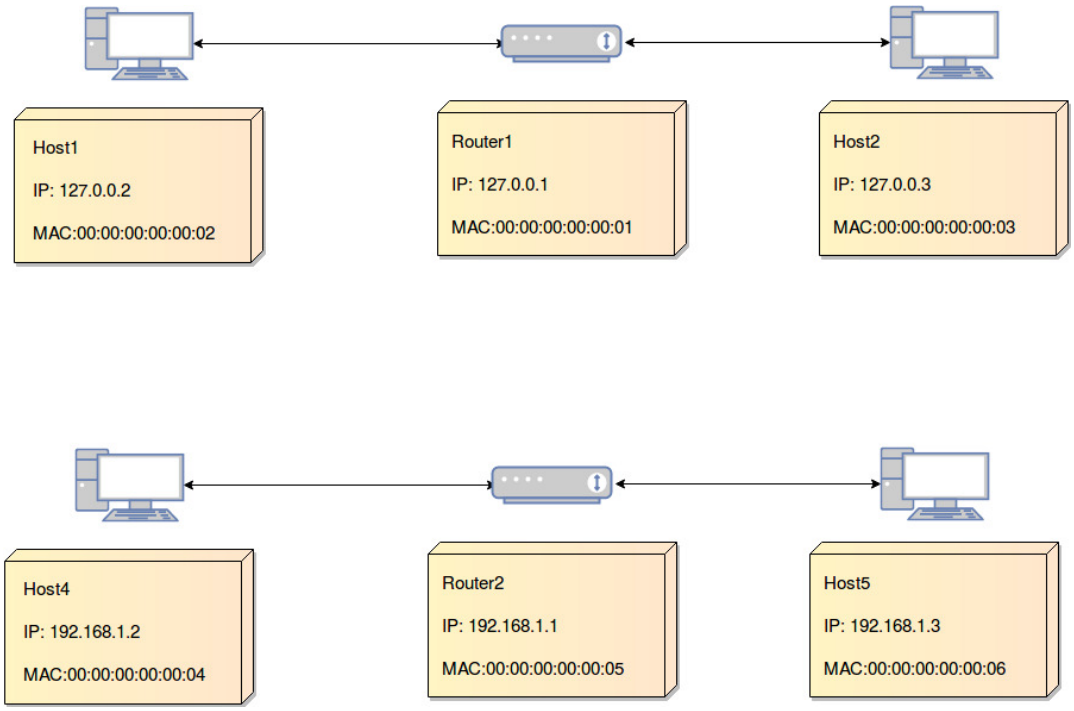


Fig. 5.4: No Path Error Network Example

The Haikunet program belows contains a *No Path Error* error when executed in the above network.

```
1 source_host := Host(ip="127.0.0.2")
2
3 destiny_host := Host(ip="192.168.1.3")
4
5 my_flow := Flow (src=source_host, dst=destiny_host, priority="55")
6
7 Intent firstExample
8 Select my_flow
```

It can be seen in Figure 5.4 that the error is thrown because there is no path between hosts *127.0.0.2* and *192.168.1.3*.

5.6. A Formal Description of Haikunet

This section formalizes the concepts previously presented and standardize vocabulary that will be use along this chapter. Towards this section errors already defined are categorized, intents and their behaviour are formally defined, and an operational semantic is introduced.

5.6.1. Definitions

5.6.1.1. Network Element

A network element is either a Link or a Host. For the purpose of this and later sections, we do not distinguish between different kinds of Hosts, such as: an end station, a router, or a switch.

5.6.1.2. Network as a Mutable Graph

A network can be easily seen as a graph that describes its topology. For instance, we can get a graph representation from the network topology by matching hosts to nodes and links to edges. There are multiple events that can caused a topology to change, for instance: 1. the failure of a network element 2. a network element being disconnected on purpose by a network operator, 3. one or several intents being applied to a controller.
All the mentioned events can alter the expected behaviour of an intent being executed in a network. For this reason, this work focus on the ability of describing how this events alter a network topology. Because a matter of complexity, this thesis narrows the study of all possible events to study only the event of one intent being applied to a controller. We left as future work the study of the remaining events.

5.6.1.3. Intent as a Sequence of Transformations over a Graph

An intent must always be applied to a network. Formally, an intent is defined as a sequence of transformations over a graph. For instance, the intent presented in section 5.4.2 can be seen as the following sequence of transformations: 1. Set *Host 5* endpoint as source, 2. Set *Host 2*, *Host 3* and *Host 4* as targets (this could be detailed in three steps, but it is collapsed into one for simplicity), 3. Create the forwarding rules needed in each node, 4. Define a new flow with source *Host 5* and targets *Host 2*, *Host 3* and *Host 4*.

Given a specific transformation and a graph representing a network, we would like to define whether a transformation can be applied over the graph. In order to achieve this goal, we provide a formal semantics for Haikunet, which formalizes all the transformations made by an intent when applied over a network. The formal semantics for Haikunet are further detailed in section 5.9.

In the following section we introduce Haikunet syntax in the *Backus-Naur Form*(BNF).

5.7. Haikunet Backus Naur Form

Haikunet programs are built upon the following Backus-Naur Form:

$$P ::= \emptyset \mid A;P \mid Intent\ V\ Select\ V;P$$

$$A ::= V := E$$

$$E ::= H_1 \mid F$$

$$F ::= Flow(src = H_2, dst = H_2, priority = N)$$

$$H_1 ::= Host((mac = MAC \mid ip = IP))$$

$$H_2 ::= [(MAC,)^*MAC] \mid [(IP,)^*IP] \mid MAC \mid IP \mid V$$

Fig. 5.5: Haikunet Backus Naur Form

P is a non-terminal that denotes a Haikunet program and stands for a sequence of expressions. The simplest Haikunet program is the empty program, denoted by $\emptyset$.

The expression $A;P$ denotes a program that starts with an assignment A and then follows as P. The non-terminal A represents the assignment of an expression E to a variable V. V is the non-terminal for identifiers (its definition is standard, and hence omitted). The non-terminal E denotes two possible types of expressions: 1. Host expressions H_1 or 2. Flow expressions F.

A Host expression H_1 has as named parameter either a MAC or an IP expression (grammar rules for MAC and IP are omitted because they are standard). The non-terminal F denotes a Flow expression, which has three parameters , namely, *src*, *dst* and *priority*. The first two named parameters respectively denote the source an target host of the flow, while priority is a natural number (the rules for natural numbers N are omitted). We remark that H_2 stands either for: 1. a single IP or MAC, 2. a sequence of either of them, 3. an identifier denoting a host expression.

Finally, the expression *Intent V Select V;P* denotes the creation of an intent. As stated before, non-terminal *V* is the non-terminal for identifiers. This expression remarks that more than one intent can be specified in a program.

5.8. Haikunet Implicit Type System

We now present a simple type system that rule out ill-formed intents, e.g., programs in which variables bound to a Flow are used when a host is expected. The type-system presented in this section uses the type-judgments analogous to the the lambda calculus. Formally, a type-judgment is a relation over typing contexts (or type environments), expressions e, and types τ. The judgment:

$$\Gamma \rhd e \ : \tau$$

is read as e has type τ in context Γ. A typing context (also called a type environment) Γ is a a partial function from variables to types. The "comma" operator stands for disjoint union.

In this work we assumed the following basic types: 1. **String:** for alphanumeric strings enclosed by quotation marks; 2.**MAC:** for the set of mac addresses; 3. **IP:** for the set of IP addresses.

We start by introducing type-judgments for expressions.

5.8.1. Expressions Type Judgments

(HOST MAC DEFINITION)

$$\frac{\Gamma \rhd x : MAC}{\Gamma \rhd Host(mac = x) \ : Host}$$

(HOST IP DEFINITION)

$$\frac{\Gamma \rhd x : IP}{\Gamma \rhd Host(mac = x) \ : Host}$$

(FLOW DEFINITION)

$$\frac{\Gamma \rhd x : Host \qquad \Gamma \rhd y : Host \qquad \Gamma \rhd z : Number}{\Gamma \rhd Flow(src = x, dst = y, priority = z) \ : Flow}$$

(ARRAY IP DEFINITION)

$$\frac{\forall i \ / e_i : IP}{\Gamma \rhd [e_1, e_2, ..., e_n] \ : Host}$$

(ARRAY MAC DEFINITION)

$$\frac{\forall i \ / e_i : MAC}{\Gamma \rhd [e_1, e_2, ..., e_n] \ : Host}$$

Host MAC Definition denotes that a Host creation expression is of type Host, whenever the expression x is of type MAC. Host IP Definition judgment is analogous to Host MAC Definition but for IP addresses.

Flow Definition states that a Flow creation is of type Flow, whenever x and y are of type Host (x and y) and z is a number.

Array IP Definition states that the sequence $[e_1, e_2, ..., e_n]$ denotes a host when every element has type IP. Array MAC Definition judgment is analogous to Array IP Definition but for MAC addresses.

We next introduce type judgments that characterized valid programs in Haikunet.

5.8.2. Program Type Judgments

(EMPTY PROGRAM)

$$\frac{}{\Gamma \rhd \emptyset : \mathit{Unit}}$$

(VARIABLE ASSIGNMENT)

$$\frac{\Gamma \rhd E : \theta \qquad \Gamma, V : \theta \rhd P}{\Gamma \rhd V := E; P : \mathit{Unit}}$$

(INTENT CREATION)

$$\frac{\Gamma \rhd P : \mathit{Unit} \qquad \Gamma \rhd x : \mathit{String} \qquad \Gamma \rhd y : \mathit{Flow}}{\Gamma \rhd \mathit{Intent}\ x\ \mathit{Select}\ y; P : \mathit{Unit}}$$

Program Type judgments use type *Unit* for denoting the well-typed programs. The first rule concerns empty programs, which are always well-typed. Variable assignment states that a program `V := E;P` is well-typed whenever E has type θ under Γ, and P is well-typed under Γ extended with the binding $V : \theta$. Intent creation states that P is well-typed under Γ and the first argument of an intent declaration must be of type String meanwhile the second one must be of type Flow.

The previous type-judgments do not guarantee that programs are error-free as it will be shown in section 5.10. In the next section we introduce the semantics of well-typed programs.

5.9. Haikunet Semantics

Haikunet semantics is given in two steps: 1. a set of graph rewriting rules describes the basic transformations on a network topology. 2. a set of reduction rules defines the meaning of Haikunet programs. Before introducing inference rules, we detailed how graphs are build upon.

5.9.1. Graph Representation of a Network Topology

As stated in section 5.6, we represent a network topology as a graph where hosts are nodes and physical connections between hosts are links. Properties in either physical links or hosts, such as MAC or IP are represented as labels in the corresponding graph element.

5.9.2. Haikunet Inference Rules

The operational semantics of Haikunet is defined as a transition relation between configurations. The relation is inductively defined by a set of inference rules. Each rule defines a reduction from one configuration to another one under certain conditions. We define a configuration in Haikunet as a pair. The first element of the pair denotes a program expression (see section 5.7), meanwhile the second element is a graph that represents the network topology.

The first inference rules concerns *Host* assignment.

(HOST ASSIGNMENT)

$$\frac{E \downarrow \mathbf{m} \qquad G \xrightarrow{host(mac=\mathbf{m})} G'}{< V := Host(mac = E); P, G > \rightarrow < P, G' >}$$

Host Assignment states that the assignment *V := Host(mac=E)* is executed by: 1. evaluating expression E to its normal form **m** (this is denoted by the down-side arrow) and 2. transforming G to G' accordingly to the graph grammar rules in section 5.9.3. When the pre-condition is achieved, then *Host Assignment* post-condition affirms that configuration $<V := Host(mac= E)\ ;\ P\ ,\ G>$ is reduced to $<\ P\ ,\ G'\ >$, where P denotes the remaining program expressions (see 5.5), and G' the result of performing *host(mac=**m**)* graph rewriting rule over G.

(FLOW ASSIGNMENT)

$$\frac{E_1 \downarrow \mathbf{h_1} \qquad E_2 \downarrow \mathbf{h_2} \qquad E_3 \downarrow \mathbf{n} \qquad G \xrightarrow{flow(src=\mathbf{h1},dst=\mathbf{h2},priority=\mathbf{n})} G'}{< V := Flow(src = E_1, dst = E_2, priority = E_3); P, G > \rightarrow < P, G' >}$$

Flow Assignment states that assignment $V := Flow(src = E_1, dst = E_2, priority = E_3)$ is executed by: 1. evaluating expressions E_1 and E_2 to their host normal forms (this is denoted by the down-side arrow), 2. evaluating expression E_3 to a number normal form and 3. transforming G to G' accordingly to the graph grammar rules in section 5.9.3. When the pre-condition is achieved, then *Flow Assignment* post-condition states that configuration $< V := Flow(src = E_1, dst = E_2, priority = E_3); P, G >$ is reduced to $< P\ ,\ G' >$, where P denotes the remaining program expressions (see 5.5), and *G'* the result of performing $flow(src = \boldsymbol{h1}, dst = \boldsymbol{h2}, priority = \boldsymbol{n})$ graph rewriting rule over G.

Next inference rule concerns intent creations.

(INTENT CREATION)

$$\frac{X \downarrow \mathbf{s} \qquad E \downarrow \mathbf{f}}{< Intent\ X\ Select\ E; P, G > \rightarrow < P, G >}$$

Intent Creation states that an intent creation *Intent X Select E* is executed by: 1. evaluating expression X to a String normal form and 2. evaluating expression E to a Flow normal form. When the pre-condition is achieved, then *Intent Creation* post-condition states that configuration $< Intent\ X\ Select\ E\ ;\ P\ ,\ G >$ is reduced to $< P\ ,\ G >$, where P denotes the remaining program expressions (see 5.5). We would like to remark that *Intent Creation* does not has a graph rewriting rule associated to it. This happens because the semantic of *Intent Creation* is currently the same as *Flow Creation*.

We will consider a Haikunet program correct whenever the program can be reduce from an initial configuration $< P, G >$ to $< \emptyset, G >$ by use of the previous inference rules.

5.9.3. Graph Rewriting Rules

Graph rewriting rules introduced in this section follow the double pushout approach.

We start with the rules for host definition, namely *host(mac=aMac)*. The argument *aMac* represents a mac address.

Fig. 5.6: host(mac=aMac) Graph Rewriting Rule: Creating a Host

There are two cases. The rule detailed in Figure 5.6 states that a host with property *aMac* can be created in the current graph whenever there is no host with the same MAC within the graph.

Fig. 5.7: host(mac=aMac) Graph Rewriting Rule: Denoting an existing Host

On the contrary, when the host with property *aMac* already exists within the graph, then *host(mac=aMac)* has no effect.

We now introduce the flow definition rule, namely *flow(src=aSrc, dst=aDst ,priority=aPriority)*. The arguments *aSrc* and *aDst* represent hosts, meanwhile *aPriority* represents a number.

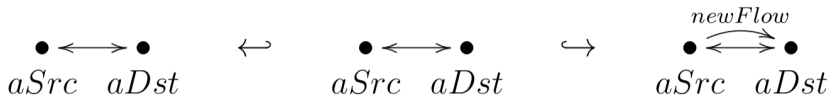


Fig. 5.8: flow(src=aSrc, dst=aDst ,priority=aPriority) Graph Rewriting Rule: Creating a new Flow

This rule states that if a path exist between hosts *aSrc* and *aDst*, then a new flow between them can be created. In this rule we have the following annotation abuses:
1. A two-headed arrow between hosts *aSrc* and *aDst* represents an existing path between them, i.e., a set of physical links connect both hosts.
2. The link *newFlow* represents a flow between the hosts instead of a physical connection, i.e., traffic data being sent from *aSrc* to *aDst*.

The advantages of having an operational semantic over Haikunet is the application of verification techniques over the programming language. In section 5.10.2 we use the presented rules over scenarios shown in section 5.5 to detect errors.

5.10. Categorizing Errors

This section categorizes errors previously defined in section 5.5. Categorizing errors provides insights about the implementation of the semantic checker. At the end of this section, error examples given in section 5.5 are analyzed accordingly to this characterization.

5.10.1. Definitions

5.10.1.1. Correct Program Computation

As stated in section 5.9.2, a Haikunet configuration $< P_0; G_0 >$, is said to be correct whenever P_0 can be reduced to $\emptyset$. This is:

$$< P_0; G_0 > \rightarrow < P_1; G_1 > \rightarrow ... \rightarrow < P_k; G_k > \rightarrow ... \rightarrow < \emptyset; G' >$$

Where each right arrow represents a reduction performed over a configuration by using the presented inference rules. $< \emptyset; G' >$ denotes the terminated configuration.

5.10.1.2. Errors

Given a configuration $< P_0; G_0 >$, an error is produced whenever $< P_0; G_0 >$ cannot be reduced to the terminated configuration. This can be summarized as follows:

$$< P_0; G_0 > \rightarrow < P_1; G_1 > \rightarrow ... \rightarrow < P_k; G_k > \not\rightarrow$$

Where $< P_k; G_k > \not\rightarrow$ represents that there is no inference rule applicable and $P_k \neq \emptyset$.
Errors presented in Haikunet configuration can be categorized in either static or dynamic errors. Depending on the category of the error, the semantic checker will behave differently. A formal definition of the mentioned categories is presented next.

5.10.1.3. Static Errors

A static error in a configuration happens when the error presented is related to the network topology structure. Examples of these types of errors are: Flow property Definition Error(5.5.1) and No path Error(5.5.2).

5.10.1.4. Dynamic Error

A dynamic error in a configuration happens when the error is triggered by monitoring traffic flow in the network. An example of this kind of error is when the buffers of some specific routers drop more that 40 % of packets, and a maximum threshold of 30 % was set. These errors are not characterized by the presented inference rules. Despite, Haikunet semantic checker was built considering them. We leave as future work the characterization of these types of errors with inference rules.

5.10.2. Formalizing Errors Using Inference Rules

This section uses Haikunet operational semantics (detailed in section 5.9) to detect errors in the examples presented in section 5.5.

5.10.2.1. Flow Property Definition Error

In the program example presented in section 5.5.1 we stated the following error: IP *"127.0.0.10"* does not reflect a property of a host in the current graph. In this section we want to identify this error by applying Haikunet inference rules over the program example.

Scenario 5.5.1 is transformed to the following configuration: $P =< source\_host := Host(mac = \text{"00:00:00:00:00:02"}); my\_flow := (src = source\_host, dst = \text{"127,0,0,10"}, priority = \text{"55"}); Intent\,firstExampleSelect my\_flow,\ G >$ where G stands for the initial network state. The first computation of this configuration is shown below:

$$< source\_host := Host(mac = \text{"00:00:00:00:00:02"}); P_0, G > \rightarrow_{HostAssignmentRule}$$

$$< my\_flow := (src = source\_host, dst = \text{"127,0,0,10"}, priority = \text{"55"}); P_1, G >$$

This computation applies *Host Assignment* rule over *source_host:=Host(mac="00:00:00:00:00:02")* expression. Because mac address *"00:00:00:00:00:02"* already exist in the network, *host(mac=aMac)* graph rewriting rule is used as the identity function, leaving G as it was before. The result is configuration $< my\_flow := (src = source\_host, dst = \text{"127,0,0,10"}, priority = \text{"55"}); P_1, G >$.

The expression that follows in the current configuration is $my\_flow := (src = source\_host, dst = \text{"127,0,0,10"}, priority = \text{"55"})$. When using *Flow Assignment* rule over the expression, the error is detected. The problem is raised because *flow(src=aSrc, dst=aDst ,priority=aPriority)* graph rewriting rule pre-condition is not fulfilled. This happens because there is no node in the graph matching the label *IP "127.0.0.10"*.

5.10.2.2. No Path Error

In the program example presented in section 5.5.2 we stated the following error: When trying to create a flow between hosts with IPs *"127.0.0.2"* and *"192.168.1.3"*, no path between them was detected. We now want to identify this error by applying Haikunet inference rules over the program example.

The mentioned program is transformed to the following configuration: $P =< source_host := Host(ip = \text{"127,0,0,2"}); destiny\_host := Host(ip = \text{"192,168,1,3"}); my\_flow := Flow(src = source\_host, dst = destiny\_host, priority = \text{"55"}); Intent\,firstExampleSelect my\_flow >$ where G stands for the initial network state. The first two computations of this configuration are shown below:

$$< source_host := Host(ip = \text{"127,0,0,2"}); P_0, G > \rightarrow_{HostAssignmentRule}$$

$$< destiny\_host := Host(ip = \text{"192,168,1,3"}); P_1, G > \rightarrow_{HostAssignmentRule}$$

$$< my\_flow := Flow(src = source\_host, dst = destiny\_host, priority = \text{"55"}); P_2, G >$$

The computations above apply *Host Assignment* rule over *source_host:=Host(ip="127.0.0.2")* and *destiny_host:=Host(ip="192.168.1.3")* expressions. Since both IPs already exist in the network, *host(mac=aMac)* graph rewriting rule is used as the identity function, leaving G as it was before. The result is configuration $< my\_flow := Flow(src = source\_host, dst = destiny\_host, priority = \text{"55"}); P_2, G >$.

The expression that follows in the current configuration is $my\_flow := Flow(src = source\_host, dst = destiny\_host, priority = \text{"55"})$. When using *Flow Assignment* rule over the expression, the error is detected. The problem is raised because *flow(src=aSrc, dst=aDst ,priority=aPriority)* graph rewriting rule pre-condition is not fulfilled. This happens because nodes representing *source_host* and *destiny_host* have no path between them. Because of this, no injective morphism exist between G and the left-hand side graph of *flow(src=aSrc, dst=aDst ,priority=aPriority)* rule, making impossible to use it over G.

5.11. Architecture

Figure 5.9 outlines Haikunet modular decomposition.

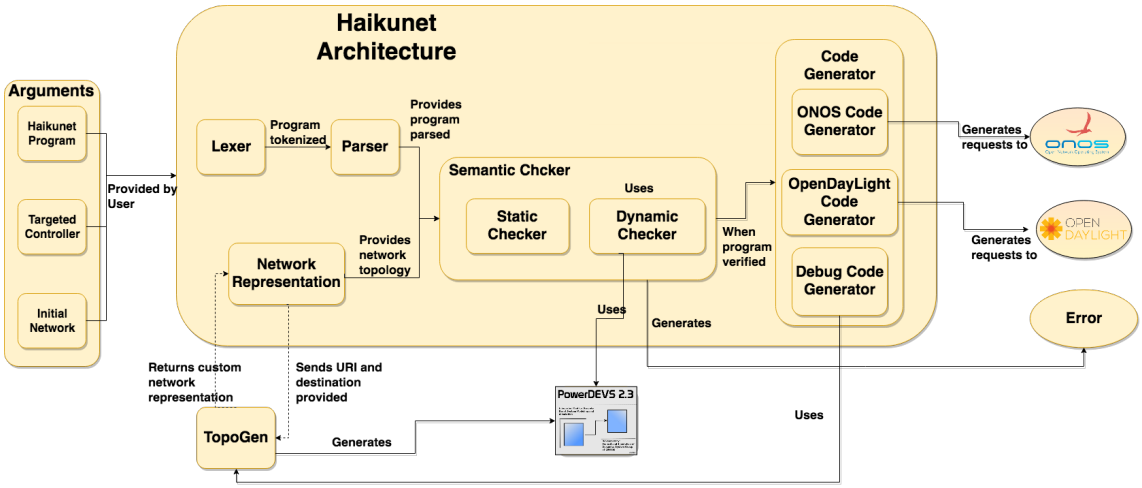


Fig. 5.9: Haikunet Architecture

Module Arguments identifies the three arguments provided by a user to Haikunet, which are: 1. a path to a file which contains a Haikunet program. Haikunet program files are denoted by the *.hk* extension, 2. a name to the targeted controller (currently supported controllers are ONOS and OpenDayLight), 3. either a URI of a controllers API (denoting the resource where the network can be retrieved from) or a path to a NTM file.

Haikunet has a pipeline architecture. First, the lexeme (i.e., the sequence of strings that represent a program) is retrieved from the file received as argument. Second, the lexeme is tokenized by the module Lexer. Third, the tokenized lexeme is parsed by the module Parser. Fourth, component Semantic Checker performs verification techniques over an abstract syntax tree and a network representation retrieved from components Parser and Network Representation respectively. Fifth and finally, module Code Generator create requests to the received targeted controller.

The mentioned modules are deeply describe in the following sections.

5.11.1. Lexer

Is the module in charge of tokenize the program lexeme. Haikunet has a context-free Grammar, and this module can be seen as a push down automaton. In case a program lexeme is badly formed, a lexical error is raised.

5.11.2. Parser

Haikunet implements a LL(1) parser that uses the syntax-direct grammar technique. In this technique, each production is converted to a function that has the responsibility of parsing the token received. The mentioned functions can raise two type of errors: syntactical errors and semantical errors. A syntactical error is raised when an unexpected token is received, e.g., when programs are badly written. On the other hand, a semantical error is raised when an identifier is not found in a program declaration, i.e., an identifier is used in a program, but was never declared. An example of this last error can be seen in the following program:

```
1 target := Host(ip="192.168.1.3")
2
3 my_flow := Flow (src=source, dst=target, priority="55")
4
5 Intent firstExample
6     Select my_flow
```

In the program above, *source* is an identifier used in a Flow declaration which was not defined in the program.

When the program is successfully parsed, this module creates a map between expressions and class instances that represent an entity in a program (e.g.,Hosts, Flows), which is later used by modules Semantic Checker and Code Generator.

5.11.3. Network Representation

This module is a proxy between Haikunet and TopoGen. Module Network Representation uses TopoGen as a Ruby gem to retrieve a graph representation of the underlying network. TopoGen's output is loaded by this module, which creates an in-memory representation of the network topology. This in-memory representation is used by other modules.

5.11.4. Semantic Checker

This module executes verification techniques that are divided in: 1. Static verification techniques and 2. Dynamic verification techniques.

Static verification techniques are performed over the graph representation created by module Network Representation and the map created by module Parser. Module Semantic checker implements methods per each error listed in section 5.5. Each method uses the input from the other modules to detect inconsistencies in the graph and the operations trying to be performed by the program. In case an error is found, a semantic exception is raised and the error detected is detailed to the user. Otherwise, the dynamic verification techniques are executed over the program.

Dynamic verification techniques start by creating a DEVS representation of the network topology. This task is performed by invoking the TopoGen tool via module Debug Code Generator. Haikunet features custom topology builders, where pre-built models are used for the creation of the DEVS simulation model. Once this representation is created, a simulation is executed by invoking the PowerDEVS toolkit. When the simulation ends, PowerDEVS sends the results to Scilab numerical environment for post-processing purposes. The Scilab process invokes pre-built functions showing the user results of having executed the simulation over the current network. As how it is currently implemented, it is an user responsibility to identify if the results given by the simulator have an error.

Simulation was used for the detection of dynamic errors, which cannot be detected analytically with the information available in the model. The simulation of a network allow us to study salient network events that can occur during a custom period of time. Since dynamic errors are defined by monitoring traffic over the network, it is in the need of the dynamic logic to have traffic data in the network. Measurements can be performed on these data to detect situations where thresholds are surpassed. This task can be easily performed using a simulator such as PowerDEVS.

Several features remain as future work, such as the implementation of graph grammar rules to be used by the semantic checker, the automation of detection of dynamic errors (by querying the simulator for violations of thresholds), and make programs capable of expressing dynamic properties, among others (see the Future Work section for more details)

5.11.5. Code Generator

This module can be divided in 3 submodules: 1. Submodule ONOS Code Generator, 2. Submodule OpenDayLight Code Generator and 3. Submodule Debug Code Generator. The first two submodules encapsulate the logic of creating requests to the respectively controller. These requests are created by using the map built in module Parser. Submodules ONOS Code Generator and OpenDayLight Code generator are used after the successful execution of module Semantic Checker.

Submodule Debug Code Generator is used by module Semantic Checker in the process of creating the simulation. This submodule executes operations from the intent that modify the network topology structure (e.g. adding hosts to the network) over the in-memory network representation provided by module Network Representation. After performing these operations, submodule Debug Code Generator provides the in-memory network representation as argument of TopoGen by using provider OBJECT and builder PowerDEVS (see Chapter 4).

Module Code Generator makes Haikunet an agnostic intent programming language regarding the targeted controller. Next section explains in more detailed how is the process of adding a new targeted controller.

5.12. Class Diagram

This section is concerned to explain Haikunet class diagram illustrated in Figure 5.10.

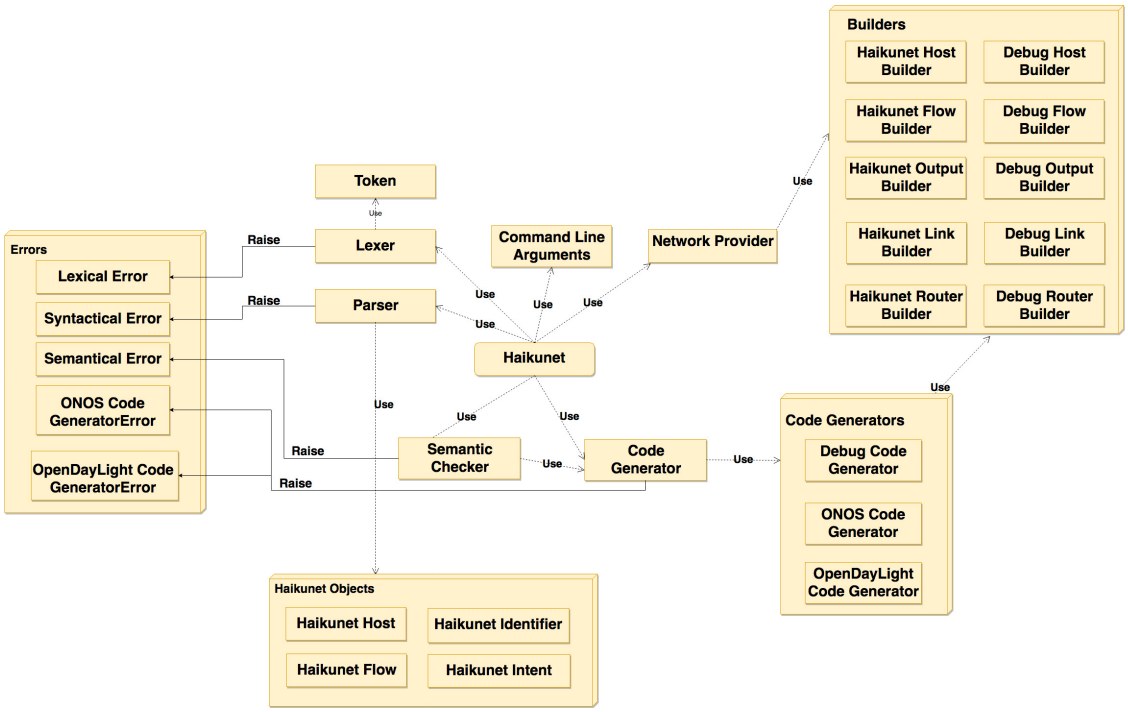


Fig. 5.10: Haikunet Class Diagram

The initial point of the code execution starts in *Haikunet* class. This class acts as an orchestrator of all other classes. When a *Haikunet* instance is created, a *Command Line Argument* is instantiated. Class *Command Line Argument* contains the command line logic for using Haikunet as a binary, and implements verifications over the arguments provided by the user (these arguments

were introduced in section 5.9). This class uses the *Commander* gem (Ref [1]) to implement the mentioned service.

Class *Lexer*, *Parser*, *Semantic Checker* and *Code Generator* can be mapped directly to their respective modules presented in the Architecture module viewtype in figure 5.9. Module *Network Representation* from figure 5.9 can be mapped to class *Network Provider*. This class uses TopoGen to generate and instance of NTM, and for doing so it uses pre-defined builders which are called *Haikunet Builders* (see Figure 5.10). These builders are particular cases of NTM Builders presented in Chapter 4. When the NTM instance is created, this class loads the NTM instance and returns the topology loaded. The parameters provided to TopoGen from Network Topology class are 1. Haikunet custom builders, 2. either a URI of a controllers API or a path to a NTM file, 3. the name of the targeted controller, and 4. an output folder.

Classes that belong to Haikunet Objects are elements used for mapping expressions in the language to internal representations. For instance, Host and Flow expressions used in a program are first parsed, and then an instance of *Haikunet Host* and *Haikunet Flow* are created respectively per each expression. When a variable expression is defined in a program, a new instance of *Haikunet Identifier* is created. Finally, when an intent is defined in a program, a new instance of *Haikunet Intent* is created. This mapping between expressions and classes is done within the parsing process.

In section 5.11, when Code Generator was introduced, ONOS and OpenDayLight submodules were presented. This submodules can be mapped into the *ONOS Code Generator* and the *OpenDayLight Code Generator* classes respectively. Both classes implement the logic of creating a bunch of requests to the targeted controller which their represent. In order to augment the targeted controllers, a new class must be implemented. Code Generators classes receive the targeted file name argument, and the context explained in section 5.11.4. Each class must implement the *generate_output* method, which receives the targeted file name as argument. This method is expected to return the desired output. This logic can be implemented because class Code Generator implements a Strategy Pattern.

Debug Code Generator class is used within the Semantic Checker to create the DEVS model which is later used by PowerDEVS. This code generator uses TopoGen (as a ruby gem) to generate from the Ruby-based network topology obtained from class *Network Provider*, a DEVS model. This model is built using the Debug Builders classes depicted in Figure 5.10. *Debug Code Generator* provides as argument to TopoGen: 1. *OBJECT* as a source; 2. The directory of the mentioned builders; 3. The topology obtained from class *Network Provider* and 4. An output directory.

5.13. Installation and Use of Haikunet

Haikunet can be installed in a Linux environment by running the following command:

```
1
2 \curl -sSL https://raw.githubusercontent.com/andyLaurito92/haikunet/
    master/download_directory.sh | bash
```

Once installed, Haikunet is meant to be used as a binary. In order to execute Haikunet, either an ONOS or OpenDayLight distribution must be installed and accessible from the installation computer.

For example, having installed ONOS locally, and assuming that the first program example introduce in this section is in a file called *firstProgramExample.hk*, Haikunet can be executed as follows:

```
1
2 haikunet -n firstProgramExample.hk -d ONOS -u http://127.0.0.1:8181/
    onos/v1/
```

In this example, *http://127.0.0.1:8181/onos/v1/* represents the ONOS local API. More examples and information are accessible by command line to the user by executing the help command. This can be done by executing this:

```
1
2 haikunet -h
```

5.14. Conclusions

In this chapter we presented Haikunet, an intent-oriented programming language, agnostic to controllers, which implements a semantic checker that performs verification techniques. These verification techniques are divided in: 1. Static verification techniques, concerning the topology structure of the network, and 2. Dynamic verification techniques, concerning the traffic data being sent in the network.

We also introduced a type system and operational semantics for the mentioned programming language. These operational semantics were given in two steps: by a set of graph rewriting rules over the network topology and a set of reduction rules over the program expressions. Haikunet operational semantic rules were successfully used for detecting errors in program. These features presented are currently not implemented because a matter of time, and their implementation are left-out as future work.

The study performed in this chapter crates new opportunities for performing verification techniques on intent-oriented programming languages. For instance, model checkers can used the presented operational semantic rules to detect if given specifications are achieved in Haikunet programs.

Finally, we would like to remark that Haikunet was successfully tested in ONOS and OpenDayLight controllers, avoiding in this way time-consuming troubleshooting scenarios.

In the next section we present future steps to work in Haikunet.

5.15. Future Work

In this section we present left-out works because a matter of time, and desired works to keep improving the language.

5.15.1. Change the Current Grammar to be SLA-oriented

The management plane of the SDN architecture usually works within SLA. Haikunet is an Intent-oriented programming language though for the management plane. Is because of this, that Haikunet grammar should be more SLA-oriented. Users of this programming languages should be abstracted from the network as much as possible. An approach for tackling this issue would be to re-write the grammar using a SLA standard notation. Haikunet internal representation should then be modified to do the mapping between a SLA and the corresponding Intent.

5.15.2. Extend the Expressive Power

Intents that can be expressed with Haikunet are very limited. Most of the Intents that can be written are not useful for a real/practice scenario. Extending the expressive power would help the language to grow and be used in real scenarios. A starting point is to extend the expressive power of the language to model dynamic properties such as the following one:

hello_world.hk

```
1 Intent firstExample
2     Select my_flow
3     Condition my_flow.packet_drops > 30%
4     Action my_flow.decreasePacketWindow(50%)
```

In the example above, when my_flow reaches a loss packet percentage of 30%, the intent describes that is expected that the flow decreases the packet window in a 50%.

5.15.3. Improve the Static Verification Techniques of the Semantic Checker

In this chapter, operational semantics and a type system for Haikunet were presented but not implemented. Next step is to extend module Semantic Checker to implement both features provided. An approach for implementing the operational semantics given is to adapt module Semantic Checker to use a library which implements Graph Grammar rewriting rules. An example of one of these libraries is GraphGen [2]. GraphGen is a library implemented in Python which allows the generation and definition of generative Graph Grammars.

5.15.4. Improve the Dynamic Verification Techniques of the Semantic Checker

The dynamic verification techniques implemented in module Semantic Checker are currently performed by the user by verifying Scilab output. Next steps in the improvement of dynamic verification techniques concern 1. the automation of detection of dynamic errors by querying the simulator for violations of thresholds, and 2. the implementation of SDN-oriented DEVS models (i.e., DEVS models that represent SDN flows, OpenFlow, SDN switches).

5.15.5. Augment Supported Targeted Controllers

Even currently supported targeted controllers are ONOS and OpenDayLight, there are still a big variety of controllers existing nowadays in the SDN realm that are not supported by Haikunet. Some examples are the OpenFlow controller and NOX. Extending Haikunet supported controllers will allow the language to target new scenarios.

6. CONCLUSIONS AND FUTURE WORK

In this thesis we presented Haikunet, and intent-oriented programming language, agnostic to controllers, that implements a semantic checker to verify intents before being applied to a network. We also introduced TopoGen, a general-purpose tool for topology serialization in the SDN paradigm which is independent of Haikunet.

We show how Haikunet can be used for detecting errors in intents before applying them to the underlying network. This is a major advantage, since intents can produce downtime, time-consuming troubleshooting and availability impact. The use of Haikunet in real scenarios was not tested, and is left-out as future work. The presented operational semantics and the usage of PowerDEVS in the semantic checker creates new possibilities to explore. For instance, model checking techniques could be applied by relying on the presented operational semantics.

TopoGen has been successfully used in Haikunet and in the CERN ATLAS-TDAQ networking team. In the first scenario, TopoGen proved to be useful for several vendor controllers, such as ONOS and OpenDayLight. Besides, the usage of TopoGen in Haikunet allows Haikunet to be used when the network topology is not physical but virtually implemented. This is due to the fact that the Network Topology Model (NTM) is implemented in TopoGen. In the second scenario, TopoGen has been used to reduce modeling time in simulations and allow the manipulation of big-size network topologies.

Different existing SDN problems were encountered in this thesis, such as the lack of a common interface for Northbound API's, the lack of verifications over intents before they are actually applied to a network, standardization issues when topology network structures are retrieved from different controllers. The work presented in this thesis opens different paths for future work, and we hope that work helps in improving the SDN paradigm.

Bibliografía

[1] Commander. https://github.com/commander-rb/commander, 2017.

[2] Robert Adams. Graphgen. https://github.com/drobertadams/GraphGen, 2015.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *ACM SIGPLAN Notices*, volume 49, pages 113–126. ACM, 2014.

[4] J Anderson, A Borga, H Boterenbrood, H Chen, K Chen, G Drake, D Francis, B Gorini, F Lanni, G Lehmann Miotto, et al. Felix: A high-throughput network approach for interfacing to front end electronics for atlas upgrades. In *Journal of Physics: Conf. Series*, volume 664, page 082050. IOP, 2015.

[5] ATLAS Collaboration. The atlas experiment at the cern large hadron collider. *Journal of Instrumentation*, 3(08):S08003, 2008.

[6] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[7] Federico Bergero and Ernesto Kofman. Powerdevs: a tool for hybrid system modeling and real-time simulation. *Simulation*, 87(1-2):113–132, 2011.

[8] M Bjorklund. Yang-a data modeling language for the network configuration protocol. RFC 6020, IETF, October 2010.

[9] Matías Bonaventura, Daniel Foguelman, and Rodrigo Castro. Discrete event modeling and simulation-driven engineering for the atlas data acquisition network. *Computing in Science & Engineering*, 18(3):70–83, 2016.

[10] Stephen L Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in SCILAB*. Springer, 2006.

[11] Gustavo Carneiro. Ns-3: Network simulator 3. In *UTM Lab Meeting April*, volume 20, 2010.

[12] R Castro and E Kofman. An integrative approach for hybrid modeling, simulation and control of data networks based on the devs formalism. In *Modeling and Simulation of Computer Networks and Systems: Methodologies and Applications*, chapter 18. Morgan Kaufmann, 2015.

[13] Xinjie Chang. Network simulations with opnet. In *Proceedings of the 1999 Winter Simulation Conference*, pages 307–314, Piscataway, New Jersey, 1999. Institute of Electrical and Electronics Engineers, Inc.

[14] J Choi. Network working group mk. shin internet-draft kh. nam intended status: Informational etri expires: December 2012 m. kang. 2012.

[15] Rogério Leão Santos De Oliveira, Ailton Akira Shinoda, Christiane Marie Schweitzer, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pages 1–6. IEEE, 2014.

[16] Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld. *Graph-grammars and their application to computer science: 3rd international workshop, Warrenton, Virginia, USA, December 2-6, 1986*, volume 3. Springer Science & Business Media, 1987.

[17] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM Sigplan Notices*, volume 46, pages 279–291. ACM, 2011.

[18] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[19] Evangelos Haleplidis, Kostas Pentikousis, Spyros Denazis, J Hadi Salim, David Meyer, and Odysseas Koufopavlou. Software-defined networking (sdn): Layers and architecture terminology. Technical report, 2015.

[20] Yannan Hu, Wendong Wang, Xiangyang Gong, Xirong Que, and Shiduan Cheng. Balanceflow: controller load balancing for openflow networks. In *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*, volume 2, pages 780–785. IEEE, 2012.

[21] Naga Praveen Katta, Jennifer Rexford, and David Walker. Logic programming for software-defined networks. In *Workshop on Cross-Model Design and Validation (XLDI)*, volume 412, 2012.

[22] Simon Knight, Hung X Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[23] Guilherme Piegas Koslovski, Pascale Vicat-Blanc Primet, and Andrea Schwertner Charao. Vxdl: Virtual resources and interconnection networks description language. In *Intl. Conf. on Networks for Grid Applications*, pages 138–154. Springer, 2008.

[24] Vasileios Kotronis, Xenofontas Dimitropoulos, and Bernhard Ager. Outsourcing the routing control logic: better internet routing based on sdn principles. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 55–60. ACM, 2012.

[25] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[26] Andres Laurito, Rodrigo Daniel Castro, Matias Alejandro Bonaventura, Pozo Astigarraga, and Mikel Eukeni. Topogen: A network topology generation architecture with application to automating simulations of software defined networks. Technical report, ATL-COM-DAQ-2017-026, 2017.

[27] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[28] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[29] Laura Victoria Morales, Andres Felipe Murillo, and Sandra Julieta Rueda. Extending the floodlight controller. In *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*, pages 126–133. IEEE, 2015.

[30] Eugene D Ngangue Ndih and Soumaya Cherkaoui. Simulation methods, techniques and tools of computer systems and networks. In Mohammad S Obaidat, Faouzi Zarai, and Petros Nicopolitidis, editors, *Modeling and Simulation of Computer Networks and Systems: Methodologies and Applications*, chapter 17. Morgan Kaufmann, 2015.

[31] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[32] Minh Pham and Doan B Hoang. Sdn applications-the intent-based northbound interface realisation for extended applications. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 372–377. IEEE, 2016.

[33] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[34] Mikel Pozo Astigarraga, Eukeni ATLAS Collaboration, et al. Evolution of the atlas trigger and data acquisition system. In *Journal of Physics: Conf. Series*, volume 608, page 012006. IOP, 2015.

[35] Luca Prete, Fabio Farina, Mauro Campanella, and Andrea Biancini. Energy efficient minimum spanning tree in openflow networks. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 36–41. IEEE, 2012.

[36] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 109–114. ACM, 2013.

[37] George F Riley and Thomas R Henderson. The ns-3 network simulator. *Modeling and tools for network simulation*, pages 15–34, 2010.

[38] Grzegorz Rozenberg. *Handbook of Graph Grammars and Comp.*, volume 1. World scientific, 1997.

[39] Jorn Schumacher, Christian Plessl, and Wainer Vandelli. High-Throughput and Low-Latency Network Communication with NetIO. In *22nd International Conf. on Computing in High Energy and Nuclear Physics, CHEP 2016*. IOP.

[40] Jeroen Van der Ham, Paola Grosso, Ronald Van der Pol, Andree Toonk, and Cees De Laat. Using the network description language in optical networks. In *Integrated Network Management, 2007. im'07. 10th IFIP/IEEE International Symposium on*, pages 199–205. IEEE, 2007.

[41] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conf. on Simulation tools and techniques for communications, networks and systems*, page 60. ICST, 2008.

[42] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[43] András Varga and György Pongor. Flexible topology description language for simulation programs. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany*, pages 225–229, 1997.

[44] Jinesh Varia and Sajee Mathew. Overview of amazon web services. *Amazon Web Services*, 2014.

[45] Andreas Voellmy, Ashish Agarwal, and Paul Hudak. Nettle: Functional reactive programming for openflow networks. Technical report, YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, 2010.

[46] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 43–48. ACM, 2012.

[47] Klaus Wehrle, Mesut Günes, and James Gross. *Modeling and Tools for Network Simulation*. Springer, 2010.

[48] Xia Yinben. NEMO: A network modeling language.

[49] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic press, 2000.