



**Pós-Graduação em Ciência da Computação**

**MODEL-DRIVEN NETWORKING: A NOVEL  
APPROACH FOR SDN APPLICATIONS  
DEVELOPMENT**

**Por**

***FELIPE ALENCAR LOPES***

**Dissertação de Mestrado**



Universidade Federal de Pernambuco  
posgraduacao@cin.ufpe.br  
www.cin.ufpe.br/~posgraduacao

RECIFE, FEVEREIRO/2015



Universidade Federal de Pernambuco

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**FELIPE ALENCAR LOPES**

***MODEL-DRIVEN NETWORKING: A NOVEL  
APPROACH FOR SDN APPLICATIONS  
DEVELOPMENT***

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

**ORIENTADOR: Prof. Stênio Flávio de Lacerda Fernandes**

**CO-ORIENTADOR: Prof. Stênio Flávio de Lacerda Fernandes**

**RECIFE, FEVEREIRO/2015**

## FICHA CATALOGRÁFICA

**Lopes, Felipe Alencar.**

*Model-Driven Network: a Novel Approach for SDN Applications Development* / Felipe Alencar Lopes. – Recife, 2015. viii, XX folhas: il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn – Ciência da Computação, 2015.

Inclui bibliografia.

1. Tecnologia da informação – Ciência da informação. I.  
Título.

004

CDD (22. ed.)

MEI2009-016

Dissertação de Mestrado apresentada por **Felipe Alencar Lopes** à Pós Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “**Model-Driven Networking: A Novel Approach for SDN Applications Development**”, orientada pelo **Prof. Stênio Flávio de Lacerda Fernandes** e aprovada pela Banca Examinadora formada pelos professores:

---

Prof. José Augusto Suruagy Monteiro  
Centro de Informática/UFPE

---

Profa. Patricia Takako Endo  
Universidade de Pernambuco

---

Prof. Stênio Flávio de Lacerda Fernandes  
Centro de Informática / UFPE

Visto e permitida a impressão.  
Recife, 27 de fevereiro de 2015.

---

**Profa. Edna Natividade da Silva Barros**  
Coordenadora da Pós-Graduação em Ciência da Computação do  
Centro de Informática da Universidade Federal de Pernambuco.

*À minha família.*

# Agradecimentos

Primeiramente, agradeço a Deus por este momento e por todas as vezes que Ele me mostrou um caminho a seguir, apesar de qualquer adversidade. Agradeço também a minha família, em especial a minha mãe Regilvane e minha irmã Rebeka que me apoiaram desde sempre na realização desta e de tantas outras etapas da minha vida. Preciso agradecer ainda a mais uma mulher especial, minha amiga, companheira e namorada, Bruna, por todo o apoio e compreensão que tanto me ajudaram a passar pelos altos e baixos dessa jornada.

Preciso destacar a minha gratidão ao meu orientador Stênio Fernandes, não só pela oportunidade de ter sido seu aluno e orientando, mas pelos ensinamentos, paciência, conselhos e incentivos que foram fundamentais para que esse momento fosse possível. Além dele, eu agradeço também ao meu co-orientador Robson Fidalgo por todas as contribuições, dicas e conversas de laboratório que me possibilitaram concluir este trabalho.

Agradeço aos professores do CIn, ressaltando Fernando Castor e Djamel Sadok, pela geração de conhecimento e motivação resultante de suas aulas.

Gostaria ainda de agradecer a todos meus professores do IFAL, em especial a prof<sup>a</sup>. Mônica Ximenes que teve uma enorme parcela de contribuição nesta fase da minha vida acadêmica. Outros professores do IFAL que merecem meu agradecimento são os meus ex-companheiros de apartamento em Recife, o prof. Fernando Kenji e a prof<sup>a</sup>. Eunice Palmeira, os quais, em meio a viagens, conversas e muita ajuda, estiveram comigo durante boa parte dessa trajetória.

Agradeço também: aos amigos de faculdade do IFAL (Toni, Gilton, Handrik, Geraldo, Uziel, Pedro, Valter e André), pelos vários momentos de descontração; aos amigos do antigo CEFET pela amizade de sempre, em especial ao meu velho amigo, e recentemente companheiro de apartamento, Nicolas Alexandre; e aos amigos e colegas do Residencial Universitário (Jean, Douglas, Mailson, sem falar do pessoal do racha de seg. e qua.) e do CIn, que tanto me ajudaram em várias ocasiões (Marcelo Santos, Marcelo Iury, Maria Silvia, Wesley e Edson).

Gostaria de agradecer ainda a todos aqueles que direta ou indiretamente contribuíram de alguma forma para que esse sonho se tornasse realidade.

Por fim, agradeço a FACEPE – Fundação de Amparo à Ciência e Tecnologia do Estado de Pernambuco – por financiar este trabalho.

# Acknowledgement

First, I thank God for this moment and for all the times He showed me a way forward, despite any adversity. I also thank my family, especially my mother, Regilvane Alencar, and my sister, Rebeka Alencar, who always supported me to realize this and so many other stages of my life. I must also thank to more one special woman. My friend, companion, and girlfriend, Bruna Albernaz, for all the support and understanding that helped me through the ups and downs of this journey.

I need to state my gratitude to my advisor Stênio Fernandes, I was fortunate not only for the opportunity to have been his student, but by teaching, patience, advice and incentives that were essential in making this moment possible. Besides him, I also thank my co-advisor Robson Fidalgo. His contributions, tips, and our laboratory conversations enabled me to finish this work.

I thank the CIN teachers, emphasizing Fernando Castor and Djamel Sadok, due to the generation of knowledge and resulting motivation of their classes.

I would also like to thank all my teachers of IFAL, especially prof. Mônica Ximenes that had a huge portion of contribution at this stage of my academic life. Other IFAL teachers who deserve my thanks are my former fellow apartment in Recife, prof. Fernando Kenji and prof. Eunice Palm, which, among trips, conversations, and a lot of help, been with me for much of this trajectory.

I also thank the college friends IFAL (Toni, Gilton, Handrik, Geraldo, Uziel, Pedro, Valter and Andre) for the various moments of fun; the friends of the former CEFET, which are my friends since always, especially my old friend, and recently roommate, Nicolas Alexandre. I would also like to thank my friends and colleagues of the Residencial Universitário (Jean, Douglas, Maílson, and all the guys that play football there) and friends from CIn, which both helped me on several occasions (Marcelo Santos, Marcelo Iury, Maria Silvia, Wesley and Edson).

I would also like to thank all those who directly or indirectly contributed in some way to this dream come true.

Finally, I thank FACEPE - Foundation for Science and Technology of Pernambuco State - to fund this work.

*“Always pass on what you have learned.”*

- Yoda



# Resumo

As Redes Definidas por Software, ou *Software-Defined Networking* (SDN), têm recebido grande atenção de comunidades acadêmicas e indústria. Uma das razões para este interesse é que SDN permite a programação da rede, devido à sua arquitetura composta por um controlador externo que suporta o uso de linguagens de programação para a construção de aplicações, eliminando o tradicional acoplamento entre plano de controle e plano de dados. Dado que o desenvolvimento destas aplicações SDN ainda é complexo, existe uma forte necessidade de metodologias e ferramentas que permitam o uso de todo potencial de abstração suportado por estas redes. Focando neste problema, este trabalho apresenta uma nova abordagem, chamada *Model-Driven Networking* (MDN), para o desenvolvimento de aplicações e especificação de políticas SDN através da diagramação de modelos. A MDN baseia-se no paradigma de Engenharia de Software Baseada em Modelos, oferecendo uma Linguagem de Modelagem Específica de Domínio para criação dos modelos SDN executáveis. Para comprovar a relevância e a viabilidade tecnológica da proposta, também foi construída uma ferramenta de modelagem para a criação de aplicações SDN seguindo a abordagem MDN. Em uma comparação da MDN com outras abordagens, identificou-se diversos benefícios na utilização de MDN, além das principais funcionalidades necessárias para o desenvolvimento de aplicações e políticas SDN, tais como o suporte aos diversos controladores existentes e a validação das aplicações modeladas. Este trabalho conclui que MDN aumenta o nível de abstração no desenvolvimento de aplicações SDN, reduzindo a complexidade para implementar estas aplicações e ajudando a evitar comportamentos errôneos da rede.

**Palavras-chave:** Redes Definidas por Software. Engenharia de Software Baseada em Modelos. Linguagem de Modelagem Específica de Domínio

# Abstract

*Software-Defined Networking* (SDN) has been receiving a great deal of attention from both academic and industry communities. One reason for this interest is that SDN enables the network programmability, due to its architecture composed by an external controller, which supports the use of programming languages to build applications, breaking the traditional bind between control and data plane. Nevertheless, the application development is still complex for such recent technology. Moreover, there is a strong need for methodologies and tools that enable the utilization of all the level of abstraction supported by these networks. Focusing on such problem, this dissertation presents a new approach, named Model-Driven Networking (MDN), to enable the development of SDN applications and specification of network rules through models. The MDN is based on the Model-Driven Engineering (MDE) paradigm, offering a Domain-Specific Modeling Language (DSML) to create SDN models. In order to show the relevance and the technical viability of MDN, this dissertation proposes a modeling tool for creating SDN applications. When comparing MDN to other approaches, our results identify several benefits of using MDN besides the major functionality needed on developing SDN applications, such as the support for several controllers and the validation of applications. This dissertation argues that MDN raises the level of abstraction in the development, thus reducing the complexity to implement SDN applications, and prevents erroneous behavior of the network.

**Keywords:** Software-Defined Networking. Model-Driven Engineering. Domain-Specific Modeling Language

# List of Figures

Figure 1.1: Traditional network architecture (a) in relation to SDN architecture (b). .....	16
Figure 2.1: Overall perspective of the control layer in SDN architecture. ....	24
Figure 2.2: The necessary steps to the Client 1 communicate with the Client 2 when the switch does not have rules that can manage this communication. ....	26
Figure 2.3: Northbound and Southbound perspective of an SDN architecture. ....	27
Figure 2.4: SDN architecture from the OpenFlow perspective.....	28
Figure 2.5: Fields of an entry in flow table. ....	29
Figure 2.6: Pyretic code to monitor network packets. ....	31
Figure 2.7: Single-way process of model-driven development. ....	32
Figure 2.8: The application model conforms to a modeling language. Such language has its abstract syntax represented by a metamodel; which conforms to a language that has an abstract syntax represented by a meta-metamodel. ....	34
Figure 2.9: Bézivin's 3+1 MDA organization (BÉZIVIN, 2005).....	35
Figure 2.10: Scheme representing the organization of visual notation regarding the semantics of DSML as well as its metamodel.....	36
Figure 2.11: Visual notation proposed by Lennox <i>et al.</i> (2004).....	36
Figure 2.12: Summarized scheme of MDD and its components.....	38
Figure 2.13: Simplified set of the Ecore meta-metamodel. ....	41
Figure 2.14: EVL rules structure. ....	41
Figure 2.15: Workflow for developing graphical modeling editors using GMF. ....	42
Figure 3.1: The releasing of SDN programming languages in timeline.....	44
Figure 3.2: Graphical interface of Miniedit. ....	48
Figure 3.3 Topology model in VND.....	48
Figure 3.4: Class diagram of CIM-SDN. ....	49
Figure 4.1: Simplified view of the MDN Architecture.....	52
Figure 4.2: Example of MDN approach being implemented in Ecore instances. ....	55
Figure 4.3: MDN's core metamodel. ....	55
Figure 4.4: Full metamodel of MDN.....	57
Figure 4.5: Example of visual notation elements in a MDN diagram. ....	60
Figure 4.6: Semantic mapping of MDN.....	62
Figure 4.7: MDN workflow. ....	68

Figure 4.8: The MDN editor. ....	69
Figure 4.9: Steps in modeling a firewall application with INVALID model. ....	70
Figure 4.10: Steps in modeling a firewall application with VALID model. ....	70
Figure 4.11: The use of <i>pingall</i> command in Mininet to simulate the application modeled. .....	71
Figure 5.1: Pyretic code causing an infinite loop in network behavior. ....	76
Figure 5.2: The modeling of network monitor application. ....	77
Figure 5.3: Network-monitoring application output when Mininet's pingall command is called. ....	78
Figure 5.4 Topology for the access control application.....	80
Figure 5.5 Validation of topology for the use case #2.....	80
Figure 5.6 Modeling of first requirement. ....	81
Figure 5.7 Modeling of second requirement. ....	82
Figure 5.8 Modeling of third requirement. ....	82
Figure 5.9: Test involving the access control application and its first requirement. ....	84
Figure 5.10: Mininet commands to open consoles in each hosts. ....	84
Figure 5.11: Output of the simulation for the second requirement. ....	84
Figure 5.12: The third network requirement of access control application. ....	85
Figure 5.13: <i>Group</i> element and <i>LOAD_BALANCE</i> action highlighted in blue. ....	85
Figure 5.14: Model of load balancing application.....	86
Figure 5.15: Output of load balancing application.....	88

# List of Tables

Table 1: Summary of characteristics for each SDN-based DSL indentified in literature.	46
Table 2: Main concepts of SDN that compose MDN approach.....	53
Table 3: Visual notation of MDN approach. ....	59
Table 4: EVL rules for <i>NetworkNode</i> semantic. ....	61
Table 5: Syntactic mapping for the <i>Rule</i> element and its relative graphical symbol. ....	63
Table 6: Summarized <i>sdn.egl</i> template. ....	65
Table 7: EGL template for ODL controller. ....	67
Table 8: Features comparison for SDN modeling. ....	75
Table 9: Snippet of code generated for network-monitoring application. ....	78
Table 10: Code generated by MDN editor to implement the access control application. .....	83
Table 11: Snippet of EGL template and the code generated from load balancing application model.....	87

# Contents

<b>1. INTRODUCTION.....</b>	<b>15</b>
1.1 WHY TO MODEL SDN APPLICATIONS? .....	17
1.2 PROBLEM STATEMENT AND RESEARCH QUESTION .....	19
1.3 GENERAL OBJECTIVE.....	20
1.4 SPECIFIC OBJECTIVES .....	20
1.5 DISSERTATION STRUCTURE.....	21
<b>2. TECHNICAL BACKGROUND .....</b>	<b>22</b>
2.1 SOFTWARE-DEFINED NETWORKING (SDN).....	22
2.1.1 SDN Architecture Overview .....	23
2.1.2 SDN Controller.....	24
2.1.3 Northbound and Southbound interfaces in SDN architecture .....	26
2.1.4 OpenFlow Protocol .....	27
2.1.5 What is an SDN Application?.....	29
2.1.6 SDN Programming Languages.....	30
2.2 MODEL-DRIVEN ENGINEERING (MDE) .....	31
2.2.1 The Model-Driven Development (MDD).....	32
2.2.2 Domain-Specific Modeling Language (DSML).....	33
2.2.3 Specifying DSMLs .....	33
2.2.4 Transformation Engines and Generators.....	38
2.2.5 Tools for Enabling MDE.....	39
2.2.6 Graphical Modeling Framework (GMF).....	40
<b>3. RELATED WORK .....</b>	<b>43</b>
3.1 DOMAIN-SPECIFIC LANGUAGES FOR SDN .....	43
3.2 MODELING APPROACHES FOR SDN .....	47
<b>4. THE MDN FRAMEWORK .....</b>	<b>51</b>
4.1 OVERVIEW OF MDN ARCHITECTURE .....	51
4.2 BUILDING PROCESS OF MDN INFRASTRUCTURE .....	52
4.2.1 Specifying the Domain.....	52
4.3 ARTIFACTS .....	54

4.3.1	<i>Abstract Syntax</i> .....	54
4.3.2	<i>Concrete Syntax</i> .....	58
4.3.3	<i>Semantic Domain</i> .....	60
4.3.4	<i>Mappings</i> .....	62
4.4	CODE GENERATION .....	63
4.4.1	<i>Templates for Code Generation</i> .....	64
4.4.2	<i>The support for different controllers</i> .....	66
4.5	APPLICATIONS DEVELOPMENT PROCESS USING MDN APPROACH.....	67
4.5.1	<i>MDN Editor</i> .....	68
<b>5.</b>	<b>EVALUATION</b> .....	<b>73</b>
5.1	USE CASE 1: NETWORK MONITORING .....	76
5.1.1	<i>Modeling Application for Network Monitoring</i> .....	77
5.1.2	<i>Code Generation</i> .....	78
5.1.3	<i>Simulation</i> .....	78
5.2	USE CASE 2: ACCESS CONTROL APPLICATION.....	79
5.2.1	<i>Modeling the access control application and its policies</i> .....	79
5.2.2	<i>Code Generation</i> .....	82
5.2.3	<i>Simulation</i> .....	83
5.3	USE CASE 3: LOAD BALANCING APPLICATION .....	85
5.3.1	<i>Simulation</i> .....	88
5.4	CHAPTER REMARKS .....	89
<b>6.</b>	<b>FINAL REMARKS</b> .....	<b>90</b>
6.1	SUMMARY OF CONTRIBUTIONS .....	91
6.2	LIMITATIONS.....	91
6.3	FUTURE WORK .....	92
A.	ONLINE.....	99
B.	ATTACHMENTS .....	99

## Chapter

## 1

## 1. Introduction

---

The complexity and rigidity of today's Internet are about to end. Its current underlying infrastructure relies on a vertical integrated architecture based on a variety of software and hardware components, which results in a little flexible environment, hard to manage (**NUNES, et al., 2014**). The need to make Internet more dynamic and to enable new management techniques has driven the development of a new paradigm, named Software-Defined Networking (SDN) (**FEAMSTER, REXFORD e ZEGURA, 2013**).

---

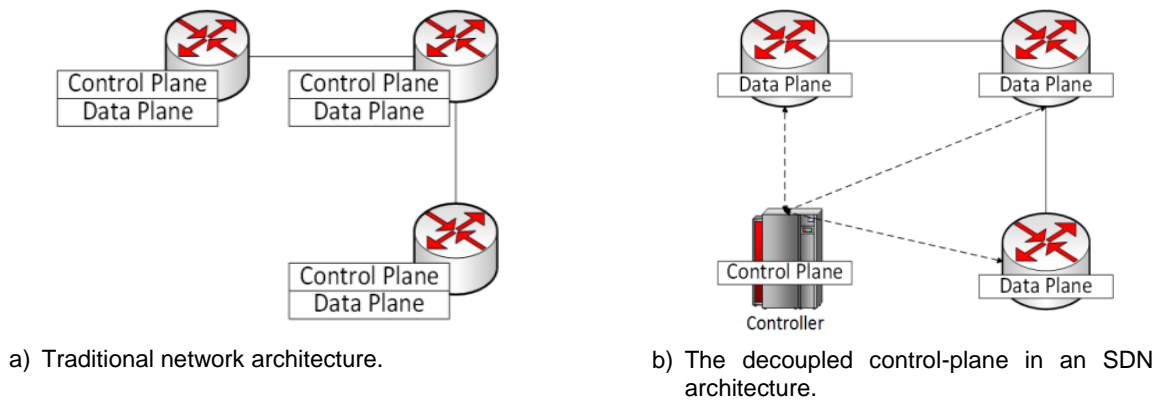
Concisely, SDN is a network architecture that separates the control plane (network intelligence) from the data one (cf. Figure 1.1). This separation moves the network logic to an external controller device that runs a software system, which enables the network programmability (**ORTIZ, 2013**). However, there is still no clear vision about how to interact with such controller to build SDN applications, avoid conflicting policies, and define the network behavior (**FOSTER, et al., 2013**).

Network programming has been a subject of research in more than a decade. For instance, Open Signaling (**CAMPBELL, et al., 1999**), Active Networking (**TENNENHOUSE, et al., 1997**), and Ethane (**CASADO, et al., 2007**), are good examples of past researches. These studies did not achieved widespread success due to many reasons, e.g., lack of compelling problems solved, the focus on data-plane



instead of control-plane programmability, and by enabling the programmability only for the developers working at specific vendors of network devices. Although SDN is not a very new idea, it integrates the concepts of programmability in the network, which is current mature enough to its deployment.

With the network programmability in hand, the first approaches enabled the development of SDN applications (e.g., access control, load balancing, traffic shaping) or even the specification of policies (e.g., limit data transfer per user) by building SDN programming languages (**FOSTER, et al., 2013**). Typically, such languages are designed specifically to express problem solutions in a particular domain (e.g., SDN), a concept known as Domain-Specific Language (DSL) (**FOWLER, 2011**). The advantage of DSL-based SDN programming languages is to abstract the complexity of low-level details relative to SDN controllers and protocols (**FOSTER, et al., 2011**).



**Figure 1.1: Traditional network architecture (a) in relation to SDN architecture (b).**

The DSL paradigm hides low-level implementation details, speeding up and making easier the software development, it is worth emphasizing that DSLs can be categorized into two separate classes: textual or visual (**ESSER e JANNECK, 2001**). Both are part of the software engineering discipline, but a visual DSL, named Domain-Specific Modeling Language (DSML), also composes the Model-Driven Engineering (MDE) paradigm (**SCHMIDT, 2006**). MDE as a software development methodology is used to address platform complexity, such as SDN applications development. Through components focused on a particular application domain (e.g., DSML), it also removes the tight platform dependency (e.g., underlying SDN controllers, operating systems, and the like). DSML is the basis of a MDE technology, it is used to create models, by enabling the execution of them as software applications.

## 1.1 Why to model SDN applications?

In SDN, its applications are responsible to define the logical aspect of network functionalities (**JARSCHEL, et al., 2014**). Such applications involve several specific components (e.g., firewall, rule-based access control, load balancing, and so on). However, there is still a considerable number of issues regarding the development of correct and effective SDN applications (**FEAMSTER, REXFORD e ZEGURA, 2013**), which are essentially composed of algorithms. Hereafter, we list the main issues that led to the development of this research work and claim the ground for model SDN applications.

- i. Networks perform multiple and parallel tasks, such as access control, routing, and traffic monitoring. The independent implementation of such tasks is effectively hard, due to dependency between tasks and the network behavior as a whole (**FOSTER, et al., 2011**).
- ii. The communication between control and data planes has a low abstraction level specified by a protocol called OpenFlow protocol (**MCKEOWN, et al., 2008**). Such low-level also defines the controllers' interfaces, making complex for developers and network operators to interact with them (**VOELLMY, KIM e FEAMSTER, 2012**). For example, the OpenFlow rules directly reflect the structure of the switch hardware (e.g., bit patterns and actions name). In this scenario, high-level concepts such as the definition of an outcome port for a packet and implementation of access control application require multiple OpenFlow rules that programmers must manage manually.
- iii. Applications and rules defined for a certain controller vendor may not work on a different environment, resulting in several different codes and network rules to implement similar behavior in networks with different controllers. For example, a controller based on Java programming language has its applications and modules wrote on a specific programming language, generally the same used to write the controller code. This dependency becomes problematic when the controller needs to be changed or different networks need a replication of some network behavior. Another example is the variation in performance achieved by different controllers (**TOOTOONCHIAN, et al., 2012**) (**LOPES, et al., 2014**). Network operators may need to replace controllers for improving network performance.

- iv. Typically, SDN applications receive events for packets when the switches do not have specific associated rule (e.g., OpenFlow rule) that handle such packets (**MCKEOWN, et al., 2008**). Besides, an earlier rule might cancel the execution of a next rule set in the application. As result, developers need to verify line by line if the execution sequence that installs rules in switches causes a wrong behavior of the network.
- v. SDN programming languages and their written codes for applications are error-prone, they do not allow a clear view of network components or even actions that a network performs, and often result in complex environments (**CASADO, FOSTER e GUHA, 2014**). The resultant problem is the difficulty in validate applications and the network behavior.
- vi. Some network operators may be unfamiliar with low-level implementation details of programming languages. The use of models can improve the communication between network operators and developers (**MOHAGHEGHI, et al., 2011**), in order to define and to implement correctly the network requirements.
- vii. Previous studies on modeling approaches and tools to model software-defined networking do not entirely addressed these issues in developing SDN applications and specifying network rules.

To minimize these problems relative to platform complexity, validation of software systems, and the like, researchers propose models built in a DSML, which are more than documentation items: they are executable objects, since one of the DSML characteristics is the code generation (**KELLY e TOLVANEN, 2008**). Models created from a DSML increase the level of abstraction in software development and help to validate an application while it is under development (**BALASUBRAMANIAN, et al., 2006**). Furthermore, codes generated by a DSML are independent of an underlying programming language (**KELLY e TOLVANEN, 2008**). In the case of SDN, this means that a DSML can support several controller vendors.

Considering the possible benefits enabled by DSML and described above, we propose the concept of Model-Driven Networking (MDN) (**LOPES, et al., 2015**), an association between MDE and SDN that consists of creating SDN applications from models by using a DSML. Instead of writing textual error-prone network algorithms, this dissertation also proposes a modeling editor to create SDN applications, which demonstrates the feasibility of our approach.

## 1.2 Problem Statement and Research Question

The SDN significance in future networks is reflected on a forecast from International Data Corporation (IDC)<sup>1</sup>, which presents that SDN market for the enterprise and cloud service provider segments will be worth \$8 billion by 2018. This scenario demonstrates the importance in providing approaches and tools to enable SDN paradigm in an easier way.

Despite the role played by SDN in enabling network programmability, the correct development of applications and specification of rules that define the network behavior are still open issues. Besides, the available body of knowledge focuses on low-level details of programming (**FOSTER, et al., 2013**) (**NELSON, FERGUSON e SCHEER, 2014**), and the specific studies on modeling SDN have not necessarily addressed the factors that enable applications development and rules specification (**FONTES e SAMPAIO, 2013**) (**PINHEIRO, et al., 2013**).

SDN programming consists of writing correct algorithms for SDN controllers that run them, i.e., algorithms running in controllers need to satisfy some network requirement. However, SDN controllers usually offer low-level platforms based on OpenFlow protocol to enable the network programmability. Thus, the direct codification of algorithms for SDN tends to be error-prone and requires a considerable knowledge in writing OpenFlow (or any other SDN protocol) rules.

In other network types, such as Ethernet (**IMTIAZ, et al., 2008**) and wireless sensor networks (**RODRIGUES, et al., 2011**), we found MDE-based solutions proving to be a feasible way in addressing several problems and in rising abstraction level for such networks. Thus, the proposal of MDN claims to address the drawbacks in developing SDN and its applications (e.g., problematic applications, conflicting rules, low-level of abstraction, and the like) treading a path similar to these approaches.

In this way, we raise two research questions in order to guide and evaluate our work:

- **RQ1** – Is MDN concept a feasible way to develop SDN applications?
- **RQ2** – What are the benefits and limitations of MDN when compared to other SDN development approaches?

These research questions led us to reason about how MDE could be applied to SDN and how such an approach could help network operators to build applications or to

---

<sup>1</sup> Worldwide Software-Defined Networking Market Expected to Reach \$8 Billion by 2018 - <http://www.idc.com/getdoc.jsp?containerId=prUS25052314>

manage network resources, by enabling the avoidance of conflicts among network rules, preventing erroneous network behavior, and validating applications.

### 1.3 General Objective

The main goal of our study is to propose a new model-based approach, called MDN, to develop SDN applications. Besides, we aim to identify its feasibility to develop SDN applications. We also try to define the main benefits of MDN, demonstrating how it improves the development process to create SDN applications, as well as its limitations in such development.

In order to achieve our goal, we provide a metamodel-based tool, called MDN editor, which resembles a Computer-Aided Software Engineering (CASE) tool (**CASE, 1985**). It enables not only the modeling of SDN and its applications, but also makes easier the development, verification, and validation of such applications. Furthermore, MDN editor supports different controller vendors without the need to recreate applications, and improves the communication between network operators and developers. In such a way, this dissertation focuses on bring to the field an alternative way for the SDN applications development.

In summary, we argue that our MDN approach offers a new development process and a modeling tool for SDN applications development, by relying on the concepts of MDE. Furthermore, although the development of SDN applications involves several problems, such as dependent tasks specification, incompatibility among applications for different controllers, low-level of abstraction, and the like, MDN approach can address such issues through a model-based perspective, which has been successfully applied into other scenarios involving networks (e.g., Ethernet, Wireless Sensor Networks).

### 1.4 Specific Objectives

- i. Present the SDN structure and demonstrate how its applications may be created based on MDE paradigm;
- ii. Develop the MDN approach;
- iii. Build a tool that enables the modeling of SDN applications based on the MDN approach;
- iv. Verify the benefits and limitations by using Model-Driven Networking;

- v. Compare the proposed approach with others in the same topic.

## 1.5 Dissertation Structure

We present this dissertation as follows:

- **Chapter 2 – Technical Background** provides the necessary background for a thorough understanding of this work, by defining key concepts and paradigms used for the development of the Model-Driven Networking approach.
- **Chapter 3 – Related Work** describes technologies and scientific findings that relate to the purpose of this dissertation.
- **Chapter 4 – The MDN Framework** presents the processes and techniques used in associating concepts, paradigms, and standards defined in the technical background.
- **Chapter 5 – Evaluation** demonstrates the benefits and potential limitations resulting from using the Model-Driven Networking approach.
- **Chapter 6 – Final Remarks** draws some conclusions and provides directions for future work.

## Chapter

## 2

## 2. Technical Background

---

The recent emergence of software-defined networking offers a platform to enable the network programmability (**FEAMSTER, REXFORD e ZEGURA, 2013**). This programmability allows software used in networks to be developed through software engineering techniques (e.g., MDE). This chapter defines the concepts related to SDN (section 2.1) and MDE (section 2.2), aiming to guide the reader in understanding the dissertation's proposal and its underlying terms.

---

### 2.1 Software-Defined Networking (SDN)

In the introduction, we stated that SDN separates the network logic from the forwarding devices. Such logic is the software control that runs on an external machine, named SDN controller. With this in mind, such controller enables the development and implementation of other softwares, similarly to what happens in traditional computing between operating systems and their applications.

SDN controller operates on a high-level, global, and consistent view of the network. Besides, the separation between control and data planes has brought great advances to datacenter, campus, and enterprise networks, such as a higher level in specifying network policies, access control (**NAYAK, et al., 2009**) and QoS (**KIM, et al., 2010**), experiments based on simulation (**SHERWOOD, et al., 2010**), easier load

balancing (**WANG, BUTNARIU e REXFORD, 2011**), and seamless migration of networks or virtual machines (**ERICKSON, et al., 2008**).

While traditional networks distributes the control plane (responsible to determine how the transmission of packets should occur) across the network forwarding devices, in SDN this control plane is logically centralized (may be physically distributed), resembling a server platform. Another possible comparison is relative to forwarding rules. In traditional networks, forwarding devices run distributed algorithms (e.g., A\* search) in an independent manner to determine the packet forwarding rule. By contrast, in SDN, controllers calculate the forwarding rule (e.g., with Bellman-Ford) and implements this rule by programming the data plane of forwarding devices.

We state that protocols proposed for SDN, such as OpenFlow (**MCKEOWN, et al., 2008**), are key components used to enable the programming of forwarding devices. Currently, OpenFlow protocol is widely accepted as an open standard that allows controllers to manage forwarding devices. Other initiatives also aim to standardize the communication between SDN controllers and forwarding devices, such as ForCES (**WANG, et al., 2010**) and OpFlex (**DVORKIN, et al., 2014**). However, the use of OpenFlow in controllers and switches from major vendors motivate us to adopt it in this work.

### 2.1.1 SDN Architecture Overview

In SDN, the control logic is decoupled from the forwarding devices and all network intelligence (e.g., decisions about routing, permissions) is moved to the SDN controller (**ONF, 2013**). We can note such decoupling strategy through flow tables present in switches. The switches receive and register network policies or rules, defined by a controller. In other words, an SDN controller defines the flow table entries present in switches to instruct them about how to handle packets or flows. It has all information about the network (e.g., where the hosts are connected, the network topology, the source or destination of each packet or flow), and goes through a number of details to deal with conflict resolution that involves general policies or to avoid misbehavior of network elements.

In order to provide a better description about the SDN control plane, the literature presents two interfaces, named the north and south bounds (**HU, HAO e BAO, 2014**). The northbound interface relates to higher-level elements to support the development of network applications, services, or to instruct controllers through a well-defined API. On



the other hand, the southbound interface relates to allow the communication between controllers with forwarding devices or to communicate with low-level protocols at the forwarding plane.

The perspective from control-plane involves a relationship between its layers (e.g., northbound, southbound) and intrinsic characteristics of controllers (e.g., vendor, multiple instances). Such perspective consists of an application layer, which includes logical, user, and business applications, a control layer (i.e., SDN controllers), and a forwarding layer (e.g., forwarding devices), as Figure 2.1 depicts.

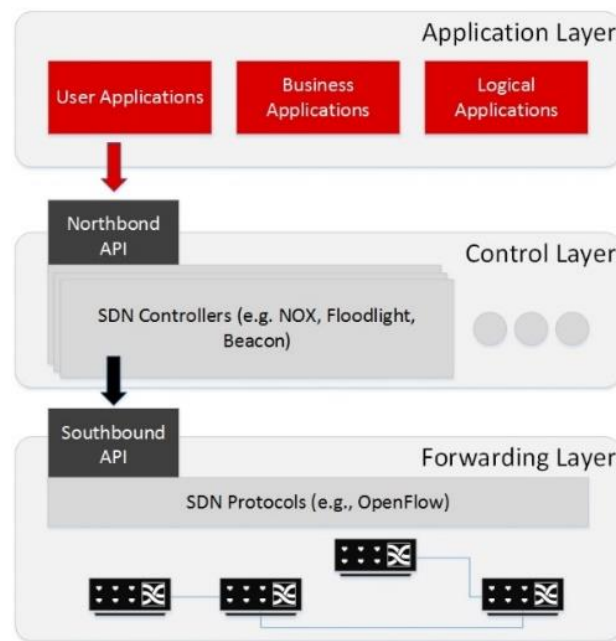


Figure 2.1: Overall perspective of the control layer in SDN architecture.

At the time of this writing, SDN-related activities at Internet Research Task Force (IRTF) proposed an RFC with its view of SDN architecture. For this RFC, the control layer is divided into control and management plane, each of them with its own respective southbound interface, but the respective implementations of SDN following such RFC have yet to publish their documents. SDN reference model as a three-layer model presented in (HU, HAO e BAO, 2014) and proposed by ONF (ONF, 2014) is well aligned with this dissertation.

### 2.1.2 SDN Controller

A controller in SDN composes the control-plane of such network architecture. It is the network “brain”. As we introduced previously, control-plane in SDN is handled apart from the data-plane. This handling is realized inside the SDN controller and research studies, as in (GUDE, *et al.*, 2008) and (ERICKSON, 2013), propose implementations

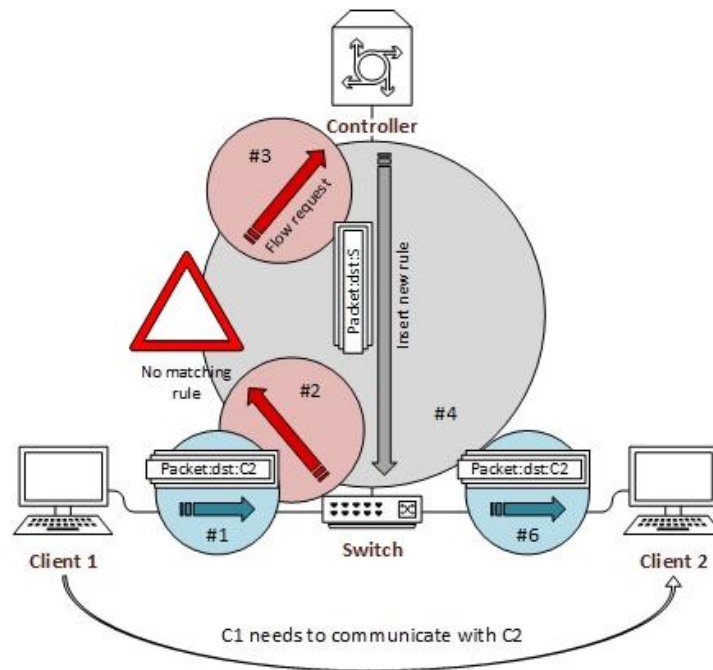
that give a general idea about how the management of forwarding devices should occur, as well as about the communication between the control-plane and data-plane, discussing the right way to deploy it.

Controllers in an SDN are strategic control elements, relaying instructions and information to underlying switches (via southbound interface) and applications on the top (via northbound interface). An SDN controller sends messages to switches disseminating packet-handling rules, e.g., routing, dropping, and the like; a developer generally defines such rules through the controller's northbound API. As Figure 2.1 depicts, the SDN architecture places a controller (control layer) between the southbound (forwarding layer) and northbound (application layer) interfaces.

The hypothetical topology depicted in Figure 2.2 helps to understand the controller's responsibility and the insertion of a new rule into a switch. In the topology below, client 1 (C1) wishes to establish a communication with client 2 (C2). From such topology, assuming that C1 sends a packet with destination to C2 (step 1); then, first, the packet arrives at switch (S), which at the time has no forwarding rule for this packet (i.e., it does not have a matching rule in its flow table) (step 2). After the arrival, S generates a new flow request to the controller (step 3). The controller responds with a new packet, which inserts a new rule at S (step 4). Switch S forwards the first packet to C2 (step 5). Then, switch S will now be able to send all packets from C1 to C2 with no need for additional communication with the controller. It is noteworthy that all communication switch-controller-switch is performed, for instance, by default with OpenFlow protocol. Figure 2.2 depicts this flow.

The possible responses (e.g., actions, instructions, and the like) from controller to switches are defined by programming algorithms based on the SDN southbound protocol (e.g., OpenFlow), in the shape of applications, modules and policies deployed on the controller at the control layer or through external applications and services in the application layer.

An important aspect of an SDN architecture is that, given any administrative domain, it is possible to have more than one controller handling a number of switches. This is a desired feature when network reliability and availability come into play. Conversely, according to the version 1.4 of OpenFlow specification, an OpenFlow switch is capable to connect with multiple controllers.



**Figure 2.2:** The necessary steps to the Client 1 communicate with the Client 2 when the switch does not have rules that can manage this communication.

We have seen several SDN controllers released over the past years. Beacon (ERICKSON, 2013), NOX (GUDE, *et al.*, 2008), POX<sup>2</sup>, and OpenDaylight<sup>3</sup> are a few examples of such releases. Although they have different characteristics, such as occurs on the traditional operating systems for PCs, such controllers have a similar operation that might be categorized on *proactive*, *reactive*, or a combination of both. *Reactive* controllers only instruct the forwarding device when the latter forwards the flow to them. *Proactive* controllers, in contrast, rely instructions to their underlying forwarding devices at any moment, without the need for such devices communicate with it. Usually, SDN controllers offer these two types of operation.

### 2.1.3 Northbound and Southbound interfaces in SDN architecture

Now we present two interfaces that involve the SDN controller, named *northbound* and *southbound*. One of the most discussed topics in SDN relates to such interfaces (FEAMSTER, REXFORD e ZEGURA, 2013). Southbound interface, located below the SDN Controller Platform in Figure 2.3, has converged to the OpenFlow standard, although there is some room for discussions, as in (DORIA, *et al.*, 2010) and (CISCO, 2014). Such interface should to enable SDN switches to communicate with controllers. As controllers relay instructions to switches about how to handle incoming flows, then,

<sup>2</sup> POX Wiki: <https://openflow.stanford.edu/display/ONL/POX+Wiki>

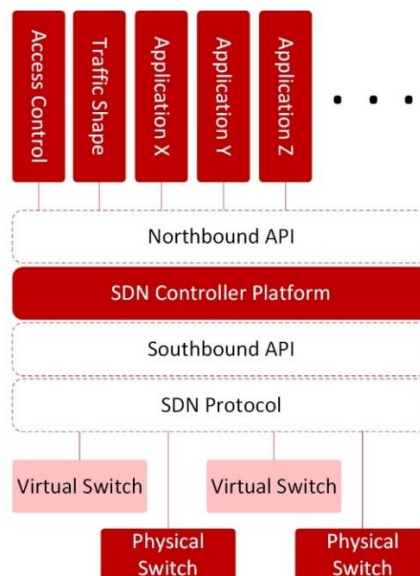
<sup>3</sup> OpenDaylight portal: <http://www.opendaylight.org/>

switches can handle incoming packets, identify network topology, and apply rules defined at some level of northbound API.

In contrast to southbound, northbound interface has no widely accepted standard specifying it. Although there are already some initiatives towards this direction (**FOSTER, et al., 2013**). Even the Open Networking Foundation (ONF) has created a working group with the aim of designing prototypes, codifying patterns, and producing artifacts that may validate the creation of a standard for the northbound (**RAZA e LENROW, 2013**).

The northbound, located above the SDN Controller Platform in Figure 2.3, encompasses the relationships between controllers, network applications or services, and user applications. This layer is responsible to abstract the underlying functions of the network, in a way that network operators or developers can implement new applications or changes into network to achieve their objectives without having to verify and know others aspects that are not related to their applications or changes.

The perspective involving the northbound and southbound layers is depicted at Figure 2.3. Such perspective demonstrates the separation of responsibilities for each layer or interface level in an SDN architecture.



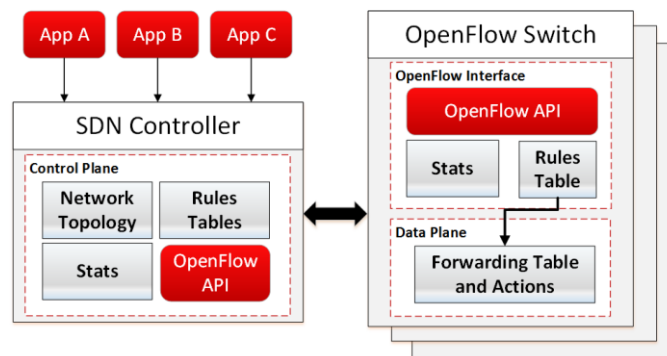
**Figure 2.3: Northbound and Southbound perspective of an SDN architecture.**

#### 2.1.4 OpenFlow Protocol

The OpenFlow (OF) protocol is primarily a specification, which standardizes how the exchanging of information between control-plane and data-plane must occur in an SDN scenario. Such specification also is part of a default architecture, created to describe the role of each component (**ONF, 2013**). Since OpenFlow is a specification and a protocol

used by many SDN solutions (HU, HAO e BAO, 2014), including the approach presented in this dissertation, we provide an overview on it in this section.

The default OpenFlow architecture defines forwarding devices (OpenFlow switches) with one or more flow tables and a higher-level layer that communicates with a controller through the OpenFlow protocol (cf. Figure 2.4). OpenFlow (or other similar protocols) has a decisive role at the SDN scenario. When an OpenFlow switch receives a packet, it verifies and compares the header fields of such packet with its own fields in flow table entries. If an entry corresponds to a packet header, the switch will realize any instructions or actions related with the flow entry (e.g. packet forwarding to Client 2 in Figure 2.2). In case that switch does not find any entry, it will act as the instructions defined at table-miss flow entry (every flow table contains a table miss entry to address this case). The switch may forward the packet to the controller (via an OpenFlow SSL/TLS channel), drop the packet, or continue to match the packet header fields with others flow tables (if any).

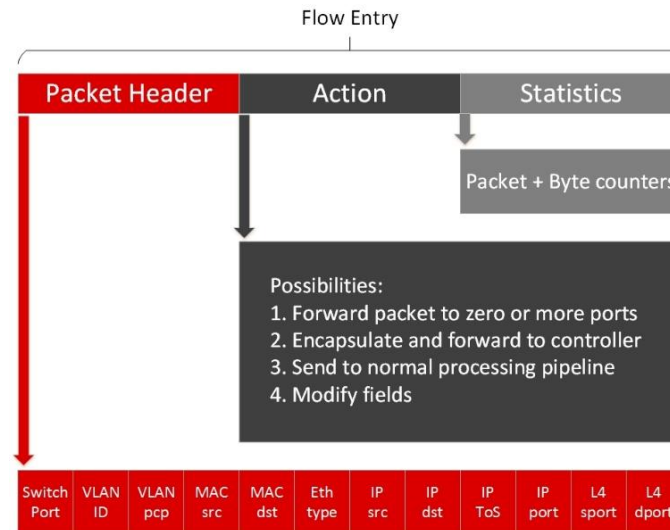


**Figure 2.4: SDN architecture from the OpenFlow perspective.**

The flow table consists of flow entries used by switches to forward the data flow. Three fields compose an entry in flow table: (1) A packet header to define the flow, (2) an action to define how packets should be processed, and (3) the statistics used to keep information about the number of packets and bytes for each flow. Figure 2.5 depicts the structure of a flow entry.

Each entry has also an identifier field (id) and its priority. Such priority is defined according to each id field or a natural sequence between entries in flow table. All entries have statistics that can be sent to controller or requested by it. Furthermore, each entry has an associated action to deal with an incoming packet flow. These actions define the behavior of SDN and enable the programmability of such networks.

Currently, we have identified several OpenFlow switches available. Vendors of network equipment such as IBM<sup>4</sup>, HP<sup>5</sup>, and NEC<sup>6</sup> already offer some type of OpenFlow switch to deploy an SDN environment. It demonstrates the massive investment on this new paradigm of networks.



**Figure 2.5: Fields of an entry in flow table.**

Although OpenFlow protocol enables programmable networks, we emphasize that OpenFlow is not SDN; OpenFlow is just part of the various components in an SDN environment, with the role of performing the communication between controllers and switches. To date, it also still has issues to be addressed, such as scalability (YEGANEH, TOOTOONCHIAN e GANJALI, 2013), performance (TOOTOONCHIAN, *et al.*, 2012), reliability (HU, *et al.*, 2013), and security (KREUTZ, RAMOS e VERÍSSIMO, 2013).

### 2.1.5 What is an SDN Application?

One of the main benefits provided by SDN is to enable the network programmability. It results in SDN applications, which can deliver highly scalable, efficient, and manageable network services (FEAMSTER, REXFORD e ZEGURA, 2013) (e.g. load balancing, access control, and the like). Although SDN enables remote applications communicating with SDN controller, hereinafter we discuss SDN applications running on it, which resembles the common organization of traditional operating systems and software applications.

<sup>4</sup>Available online: <http://bit.ly/sdn-switches>

<sup>5</sup>Available online: <http://bit.ly/hp-switches>

<sup>6</sup>Available online: <http://bit.ly/nec-switches>

According to ONF (**ONF, 2014**), SDN applications are deployed at the application layer, which completes the three main layers of SDN architecture (i.e. forwarding layer, control layer, and application layer). As previously mentioned, such application layer communicates with controller through an interface (e.g., northbound) (**JARSCHEL, et al., 2014**). This interface enables applications to specify rules or to request some network resource. The specification of rules or network behavior by an SDN application passes through underlying SDN layers to get reach the forwarding devices (e.g., OpenFlow switches).

Typical SDN applications include features such as granular firewall monitoring, user identity management, access control policies, etc. (for an extensive list see (**HU, HAO e BAO, 2014**)). While most discussions involves the use of SDN in datacenters, recent research efforts started to explore the use and creation of SDN applications for networks with flexible and more secure managements (**FOSTER, et al., 2013**) (**MONSANTO, et al., 2013**).

We emphasize that there is not yet a widely well-accepted standard process for developing SDN applications. Currently, such development uses some General Purpose Language (GPL) (e.g., Python, Java, and C++) to write SDN applications or to specify policies through SDN controllers' APIs. In addition, a well-known approach is to use the *Domain-Specific Languages* (DSL) paradigm in building SDN programming languages to support the development of applications (**FOSTER, et al., 2013**).

Although each SDN layer has its own responsibility, note that there is a tight dependency between current SDN applications development and SDN controllers. Such a dependency might hinder some benefits of the SDN architecture, such as flexibility and scalability. For instance, an application developed to run with the NOX controller (**GUDE, et al., 2008**) cannot be executed on an SDN environment with Floodlight as controller due to incompatibility issues (**FLOODLIGHT, 2014**).

### 2.1.6 SDN Programming Languages

The need to build network applications that run on an SDN environment and the low-level of abstraction existent to specify OpenFlow-based rules motivated the emergence of several SDN languages, such as Procera (**VOELLMY, KIM e FEAMSTER, 2012**), Pyretic (**MONSANTO, et al., 2013**), and Flowlog (**NELSON, FERGUSON e SCHEER, 2014**). SDN controllers use such languages to support the development of applications and to provide abstractions for operation, administration, and managing of networks.

Such network management is characterized through diverse features, such as security (e.g., avoiding that unauthorized users access the network), performance (e.g., eliminating bottlenecks or erroneous network behavior), and reliability (e.g., ensuring that the network is available to its users). In SDN, the literature defines three stages to provide the network management through SDN programming languages: i) traffic monitoring, ii) policies and rules specification, and iii) updating of flow tables (**FOSTER, et al., 2013**). An instance of such languages is the following example of Pyretic code (cf. Figure 2.6):

```
Select(packets) *
GroupBy([srcmac]) *
SplitWhen([inport]) *
Limit(1)
```

**Figure 2.6: Pyretic code to monitor network packets.**

The piece of code above exhibits code of an SDN programming language to create an application that will select a group of packets in a given port. Such language is built with an underlying GPL (Python). The use of GPL to propose an SDN language has been based on the DSL paradigm, previously mentioned. The DSL paradigm is present in most SDN languages available, but they differ by their components, methods, and scope (e.g., quality of service, load balancing, network monitoring). Unlike GPL, a DSL is designed for a specific domain (e.g., SQL in databases) (**FOWLER, 2011**).

## 2.2 Model-Driven Engineering (MDE)

The term MDE describes software development processes that create models of softwares methodically transformed to concrete implementations (e.g., source code) (**FRANCE e RUMPE, 2007**). MDE combines *Domain-Specific Modeling Languages* (DSML) (**SCHMIDT, 2006**), which formalize the application structure, behavior, and requirements within particular domain (e.g., middleware platforms, mobile applications, software-defined networking), with *Transformation Engines* and *Generators* that verify properties of models and transform them in software artifacts (e.g., source code, configuration files, simulation inputs, documentation, or even in another model).

We emphasize that the overall concept behind the MDE exists since 1980s and 1990s. The prior works, such as Computer-Aided Software Engineering (**CASE, 1985**), served as lessons learned for recent MDE technologies. For instance, pioneer CASE tools also enabled features like the detection of errors and code generation through graphical modeling, but the design of such tools was defined in a low abstraction-level,

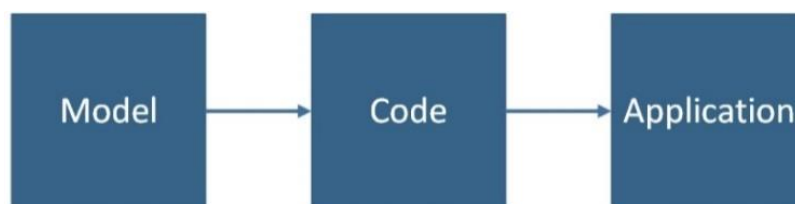


often hard coded. Today's platforms based on MDE can structure and offer more functionalities in a higher abstraction-level (e.g., *metamodels*) and a modular way. For instance, the transformation engine for code generation is not coupled with the graphic model, it consists of several components that may be implemented independently (e.g., templates, validation rules).

### 2.2.1 The Model-Driven Development (MDD)

While some authors define MDD equivalently to the MDE (**SCHMIDT, 2006**) (**FRANCE e RUMPE, 2007**), we consider the definition of Selic (2003) in which MDD involves the process of software development that primarily focuses and produces models rather than codes of computer programs (**SELIC, 2003**). Thus, models in MDD are the primary artifacts in the development process (**HAILPERN e TARR, 2006**). Kelly and Tolvanen (2008) reinforce this view, they claim that MDD uses its source models instead of source codes from traditional development in implementing software (**KELLY e TOLVANEN, 2008**).

The development process based on models should be applied whenever possible, because it increases the level of abstraction and reduces complexity of the development. Essentially, MDD uses the automated transformation of models to applications, similarly to what happens in the compilation of applications code. When the model of an application is done, the target code may be generated and then interpreted or compiled for execution, without the need for modification in such code. Figure 2.7 depicts this single-way of MDD.



**Figure 2.7: Single-way process of model-driven development.**

One strategy to provide a MDD solution is through a DSML. In this way, the underlying modeling language of MDD and the generation of code need to be domain-specific, enabling the development of only certain applications in a particular domain (**KELLY e TOLVANEN, 2008**). The focus on a narrow domain makes possible for MDD (with an underlying DSML) to map the modeling closer to the main problem and to enable code generation effectively.

### 2.2.2 Domain-Specific Modeling Language (DSML)

The aforementioned Domain-Specific Language (DSL), used in several proposals for SDN applications development, is a software language built and tailored to solve problems through some domain-specific application. Compared to GPLs, the DSLs have a higher level of abstraction due to providing constructs that represent concepts of the domain. According to (CUADRADO e MOLINA, 2009) and (KELLY e TOLVANEN, 2008), a DSL is fundamentally composed of three components, namely the *Abstract Syntax*, which defines language concepts as well as their constraints and relationships, the *Concrete Syntax*, corresponding to notation available for end-user in specifying applications based on abstract syntax, and the *Semantics*, used to describe the meaning of notation elements.

The concrete syntax of a DSL can have textual and/or graphical notations. In the latter case, it is named Domain-Specific Modeling Language (DSML) (KELLY e TOLVANEN, 2008). Another difference of DSML as compared to textual DSL is that DSML has two additional components in its fundamental composition, i.e., two mappings components named *Semantic Mapping*, for the relationship between abstract syntax and semantic domain, and *Syntactic Mapping*, which assigns syntactic constructions (e.g., graphical, textual, or both) with elements of abstract syntax. Thus, formally, a DSML is a 5-tuple composed of Concrete Syntax (CS), Abstract Syntax (AS), Semantics (S), Semantic Mapping (MC), and Syntactic Mapping (MS) (HAREL e RUMPE, 2004), as defined in (1):

$$L = \langle CS, AS, S, M_c, M_s \rangle \quad (1)$$

Any DSML requires a precise specification of these five components (KELLY e TOLVANEN, 2008). Such specification follows the MDE requirements to offer a model-based development platform.

### 2.2.3 Specifying DSMLs

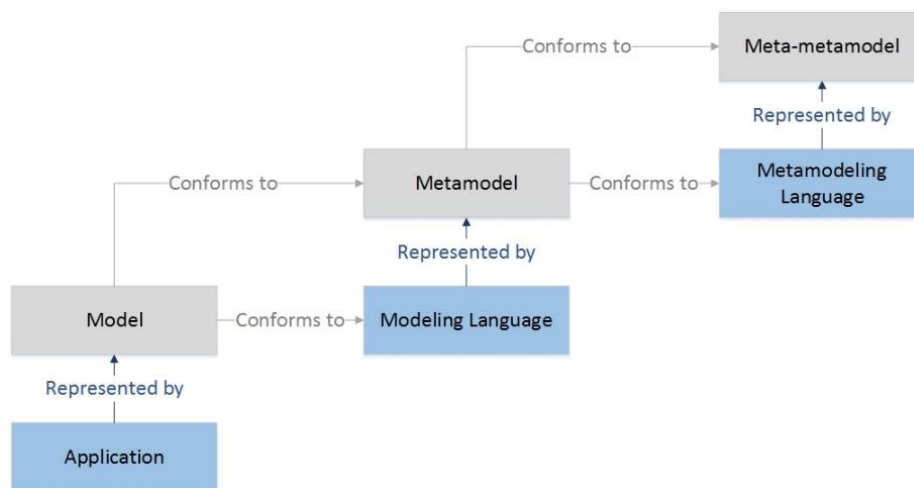
The more specific and restricted the domain, the easier it becomes to be structured in a DSML and in its components. Thus, we cannot use DSML for creating applications out of the scope previously defined. However, it is not a limitation, because we can extend a DSML to achieve other features, since they are within the same domain.

The first step in DSML specification is the definition of the target domain and its concepts. The identification of main concepts for a DSML is strongly dependent on a

creative view and knowledge in the domain. However, some sources can help in this investigation, such as domain architecture or available specifications (e.g., OpenFlow), existing products (e.g., SDN controllers), patterns, target environment, and code (KELLY e TOLVANEN, 2008).

As aforementioned, to create a DSML we need to specify five basic components. In the following paragraphs are clearer definitions and descriptions of forms of specification for each DSML component:

**Abstract syntax.** Such component is normally specified through a *metamodel*. The metamodel (or abstract syntax) has the conceptual structure of a DSML, e.g., the entities (or concepts) of a modeling language, their attributes, possible relationships, and well-formedness rules. When a DSML needs to be created, the developer uses a *metamodeling language* to specify the elements of metamodel, which describes the domain knowledge. For metamodeling, there are several languages, e.g., MetaGME (LEDECZI, *et al.*, 2001) (KARSAI, *et al.*, 2004) or Meta-Object Facility (MOF) (OMG, 2000). According to (KELLY e TOLVANEN, 2008), a metamodeling language is used for specifying the abstract syntax (or metamodel) of a DSML, which supports the production of CASE tools that follow such abstract syntax. The authors also consider the metamodeling language as a *meta-metamodel*, i.e., a model that describes another metamodel. Indeed, the metamodel is just part of this cascade instance, as Figure 2.8 depicts.

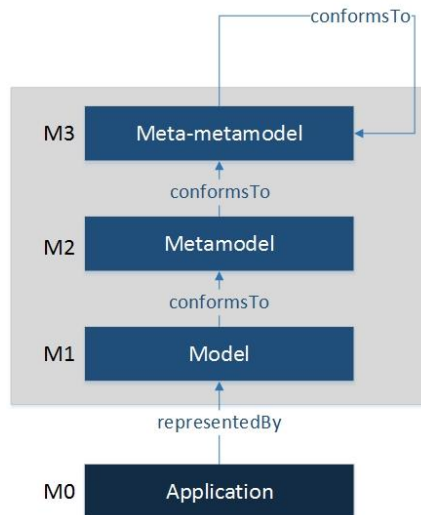


**Figure 2.8:** The application model conforms to a modeling language. Such language has its abstract syntax represented by a metamodel; which conforms to a language that has an abstract syntax represented by a meta-metamodel.

The idea of conformity between models is shared yet by the Object Management Group (OMG) and its four-level modeling framework (OMG, 2014), as well as the Bézivin's version of metalayers (BÉZIVIN, 2005). For OMG's *Model-Driven Architecture*

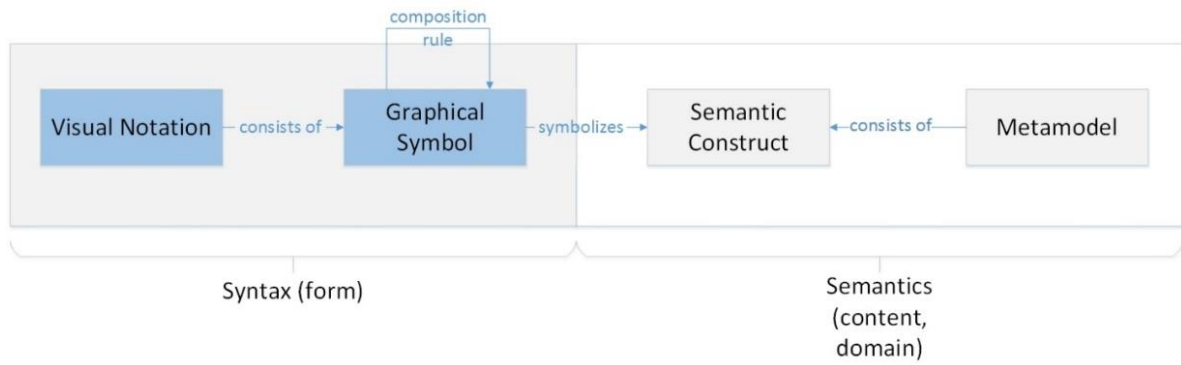
(MDA), i.e., its particular view of MDE, a metamodel is a “class of models”. The derived models of this class of models are instances of the metamodel. Such structure has four levels, namely levels M0, M1, M2, and M3. It is worth to mention that for MDA the level M2 corresponds to the metamodel and it is expressed using MOF.

On the other hand, Bézivin (2005) removes the dependency on MOF and Unified Modeling Language (UML) present in OMG’s MDA and defines a more general infrastructure, named 3+1 MDA organization. Such infrastructure is closer to that used in MDN approach. Figure 2.9 shows the Bézivin’s architecture.



**Figure 2.9: Bézivin's 3+1 MDA organization (BÉZIVIN, 2005).**

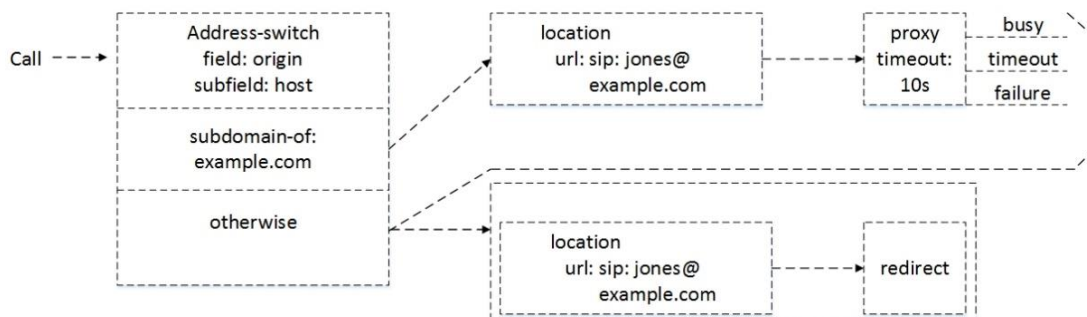
**Concrete syntax.** In DSMLs, graphic elements compose such syntax. They may have a close iconic meaning to its corresponding entity (or concept) in abstract syntax. These graphic elements are visual representations for conceptual elements of abstract syntax, forming a visual notation, which is a representation focused on humans, with the purpose to facilitate the communication and problem solving (HAREL, 1988). To encode information of underlying models or metamodels, the visual notation uses spatial arrangements of graphic (and textual) elements (MOODY, 2009). The scheme of Figure 2.10 depicts the composition of visual notations.



**Figure 2.10: Scheme representing the organization of visual notation regarding the semantics of DSML as well as its metamodel.**

According to Moody (2009), the visual notation (or visual language, graphical notation, diagramming notation) is composed of a set of graphical symbols, a set of compositional rules (visual grammar), and descriptions of the meaning for each graphical symbol (visual semantic). Thus, the Moody's definition for visual notation (or concrete syntax) of a DSML, which is used in this dissertation, is the combination between graphical symbols (e.g., lines, circles, spatial relationships, and the like) and compositional rules.

The typical starting point in defining a concrete syntax for a DSML is specify a basic visual notation for the language. For instance, Lennox *et al.* (2004) proposes a notation composed simply of directed arrows between boxes (cf. Figure 2.11). The main principle argued was read the whole specification for a Call Processing Service from visual notation representing a model (**LENNOX, SCHULZRINNE e WU, 2004**). It is worth to mention that such specification of concrete syntax normally arises from the creativity and domain knowledge owned by the creator of a DSML. In some cases, empiric aspects (e.g., cognitive and colors) may be used to a better definition of the visual notation.



**Figure 2.11: Visual notation proposed by Lennox *et al.* (2004).**

**Semantic Domain.** Abstract syntax conveys limited information about what the concepts in a DSML actually mean. The definition of semantics is crucial for clarify what

the DSML represents and means. Without such definition, misinterpretations and personal assumptions may occur in the use of the language. For instance, in the context of modeling languages, the understanding involving the meaning of State Machines form a key part in choosing it to model a behavior of a problem domain.

The concepts that compose a DSML have some meaning, also referred to as semantics. If some modeler creates elements into a diagram and design a link connecting them, it is a meaning for the model. Such meaning come from the problem domain (**KELLY e TOLVANEN, 2008**). In a hypothetical scenario, when a developer is creating an information system for a university, the modeling concepts, such as “teachers”, a “campus”, “classrooms”, and their properties and connections are well-defined meanings inside the application domain. These relationships between modeling concepts and meanings are the semantics of a DSML. In other words, the semantics of a DSML defines the meaning of the language, using terms as behavior and static properties.

There are several ways to describe the meaning of a language concept (**CLARK, SAMMUT e WILLANS, 2008**). An example is the use of natural language in describing the meaning of concepts:

- i. By describing concepts with well-defined meaning, e.g., “a professor has a name and an office”;
- ii. Defining the properties and behavior for a concept, e.g., “a professor can be in vacation, or can be teaching classes”, as well as the common properties for all instances of a concept, e.g., “professors have Ph.Ds”;
- iii. Specializing another concept, e.g., “a hardware lab is a classroom with hardware equipment”.

Besides the use of natural language in specifying semantics, Clark *et al.* (2008) present more four approaches to describe the semantics of languages. The main difference of these four approaches in relation to natural language in specifying semantics is the use of metamodels to express such semantics. The approaches are: (1) *Translational*, which translates concepts from one language into concepts in a different language; (2) *Operational*, which models the operational behavior involving language concepts; (3) *Extensional*, used for extend the semantics of an existing language; (4) *Denotational*, for model the mapping to semantic domain concepts.

Generally, the use of more than one approach in specifying the language semantics occurs. The MDN combines natural language with Object Constraint

Language (OCL) and a metamodel to specify the semantics of the proposed DSML, mixing translational and operational approaches. OCL is one of the most efficient ways to apply rules of architecture and to perform validation in system models (**OMG, 2006**).

**Mappings.** In order to complete the specification of a DSML, the mapping between the semantic domain and metamodel (abstract syntax) must be made. Such association, usually called *semantic mapping*, can also serves to define the semantics of the DSML. It enables valid instances of models and it can be performed through the specification of constraints or rules (e.g., cardinality in metamodels). Besides the semantic mapping, we already introduced the *syntactic mapping*, which combines the elements of visual notation with the concepts or entities of abstract syntax. Some tools help in such mapping (e.g., Eclipse Modeling Framework (**ECLIPSE, 2014**)), they associate graphics, codes and the like to the concepts in metamodel. Both mappings are dependent on previous components of DSML to be satisfied.

Besides the specification of a DSML, in order to build a MDE infrastructure, there is the need to generate code and to enable the model transformation, but such requirements are out of the scope DSML structure, as depicted in Figure 2.12. Thus, we discuss the model transformation and generators in the next subsection.

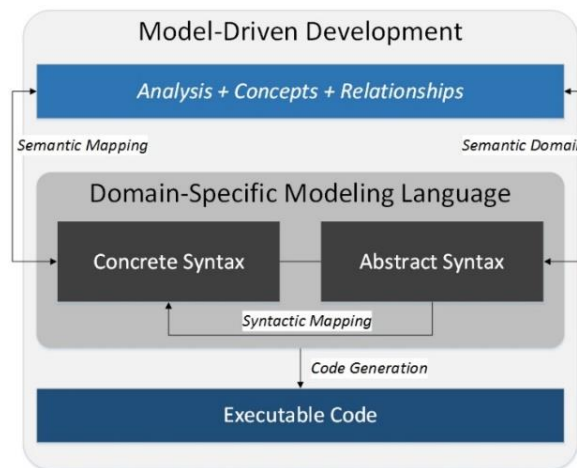


Figure 2.12: Summarized scheme of MDD and its components.

#### 2.2.4 Transformation Engines and Generators

To be useful, MDE technologies require both the DSML and the features to generate code or to transform models. In the previous subsection, we already discussed DSML and its specification. Hereinafter, we will briefly present the role of the transformation engines and generators to enable the synthesizing of artifacts from models also helping to ensure the consistency between applications and their models (**SCHMIDT, 2006**).

Note that one of the benefits in using MDD is its improved compatibility with diverse scenarios. We observe this benefit in the transformation of models, e.g., a model of a software-defined network can become a traditional network model. There are wide accepted solutions to enable the transformation between models, such as the OMG's *Query/View/Transformation* (QVT) (**BAST, et al., 2011**), although it is not the focus of this dissertation, an extensible work can be found in (**MENS e VAN GORP, 2006**) and (**CZARNECKI e SIMON, 2003**). According to Cuadrado and Molina (2009), we can even consider the code generation as a transformation, named *model-to-text*, which also has standards under development (**OMG, 2008**).

The code generator is one of the key parts in a MDE structure; it helps in automating the execution of models as applications. From the perspective of developer (or modeler), the generated code is complete and it does not require modifications after the action of code generator (**KELLY e TOLVANEN, 2008**). Thus, such action means that such generator and its underlying framework have the logic and methods to translate automatically the models to executable code, being the last validation step in the MDD process.

An overview about tools that enable and make easier the implementation of MDE structure is presented in the next subsection.

### 2.2.5 Tools for Enabling MDE

The benefits achieved with MDE have raised some tools that help in the creation of infrastructures to support the development based on models. These tools need to offer support for the specification of DSML components, domain-specific constraints and perform model checking to detect and prevent several errors early in MDD life cycle. Furthermore, the tools for enabling MDE need to allow the generation of code or even the transformation between models.

In this scenario, there is a number of available tools we can use to create a MDE-based editor, such as Graphical Modeling Framework (GMF) (**STEINBERG, et al., 2009**), MetaEdit+ (**TOLVANEN e ROSSI, 2003**), and AToM<sup>3</sup> (**DE LARA e VANGHELUWE, 2004**). These tools enable the creation of metamodels, visual notations, and so on, but the main benefit is in automating the mappings and the code generation.

Although the tools for implement MDE have similarities, due to reasons like public license, compatibility, and code generation, the Model-Driven Networking (MDN)



editor uses GMF as underlying framework. Thus, we perform a more comprehensive explanation of this framework in the next subsection.

### 2.2.6 Graphical Modeling Framework (GMF)

The GMF is a framework for building Eclipse-based graphical editors. For example, an SDN modeling tool proposed in this dissertation, UML editors, flow editor, and so on. Such framework consists of two other frameworks, namely the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). Before we explain how such composition occurs, we expose what each of them is:

**Eclipse Modeling Framework (EMF).** The EMF project is a modeling framework and a tool for code generation to build applications based on a structured data model. From a model specification described in the XML Metadata Interchange (XMI) standard, the EMF provides features to create a set of classes for the model (relating the model described in XMI with such classes). It also joins such features with adapter classes to enable the editing based on commands of model (e.g., insert an entity).

EMF includes a metamodel Ecore that is indeed a reference implementation of OMG's EMOF (Essential Meta-Object Facility), which by its turn is the simplified version of the more comprehensive MOF. EMF uses an Ecore metamodel to specify the abstract syntax of a DSML. It makes reasonably easy the mapping between application-level models and Ecore metamodel.

Ecore is the model used to represent models in EMF. A simplified set of the Ecore model is depicted in Figure 2.13, which only shows the elements of Ecore needed to model the MDN approach, avoiding, for example, the base class *ENamedElement*, which relates with *EClass*, *EAttribute*, and *EReference*, defining the *name* attribute which is presented explicitly in the classes mentioned here.

As the Figure 2.13 shows, there are four Ecore classes composing this meta-metamodel (**STEINBERG, et al., 2009**):

1. *EClass* represents a modeled class, having a name, zero or more attributes (*EAttributes*), and zero or more references (*EReferences*).
2. *EAttribute* represents a modeled attribute, which has a name and a type.
3. *EReference* represents an association between classes. It has a name, a boolean flag (it indicates if the reference represents a containment), and a reference target, which is another class of *EClass* type.

4. *EDataType* represents the type of attributes (e.g., int or float).

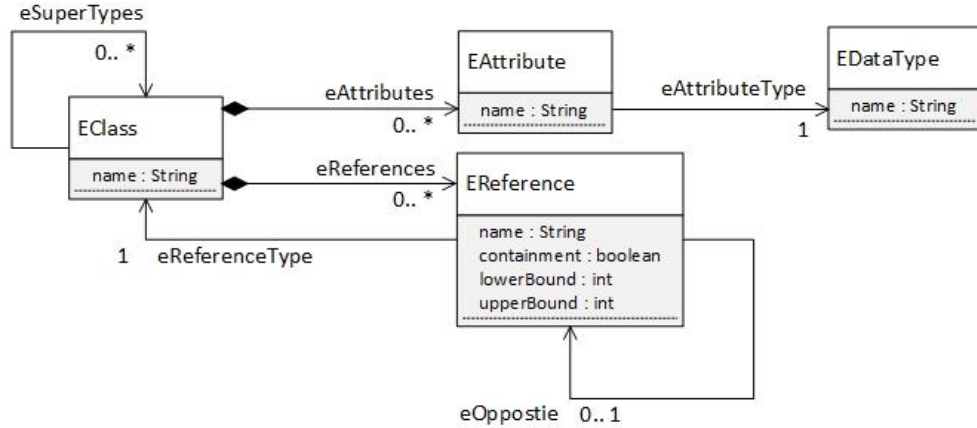


Figure 2.13: Simplified set of the Ec core meta-model.

From the perspective of DSML specification and its components, EMF enables the specification of abstract syntax and code generator. Automating the mappings already discussed in subsection 2.2.3.

EMF also has a language based on Object Constraint Language (OCL) for verification of models, named Eclipse Validation Language (EVL).

Rules written with EVL have the structure depicted in Figure 2.14 below:

```

context <name> {
    (constraint|critique <name>) {
        (guard (:expression)|({statementBlock}))?
        (check (:expression)|({statementBlock}))?
        (message (:expression)|({statementBlock}))?
        (fix)*
    }
}

```

Figure 2.14: EVL rules structure.

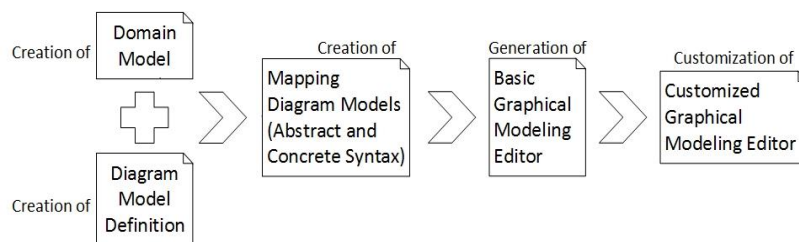
Each *context* is related to an instance of *EClass* (e.g., network node, host, and rules). Thus, inside the specified *context*, there is the concept of *invariants* (i.e., *constraint* and *critique*), which defines the applicability of rules to a subset of the instances specified by the *context*. Such definition is performed by writing the *guard* and/or *check* expressions. If some instance matches the *guard* and does not satisfy the expression defined in *check* invariant, the model is not valid and a fix is needed.

**Graphical Editing Framework (GEF).** The GEF framework specifies technology used to create graphical editors and views of Eclipse IDE and its workbench user interface. In more detail, there are six components consisting such editors:

- i. Model diagram editor, including a tool palette;
- ii. Figures graphically representing the data elements of underlying models;
- iii. *EditParts* relating figures and their respective models elements;
- iv. Request objects for users inputs;
- v. *EditRule* objects that evaluate requests and create appropriated command objects;
- vi. Command objects, which edit the model and provide features to *undo* or *redo*.

GEF is intrinsically linked with the concrete syntax in specifying DSML, as well as with the mapping between visual notation and its semantics. Furthermore, GEF is a key part in creating the MDN editor as a feasible tool.

In summary, GMF automates the joining between EMF and GEF. Thus, developer or modeler need only perform the definition of abstract and concrete syntaxes in these frameworks, as Figure 2.15 depicts. Through several standard rules (e.g., symbols connections, actions for create/edit/delete models) and models described in XML, it automatically relates each element of metamodel described in EMF with the visual notation specified in GEF, generating an editor based on the Eclipse user interface.



**Figure 2.15: Workflow for developing graphical modeling editors using GMF.**

## 3. Related Work

---

This chapter presents and discusses recent research studies related to SDN applications development. In section 3.1, we present the DSLs for SDN, while in section 3.2 we show model-based approaches.

---

Since the first OpenFlow specification, released in 2008, several approaches emerged trying to raise the abstraction level of the protocol that enables SDNs and its applications. These approaches achieve their goals through the controller's northbound interface. Such interface enables and even requires a higher-level concept to define the desired behavior in an SDN-based environment. Due to network dynamics and complexity for the developer to handle all aspects of an SDN application, such applications have been achieved through high-level programming languages that hides the complexity involved in implementing and handling OpenFlow protocol directly.

### 3.1 Domain-Specific Languages for SDN

Some SDN programming languages have emerged to enable network operators in creating network applications for the controllers' northbound interface. Currently, the following SDN programming languages are available: FML (**HINRICHS, *et al.*, 2009**), Nettle (**VOELLMY, AGARWAL e HUDAK, 2011**), Procera (**VOELLMY, KIM e FEAMSTER, 2012**), Flog (**KATTA, REXFORD e WALKER, 2012**), NetCore

(MONSANTO, *et al.*, 2012), Pyretic (MONSANTO, *et al.*, 2013), Frenetic (FOSTER, *et al.*, 2011), FatTire (REITBLATT, *et al.*, 2013), Nlog (KOPONEN, *et al.*, 2014), and Flowlog (NELSON, FERGUSON e SCHEER, 2014). It is worth emphasizing that all those languages are based on the DSL paradigm. Figure 3.1 depicts the timeline of releases of the SDN programming languages.

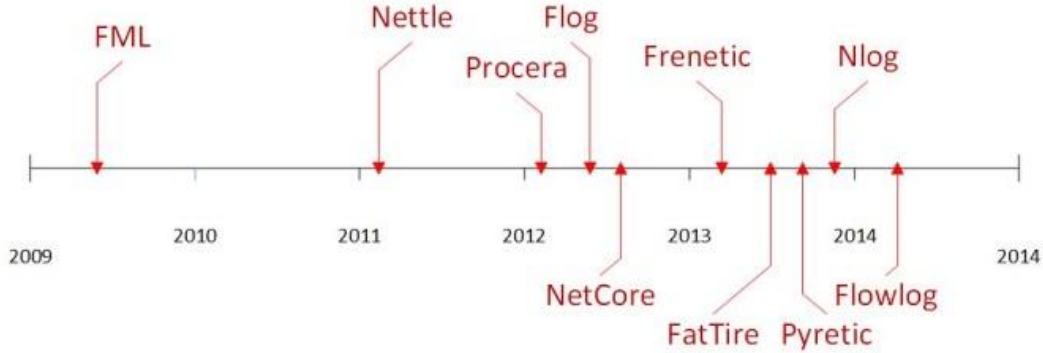


Figure 3.1: The releasing of SDN programming languages in timeline.

The first SDN programming language released was FML, using a declarative style to express network policies and abstract the complexity of OpenFlow rules. FML was released just one year after the release of the first OpenFlow specification and it is based on NOX controller (GUDE, *et al.*, 2008). The following FML policy snippet present in (HINRICHS, *et al.*, 2009) exemplifies its declarative style. In such snippet the user *todd* is defined as superuser without communication restrictions:

```
allow( $U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req$ )  $\Leftarrow$  superuser( $U_s$ )
superuser(todd)
```

Where  $U_s$  means source user,  $U_t$  means target user;  $H_s$  means source host and  $H_t$  target host;  $A_s$  means source access point and  $A_t$  means target access point; *Prot* means protocol and *Req* verifies if the flow is a request.

Nettle language also enables the writing of network policies for developing SDN applications, starting the discussion about proactive and reactive SDN applications development. Besides the DSL, Nettle uses *Functional Reactive Programming* (FRP) paradigm in its design, which means it is compatible with reactive controllers by supporting the implementation of policies that reacts according to events.

Procera and NetCore bring the offering of support to avoid conflicts among network policies, a desirable characteristic in programming SDN applications, which reduces the complexity in development. The FRP, DSL, and declarative paradigms, used earlier in Nettle and FML, were also used in Procera and NetCore. Such use has

particularities relative to expressiveness achieved by each language. The following code snippet shows an example of Procera in practice to illustrate the declarative style of such languages.

**proc** *world* → **do**

**returnA** -<

*λreq* → **if** *destIP req 'inSubnet' ipAddr 128 36 5 0 // 24*

**then** *allow* **else** *deny*

The code above *declares* that such procedure might allow only traffic to IP addresses in subnet 128.36.5.0 / 24. Thus, this type of building does not requires a programmer or network operator to know how to implement the blocking of traffic destined to the IP mentioned, these agents only need to declare what is allowed. Such scenario is the main characteristic of declarative paradigm.

Further, after Procera and NetCore releases, there are the Frenetic and Pyretic languages, which are slightly different, but make part of the same project (Frenetic project). They have distinct characteristics involving how modules are interpreted. For example, Frenetic allows parallel modules (e.g., access control parallel to load balancing), while Pyretic adds sequential execution to parallel support. Pyretic might execute an access control module in parallel to load balancing, defining after such execution the start of monitor usage module. These languages also use DSL, declarative, and FRP (for Frenetic) paradigms.

The Flog language applies ideas of FML, by allowing event-driven development (i.e., FRP paradigm), network state query, and processing information according to the facts generated by network events (e.g., new flow, unreachable device).

Nlog is another proposal for computing the network forwarding state, based on DSL and declarative paradigms, and it resembles FML semantically and in its characteristics of network state query.

Flowlog is a finite-state language, which restricts some buildings in order to simplify the reasoning about correctness and to install proactively rules into switches.

Last but not least, the language FatTire has one of the most narrow scopes when compared to other languages due to its focus on fault-tolerance, by offering features to handle the failures in an SDN application.

In order to summarize the description of each DSL discussed in this section we present the Table 1 below.

	Use Cases	Paradigm	Objective	Year	Limitations
<b>FML/FSL</b>	Routing,	DSL, Declarative, Rule-based, Logic Programming	Replace the various different configuration mechanisms and offer a higher level abstraction to express network behavior.	2009	Does not allow arithmetic constraints; Static policies; Does not address conflicting rules; Does not allow explicit negation in rule bodies.
<b>Nettle</b>	QoS, policy-based routing, load balancing.	DSL, Declarative, FRP.	Allow programming OpenFlow networks in a declarative style.	2011	Does not address conflicting rules.
<b>Procera</b>	ACLs, QoS, policy-based routing, load balancing.	DSL, Declarative, FRP.	Express reactive dynamic policies in a declarative way.	2012	Does not directly support issuing events or external queries ( <b>NELSON, et al., 2014</b> ).
<b>Flog</b>	ACLs, QoS, policy-based routing, load balancing.	DSL, Declarative, Rule-based, Logic Programming.	Provide an event-driven and forward-chaining language to each time a network event occurs the logic program executes.	2012	Does not allow explicit negation in rule bodies.
<b>NetCore</b>	Network monitoring, ACLs, QoS, policy-based routing, load balancing.	DSL, Declarative, FRP.	Allow programmers to describe <i>what</i> network behavior they want, without <i>how</i> it should be implemented.	2012	Can't reference the state on the controller.
<b>Frenetic</b>	Network monitoring, ACLs, QoS, policy-based routing.	DSL, Declarative.	Raise the level of abstraction for writing controller programs for SDN, offering ways to query the network state, and define forwarding policies.	2013	Only provides consistency on a single switch for each flow.
<b>FatTire</b>	Network monitoring, policy-based routing, fault-tolerance.	DSL, Declarative, FRP.	Write programs in terms of paths through the network and explicit fault-tolerance requirements.	2013	Failure-recovery and detection mechanisms not integrated; Does not address QoS and performance requirements.
<b>Pyretic</b>	Network monitoring, ACLs, QoS, policy-based routing.	Imperative, DSL.	Specify network policies at a high level of abstraction.	2013	Only provides consistency on a single switch for each flow.
<b>Nlog</b>	Network monitoring, policy-based routing.	DSL, Declarative, Rule-based, Logic Programming.	Compute the network forwarding state and separate the logic specification from the controller that executes such logic.	2013	Does not offer a way to verify the correctness of SDN applications; Does not allow explicit negation in rule bodies.
<b>Flowlog</b>	ACLs, QoS, policy-based routing.	DSL, Rule-based, Declarative, Logic Programming	Abstract the data-plane and control-plane behaviors, allowing reason about the semantics of SDN applications and its code.	2014	Does not have abstractions for queries.

**Table 1: Summary of characteristics for each SDN-based DSL identified in literature.**

Most of information present in Table 1 can also be found at (**KREUTZ, *et al.*, 2014**), the main difference is relative to our view on limitations for each SDN programming language, in order to verify if our proposal can contribute to address some of them.

### 3.2 Modeling Approaches for SDN

Increasing the abstraction in development or management of SDN applications is the objective of several research projects as abovementioned. To the best of our knowledge, besides the proposals involving textual DSL-based programming languages for SDN discussed above, the closest related studies found in literature involving the modeling of SDN and its applications are the following: Miniedit (**LANTZ, HELLER e MCKEOWN, 2010**), Virtual Network Descriptor (VND) (**FONTES e SAMPAIO, 2013**), and Common Information Model for SDN (CIM-SDN) (**PINHEIRO, *et al.*, 2013**).

Miniedit refers to the creation of virtual topologies in SDN through a graphical interface. These topologies are simulated at Mininet, the underlying SDN simulator (**LANTZ, HELLER e MCKEOWN, 2010**). From Miniedit, it is possible to create and organize the network nodes, such as controllers, forwarding devices, and hosts. The graphical interface of Miniedit with a topology to illustrate its features is depicted at Figure 3.2. Such topology is composed of four hosts, seven switches, and three controllers. The hosts h1 and h2 are physically connected to switches s2 and s3, which are connected with switch s1. The controller c0 manages the logic of such switches. On the other hand, the controller c2 manages the logic of switches s5, s6, and s7, the last two connect the hosts h3 and h4. Finally, the switch s4 connects switches s1 and s5, it is logically managed by controller c1.



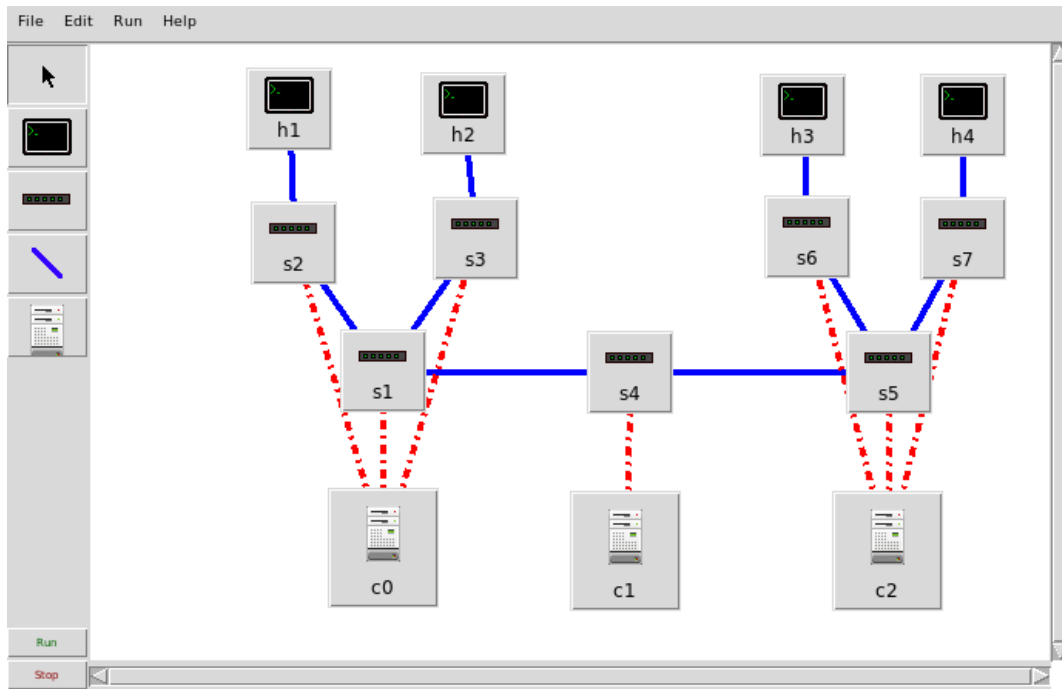


Figure 3.2: Graphical interface of Miniedit.

Another tool for modeling SDN is the VND. An editor aims to describe the structure and part of the SDN behavior through a Graphical User Interface (GUI). It uses a framework called Network Scenario Description Language (NSDL) (**MARQUES e SAMPAIO, 2012**) to define the SDN elements, also supporting the generation of scripts for Mininet simulator and OpenFlow controllers. The proposal of VND was adapt the NSDL framework to support the description of SDNs, since the original proposal of NSDL was to describe traditional networks. Figure 3.3 depicts the modeling of SDN topologies in VND.

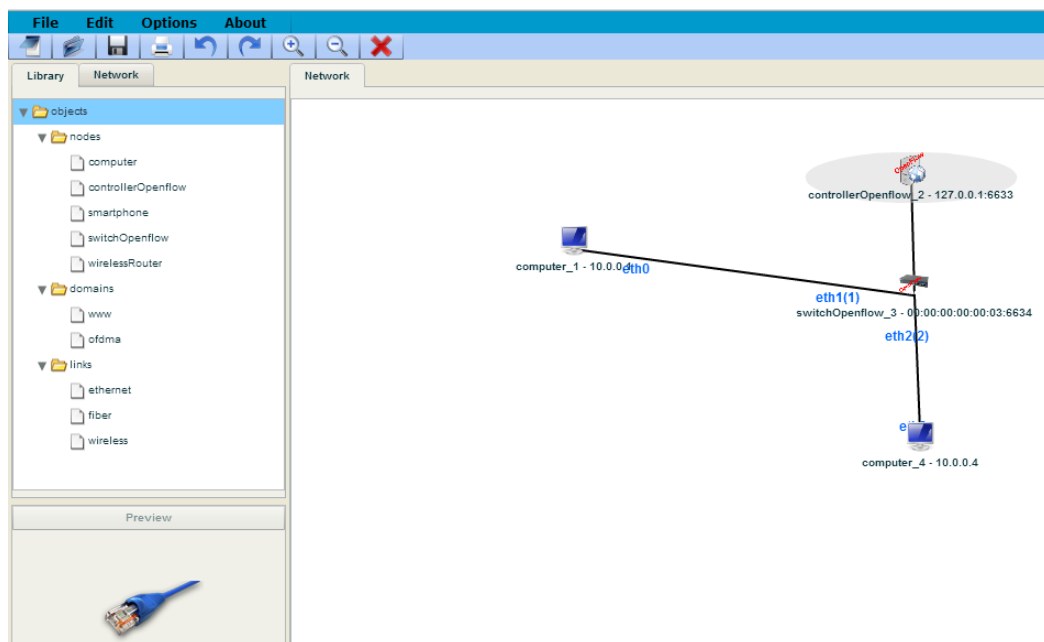
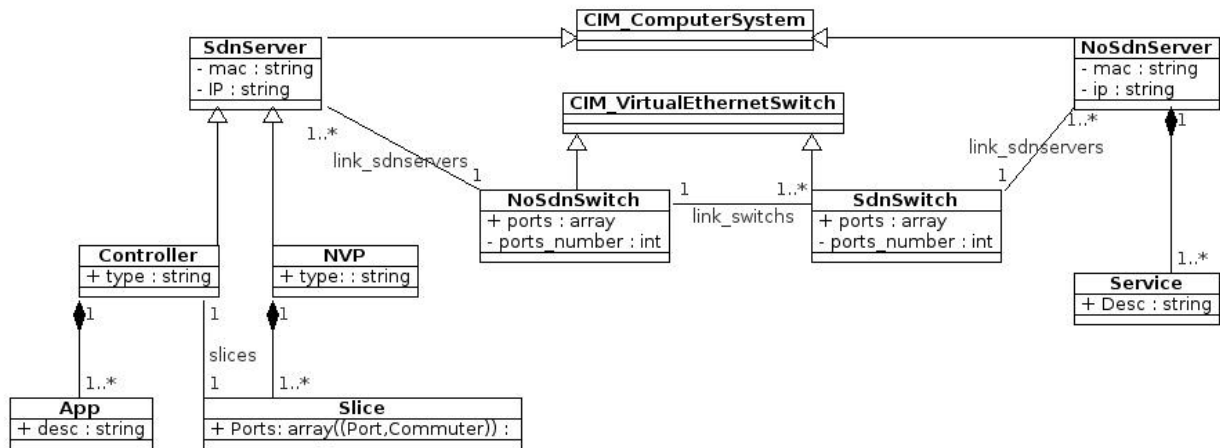


Figure 3.3 Topology model in VND.

The NSDL framework used in VND provides a vocabulary that composes a set of rules, which enable the description of traditional networks (e.g., topology, characteristics, and network perspectives). NSDL also offers a *Graphical User Interface* (GUI) for topology generation (**MARQUES e SAMPAIO, 2012**). Although VND can describe SDN topologies, it only enables the definition of QoS for network elements and cannot model applications, such as firewall or load balancing.

Last, but not least, the *Common Information Model extension for Software-Defined Networking* (CIM-SDN) was recently proposed (**PINHEIRO, et al., 2013**). It offers an abstraction model and a client editor for SDN management, by modeling SDN elements (e.g., hosts, controllers, switches) and by reducing the complexity in defining their properties. CIM-SDN also validates an SDN structure through OCL, finding inconsistencies and avoiding error-prone buildings in network. The abstract syntax of MDN resembles the CIM-SDN.

The CIM-SDN approach extends the MDA's *Common Information Model* (CIM) to support the description of SDN elements. It is based on an SDN controller named FlowVisor (**SHERWOOD, et al., 2009**), since it deals with slices of the network (i.e., *Slice* class) that are the focus of such controller. The extension of CIM model for SDN is depicted at Figure 3.4.



**Figure 3.4: Class diagram of CIM-SDN.**

CIM-SDN enables the building of CIM models for describing of slices in an SDN environment. It also may model the nodes of SDN, as well as its properties (e.g., *SdnSwitch* class enables the definition of how many ports a switch has). However, due to its foundation on CIM model, it is dependent on UML to perform the modelling of SDNs.

To the best of our knowledge, our MDN approach is the first DSML for SDN. Moreover, a number of features make MDN a novel approach to address the issues that currently exists for SDN application development. At the time of this writing, MDN is the only one to unify the description of SDN scenarios (e.g., topology, properties) with the development of SDN applications through high-level models. MDN addresses issues not found in other proposals, such as the compatibility between applications and different controller vendors, as well as the validation of SDN topologies and applications.

## 4. The MDN Framework

---

This chapter describes the methods and the infrastructure of MDN, which consists of techniques and software built following the MDE paradigm to support the approach and technology presented in this dissertation. We start describing the MDN architecture at section 4.1. After we present the building process MDN in section 4.2 and its artifacts (section 4.3). Then, we present details of code generation (section 4.4) and conclude this chapter by presenting the MDN Editor and its development process (section 4.5).

---

### 4.1 Overview of MDN Architecture

The general perspective of MDN involves elements of a MDE technology applied to SDN applications domain. In this context, we consider the MDN architecture with three layers related to each other (i.e., Model, MDE Technology, and SDN Domain-Specific Elements). The base of such architecture, depicted in Figure 4.1 is the set of concepts and specification of SDN domain used.

Due to MDN architecture layers, we can achieve a higher abstraction level in creating SDN applications through modeling.

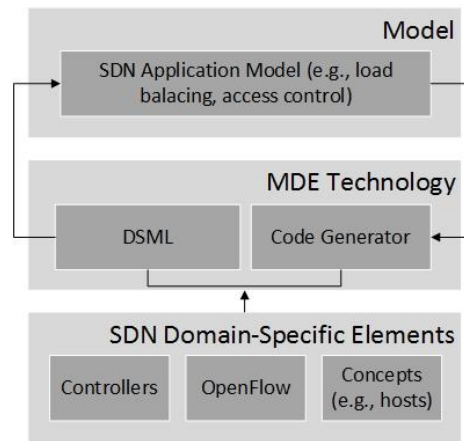


Figure 4.1: Simplified view of the MDN Architecture.

## 4.2 Building Process of MDN Infrastructure

The concepts defined in the previous chapters provide the necessary background to understand a prospective relationship between MDE and SDN, which results in the MDN approach. In this section, we present the workflow for the creation of MDN infrastructure, used to offer a new way in developing SDN applications. Such workflow follows a main process defined to create a model-based tool using the Eclipse's framework (i.e., GMF), which relates with MDE concepts.

### 4.2.1 Specifying the Domain

Initially, in order to offer the DSML and code generator of a model-based approach, such as MDN, first it is necessary to define the domain scope and its modeling concepts for such modeling language, which may be coming from available specifications, architecture, existing products, code, and so on (**KELLY e TOLVANEN, 2008**). Although to date there is no well-known standard specification for SDN applications, this work used their underlying concepts (e.g., SDN controllers, OpenFlow), relationships, and the basic elements of SDN architecture, already defined in the literature (cf. section 2.1.1).

Furthermore, we carried out an investigation about SDN programming languages, identifying several domain-specific languages in literature, such as FML (**HINRICHS, et al., 2009**), Nettle (**VOELLMY, AGARWAL e HUDAK, 2011**), and Pyretic (**MONSANTO, et al., 2013**) (for an extensive list see chapter 3, (**HU, HAO e BAO, 2014**) and (**LARA, KOLASANI e RAMAMURTHY, 2013**)). Due to common characteristics of such languages in abstracting OpenFlow complexity for specifying network policies, such as relational operators, packet flow identification, and performing

actions, the MDN approach was inspired by the plusses of them in its domain specification.

According to Kelly and Tolvanen (2008), it is possible to use natural language in describing the domain scope. Thus, Table 2, based on OpenFlow and SDN use cases (JARSCHER, *et al.*, 2014), presents an overview of the domain scope and its concepts involved in specifying the MDN approach. Such table is composed of the main elements that enable the creation of SDN applications by modeling.

Concepts	Description
<b>Network Node</b>	A general concept involving specific network components, such as host, controller, and switch.
<b>Link</b>	Component to connect network elements (e.g., controllers, switches, hosts, rules).
<b>Controller</b>	The logical part of SDN environment in which switches are managed, applications deployed, and network behavior programmed.
<b>Switch</b>	It makes part of forwarding layer, connecting hosts, controllers, or even other switches.
<b>Host</b>	Clients or servers in the network.
<b>Traffic</b>	Network traffic comprehends the amount of data used or the network traffic rate.
<b>Flow</b>	The SDN with OpenFlow deals with traffic by verifying its flows properties (e.g., packet header, incoming/outgoing port, source, and destination) and installing rules in flow tables.
<b>Rule</b>	In SDN, network rules are specified through algorithms running on controllers.
<b>Relational Operator</b>	In SDN programming languages, relational operators are used to compose conditional rules in applications (e.g., equal to, greater than, less than).
<b>Action</b>	The OpenFlow specification (version 1.4) defines actions for each flow entry (e.g., discard, forward, deny).
<b>Time</b>	Network behavior may change over the time (e.g., vacation period, big time rush).
<b>Condition</b>	Some rules of SDN applications are applied according to certain conditions.

**Table 2: Main concepts of SDN that compose MDN approach.**

In section 2.1.1, we have stated the main components of the SDN architecture: controllers, forwarding devices, SDN protocols (e.g., OpenFlow), hosts and applications. Except the SDN southbound protocol, which is underlying to SDN controller and switch, such components are presented in domain specification. Besides, some characteristics from literature (MONSANTO, *et al.*, 2013) involving network behavior and network structure are also described at Table 2, i.e., *Traffic*, *Flow*, and *Rules*.

After we have used natural language to express the main concepts of domain scope, we have specified the building process of MDN metamodel, defining the relationships and attributes for each concept above. The next section outlines such metamodel, among other artifacts that compose the MDN approach.

### 4.3 Artifacts

Besides domain scope, the following elements compose a DSML: abstract and concrete syntaxes, semantic domain, and mappings (semantic and syntactic). Hereinafter, we present these elements in the MDN specification, in order to demonstrate its structure. We already defined the use of framework GMF to develop the MDN infrastructure. The new element in this scenario is the Epsilon framework (**KOLOVOS, et al., 2014**), a family of languages and tools for code generation, model validation, migration, among others, which provides facilities in implementing EMF and GMF components.

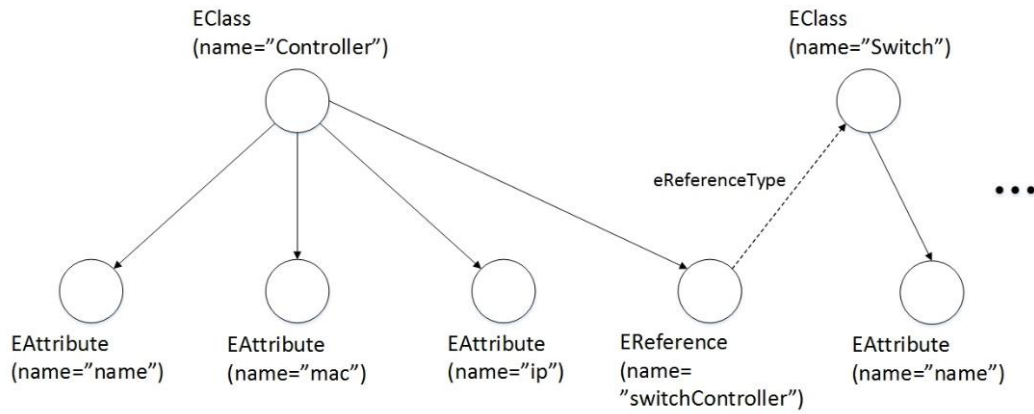
#### 4.3.1 Abstract Syntax

The metamodel discussed in this section, also known as abstract syntax of DSML, represents a view that involves the structural characteristics of MDN core. Such view is the base for the others artifacts of DSML (i.e., concrete syntax, mappings, and semantics).

Besides the identification of concepts defined at Table 2, we also have used the version 1.4 of OpenFlow specification in defining abstract syntax (**ONF, 2013**). For example, such specification defines that one controller can be linked with many switches, as well as one switch can be linked with many controllers. Thus, by following the concept of cardinality present in modeling approaches (e.g., classes diagram, UML), this is a *many-to-many* relationship, which needs to be specified in metamodel.

Firstly, our methodology in building MDN related each main component of SDN architecture (e.g., controller, switch) and described it in the metamodel by using an essential version of MOF, named EMOF. As pointed out in section 2.2.6, a reference implementation of EMOF was created by Eclipse, namely Ecore. It grants that concepts of domain are formally described in abstract syntax with a meta-metamodel (M4 of Bézivin's 3+1 MDA organization), hereinafter named Ecore model. Furthermore, Ecore is part of EMF/GMF frameworks.

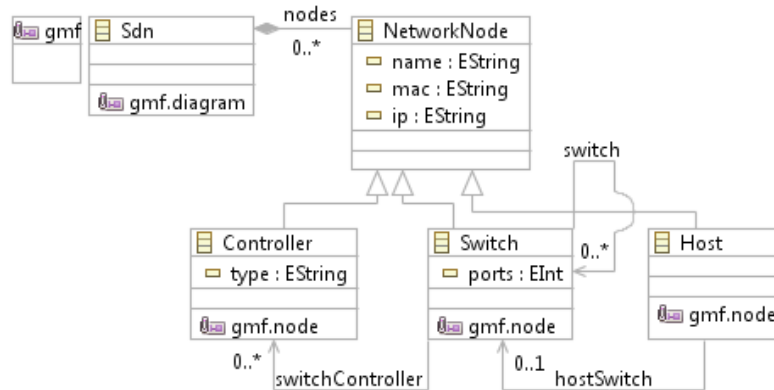
Figure 4.2 shows a brief example of implementing domain concepts of SDN architecture using Ecore and its classes. To perform such implementation, EMF provides an Ecore editor in which graphical elements and Java annotations that define an Ecore model. For another view of this initial implementation, the script describing this metamodel of SDN architecture in MDN core is available in the appendix A.



**Figure 4.2: Example of MDN approach being implemented in Ecore instances.**

Thus, it is feasible to use instances of Ecore classes to describe the infrastructure of MDN. For instance, Figure 4.2 shows the description of SDN controller as an EClass instance named *Controller*. It contains three attributes: *name*, *mac*, and *ip*, for which its target reference (EReference) is equal to another EClass instance named *Switch*. We performed such a description for each item listed in Table 2 through a direct editing using the Ecore editor present in Epsilon.

After we had implemented the domain concepts of SDN architecture in Ecore, it is possible to view at Ecore editor an UML-based diagram exhibiting the abstract syntax and core metamodel with the classes and relationships of SDN architecture defined for the MDN approach. In other words, the elements of SDN architecture such as *Controller*, *Switch*, and even *Host* are nodes of *Sdn*, which is the diagram to model these elements. The nodes of *Sdn* diagram also have the following relationships connecting them: *switchController* (switch may link with controller), *hostSwitch* (host may link with switch), and *switch* (switch may link with other switch). Such relationships are consistent with OpenFlow specification and SDN architecture itself, where hosts are connected with switches in the forwarding plane. Figure 4.3 depicts this scenario.



**Figure 4.3: MDN's core metamodel.**

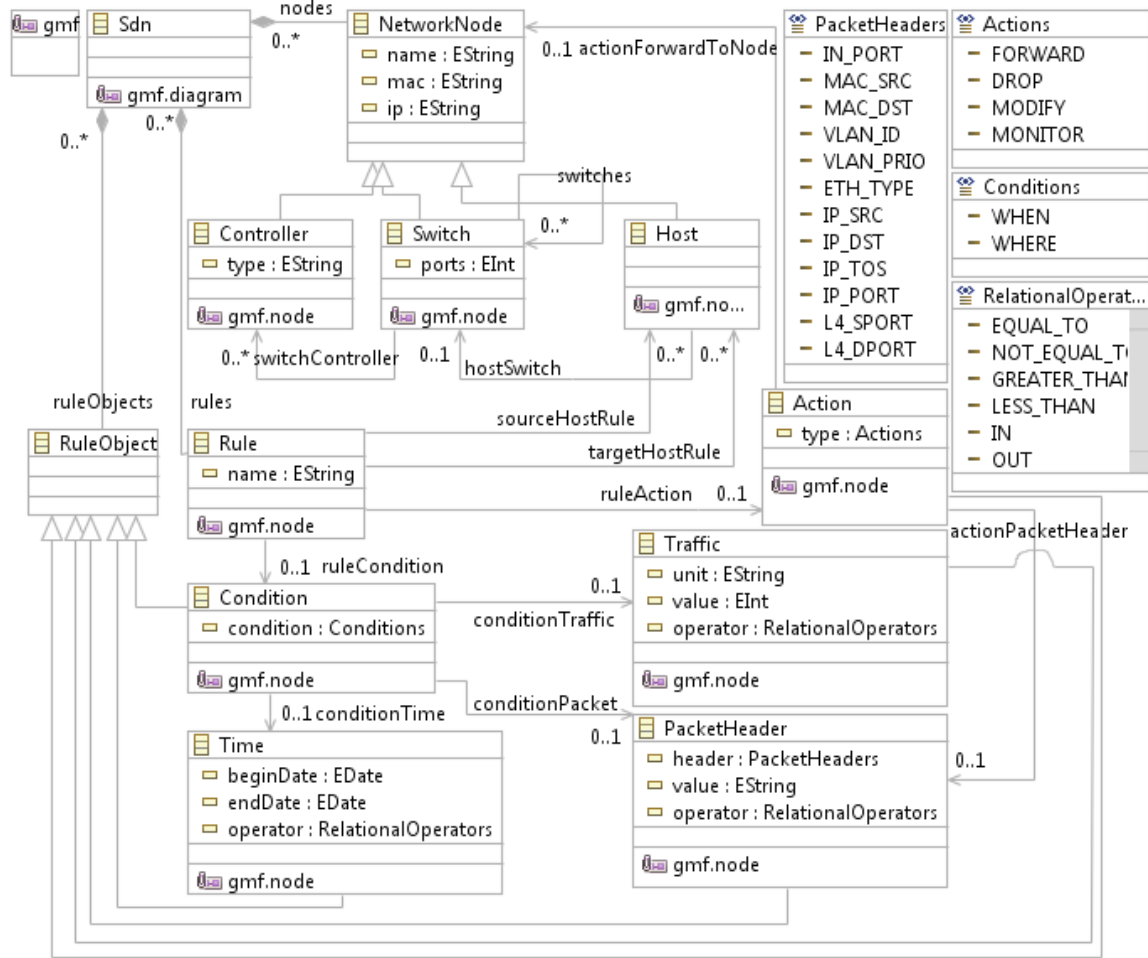


In order to clarify the definition of abstract syntax and its specification process, the building shown in Figure 4.3 represents a metamodel in which only the modeling of SDN architecture is possible. Thus, it is need to outline the increments performed for that MDN structure enables the development of SDN applications. The organization of such increments in abstract syntax still has relation with the OpenFlow specification in its version 1.4 and SDN architecture (**ONF, 2014**).

As seen in sections 2.1.5 and 2.1.6, the programming of SDN is performed in controllers through rules in the shape of algorithms. Such programming may involve several use cases. However, based on literature, we have defined three main features related to SDN applications: traffic monitoring, policies and rules specification, and validation (**MONSANTO, et al., 2013**).

These features involve common elements, such as traffic, concessions (e.g., allow or deny some behavior), and monitoring of network nodes or devices. In general, the elements of Table 2 cover the components necessary to program applications in SDNs. In this scenario, we have inserted these components into abstract syntax, but the use of them depends on the SDN application to be modeled.

Thus, MDN metamodel should enable that rules to be modified and related with the respective controller, which will relay such rules to underlying switches. We have achieved such actions by adding the *rule* element into the metamodel and relating it with the controller (*Controller*) and network nodes (*NetworkNode*). Thus, the *rule* element may define the behavior of both (i.e., controller and network nodes). However, as seen earlier, in SDNs, rules are algorithms that may be related to several network elements (e.g., traffic, switches, and hosts). Then, there is a need to enable the modeling of such elements, also by inserting them into MDN metamodel. The following elements were inserted directly or indirectly (i.e., a combination of classes in metamodel): *Traffic*, *Flow*, *Rule*, *Concession*, *RelationalOperator*, *Action*, *Time*, and *Condition*. Figure 4.4 depicts this addition and the metamodel correlated with M2 level of Bézivin's 3+1 architecture (**BÉZIVIN, 2005**).



**Figure 4.4: Full metamodel of MDN.**

The description of each element inserted in MDN metamodel (as an *EClass*, *EEnum*, or *EReference*) is present in Table 2. However, it is worth to underline the meaning of the relationships between them and the reason why they were inserted in the metamodel:

- i. The *Sdn* entity and its aggregations (e.g., *ruleObjects*, *nodes*) is a concept relative to the GMF in creating the graphical editor. Similar purpose is related to *RuleObject* entity. It only present in such metamodel to enable the creation of classes that are inheriting its compositionality in *Sdn* diagram. Annotations such as *gmff.diagram* and *gmff.node* also appear on MDN metamodel due to their use by GMF.
- ii. *Rule* is one of the most important elements of MDN metamodel. In SDN, the specification of rules in MDN is equivalent to the writing of algorithms that composes an SDN application. The result of such specification involves the insert of rules in flow tables, which are related with the source and destination of a network flow or packet depending on certain characteristics (i.e., *Condition*) of

the network or flow itself (i.e., *Time*, *Traffic*, *PacketHeader*). As we have stated in section 2.1.4, such rules define some type of action to be made (i.e., *Action*) when a flow (e.g., packet forwarding between two *hosts*) matches the characteristics or conditions present in flow tables. Thus, we linked the *Rule* element with the abovementioned elements. Note that such *Rule* element is not related with *Controller* element in our metamodel due to semantic of our concrete syntax. However, the code generation verifies the hosts related in a *Rule* element. Such verification goes deeper by identifying the switch that connects such hosts. After, it finds the controller that manages this switch.

- iii. *PacketHeader*, *Actions*, *Conditions*, and *RelationalOperators* are enumerations containing the accepted values for each homonymous class. For instance, OpenFlow specification defines that switches identify the packets by verifying their headers, e.g., incoming port (*IN\_PORT*), source IP (*IP\_SRC*), destination MAC (*MAC\_DST*), and so on (for extensive list see Figure 2.5 or (**ONF**, 2013)). Another definition from OpenFlow specification is the possible actions in switches. MDN approach defined such actions (*Actions*) in three values to be accepted by a model of SDN application, namely, (i) *FORWARD*, which may send the packet flow to the normal processing pipelining, to the controller, or to another port, or may multicast; (ii) *DROP*, which discards the packet; and (iii) *MODIFY*, which changes some value or characteristic of the packet. On the other hand, *Conditions* and *RelationalOperators* have their utility when there is a need to compare the packet flow with some network requirement, e.g., *WHEN* the traffic transmitted from host A is *GREATER\_THAN* 1 gigabyte, discard its packets.












These statements describe more than abstract syntax of MDN through metamodeling, they build a base and give an overview about how the modelling of SDN and its applications may occurs with MDN.

### 4.3.2 Concrete Syntax

The concrete syntax defines a graphical element for each concept present in abstract syntax (e.g. *Controller*). The same occurs with the links type involving the relationships among entities (e.g., links between hosts and switches). Although we present and discuss the concrete syntax of MDN and its graphical elements, there is room for improvements such as the verification of cognitive aspects in their characteristics.

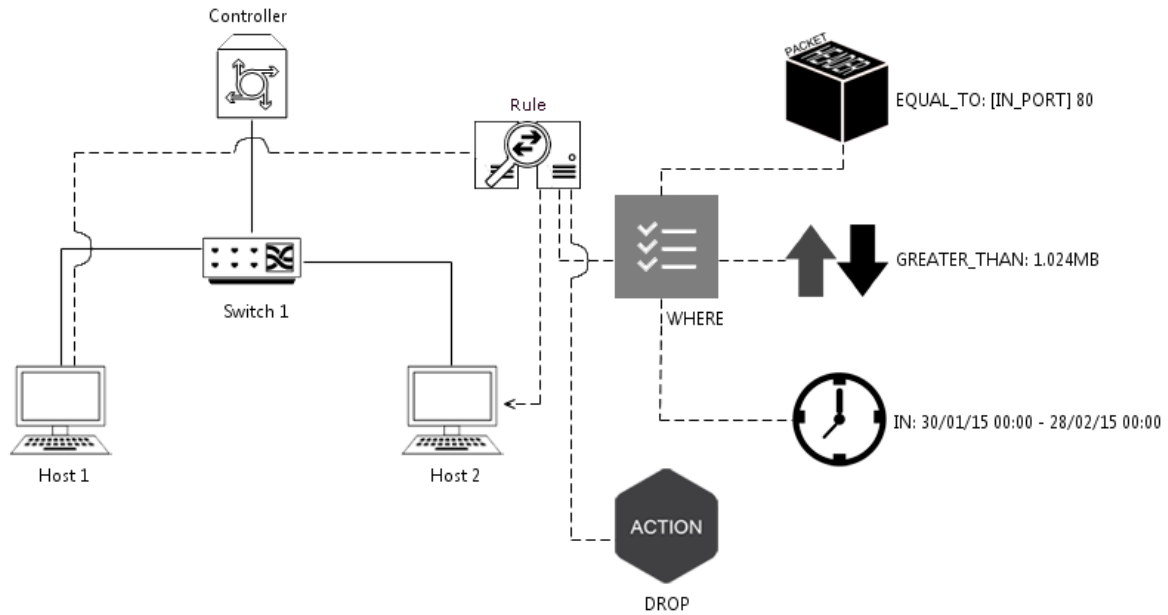
Currently, MDN offers simple black and white graphical elements, but with denotational meaning in relation to the concepts that they represent.

In order to represent graphically each element of SDN domain scope, our MDN approach uses several symbols (cf. Table 3). For instance, due to the wide use in documenting network diagrams, we used network shapes present in Microsoft Visio in the graphical representation of topology elements (e.g., host, switch, and controller). On the other hand, we added specific graphical style for each type of connection present in an SDN application model (i.e., solid lines for physical links and dashed lines for rule links) in order to differentiate the connections between network nodes and rules. Furthermore, MDN also defines new icons to enable the modelling of graphical underlying concepts such as *packet header*, *traffic*, and *relational operators*, increasing the level of abstraction in using such concepts.

Visual Notation	Description	Properties
	<b>Controller:</b> represents the SDN controllers.	<i>name (EString); mac (EString); ip (EString); type (EString).</i>
	<b>Switch:</b> represents the forwarding devices.	<i>name (EString); mac (EString); ip (EString); ports (EInt).</i>
	<b>Host:</b> represents the end nodes of the network.	<i>name (EString); mac (EString); ip (EString).</i>
	<b>Rule:</b> represents the constraints that define the network behavior.	<i>name (EString).</i>
	<b>Condition:</b> composes the network rules, conditioning their effectiveness.	<i>condition (Conditions).</i>
	<b>Packet Header:</b> represents the headers of packets present in flows. It is used to describe some <b>Condition</b> or perform some <b>Action</b> .	<i>header (PacketHeaders); value (EString); operator (RelationalOperators).</i>
	<b>Time:</b> represent the time dimension in the network. It is used to describe some <b>Condition</b> .	<i>beginDate (EDate); endDate (EDate); operator (RelationalOperators).</i>
	<b>Traffic:</b> represent the network traffic, specially from hosts. It is used to describe some <b>Condition</b> and perform some <b>Action</b> .	<i>unit (EString: MB, GB); value (EInt); operator (RelationalOperators).</i>
	<b>Action:</b> network rules define some action to perform when some flow matches the conditions.	<i>action (Actions).</i>
	<b>Physical Link:</b> represents the element used to connect two network nodes.	
	<b>Rule Link:</b> represents the element used to connect virtually rules and nodes.	

**Table 3: Visual notation of MDN approach.**

In order to illustrate how the visual notation elements of concrete syntax are organized in MDN, we have created the example model depicted in Figure 4.5. Such model demonstrates the graphical representation and relationships of each element that compose the concrete syntax. It is a simple topology (similar to Figure 2.2) composed of one controller, two hosts, and one switch connecting them. Furthermore, there are elements that compose a *Rule* entity, i.e., *Condition*, *Traffic*, and *Action* (cf. Table 3).



**Figure 4.5: Example of visual notation elements in a MDN diagram.**

The meaning of model depicted in Figure 4.5 may be defined as an SDN environment composed of two hosts (*Host 1* and *Host 2*), one switch (*Switch 1*) connecting them, and one controller (*Controller*) managing this switch. Such environment has a rule (*Rule*) programming the network, specifically hosts 1 and 2, which drops (*Action*) the packets at port 80 (see the property of *PacketHeader* symbol<sup>7</sup>) from *Host 1* to *Host 2* if the traffic reaches more than 1024MB. The date and time in which such action performs need to be in the period specified at *Time* element.

### 4.3.3 Semantic Domain

The abstract syntax conveys limited information in defining the meaning of concepts in a DSML. The same limitation applies to concrete syntax. Thus, the artifact that is able to represent the domain concepts describing particularities of each element, particularly those elements that are present in the abstract syntax, is the semantic domain, which

<sup>7</sup>*EQUAL\_TO: [IN\_PORT] 80*).

consists of a set of descriptions and constraints for concepts belonging to domain scope of the DSML. Indeed, it is used to define the behavior and meaning of abstract syntax elements.

We have based the definition of MDN's semantic domain on *translational* and *operational* approaches (cf. section 2.2.3) by using natural language for presenting and well-formed rules written with EVL for implementation (KOLOVOS, *et al.*, 2014) (KOLOVOS, PAIGE e POLACK, 2009). For instance, considering the element *NetworkNode* from MDN metamodel, it is the *EClass* reference for the elements *Controller*, *Switch*, and *Host*. Thus, according to the body of knowledge in networks and the OpenFlow specification, there are three rules that are not possible to grant by metamodeling and need to be set out with EVL: 1) elements of *NetworkNode* type should have a name; 2) such elements must not have the same MAC; and 3) They need valid IPs.

Our example present in Table 4 shows a summary of rules defined by using natural language and EVL. In such a way, we have defined the resting semantic domain of MDN and it is fully available at appendix A.

Element	<i>NetworkNode</i>
Rule #1	Should has a name.
Rule #2	Should not has two identical MACs in the network.
Implementation	<pre> 1. context NetworkNode { 2.   critique hasName { //Rule #1 3.     check : self.name.isDefined() 4.     message : 'Unnamed ' + self.eClass().name.toUpperCase() + ' not allowed' 5.     fix { 6.       title : 'Define the name of the node ' 7.       do { 8.         var type := UserInput.prompt('What is the name?'); 9.         if (type.isDefined()) self.type := type; 10.      } 11.    } 12.  } 13.  constraint uniqueMAC { //Rule #2 14.    check { 15.      var networkNodes = 16.        NetworkNode.all.select(nn nn.mac = self.mac); 17.      return networkNodes.size() = 1; 18.    } 19.    message : 'Not unique MAC ' + self.mac + ' not allowed' 20.    fix { 21.      title : 'Define the correct MAC ' 22.      do { 23.        var mac := UserInput.prompt('What is the MAC?'); 24.        if (mac.isDefined()) self.mac := mac; 25.      } </pre>

Table 4: EVL rules for *NetworkNode* semantic.

### 4.3.4 Mappings

In section 2.2.3, the specification process of DSMLs was introduced. As seem, such specification has two types of mappings to grant well-formed models by using DSMLs, i.e., semantic mapping and syntactic mapping. The former was made by relating the underlying concepts of SDN with the abstract syntax of MDN and its EVL rules. The latter was defined through the EMF framework by relating each *EClass* and *EReference* of metamodel with their corresponding graphical elements. The following lines attempt to clarify these mappings.

The semantic mapping was defined through the scheme present in Figure 4.6, relating concepts of MDN with its abstract syntax.

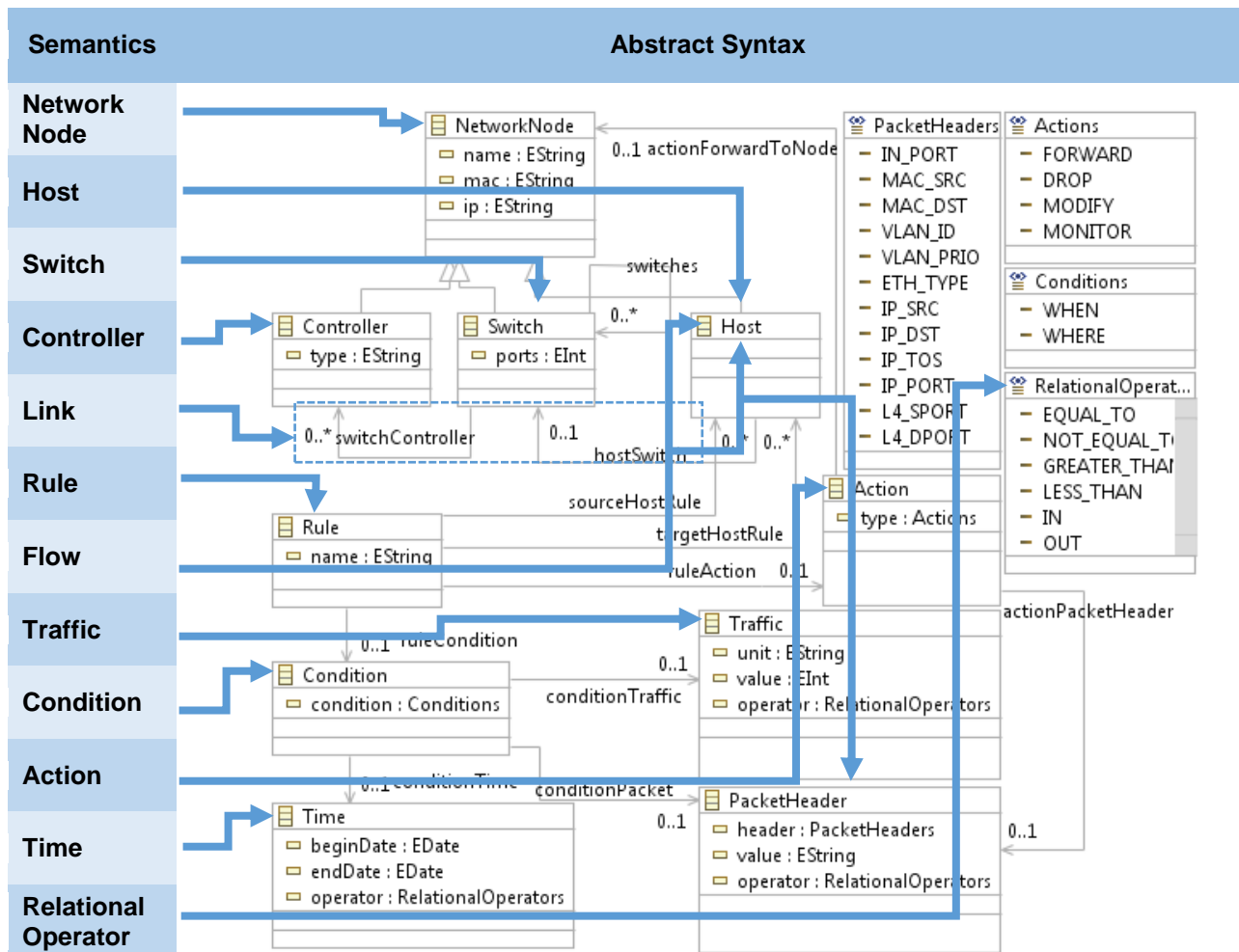


Figure 4.6: Semantic mapping of MDN.

Regarding the syntactic mapping, MDN approach enables to specify a graphical symbol for each element present in abstract syntax through the EMF framework and the use of Java annotations. For instance, the syntactic mapping between *Rule* element and its visual notation was performed from the code snippet present in Table 5. The Java annotation `@gmf.node` defines the graphical representation for the respective

*class Rule*, which is the *RuleFigure* (cf. section 4.3.2). Inside such class, there are three Java annotations of *@gmf.link* type, which defines the style of connections between rule and other three elements (i.e., *Host*, *Condition*, and *Action*) by setting the style of such connections and in customizing four attributes: *target-decoration*, *source-decoration*, *style*, and *color*.

Element	Rule
Mapping	<pre> 1. @gmf.node(figure="figures.RuleFigure") 2. class Rule {  3.     @gmf.link(target-decoration="arrow", source-decoration="none", 4.         style="dash", color="0,0,0") 5.     ref Host targetHostRule; 6.     attr String name;  7.     @gmf.link(target-decoration="none", source-decoration="none", 8.         style="dash", color="0,0,0") 9.     transient ref Condition ruleCondition;  10.    @gmf.link(target-decoration="none", source-decoration="none", 11.        style="dash", color="0,0,0") 12.    ref Action ruleAction; 13. } </pre>

**Table 5: Syntactic mapping for the *Rule* element and its relative graphical symbol.**

The entire syntactic mapping of MDN approach followed this correlation using annotations and it is available at appendix A.

## 4.4 Code Generation

The feature of code generation is a fundamental component of any DSML. Furthermore, the EMF framework, used in MDN infrastructure in defining our abstract syntax, also provides code generation support through the specification of code templates. Such templates are specified by writing Eclipse Generation Language (EGL) tags, operations, and static text. EGL tags are replaced by information present in the model of a particular application (e.g., host IP and rule name), EGL evaluates a model and retrieves the modeled elements as well as their properties.

SDN controllers have two main different types of operation to handle packets or flows, namely proactive and reactive. Such operation modes are implemented according to message patterns (e.g., OpenFlow messages (**ONF, 2013**)) exchanged between controller and switch. Thus, the MDN approach enables modelling of applications in such modes; it is performed by verifying the conditions to apply rules



desired by an end-user or required by an application (e.g., time conditions, traffic limit, and the like).

Although there are several controllers available to deploy an SDN environment, currently the MDN infrastructure offers support for the POX controller due to characteristics like open-source development model, Apache license, and modularity (e.g., packet forwarding and firewall as modules). Furthermore, in order to simulate and validate the modeled applications, MDN enables the transformation of topologies present in MDN models into Mininet scripts.

The code generation feature was enabled by adding an action button at MDN graphical editor (cf. section 0). When the SDN application modeling is done, the end-user pushes such button and the MDN tool performs a process similar to a compilation in traditional programming. Then, the tool uses the template associated with the button and creates a new file with generated code and ready to run as application module in the controller.

#### 4.4.1 Templates for Code Generation

At beginning of this section, we defined that EMF framework and its EGL language make use of code templates to perform the code generation, by verifying created models and replacing tags present in templates for elements and properties of such models. In the case of MDN approach and its underlying DSML, the specification of abstract and concrete syntaxes demonstrated that an SDN application may consider the network topology in its behavior and it is based on rules, which must perform certain action depending on some condition (cf. Figure 4.5). Thus, the EGL templates of MDN editor to generate code must have tags and operations that satisfy the concepts involved in SDN applications.

In such a way, the code generation of MDN approach considers available information in application models. It makes the automatic building and declaration of several variables, structures, and methods without the direct involvement of end-user (e.g., network operator). The Table 6 demonstrates part of main template *sdn.egl* used in generating code. For a complete listing, the appendix A has the full *sdn.egl* and other templates used.

At Table 6 is displayed the combination between EGL tags (e.g., operations, conditional commands) and static text out of them (the text out of “[% %]”), composing the Python code used to program the POX controller. The result in replacing EGL tags

by information present in the model is the SDN application that is executed by POX. Furthermore, such template is composed of five other *.egl* files (i.e., it imports). Each of them is responsible to perform some specific action on final code. For instance, the *header.egl* imports the possible libraries of POX controller that may be used on SDN applications.

Template	sdn.egl
Objective	It defines the possible libraries used in code generated of MDN models.
<pre> 1. [% 2. import "header.egl"; 3. import "utils.egl"; 4. import "condition.egl"; 5. import "firewall.egl"; 6. import "monitor.egl"; 7. %] 8. [% var header = getCodeHeader(); %] 9. [%=header%] 10.  [% var counterActionDrop : Integer = 0; %] 11.  [% var counterActionMonitor : Integer = 0; %] 12.  def mdn_handler (event): 13.  # Handles packet events and kills the ones with blocked  property 14.  packet = event.parsed 15.  [% 16.  for (rule in Rule.all) { 17.    if (rule.ruleAction.isDefined()) { 18.      //IF ACTION EQUALS TO DROP 19.      if (rule.ruleAction.type.value = Actions#DROP.value) { 20.        if (counterActionDrop == 0) { 21.          counterActionDrop = counterActionDrop + 1; 22.        } 23.        actionDrop(rule); 24.      } 25.      //IF ACTION EQUALS TO MONITOR 26.      if (rule.ruleAction.type.value = Actions#MONITOR.value){ 27.        if (counterActionMonitor == 0) { 28.          counterActionMonitor = counterActionMonitor + 1; 29.        } 30.        actionMonitor(rule); 31.      } 32.    } 33.  } 34.  %] 35.  def launch (): 36.  [% if (counterActionDrop &gt; 0) { %] 37.    core.openflow.addListenerByName("PacketIn", mdn_handler) 38.  [% } %] 39.  [...]</pre>	

**Table 6: Summarized *sdn.egl* template.**

Indeed, the SDN programming occurs in method `mdn_handler` (event) at line 11 from Table 6, it holds the verification of rules present at modeled application, identifying the required actions. It is from the actions of each rule (i.e., *rule* object at line 16) that the body of code is generated by retrieving information of network elements and conditions. The verifications of counters (e.g., lines 20 and 27) are used to avoid duplicated code in implementing such rules. For instance, the method `addListenerByName` at line 37 can not be defined twice with the same *PacketIn* parameter or it would cause an inconsistency.

The reactive and proactive operation modes are applied according to the message type exchanged between switch and controller. For instance, in line 37 it is defined a listener for *PacketIn* messages (**ONF, 2013**), which are output from switches to the controller. The listener is the way in which the controller reacts to such messages, in MDN it is handled by *mdn\_handler* method. The OpenFlow specification defines more than one type of message for reactive and proactive operation modes (see (**ONF, 2013**) for an exhaustive list).

#### 4.4.2 The support for different controllers

It is worth mentioning that the templates used to generate code must be defined not only according to modeled application characteristics but also with the underlying controller vendor. For instance, although this dissertation presents the MDN approach supporting the code generation for POX (based on Python), it is possible to generate compatible code for the OpenDaylight<sup>8</sup> (ODL) controller, which is based on Java programming language. MDN enables such support by using different EGL templates for each underlying controller.

This extensible characteristic enables the compatibility of MDN models with any SDN controller vendor without the need to perform any type of modification in syntaxes and mappings of underlying DSML. Thus, an application created following our MDN approach can be migrated to different SDN scenarios (e.g., testing applications performances for distinct controllers) without code refactoring. There is the need only to generate code to the target controller.

The example at Table 7 shows the code template for ODL controller. Such code snippet is just a monitor application that notifies when the incoming packet IP is equal to that is defined in a rule of model.

---

<sup>8</sup> OpenDaylight Project - [opendaylight.org](http://opendaylight.org).

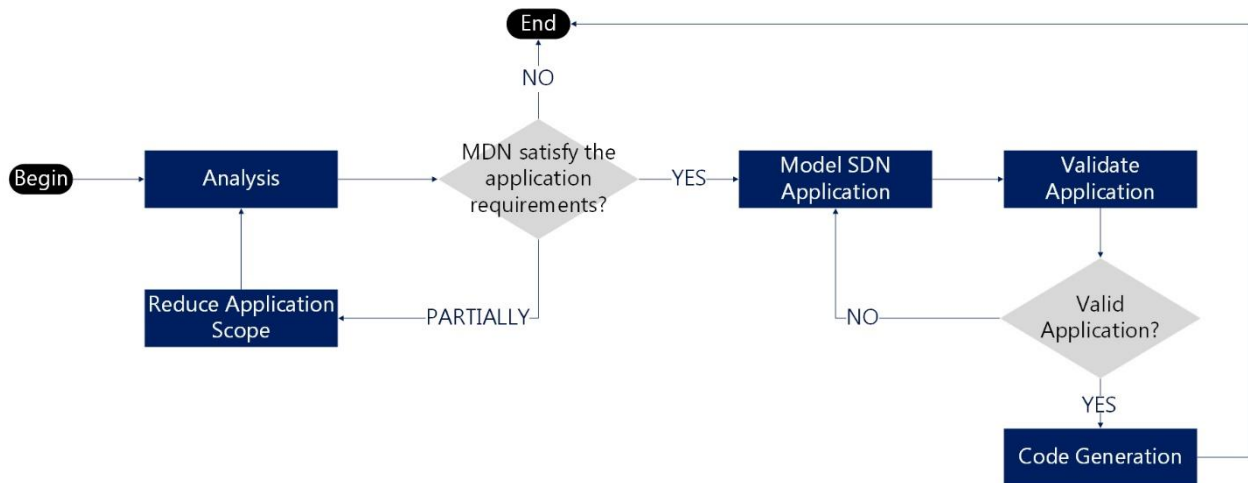
Template	sdn-odl-monitor.egl
Objective	Generate code for a monitor application compatible with ODL controller.
<pre> 1. [% import "header.egl"; %] 2. [% var ruleCounter : Integer = 0; %] 3. [%for (rule in Rule.all) { 4.   if (rule.ruleAction.isDefined()) { 5.     if (rule.ruleAction.type.value = Actions#MONITOR.value and         rule.sourceHostRule.isDefined() and rule.targetHostRule.isDefined()) { 6. %] 7.     public class Rule[%=ruleCounter%]PacketInDispatcherImpl implements         PacketProcessingListener { 8.       private Map&lt;InstanceIdentifier&lt;Node&gt;, PacketProcessingListener&gt;           handlerMapping; 9.       public PacketInDispatcherImpl() { 10.        handlerMapping = new HashMap&lt;&gt;(); 11.      } 12.      public void onPacketReceived(PacketReceived notification) { 13.        InstanceIdentifier&lt;?&gt; incomingIp = 14.          [%=rule.sourceHostRule.hostSwitch.ip    %]; 15.        InstanceIdentifier&lt;Node&gt; nodeOfPacket = 16.          incomingIp.firstIdentifierOf(Node.class); 17.        PacketProcessingListener nodeHandler = 18.          handlerMapping.get(nodeOfPacket); 19.        [...] 20.        [% ruleCounter = ruleCounter + 1; 21.      } 22.    } 23.  } %] </pre>	

Table 7: EGL template for ODL controller.

## 4.5 Applications Development Process using MDN approach

The process involving SDN applications development in MDN approach is rather similar to traditional software development. The main difference resides in using the concrete syntax of each of them. To create an application using MDN, an end-user needs to verify if the application domain fits in domain specified for the underlying DSML of MDN. If it is so, end-user models the application just by using visual elements of MDN's concrete syntax. At the end, if the created model is valid, it is possible to generate the code of such application and run it in the target controller.

Due to its foundation on MDD process, the MDN approach requires that any modification or correction to be performed in some application must be made through the model, excluding the direct modification of code generated, as Figure 4.7 depicts. Such figure also depicts the workflow of SDN applications development in MDN.



**Figure 4.7: MDN workflow.**

#### 4.5.1 MDN Editor

After we have specified each element of MDN's DSML and the components of GMF, the latter enables to generate a graphical editor, the MDN editor. Such editor has an area to create projects as well as *mdn\_diagram* files. These files enable the diagramming model in a space for the modeling. The editor also lists elements of visual notation in a palette tool, grouping the nodes of MDN models, such as hosts, switches, and controllers, separating them from elements that creates the possible relationships between such nodes.

The user interface has several similarities with Eclipse GUI, however it only provides the perspective to model SDN applications. As Figure 4.8 depicts, MDN editor provides the aforementioned project explorer (as well as Eclipse), one diagram area, and palette with network objects and connections. The already mentioned button for code generation, named "*Export to script*" is located at the top of graphical interface, when some generation is performed it appears at the *Application Code* tab at bottom. There is also a button to generate Mininet script; it follows the same functioning of the former.

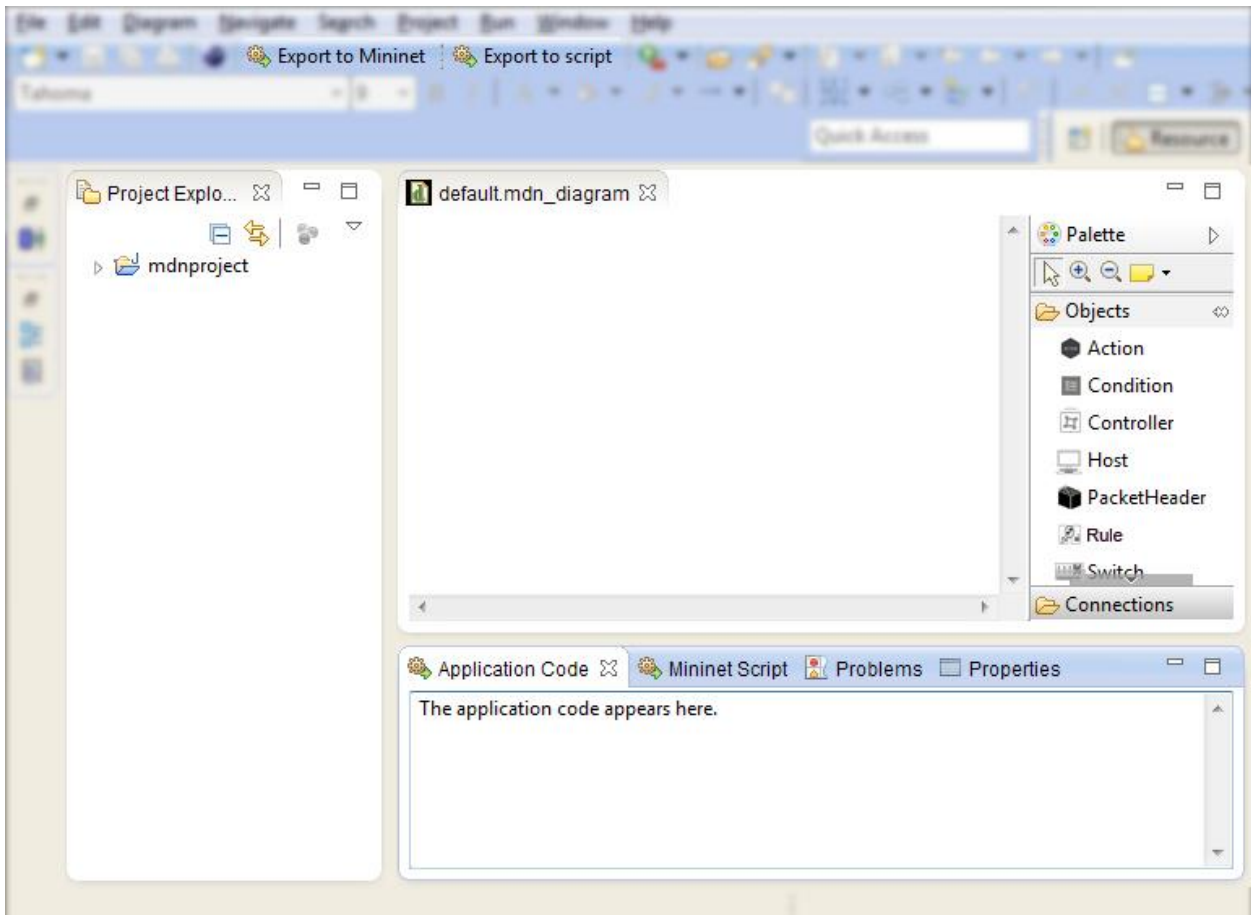


Figure 4.8: The MDN editor.

**Creating, Validating, and Simulating SDN Applications.** In order to demonstrate a simple example in using MDN editor to model SDN applications, we specified a firewall application as follows. Considering a hypothetical scenario where the network topology is composed of three hosts ( $h1$ ,  $h2$ , and  $h3$ ) and two switches ( $s1$  and  $s2$ ) managed by one controller ( $C$ ), the firewall application must block the flows between  $h1$  and  $h3$ . Such application is created by modeling this network topology (or simply the hosts) (step 1) and specifying the rule object in MDN editor to perform a drop action for the required hosts (step 2). After these steps, the end-user must validate the model before the code generation (step 3) (cf. Figure 4.9). For instance, if the host  $h1$  has no value for IP attribute the MDN editor describes the error in *Problems* tab (cf. Figure 4.8). However, if the model is valid, we can generate the application code and deploy it in SDN controller (step 4 in Figure 4.10).

If there is a need to simulate SDN applications, we need to generate the Mininet script following the same process to generate application code (cf. optional step 5 in Figure 4.10). Thus, the SDN application runs in the controller, and the topology part of MDN model can be simulated in Mininet.

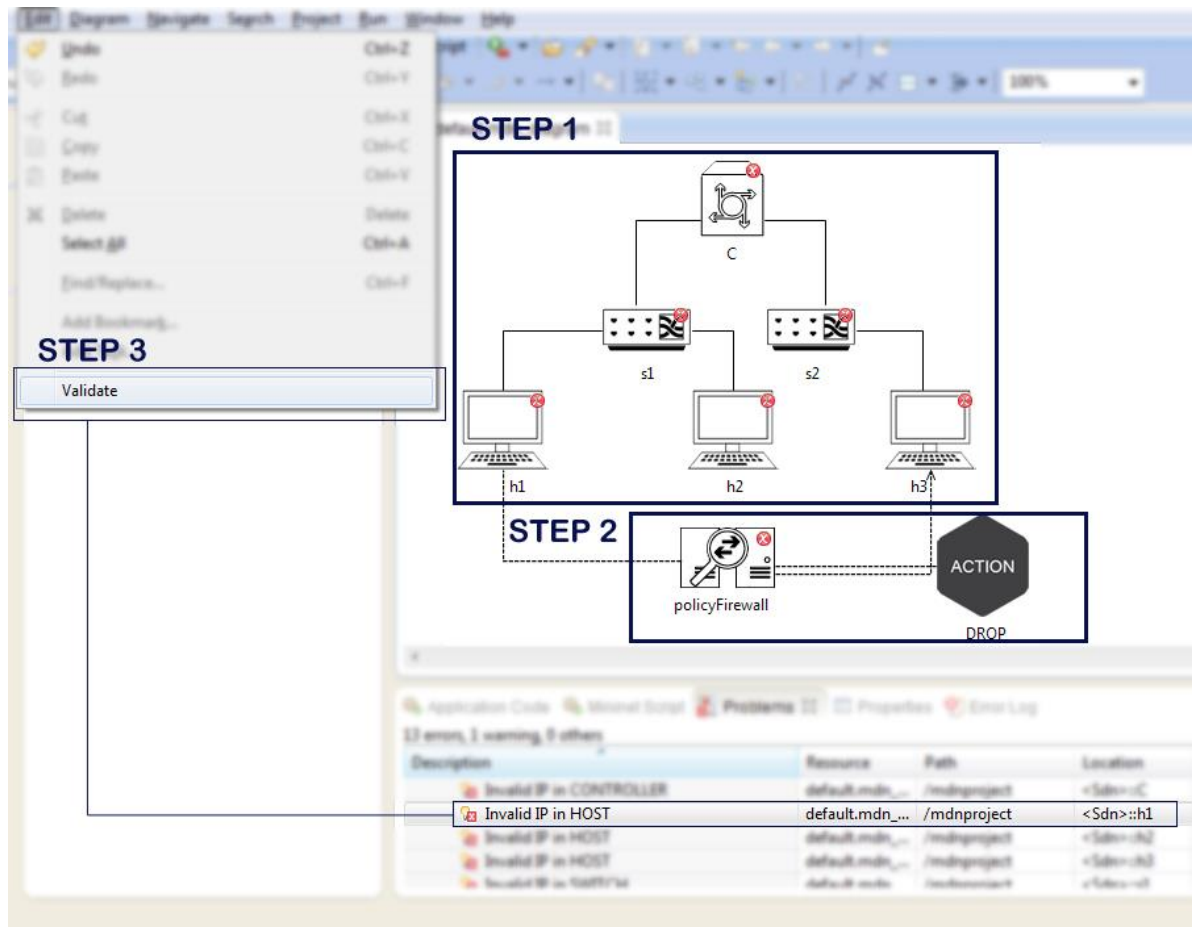


Figure 4.9: Steps in modeling a firewall application with INVALID model.

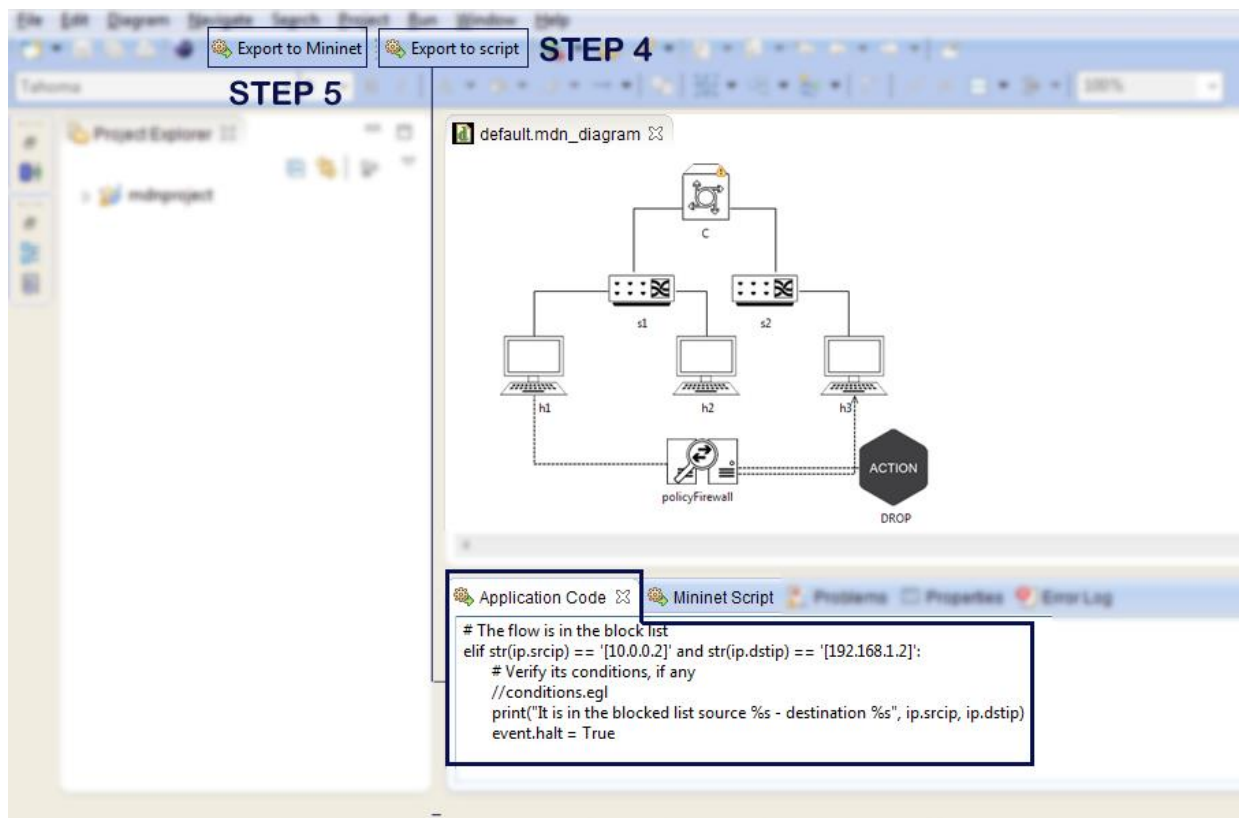


Figure 4.10: Steps in modeling a firewall application with VALID model.

The simulation for this scenario was performed in Linux-based environment with an installation of Mininet in which the MDN model-based topology is simulated and connected to a POX controller that by its turn runs the code generated from application model. Thus, the code generated by MDN editor to implement such SDN application and the network topology modeled must be imported to files accessible by POX and Mininet, such files are available at appendix A. In this case, as the following commands show, the file *mdn.py* was used in POX and the *MultipleSwitchTopo.py* in Mininet, as follows:

```
$ sudo mn --custom ~/mininet/custom/MultipleSwitchTopo.py --topo
multipleswitchtopo -controller remote,127.0.0.1 &
$ ~/pox/pox.py forwarding.l2_learning mdn
```

It is worth to mention that MDN uses POX controller with *l2\_learning* module, which makes SDN switches act as a L2 learning switch. Even if this module learns L2 addresses, the flows entries it installs have many fields as possible (e.g., source/target IP, incoming port). Such module enables the controller to paths across the network. Besides, in order to test if the firewall constraint in which host *h1* can not get connection with host *h2*, the *pingall* command of Mininet was used (cf. Figure 4.11), obtaining the following output:

```
root@ubuntu:/home/felipealencar/mininet/custom# mn --custom multipleswitchtopo.py --topo multipleswitchtopo --controller remote,127.0.0.1
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (s1, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 2 switches
s1 s2
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X
h2 -> h1 h3
h3 -> X h2
*** Results: 33% dropped (4/6 received)
mininet>
```

Figure 4.11: The use of *pingall* command in Mininet to simulate the application modeled.



This description about implementing a simple firewall SDN application demonstrated the MDN workflow in practice, in order to clarify its development process that involves modeling, validation, code generation, and execution of modeled applications.

## 5. Evaluation

---

This chapter presents the analysis of MDN approach in practice, by comparing it with other related approaches in developing SDN applications. It also presents the modeling of three use cases in order to demonstrate the feasibility of MDN in several scenarios, including its flexible characteristic. We end this chapter by present a summary of our simulations and concluding remarks of it.

---

In order to evaluate the benefits of MDN and to detect any possible drawback of it in relation to other approaches, we have performed a comparison considering the required features to support the modeling and creation of SDN applications. Such comparison analyzed the related approaches by verifying the support for each feature below:

**Topology model.** A common characteristic in creating SDN models is to diagram the network topology. It is achieved by enabling the modeling of network nodes such as hosts, switches, and controllers.

**Model behavior and rules.** SDN has rules and behaviors, which SDN applications define. The modeling of scenarios with such particularities may be valuable in granting

the network functioning according to a network requirement or organizational policy (e.g., block all connections after 6pm).

**Model applications.** As discussed in whole dissertation, the SDN enables the network programmability, which results in several applications handling the network. The modeling of SDN applications specifies through models the actions, nodes, and conditions in which the actions of certain application are performed.

**Support for multiple controllers.** Several vendors provides SDN controllers with distinct characteristics. Each of them with its advantages and drawbacks. Thus, the purpose of making models and SDN applications compatible with these possible scenarios (e.g., different controllers in networks) is to avoid the refactoring of applications or network configuration.

**Support for DSLs.** The code generation feature may use more than one programming language to implement SDN applications. It includes the use of DSLs. However, if the modeling approach is hard coded it becomes difficult to provide the support for different underlying languages or controllers and it does not takes advantage of the abstraction benefits in using SDN and, for instance, MDE.

**Generated executable code.** The use of models may be interesting in just documenting the network. However, the support of code generation through a paradigm like MDE avoids inconsistent models and provides easier ways to implement applications.

**Validation.** Although there are several ways to model and implement SDN applications (e.g., modeling approaches, DSLs, native APIs of SDN controllers), the validation of such applications has been performed only by compiling the code or testing the applications in execution environments. The feature that validates an SDN application during its creation is a way to avoid defective applications.

**Descriptive graphical elements.** Although SDN enables the network programmability, it does not makes such programming as easy. The use of graphical elements describing the behavior of SDN applications facilitates the understanding by end-users like network operators that do not have wide knowledge in programming.

Now, considering the modeling of simple firewall application at section 0, we used the related works involving modeling in an attempt to perform the creation of such application (cf. section 3.2).

Although all the approaches provide the basic feature in modeling network topology and support to program or configure multiple controllers, they do not enable the modeling of applications or network behavior and rules. Consequently, there is no support for DSLs. On the other hand, Miniedit and VND generate executable code to configure Mininet and SDN controllers, respectively, while CIM-SDN does not. The CIM-SDN is the only one to perform validation on models. MDN provides all features mentioned here.

Thus, the attempt of modeling showed gaps involving the modeling approaches when compared to MDN when the evaluation considered expressiveness of them. We present the general result of our comparison in Table 8 below.

Main Features for SDN Modeling	Miniedit	VND	CIM-SDN	MDN
Topology Model	YES	YES	YES	YES
Model behavior and rules	NO	NO	NO	YES
Model applications	NO	NO	NO	YES
Support for multiple controllers	YES	YES	YES	YES
Support for DSL	NO	NO	NO	YES
Generate executable code	YES	YES	NO	YES
Validation	NO	NO	YES	YES
Descriptive graphical elements	YES	YES	NO	YES
<b>Percentage</b>	50%	50%	37.5%	100%

**Table 8: Features comparison for SDN modeling.**

Regarding the textual DSLs for SDN (e.g., Procera, Nettle, Pyretic), MDN also demonstrates its effectiveness. Although the related DSLs have a number of features close to that MDN provides, excepting topology model and graphical elements due to their textual focus, they do not validate the applications like MDN. Furthermore, they allow error-prone expressions, as follows in Figure 5.1:

```

1. from pyretic.lib.corelib import *
2. from pyretic.lib.std import *
3.
4. def infinite_loop (pkt):
5.     self.rule = if_(match(srcip=pkt['srcip']),
                       modify(dstip=self.controller.getip(),
                              self.rule)
6. def main():
7.     pkt = packets()
8.     pkt.register_callback(infinite_loop)
9.     return pkt

```

**Figure 5.1: Pyretic code causing an infinite loop in network behavior.**

The figure above displays a code snippet in which lines 5-6 compose a network rule has a hypothetical method named `infinite_loop`. Such method receives as parameter the network state and certain packet. When the controller invokes `infinite_loop`, it tries to match the packet's source IP (`pkt['srcip']`) with flow entries. If there is a flow entry for such packet, the method changes the packet's destination IP forwarding it to controller. Thus, every flow present in flow entries will change its destination IP to controller.

Another gap found at DSLs for SDN is the controller-dependency. For instance, the Pyretic language creates applications for POX controller and do not offer an option to change it. On the other hand, MDN avoids such dependency by creating applications based on templates, which can have any syntax according to target controller.

In order to verify the completeness of MDN and possible limitations in using it to develop SDN applications, we performed the modeling of three use cases that are common in testing SDN proposals (JARSCHER, *et al.*, 2014). Such modeling answers to our first research question, concerning the feasibility of MDN in developing SDN applications. The simulation environment used to create and to simulate such use cases consists of Ubuntu as operational system and Mininet as SDN simulator. The following sections describes each use case, by presenting their modeling process and simulation.

## 5.1 Use Case 1: Network Monitoring

Our first use case aims to model an application that provides traffic monitoring flow characteristics defined by network operator. As hypothetical scenario, we created a topology composed of two hosts (*h1* and *h2*), which connects to one switch (*s*) (managed by an SDN controller). This application will monitor the traffic between *h1* and *h2* and print the statistics of flows. As we used Linux-based environment to run the SDN controller and to test the application, the printing is displayed at an Ubuntu's console.

The template used to generate code extends the default network-monitoring module of POX controller<sup>9</sup>.

### 5.1.1 Modeling Application for Network Monitoring

In order to create network-monitoring application, we modeled the hypothetical scenario by creating two *host* objects (setting the IP 10.0.0.2 for *h1* and the IP 10.0.0.3 for *h2*), one *switch* object, and the *controller* object. The *host* objects have links to *switch* object, which by its turn connects to *controller*. Furthermore, we used the *rule* object of MDN, with its *type* attribute set to *MONITOR*. After the association of *h1* and *h2* with the *rule* object, our model is ready. Thus, MDN Editor will validate the model and after enable the code generation. Figure 5.2 depicts such modeling.

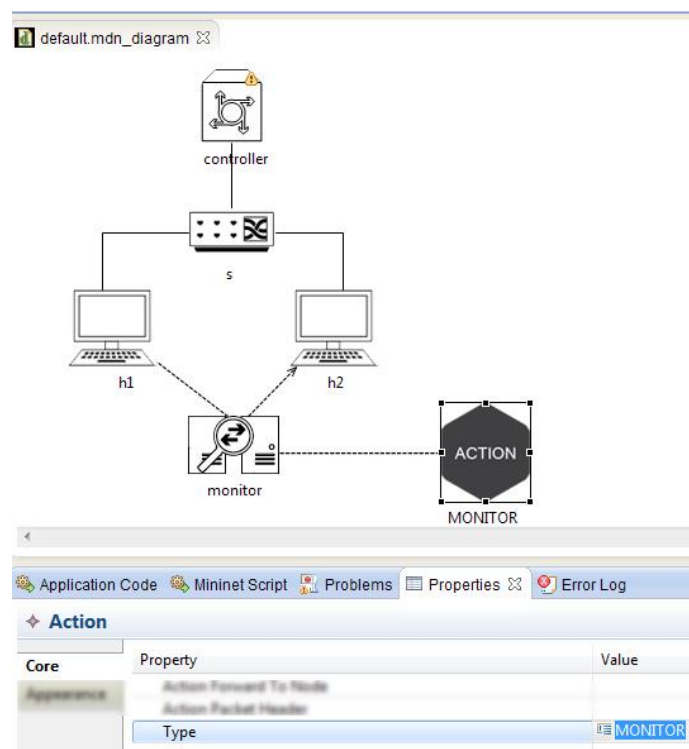


Figure 5.2: The modeling of network monitor application.

After the modeling, the MDN can generate the executable code as described in section 5.1.2.

<sup>9</sup> POX repository - [https://github.com/noxrepo/pox/tree/carp/pox/samples/flow\\_stats.py](https://github.com/noxrepo/pox/tree/carp/pox/samples/flow_stats.py)

### 5.1.2 Code Generation

The code generated from the modeling needs to define an application that 1) identifies the network flows, 2) verifies if certain flow involves a traffic between  $h1$  and  $h2$ , and 3) performs a *MONITOR* action when the flow matches the verification.

As result for this application, MDN uses its EGL templates (available at appendix B) to generate the code file with our needs of creating an SDN application that monitors the network under the conditions specified at model. We show part of the code generated and its related EGL template at Table 9 (we omitted some parts such as importing of libraries).

EGL Template
<pre> 1. web_bytes = 0 2. web_flows = 0 3. Web_packet = 0 4. for f in event.stats: 5. [% for (rule in Rule.All) { %] 6. [%   var srcIP = rule.sourceHostRule.ip.toString(); %] 7. [%   var dstIP = rule.targetHostRule.ip.toString(); %] 8. [%   srcIP = srcIP.substring(1, srcIP.length()-1); %] 9. [%   dstIP = dstIP.substring(1, dstIP.length()-1); %] 10.     if f.match.ip_dst == [%=dstIP%] or f.match.ip_src == [%=srcIP%]: 11.         web_bytes += f.byte_count 12.         web_packet += f.packet_count 13.         web_flows += 1 14. [% } %]</pre>
Code Generated
<pre> 1. web_bytes = 0 2. web_flows = 0 3. web_packet = 0 4. for f in event.stats: 5.   if f.match.ip_dst == 10.0.0.3 or f.match.ip_src == 10.0.0.2: 6.       web_bytes += f.byte_count 7.       web_packet += f.packet_count 8.       web_flows += 1</pre>

Table 9: Snippet of code generated for network-monitoring application.

### 5.1.3 Simulation

When the controller executes our code generated as a module (cf. Figure 5.3), and we perform a `pingall` command at Mininet, the following output appears at Ubuntu's console:

```

root@ubuntu:/home/felipealencar/pox# ./pox.py forwarding.l2_learning statistic
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.1.0 (betta) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
INFO:statistic:Web traffic from 00-00-00-00-00-01: 294 bytes (3 packets) over 3
flows

```

Figure 5.3: Network-monitoring application output when Mininet's `pingall` command is called.

Thus, the application identified three packets, two used for pingall and one for the route learning. It also shows the bytes amount used to perform such command in Mininet. The string 00-00-00-00-00-01 refers to the switch that connects the two hosts (*h1* and *h2*), it is an OpenFlow switch identifier, named DPID (ONF, 2013). Although the textual presentation, the application classifies the network flows for each switch, as well as the bytes transferred. Such application could provide this information through a graphical interface, but for now, it is out of the scope. Our intent is to demonstrate the correct execution of application model created in MDN Editor.

## 5.2 Use Case 2: Access Control Application

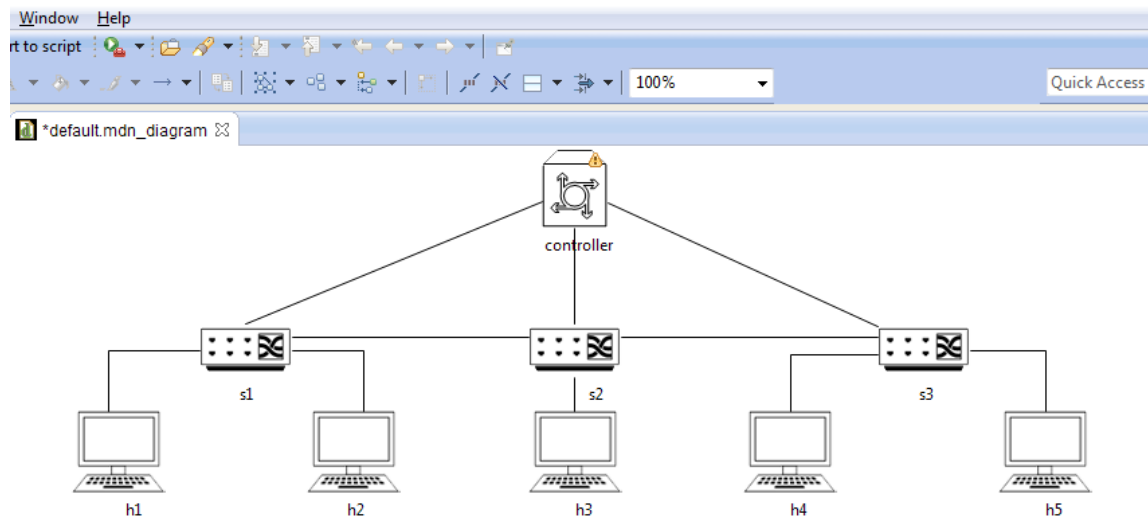
In order to clearly show the MDN potential to express fine-grained access control, this use case considers a hypothetical scenario in which the network topology consists of five hosts (*h1*, *h2*, *h3*, *h4*, and *h5*) connected to three switches interconnected (*s1*, *s2*, and *s3*). Switch *s1* connects the hosts *h1* and *h2*, switch *s2* links the host *h3*, and switch *s3* connects the hosts *h4* and *h5*. One SDN controller manages the switches. Besides, the network needs to satisfy following requirements:

1. Hosts connected to switch *s1* cannot access the network in the 6am - 9am period;
2. Host *h3* cannot establish connections when the application port is equal to 8080;
3. Hosts connected to switch *s3* cannot access the network.

### 5.2.1 Modeling the access control application and its policies

We start to create the network topology by modeling each network node, defining their names and their links, as Figure 5.4 depicts.





**Figure 5.4 Topology for the access control application.**

In order to emphasize the feature of validation that MDN provides, when we try to validate such topology, the MDN editor provides the following errors about invalid structures:

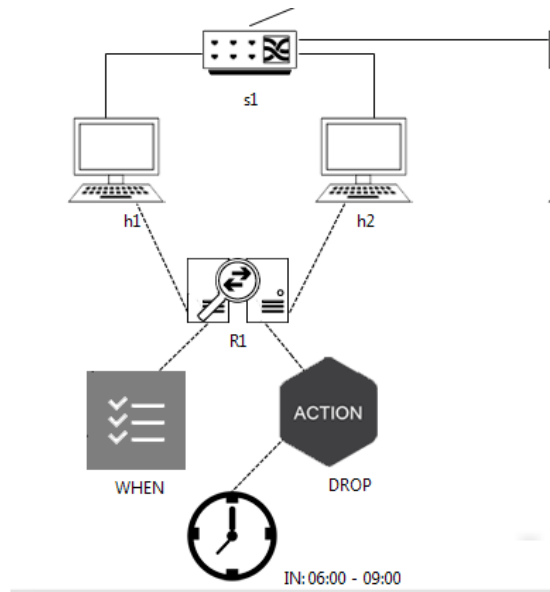
Description	Resource	Path	Location
Invalid IP in HOST	default.mdn_...	/mdnproject	<Sdn>::h1
Invalid IP in HOST	default.mdn_...	/mdnproject	<Sdn>::h2
Invalid IP in HOST	default.mdn_...	/mdnproject	<Sdn>::h3
Invalid IP in HOST	default.mdn_...	/mdnproject	<Sdn>::h4
Invalid IP in HOST	default.mdn_...	/mdnproject	<Sdn>::h5
Invalid IP in SWITCH	default.mdn_...	/mdnproject	<Sdn>::s1
Invalid IP in SWITCH	default.mdn_...	/mdnproject	<Sdn>::s2
Invalid IP in SWITCH	default.mdn_...	/mdnproject	<Sdn>::s3
More than one identical IP "" connected to switch	default.mdn_...	/mdnproject	<Sdn>::h1

**Figure 5.5 Validation of topology for the use case #2.**

Then, we need to correct the properties of topology elements by defining their IP as follows:

- $s1$  – IP: 10.0.0.1
  - $h1$  – IP: 10.0.0.2
  - $h2$  – IP: 10.0.0.3
- $s2$  – IP: 128.0.0.1
  - $h3$  – IP: 128.0.0.2
- $s3$  – IP: 192.0.0.1
  - $h4$  – IP: 192.0.0.2
  - $h5$  – IP: 192.0.0.3

After, we created the rules to satisfy the network requirements as follows: the first requirement defines a period (i.e., 6am - 9am) in which hosts of switch  $s1$  can access network resources. Then, we inserted a rule  $R1$  that relates hosts  $h1$  and  $h2$  to a condition specifying the period with MDN's time element. Besides, we defined the action DROP for  $R1$  (cf. Figure 5.6).



**Figure 5.6 Modeling of first requirement.**

For the blocking of host  $h3$  when it tries to establish a connection through port 8080, we defined more one rule in which its action drops the flows of  $h3$  when it refers to such port. The Figure 5.7 depicts such modeling. Note that our model reutilizes the ACTION element in order to simplify the modeling (the blurred part is our first requirement modeled – cf. Figure 5.6).

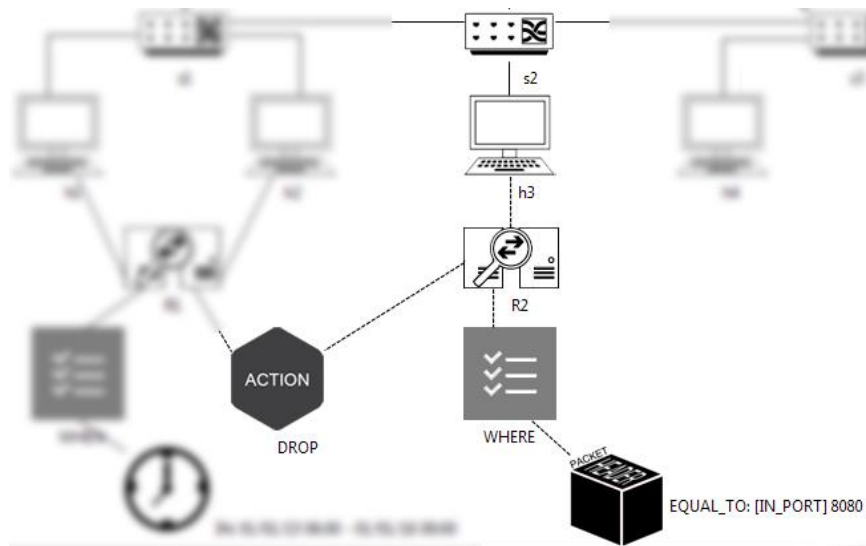


Figure 5.7 Modeling of second requirement.

Finally, to satisfy the third requirement, we added another rule ( $R3$ ) that block all flows from  $h4$  or  $h5$ . Once again, note in Figure 5.8 that we can reuse the ACTION element to drop the respective flows (the blurred part is our second requirement modeled - cf. Figure 5.7).

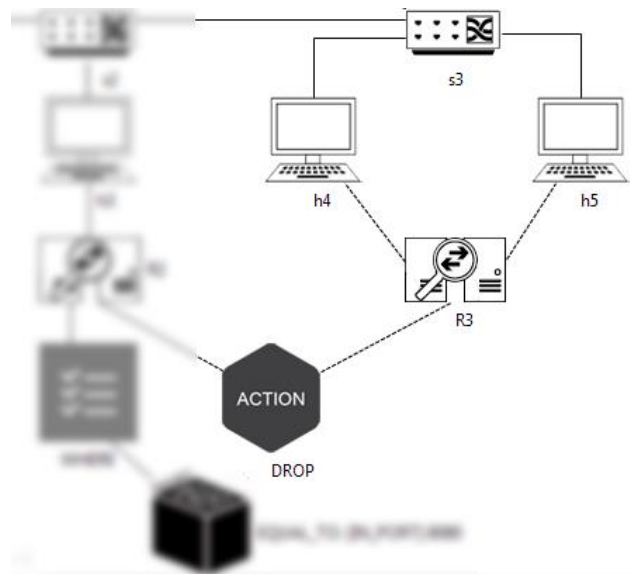


Figure 5.8 Modeling of third requirement.

### 5.2.2 Code Generation

The code generated from the modeling above needs to implement the topology in Mininet and an application to meet the three requirements described in section 5.2. Then, in Table 10, we present a snippet of code generated by our template, named *accesscontrol.egl* (fully available at appendix A).

Code Generated
<pre> 1.ip = packet.find('ipv4') 2.# This packet isn't IP! 3.if ip is None: 4.    return 5.# Rule R1 6.# The flow is in the block list 7.elif str(ip.srcip) == '10.0.0.2' or str(ip.srcip) == '10.0.0.3': 8. 9.    # Verify rule conditions, if any 10.        if (datetime.now().strftime('%H:%M') &gt; datetime.strptime('06:00AM', '%I:%M     %p') and datetime.now().strftime('%H:%M') &lt; datetime.strptime('09:00AM', '%I:%M%p'     )): 11.            print("It is in the blocked list source %s - destination     %s", ip.srcip, ip.dstip) 12.            event.halt = True 13.        else 14.            event.halt = False 15. 16.    # Rule R2 17.    # The flow is in the block list 18.    elif str(ip.srcip) == '128.0.0.2': 19.        # Verify rule conditions, if any 20.        if (str(packet.find("tcp").srcport) == '8080'): 21.            print("It is in the blocked list") 22.            event.halt = True 23.        else 24.            event.halt = False </pre>

Table 10: Code generated by MDN editor to implement the access control application.

In summary, once again we defined a scaffold that MDN Editor uses to insert data from elements modeled. For instance, line 7 receives the IPs of hosts tied to rule R1, if these hosts perform some type of connection or transmission on network, the controller verifies if the period allows such action (lines 10-13).

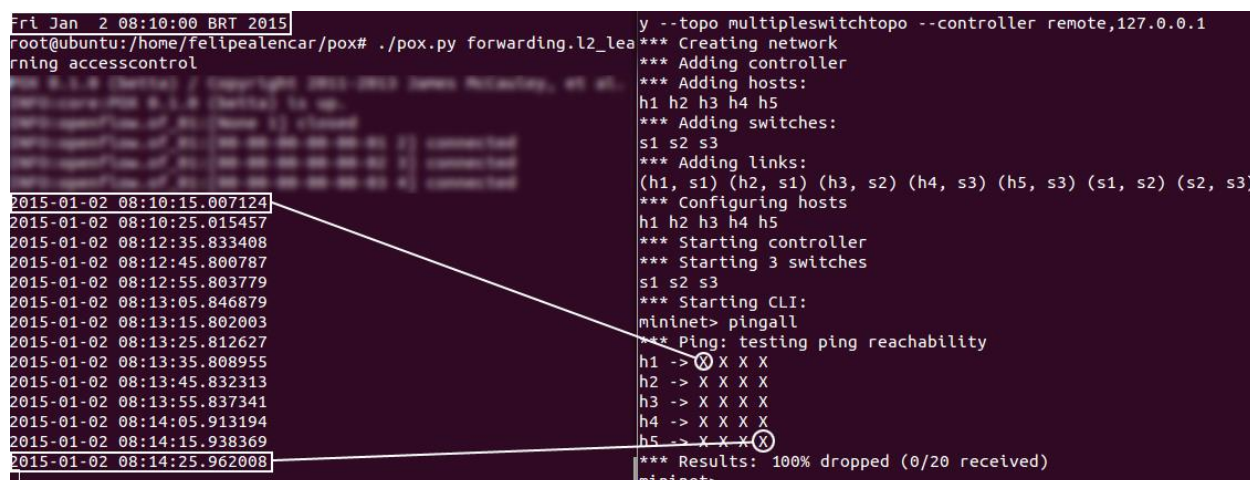
### 5.2.3 Simulation

In order to test our application modeled, as well as its code generated, we imported the Mininet script generated from topology specified in Figure 5.4 and simulated the scenarios involved in the use case requirements defined at the beginning of section 5.2.

The first requirement was tested by verifying if the hosts *h1* and *h2* could receive or rely packets across the network in the specified (6am - 9am). Then, for such requirement, we obtain the output depicted at Figure 5.9 of Mininet and our access control application.

Figure 5.9 depicts two consoles of Ubuntu, the left console shows POX controller executing our access control application, at right the console presents the Mininet creating the topology and the output of `pingall` command. Note that hosts *h3*, *h2*, and *h5* do not have any restrictions preventing the transmission of data between them (right

console). On the other hand, hosts *h1* and *h2* cannot rely or receive any data considering the time displayed in the upper left corner of the figure.



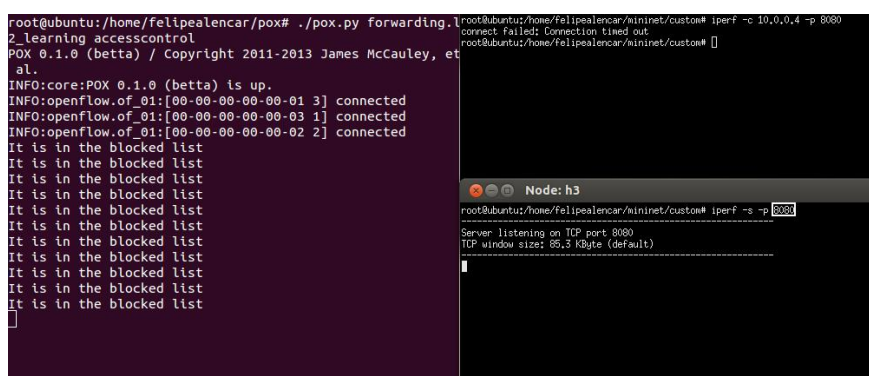
**Figure 5.9: Test involving the access control application and its first requirement.**

The simulation of second requirement used a feature of Mininet named `xterm`, which opens a console for host of topology. We opened one console for host *h2* and other console for host *h3* (cf. Figure 5.10), as follows:

```
mininet> xterm h2
mininet> xterm h3
```

**Figure 5.10: Mininet commands to open consoles in each hosts.**

After, we set *h3* to act as a server listening at application port 8080. Then, from console opened for *h2*, we used the Mininet's command `iperf`, which generates TCP traffic, in order to test if the *h3* could receive packets to such port. Figure 5.11 depicts this scenario.



**Figure 5.11: Output of the simulation for the second requirement.**

Finally, we verified if the application blocks the hosts connected to switch s3 (dpid: 00-00-00-00-00-03). Considering the hour of simulation out of the period between 6-9am, pingall command returned the reachability displayed in Figure 5.12.

```

root@ubuntu:/home/felipealencar/pox# ./pox.py forwarding.lmininet> pingall
2_learning accesscontrol
POX 0.1.0 (beta) / Copyright 2011-2013 James McCauley, et
al.
INFO:core:POX 0.1.0 (beta) is up.
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:openflow.of_01:[00-00-00-00-00-02 3] connected
INFO:openflow.of_01:[00-00-00-00-00-03 4] connected
h1 -> h2 h3 X X
h2 -> h1 h3 X X
h3 -> h1 h2 X X
h4 -> X X X X
h5 -> X X X X
*** Ping: testing ping reachability
*** Results: 70% dropped (6/20 received)
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list
It is in the blocked list

```

Figure 5.12: The third network requirement of access control application.

### 5.3 Use Case 3: Load Balancing Application

The load balancer we describe hereinafter receives the HTTP requests from a client and distributes them, based on a round-robin algorithm, to a list of four hosts. However, as presented earlier in section 4.3.1, the MDN infrastructure does not offer any class or relationship that provide a way to model such load balancer. Thus, we took this use case to demonstrate the extensibility of our approach. First, we modify the MDN metamodel by adding a new element called *group*, which groups the hosts of network topology. Then, we use such group element to define the hosts involved in load balancing application. Note that the act of extending metamodel is not role of network operators, we only use such an extension to present the possibilities enabled by MDN approach.

The *group* element was inserted in MDN metamodel as a diagram element. It works as a container to network nodes of *host* type. Figure 5.13 depicts the new element in MDN metamodel (we have omitted some links in order to provide a better visualization), it has an *ip* attribute, which refers to the virtual IP of load balancing described above.

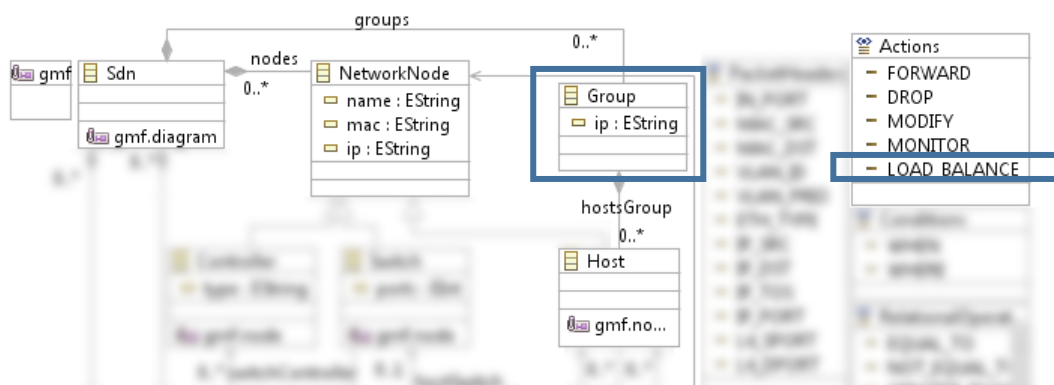
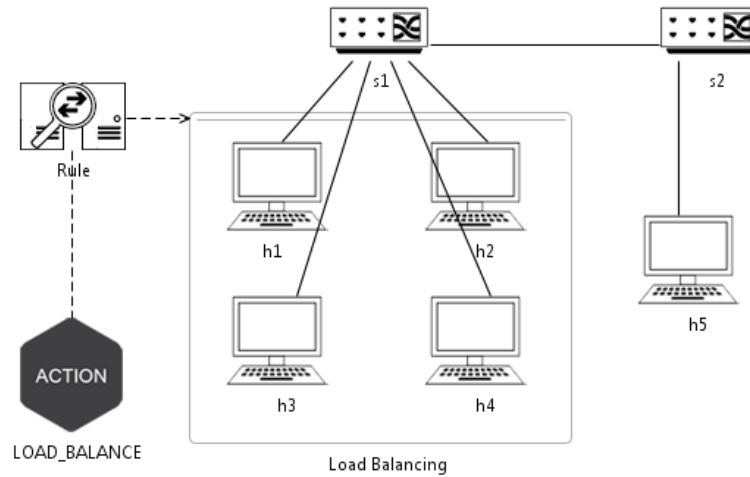


Figure 5.13: *Group* element and *LOAD\_BALANCE* action highlighted in blue.

Besides, with the aim of verify if the network operator needs to perform load balancing for the grouped hosts at model, we add an action named *LOAD\_BALANCE* (cf. Figure 5.13). After these extensions performed in MDN metamodel, following the same process used to specify our DSML (cf. section 4.3), we have defined a simple white box for the *group* element as part of MDN's visual notation, or concrete syntax. We also have mapped such visual notation with its underlying concept. Figure 5.14 depicts the visual notation used to model load balancing applications.



**Figure 5.14: Model of load balancing application.**

Another addition in MDN infrastructure is relative to templates for code generation. We have implemented a new template to capture the grouped hosts, which is used in code generation if the modeler defines any *LOAD\_BALANCE* action. If there is such action, this template selects the IPs of grouped hosts and inserts them into a list. From such list, a round robin algorithm performs the selection of one host and directs the traffic to it. Table 11 depicts the structure of code generation for such scenario.

EGL Template
<pre> 1. [% operation actionLoadBalancing(rule : Any) : String { %] 2. virtual_ip = IPAddr("[%=rule.targetGroupRule.ip %]") 3. host = {} 4. [% 5.   var index = 0; 6.   for (host in rule.targetGroupRule.hostsGroup.all) { 7. [% 8.   host[%=index%]] = {'ip':IPAddr("[%=host.ip%]")}} 9. [%   index++; %] 10. [% } %] 11.   total_hosts = len(host) 12.   host_index = 0 13.   [...] 14.   # Only handle IPv4 packets 15.   if (not event.parsed.find("ipv4")): 16.       return EventContinue </pre>



```

17. # Only verify traffic with virtual IP as destination
18. if (msg.match.nw_dst != virtual_ip):
19.     return EventContinue
20. [...]
21. # Round-robin
22. index = host_index % total_hosts
23. selected_host_ip = host[index]['ip']
24. host_index += 1
25. [...]
26. # Set route to the selected host
27. msg.actions.append(of.ofp_action_nw_addr(of.OFPAT_SET_NW_DST, selected_server_ip))
28. event.connection.send(msg)
29. [...]
30. [%}%]

```

#### Code Generated

```

1. virtual_ip = IPAddr("10.0.0.2")
2. host = {}
3. host[0] = {'ip':IPAddr("10.0.0.3")}
4. host[1] = {'ip':IPAddr("10.0.0.4")}
5. host[2] = {'ip':IPAddr("10.0.0.5")}
6. host[3] = {'ip':IPAddr("10.0.0.6")}
7. total_hosts = len(host)
8. host_index = 0
9. [...]
10. # Round-robin
11. index = host_index % total_hosts
12. selected_host_ip = host[index]['ip']
13. host_index += 1
14. # Setup route to the selected server
15. [...]
16. msg.actions.append(of.ofp_action_nw_addr(of.OFPAT_SET_NW_DST, selected_server_ip))
17. event.connection.send(msg)
18. [...]

```

**Table 11: Snippet of EGL template and the code generated from load balancing application model.**

The EGL template (cf. Table 11), at lines 1-10, generates code to perform the verification of virtual IP and hosts related to load balancing. Lines 14-24 verify if some traffic is sent to virtual IP and if such traffic is IPv4. For such cases, the application runs round-robin algorithm to select the host that will receive the traffic. The following lines (i.e., lines 16-18) consist of OpenFlow messages to set the traffic route to the selected server.

Finally, we have generated the MDN Editor once again, in order to enable the modeling of load balancing elements and code generation. Although the code generated in Table 11 already presents some properties of network nodes modeled for the use case in discussion (cf. Figure 5.14), we clarify the properties for switches 1 (s1) and 2 (s2) as well as for their underlying hosts (h):

- s1 – IP: 10.0.0.1
  - Load Balancing – Virtual IP: 10.0.0.254
    - h1 – IP: 10.0.0.2
    - h2 – IP: 10.0.0.3



- h3 – IP: 10.0.0.4
- h4 – IP: 10.0.0.5
- s2 – IP: 192.168.0.1
  - h5 – IP: 192.168.0.2

Then, in summary, our MDN Editor can now create a load balancing application model, which consists of a rule that handles HTTP requests from host *h5*, by directing such requests to a group of hosts (i.e., *h1*, *h2*, *h3*, and *h4*) based on a round-robin algorithm.

### 5.3.1 Simulation

We have tested the modeled application by creating the topology in Mininet and running the generated code at POX controller. From Mininet, the host *h5* throw HTTP requests through the `wget`<sup>10</sup> command to the virtual IP of load balancing group. In response to such requests, we have started a HTTP server for each host grouped in load balancing.

Thus, from host *h5*, `wget` command was used as follows:

```
mininet> h5 wget http://10.0.0.254:80
```

Such command performs an HTTP request to the address specified as parameter, which refers to our virtual IP. We have performed the `wget` call five times in order to verify if the application balanced the traffic based on round-robin algorithm. Figure 5.15 depicts the output obtained from such calls. At the left side of figure, our application prints the host IP that is sending HTTP responses for each request. At the right side, there is the output of `wget` commands when we have performed the five HTTP requests to virtual IP.

```

root@ubuntu:/home/felipealencar/pox# ./pox.py loadbalancing fo
POX 0.1.0 (beta) / Copyright 2011-2013 James McCauley, et al.
INFO:loadbalancing:Stateless LB running.
INFO:core:POX 0.1.0 (beta) is up.
INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
Host: 10.0.0.2
Host: 10.0.0.3
Host: 10.0.0.4
Host: 10.0.0.5
Host: 10.0.0.2
mininet> h5 wget http://10.0.0.254:80
--2015-02-09 15:35:27-- http://10.0.0.254/
Connecting to 10.0.0.254:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 326 [text/html]
Saving to: 'index.html'

      0K      100% 29.0M=0s

2015-02-09 15:35:27 (29.0 MB/s) - 'index.html' saved [326/326]

```

Figure 5.15: Output of load balancing application.

After we present the correct behavior of load balancing application generated from our MDN model, note that a network operator can execute the applications

<sup>10</sup> GNU Wget 1.16.1 Manual - <http://www.gnu.org/software/wget/manual/wget.html>

presented until here as modules of an SDN controller. Thus, for instance, the load balancing application discussed here could be executed with a network monitor application in order to collect statistics from network. Such modularity might depend on the underlying controller. However, we claim that MDN supports this modular scenario of SDN applications.

## 5.4 Chapter Remarks

In order to verify the feasibility of MDN to model (and generate) SDN applications, we have created three models of SDN applications (i.e., network monitoring, access control, and load balancing). Such step demonstrated that MDN achieves a higher abstraction level in developing SDN applications, as well as providing the validation of such applications. From MDN Editor, the network operator does not need to verify if the OpenFlow rules are consistent or even codify them.

Note that although some use cases may be considered simple, MDN allows the modeling of complex topologies and scenarios. The code generation of MDN mapped with OpenFlow specification (version 1.4) considered the use of wildcards in defining rules for flow entries. If there is a *Host* element in some MDN model with a partial IP with the following format: `10.0.0.*` and a *Rule* element related to such *Host*, the rule will be applied to all hosts in `10.0.0.0 /24`. Thus, for instance, a network operator can use such wildcards in MDN Editor to create a *Rule* that achieves several hosts according to some match pattern (or wildcard) without the need to insert visual notation for each of them.

# Chapter

# 6

## 6. Final Remarks

---

This dissertation discussed how our approach to develop SDN applications might be applied to abstract the complexity in network programmability, avoiding error-prone implementations and providing consistent network executable models. This chapter presents a summary of contributions (section 6.1) and the limitations (section 6.2) of MDN. We conclude this dissertation providing future work directions (section 6.3).

---

First, we presented an infrastructure that consists of DSML specification considering all its components to describe an SDN scope in the concept that we called Model-Driven Networking (MDN), which also defines a development process based on modeling. In order to verify and test such concept, we have implemented a tool named MDN Editor that supports the modeling of SDN applications. After modeling, MDN Editor enables code generation, exporting SDN applications to files that an SDN controller can execute.

We have evaluated our approach in terms of effectiveness by comparing its features with three other modeling approaches identified in the literature. MDN has demonstrated to be more complete in this comparison. We have also verified that a

MDE-based approach such as MDN has one advantage over textual DSLs for SDN, relative to its independence from specific controllers.

At last, we have demonstrated the use of MDN in developing three common use cases for SDN applications (i.e., network monitoring, access control, and load balancing). The analysis brings preliminary evidences that several SDN concepts and low-level specifications can be mapped into our MDN approach, which provides a consistent way to facilitate the interaction between network operators and this new network paradigm, with the potential to improve the productivity and to reduce error-prone actions.

In summary, we believe that since MDN can model several concepts related to an SDN application, it provides an easier and alternative way to perform the development of such applications.

## 6.1 Summary of Contributions

The benefits of the MDN proposal are manifold:

- i. The description of SDN architecture and underlying concepts through metamodeling;
- ii. Identification how MDE concepts can help in SDN applications development;
- iii. The support for any controller vendor;
- iv. The support for any underlying programming language;
- v. The graphical description of SDN by MDN models, offering a high-level platform to develop SDN applications;
- vi. Validation of SDN topologies and applications before the deploying.

Such items answer to our second research question about the benefits in using MDN when we compare it with other approaches of development.

## 6.2 Limitations

SDN applications tend to have infinite possibilities of use cases, and thus, the underlying *domain-specific* characteristic of MDN may limit the possible applications that our approach can model and develop. For example, currently MDN cannot model an application for Deep Packet Inspection (DPI). It would be necessary to extend MDN metamodel and its code generation strategy to enable the developing of DPI applications.

Currently, MDN is restricted to some features and application types such as traffic monitoring, network policies and rules specification, and validation. Another

limitation is related to our evaluation, which does not verify cognitive aspects of concrete syntax. Besides, although MDN is not dependent on a specific SDN controller vendor due to its extensible engine of code generation, currently, it supports only code generation for POX controller.

Such drawbacks provide the answer for our second research question about the limitations of MDN approach in developing SDN applications when compared to another development method.

### 6.3 Future Work

There are several ways to enhance and to extend our work. The first step is to consider a qualitative evaluation of our approach, by checking the expressiveness of its concrete syntax. In addition, we envisage a formal mathematical definition of our DSML in order to prove that MDN syntaxes address fully the domain scope defined for SDN and its applications. In order to avoid error-prone models, another valuable increment is to detect the loop connections at model validation.

We also plan to explore model transformations from MDN models to standard network models, such as NETCONF, and vice-versa. Regarding transformations, it is also possible to investigate how to support reverse engineering through a *code-to-model* transformation (e.g., to transform Python code into SDN models).

We have verified the benefits of MDN considering only the applications development. However, we intend to synchronize automatically the network structure with MDN models without the need to model manually its nodes and links. MDN Editor should to identify the network topology and generate a model for it.

## References

- BALASUBRAMANIAN, K. et al. Developing applications using model-driven design environments. **Computer**, v. 39, n. 2, p. 33-40, 2006.
- BAST, W. et al. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). **Object Management Group**, 2011. Available in: <<http://www.omg.org/spec/QVT/1.1>>. Accessed in: 21 Dec 2014.
- BÉZIVIN, J. On the unification power of models. **Springer-Verlag. Software & Systems Modeling**, v. 4, n. 2, p. 171-188, May 2005.
- CAMPBELL, A. T. et al. A survey of programmable networks. **ACM SIGCOMM Computer Communication Review**, v. 29, n. 2, p. 7-23, 1999.
- CASADO, M. et al. Ethane: Taking control of the enterprise. **ACM SIGCOMM Computer Communication Review**, New York, NY, v. 37, n. 4, p. 1-12, Oct 2007.
- CASADO, M.; FOSTER, N.; GUHA, A. Abstractions for software-defined networks. **Communications of the ACM**, New York, NY, v. 57, n. 10, p. 86-95, Oct. 2014.
- CASE, A. F. Computer-Aided Software Engineering (CASE): Technology for Improving Software Development. **ACM SIGMIS Database**, v. 17, n. 1, p. 35-43, 1985.
- CISCO. **OpFlex: An Open Source Approach**. [S.l.]: [s.n.], 2014.
- CLARK, T.; SAMMUT, P.; WILLANS, J. **Applied metamodeling: a foundation for language driven development**. 2. ed. [S.l.]: Ceteva, v. 1, 2008.
- CUADRADO, J. S.; MOLINA, J. G. A model-based approach to families of embedded domain-specific languages. **IEEE Transactions on Software Engineering**, 35(6), 2009. 825-840.
- CZARNECKI, K.; SIMON, H. Classification of model transformation approaches. **Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture**, Citeseer, v. 45, n. 3, p. 1-17, 2003.
- DE LARA, J.; VANGHELUWE, H. Meta-modelling and graph grammars for multi-paradigm modelling in ATOM<sup>3</sup>. **IN ATOM<sup>3</sup>, SOFTWARE AND SYSTEMS MODELING (3)**, August 2004. 194-209.
- DORIA, A. et al. Forwarding and Control Element Separation. **RFC 5810 (Proposed Standard)**, March 2010.
- DVORKIN, M. et al. OpFlex Control Protocol. **IETF**, 2014. ISSN Work in progress. Available in: <<http://tools.ietf.org/html/draft-smith-opflex-00>>. Accessed in: 15 September 2014.

ECLIPSE. Documentation. **Graphical Modeling Framework**, 2014. Available in: [http://wiki.eclipse.org/GMF\\_Documentation](http://wiki.eclipse.org/GMF_Documentation). Accessed in: 20 Dec 2014.

ERICKSON, D. The beacon openflow controller. **HotSDN '13 Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking**, 2013. 13-18.

ERICKSON, D. et al. A Demonstration of Virtual Machine Mobility in an OpenFlow Network. **ACM Sigcomm**, 17-22 Aug 2008.

ESSER, R.; JANNECK, J. W. A framework for defining domain-specific visual languages. **Workshop on Domain-Specific Visual Languages, in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA-2001**, Tampa Bay, Florida, 2001.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The Road to SDN. **ACM Queue**, v. 11, n. 12, p. 20, 2013.

FLOODLIGHT, P. Floodlight OpenFlow Controller, 2014. Available in: <http://www.projectfloodlight.org/floodlight/>. Accessed in: 15 Sep 2014.

FONTES, R.; SAMPAIO, P. Visual Network Description: A Customizable GUI for the Creation of Software Defined Network Simulations. **27th European Simulation and Modelling Conference - ESM'2013**, 23-25 October 2013.

FOSTER, N. et al. Frenetic: a network programming language. **Proceedings of the 16th ACM SIGPLAN international conference on Functional programming**, September 2011. 279-291.

FOSTER, N. et al. Languages for Software-Defined Networks. **IEEE Communications Magazine**, v. 51, n. 2, p. 128-134, February 2013.

FOWLER, M. **Domain-Specific Languages**. 1<sup>a</sup>. ed. [S.l.]: Addison-Wesley, 2011.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. **Future of Software Engineering. IEEE Society.**, 2007. 37-54.

GUDE, N. et al. NOX: Towards an Operating System for Networks. **ACM SIGCOMM Computer Communication Review**, v. 38, n. 3, p. 105-110, 2008.

HAILPERN, B.; TARR, P. Model-driven development: The good, the bad, and the ugly. **IBM systems journal**, v. 45, n. 3, p. 451-461, 2006.

HAREL, D. On Visual Formalisms. **ACM Communications**, v. 31, n. 5, p. 514-530, 1988.

HAREL, D.; RUMPE, B. Meaningful Modeling: What's the Semantics Much confusion surrounds the proper definition of complex modeling. **IEEE Computer**, v. 37, n. 10, p. 64-72, 2004.

HINRICHS, T. L. et al. Practical Declarative Network Management. **Proceedings of the 1st ACM workshop on Research on enterprise networking**, 21 August 2009. 1-10.

HU, F.; HAO, Q.; BAO, K. A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation. **IEEE Communications Surveys & Tutorials**, v. 16, n. 4, p. 2181-2206, May 2014.

HU, Y. et al. Reliability-aware controller placement for Software-Defined Networks. **Integrated Network Management (IM 2013)**, 2013. 672-675.

IMTIAZ, J. et al. A novel method for auto configuration of Realtime Ethernet Networks. **IEEE International Conference on Emerging Technologies and Factory Automation, ETFA.**, Hamburg, 15-18 Sep 2008. 861-868.

JARSCHEL, M. et al. Interfaces, attributes, and use cases: A compass for SDN. **IEEE Communications Magazine**, v. 52, n. 6, p. 210-217, 2014.

KARSAI, G. et al. Composition and cloning in modeling and meta-modeling. **IEEE Transactions on Control Systems Technology**, v. 12, n. 2, p. 263-278, Mar 2004.

KATTA, N. P.; REXFORD, J.; WALKER, D. Logic Programming for Software-Defined Networks. **ACM SIGPLAN Workshop on Cross-Model Language Design and Implementation**, 2012.

KELLY, S.; TOLVANEN, J.-P. **Domain-Specific Modeling: Enabling Full Code Generation**. [S.l.]: Wiley-IEEE Computer Society Press, 2008.

KIM, W. et al. Automated and Scalable QoS Control for Network Convergence. **Internet Network Management Workshop on Research on Enterprise Networking (INM/WREN)**, 2010. 1-1.

KOLOVOS, D. et al. The Epsilon Book. **Eclipse**, 2014. Available in: <<http://eclipse.org/epsilon/doc/book/>>. Accessed in: 16 Dec 2014.

KOLOVOS, D. S.; PAIGE, R. F.; POLACK, F. A. C. On the evolution of OCL for capturing structural constraints in modelling languages. In: **ACM Rigorous Methods for Software Construction and Analysis**. Berlin, Heidelberg: Springer-Verlag, 2009. p. 204-218.

KOPONEN, T. et al. Network virtualization in multi-tenant datacenters. **11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)**, April 2014. 203-216.

KREUTZ, D. et al. Software-Defined Networking: A Comprehensive Survey. **ArXiv e-prints**, Jun 2014.

KREUTZ, D.; RAMOS, F. M. V.; VERÍSSIMO, P. Towards secure and dependable software-defined networks. **Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking**, Sep 2013. 55-60.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. **Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks**, 2010. 19.



- LARA, A.; KOLASANI, A.; RAMAMURTHY, B. Network Innovation using OpenFlow: A Survey. **IEEE Communications Surveys & Tutorials**, v. 16, n. 1, p. 493-512, August 2013.
- LEDECZI, A. et al. Composing Domain-Specific Design Environments. **IEEE Computer**, v. 34, n. 11, p. 44-51, 2001.
- LENNOX, J.; SCHULZRINNE, H.; WU, X. Call Processing Language (CPL): A Language for User Control of Internet Telephony Services. **IETF**, 2004. Available in: <<http://www.ietf.org/rfc/rfc3880.txt>>. Accessed in: 15 Dec 2014.
- LOPES, F. A. et al. How Software Aging Affects SDN: A View on the Controllers. **Global Information Infrastructure and Networking Symposium (GIIS)**, Montreal, CA, 15-19 September 2014.
- LOPES, F. A. et al. **Model-Driven Networking: A Novel Approach for SDN Applications Development**. IFPE/IEEE International Symposium on Integrated Network Management. [S.l.]: [s.n.]. 2015. p. xx-xx.
- MARQUES, E. M.; SAMPAIO, P. N. NSDL: an integration framework for the network modeling and simulation. **International Journal of Modeling and Optimization**, v. 2, n. 3, p. 304-308, 2012.
- MCKEOWN, N. et al. OpenFlow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, v. 38, n. 2, p. 69-74, Apr 2008.
- MENS, T.; VAN GORP, P. A taxonomy of model transformation. **Electronic Notes in Theoretical Computer Science**, v. 152, p. 125-142, 2006.
- MOHAGHEGHI, P. et al. Where does model-driven engineering help? Experiences from three industrial cases. **Software & Systems Modeling**, Springer Berlin Heidelberg, v. 12, n. 3, p. 619-639, October 2011.
- MONSANTO, C. et al. A Compiler and Run-time System for Network Programming Languages. **ACM SIGPLAN Notices**, v. 47, n. 1, p. 217-230, 25-27 January 2012.
- MONSANTO, C. et al. Composing Software-Defined Networks. **Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation**, 2013. 1-14.
- MOODY, D. L. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. **IEEE Transactions on Software Engineering**, v. 35, n. 6, p. 759-779, November 2009.
- NAYAK, A. et al. Resonance: Dynamic access control for enterprise networks. **Proceedings of the 1st ACM workshop on Research on enterprise networking**, August 2009. 11-18.
- NELSON, T. et al. Tierless Programming and Reasoning for Software-Defined Networks. **Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation**, 2-4 April 2014.

NELSON, T.; FERGUSON, A. D.; SCHEER, M. J. G. Tierless Programming and Reasoning for Software-Defined Networks. **Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation**, 2014.

NUNES, B. A. A. et al. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. **IEEE Communications Surveys & Tutorials**, v. 16, n. 3, p. 1617-1634, August 2014. ISSN 1553-877X.

OMG. Meta Object Facility (MOF) Specification. **Object Management Group**, 2000.

OMG. OCL 2.0 Specification. **Object Management Group**, 2006. Available in: <<http://www.omg.org/spec/OCL/2.0/>>. Accessed in: 27 Dec 2014.

OMG. MOF Model To Text Transformation (MOFM2T) 1.0. **Object Management Group**, 2008. Available in: <<http://www.omg.org/spec/MOFM2T/1.0/>>. Accessed in: 21 Dec 2014.

OMG. Meta Object Facility (MOF™) Core v2.4.2. **OMG**, 2014. Available in: <<http://www.omg.org/spec/MOF/2.4.2/>>. Accessed in: 14 Dec 2014.

ONF. OpenFlow Switch Specification 1.4.0. **Open Networking Foundation**, 14 Oct 2013. Available in: <<https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf>>. Accessed in: 20 Apr 2014.

ONF. SDN architecture. **Open Networkin Foundation**, 2014. Available in: <[https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR\\_SDN\\_ARCH\\_1.0\\_06062014.pdf](https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf)>. Accessed in: 23 Dec 2014.

ORTIZ, S. Software-Defined Networking: On the Verge of a Breakthrough? **IEEE Computer**, v. 46, n. 7, p. 10-12, Jul. 2013.

PINHEIRO, B. et al. CIM-SDN: A Common Information Model extension for Software-Defined Networking. **IEEE Globecom Workshops**, 2013. 836-841.

RAZA, S.; LENROW, D. **North Bound Interface Working Group (NBI-WG) Charter**. [S.l.]: Open Networking Foundation, 2013.

REITBLATT, M. et al. FatTire: Declarative Fault Tolerance for Software-Defined Networks. **Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking**, 2013. 109-114.

RODRIGUES, T. et al. Model-Driven Development of Wireless Sensor Network Applications. **IFIP 9th International Conference on Embedded and Ubiquitous Computing (EUC)**, Melbourne, VIC, 24-26 Outubro 2011. 11-18.

SCHMIDT, D. C. Guest Editor's Introduction: Model-Driven Engineering. **Computer**, Long Beach, California, v. 39, p. 25-34, February 2006.

SELIC, B. The pragmatics of model-driven development. **IEEE software**, v. 20, n. 5, p. 19-25, 2003.

SHERWOOD, R. et al. FlowVisor: A Network Virtualization Layer. **OpenFlow Switch Consortium, Tech. Rep**, Oct 2009.

SHERWOOD, R. et al. Can the Production Network Be the Testbed? **Symposium on Operating Systems Design and Implementation (OSDI)**, Vancouver, BC, Canada, 2010. 1-6.

STEINBERG, D. et al. **EMF: Eclipse Modeling Framework 2.0**. [S.l.]: Addison-Wesley Professional, 2009.

TENNENHOUSE, D. L. et al. A survey of active network research. **IEEE Communications Magazine**, v. 35, n. 1, p. 80-86, 1997.

TOLVANEN, J.-P.; ROSSI, M. MetaEdit+: defining and using domain-specific modeling languages and code generators. **OOPSLA '03 Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**, 2003. 92-93.

TOOTOONCHIAN, A. et al. On controller performance in software-defined networks. **USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)**, 54, 2012. 10.

VOELLMY, A.; AGARWAL, A.; HUDAK, P. Nettle: Functional Reactive Programming for OpenFlow Networks. **DTIC Document**, July 2011.

VOELLMY, A.; KIM, H.; FEAMSTER, N. Procera: a language for high-level reactive network control. **HotSDN '12 Proceedings of the first workshop on Hot topics in software defined networks**, 2012. 43-48.

WANG, R.; BUTNARIU, D.; REXFORD, J. OpenFlow-based server load balancing gone wild. **Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services. Hot-ICE'11**, Berkeley, 2011. 12-12.

WANG, W. et al. Forwarding and Control Element Separation (ForCES) Protocol Specification. **IETF**, 2010. Available in: <<http://tools.ietf.org/html/rfc5810>>. Accessed in: 17 October 2014.

YEGANEH, S. H.; TOOTOONCHIAN, A.; GANJALI, Y. On scalability of software-defined networking. **IEEE Communications Magazine**, v. 51, n. 2, p. 136-141, Feb 2013.

# APPENDIX

## A. Online

In this appendix, we put the following links to source code, models, and analysis mentioned in this work:

- The MDN Editor is available at:
  - <https://github.com/felipealencar/mdn/tree/master/mdneditor>
- The EVL rules:
  - <https://github.com/felipealencar/mdn/tree/master/mdn.validation>
- The code for syntactic mapping:
  - <https://github.com/felipealencar/mdn/tree/master/mdn>
- The EGL templates:
  - <https://github.com/felipealencar/mdn/tree/master/mdn/m2t>
- The code generated for our simulations:
  - <https://github.com/felipealencar/mdn/tree/master/mdn.simulation>

## B. Attachments

Element	<i>NetworkNode</i>
Rule #1	It may has a name.
Rule #2	It may not has two identical MACs in the network.
Rule #3	It may not has invalid IPs.
Implementation	<pre> <b>context</b> NetworkNode {   <b>critique</b> hasName { //Rule #1     <b>check</b> : <i>self</i>.name.isDefined()     <b>message</b> : 'Unnamed ' + <i>self</i>.eClass().name.toUpperCase() + ' not allowed'     <b>fix</b> {       <b>title</b> : 'Define the name of the node '       <b>do</b> {         <b>var</b> type := UserInput.prompt('What is the name?');         <b>if</b> (type.isDefined()) <i>self</i>.type := type;       }     }   }    <b>constraint</b> uniqueMAC { //Rule #2     <b>check</b> {       <b>var</b> networkNodes = NetworkNode.all.select(nn nn.mac = <i>self</i>.mac);        <b>return</b> networkNodes.size() = 1;     }     <b>message</b> : 'Not unique MAC ' + <i>self</i>.mac + ' not allowed'     <b>fix</b> {       <b>title</b> : 'Define the correct MAC ' </pre>

	<pre> do {   var mac := UserInput.prompt('What is the MAC?');   if (mac.isDefined()) self.mac := mac; } }  constraint notNullIP { //Rule #3   check : self.ip.isDefined()   message : 'Invalid IP in ' + self.eClass().name.toUpperCase()   fix {     title : 'Define the IP of the node '     do {       var ip := UserInput.prompt('What is the IP?');       if (ip.isDefined()) self.ip := ip;     }   } } </pre>
<b>Element</b>	<i>Switch -&gt; NetworkNode</i>
<b>Rule #1</b>	The maximum ports number should not be greater than [OF 1.4 SPEC]
<b>Rule #2</b>	A switch can not be connected to itself.
<b>Rule #3</b>	It may not has two identical IPs connected to the same switch.
<pre> context Switch {   constraint maxNumPorts { //Rule #1     check {       if(self.ports &gt; 65280){         return false;       }       return true;     }   } }  context Host {   constraint noldenticalIpsConnectedToSwitch { //Rule #2     check {       var networkNodes = Host.all.select(h h.ip = self.ip);       if(networkNodes.size() &gt; 1){         var hostSwitch = 0;         for (host in networkNodes) {           if(host.hostSwitch &lt;&gt; hostSwitch)             hostSwitch = host.hostSwitch;           else             return false;         }       }       return true;     }   }   message : 'More than one identical IP ' + self.ip + ' connected to switch'   fix {     title : 'Define the correct IP '     do {       var ip := UserInput.prompt('What is the IP?');       if (ip.isDefined()) self.ip := ip;     }   } } </pre>	

<pre>     }   } } </pre>	
Element	<i>Controller -&gt; NetworkNode</i>
Rule #1	Controller type must be 'PROACTIVE' or 'REACTIVE'.
<pre> context Controller {   critique checkType { //Rule #1     check {       if(self.type &lt;&gt; 'PROACTIVE' or self.type &lt;&gt; 'REACTIVE' or self.type &lt;&gt; 'MIX'){         return false;       }       return true;     }     message : 'Controller type should be REACTIVE, PROACTIVE or MIX'     fix {       title : 'Define the controller type '       do {         var type := UserInput.prompt('REACTIVE, PROACTIVE or MIX?');         if (type.isDefined()) self.type := type;       }     }   } } </pre>	
Element	<i>Rule</i>
Rule #1	Name property cannot be empty.
Rule #2	If it has only one source host, such host cannot be the target host.
Rule #3	It can be linked to one <i>Condition</i> at maximum.
<pre> context Rule {   critique hasName { //Rule #1     check : self.name.isDefined()     message : 'Unnamed ' + self.eClass().name.toUpperCase() + ' not allowed'     fix {       title : 'Define the name of the rule '       do {         var name := UserInput.prompt('What is the name?');         if (name.isDefined()) self.name := name;       }     }   } }  constraint ruleWithOneSourceHostRequiresDifferenteTargetHost { //Rule #2   check {     var rules = Rule.all.select(p p = self);     var sourceHosts = 0;     var targetHosts = 0;     for (r in rules) {       sourceHosts = p.sourceHostRule;       targetHosts = p.targetHostRule;       if(sourceHosts.size() = 1){         for (tH in targetHosts) {           if(sourceHosts.ip = tH.ip or sourceHosts.ip = null or tH.ip = null){             return false;           }         }       }     }   } } </pre>	

```

    }
  }
}
return true;
}
message : 'Rules can not be null and/or have the same source and target nodes.
Change some.'
}

constraint hasOneCondition { //Rule #3 (It is also guaranteed by metamodel)
  check {
    var rules = Rule.all.select(p|p = self);
    var conditions = 0;
    for (r in rules) {
      conditions = r.ruleCondition;
      if(conditions.size() > 1){
        return false;
      }
    }
    return true;
  }
  message : 'Rules can only have one Condition'
}
}

```

Element	Condition
Rule #1	It can be linked to one <i>Time</i> at maximum.
Rule #2	It can be linked to more than one PacketHeader, if these are different.
Implementation	
Element	Time
Rule #1	The “beginDate” attribute can not be great than “endDate” attribute.
<pre> context Time {   constraint validDate { //Rule #1     check : self.beginDate &lt; self.endDate     message : 'The Time clause requires that begin date less than end data.'   } } </pre>	
Element	Traffic
Rule #1	The “value” attribute can not be negative or empty.
Rule #2	The “unit” attribute can not be different from ‘mb’ or ‘gb’.
<pre> context Traffic {   constraint validValue {     check : self.value &gt; 0 </pre>	

```

message : 'The value of traffic can not be less than or equal to 0.'
fix {
    title : 'Define the correct value for traffic '
    do {
        var value := UserInput.prompt('What is the value for traffic?');
        if (value.isDefined()) self.header := header;
    }
}
}
constraint validUnit {
    check {
        var unit = self.unit;
        unit = unit.toUpperCase();
        if(unit <> 'MB' or self.unit <> 'GB') {
            return false;
        }
        return true;
    }
}
message : 'The unit of traffic can not be different from MB or GB.'
fix {
    title : 'Define the correct unit for traffic '
    do {
        var unit := UserInput.prompt('What is the unit?');
        if (unit.isDefined()) self.unit := unit;
    }
}
}
}

```

Element	<i>PacketHeader</i>
Rule #1	The “value” attribute can not be negative or empty.

```

constraint validValue { //Rule #1
    check : self.value > 0
    message : 'The value of ' + self.header + ' can not be less than or equal to 0.'
    fix {
        title : 'Define the correct value for packet header '
        do {
            var value := UserInput.prompt('What is the value for packet header' + self.header + '?');
            if (value.isDefined()) self.header := header;
        }
    }
}

```

Element	<i>Action</i>
Rule #1	It only can be linked with <i>PacketHeader</i> if its type is “MODIFY”.
Rule #2	It only can be linked with <i>NetworkNode</i> if its type is “FORWARD”.



```

context Action {
  constraint ifActionEqualToModifyThenActionPacketHeaderCanNotBeNull { //Rule #1
    check {
      if(self.type.asString() = 'MODIFY' and self.actionPacketHeader = null) {
        return false;
      }
      return true;
    }
    message : 'The MODIFY action needs to relate with a packet header.'
  }
  constraint ifActionEqualToForwardThenActionForwardToNodeCanNotBeNull { //Rule #2
    check {
      if(self.type.asString() = 'FORWARD' and self.actionForwardToNode = null) {
        return false;
      }
      return true;
    }
    message : 'The FORWARD action needs to relate with a host.'
  }
}

```

<b>Controller</b>	<pre> @gmf.node(figure="figures.ControllerFigure", label="name", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/controller.gif", label.placement="external") class Controller extends NetworkNode {   attr String type; } </pre>
<b>Host</b>	<pre> @gmf.node(label="name", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/host.gif", figure="figures.HostFigure", label.placement="external") class Host extends NetworkNode {    @gmf.link(target.decoration="none", source.decoration="none", style="solid", color="0,0,0")   ref Switch hostSwitch;    @gmf.link(target.decoration="none", source.decoration="none", style="dash", color="0,0,0")   ref Rule sourceHostRule; } </pre>
<b>Switch</b>	<pre> @gmf.node(label="name", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/switch.gif", figure="figures.SwitchFigure", label.placement="external") class Switch extends NetworkNode {   attr int ports;    @gmf.link(target.decoration="none", source.decoration="none", style="solid", color="0,0,0")   ref Controller[*] switchController;    @gmf.link(target.decoration="none", source.decoration="none", style="solid", color="0,0,0", source.constraint="self &lt;&gt; oppositeEnd")   ref Switch[1] source; } </pre>

	<pre> @gmf.link(target.decoration="none", source.decoration="none", style="solid", color="0,0,0", source.constraint="self &lt;&gt; oppositeEnd") ref Switch[1] target; } </pre>
<b>Rule</b>	<pre> @gmf.node(figure="figures.RuleFigure", label="name", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/rule.gif", label.placement="external") class Rule {      @gmf.link(target.decoration="arrow", source.decoration="none", style="dash", color="0,0,0")     ref Host targetHostRule;     attr String name;      @gmf.link(target.decoration="none", source.decoration="none", style="dash", color="0,0,0")     transient ref Condition ruleCondition;      @gmf.link(target.decoration="none", source.decoration="none", style="dash", color="0,0,0")     ref Action ruleAction; } </pre>
<b>Action</b>	<pre> @gmf.node(figure="figures.ActionFigure", label="type", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/action.gif", label.placement="external") class Action extends RuleObject {     attr Actions type;      @gmf.link(target.decoration="arrow", source.decoration="none", style="dash", color="0,0,0")     ref PacketHeader actionPacketHeader;      @gmf.link(target.decoration="arrow", source.decoration="none", style="dash", color="0,0,0")     ref NetworkNode actionForwardToNode; } </pre>
<b>Condition</b>	<pre> @gmf.node(label="condition", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/condition.gif", label.placement="external", figure="figures.ConditionFigure") class Condition extends RuleObject {     attr Conditions condition = "WHERE";      @gmf.link(target.decoration="none", source.decoration="none", style="dash", color="0,0,0")     ref Time conditionTime;      @gmf.link(target.decoration="none", source.decoration="none", style="dash", color="0,0,0")     ref Traffic conditionTraffic;      @gmf.link(target.decoration="none", source.decoration="none", style="dash", color="0,0,0")     ref PacketHeader conditionPacket; } </pre>

<b>Traffic</b>	<pre>@gmf.node(figure="figures.TrafficFigure", label="operator,value,unit", label.pattern="{0}: {2}{1}", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/traffic.gif", label.placement="external") class Traffic extends RuleObject {     attr RelationalOperators operator;     attr String unit = "MB";     attr int value; }</pre>
<b>Time</b>	<pre>@gmf.node(figure="figures.TimeFigure", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/time.gif", label.placement="external", label="operator,beginDate,endDate", label.pattern="{0}: {1} - {2}") class Time extends RuleObject {     attr RelationalOperators operator;     attr EDate beginDate;     attr EDate endDate; }</pre>
<b>Packet Header</b>	<pre>@gmf.node(figure="figures.PacketHeaderFigure", label.icon="false", tool.small.bundle="mdn.edit", tool.small.path="/icons/full/obj16/packetheader.gif", label.placement="external", label="operator,header,value", label.pattern="{0}: [{1}] {2}") class PacketHeader extends RuleObject {     attr RelationalOperators operator;     attr PacketHeaders header = "";     attr String value; }</pre>

<b>Template</b>	header.egl
<b>Objective</b>	It defines the possible libraries used in code generated of MDN models.
<b>Implementation</b>	<pre>[% operation getCodeHeader() : String {     var header : String;      header = "         import pox         import pprint         import datetime         from pox.lib.packet.ethernet import ethernet,                                 ETHER_BROADCAST          from pox.lib.packet.ipv4 import ipv4         from pox.lib.packet.arp import arp         from pox.lib.addresses import IPAddr, EthAddr         from pox.lib.util import str_to_bool, dpidToStr         from pox.lib.recoco import Timer         from pox.core import core         from pox.lib.util import dpidToStr         import pox.openflow.libopenflow_01 as of          # include as part of the betta branch</pre>

	<pre> from pox.openflow.of_json import *  log = core.getLogger();  return header; } %]</pre>
--	--

Template	firewall.egl
Objective	It defines the actions to drop flows.
Implementation	<pre> [% operation actionDrop(rule : Any) : String { %]     ip = packet.find('ipv4')     # This packet isn't IP!     if ip is None:         return     # The flow is in the block list     [% var srcIP = rule.sourceHostRule.ip.toString(); %]     [% var dstIP = rule.targetHostRule.ip.toString(); %]     [% srcIP = srcIP.substring(1, srcIP.length()-1); %]     [% dstIP = dstIP.substring(1, dstIP.length()-1); %]     elif str(ip.srcip) == [%='\'+srcIP+'\'%] and str(ip.dstip) == [%='\'+dstIP+'\'%]:         # Verify its conditions, if any         [% var stringConditions : String = verifyConditions(rule); %]         [% if (stringConditions.length() &gt; 1) { %]             [%='if ('+stringConditions+')'%]:                 print("It is in the blocked list source %s - destination %s", ip.srcip, ip.dstip)                 event.halt = True             else                 event.halt = False         [% } %]         [% else { %]             print("It is in the blocked list source %s - destination %s", ip.srcip, ip.dstip)             event.halt = True         [% } %]     else:         print("Allowed source %s - destination %s", ip.srcip, ip.dstip) [% }%]</pre>

Template	monitor.egl
Objective	It defines the monitoring for specified hosts.
Implementation	<pre> [% operation monitorAction(rule : Any) : String { %] def _timer_func():     for connection in core.openflow._connections.values():</pre>

```

        connection.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))

        connection.send(of.ofp_stats_request(body=of.ofp_port_stats_request()))
        log.debug("Sent %i flow/port stats request(s)",
len(core.openflow._connections))

def _handle_flowstats_received (event):
    stats = flow_stats_to_list(event.stats)
    log.debug("FlowStatsReceived from %s: %s",
        dpidToStr(event.connection.dpid), stats)

    # Get number of bytes/packets in flows for web traffic only
    web_bytes = 0
    web_flows = 0
    web_packet = 0
    for f in event.stats:

        [% for (rule in Rule.All) { %]
        [% var srcIP = rule.sourceHostRule.ip.toString(); %]
        [% var dstIP = rule.targetHostRule.ip.toString(); %]
        [% srcIP = srcIP.substring(1, srcIP.length()-1); %]
        [% dstIP = dstIP.substring(1, dstIP.length()-1); %]
        if f.match.ip_dst == [%=dstIP%] or f.match.ip_src ==
[%=srcIP%]:
            web_bytes += f.byte_count
            web_packet += f.packet_count
            web_flows += 1
        [% } %]
        log.info("Web traffic from %s: %s bytes (%s packets) over %s
flows",
            dpidToStr(event.connection.dpid), web_bytes, web_packet,
web_flows)

# handler to display port statistics received in JSON format
def _handle_portstats_received (event):
    stats = flow_stats_to_list(event.stats)
    log.debug("PortStatsReceived from %s: %s",
        dpidToStr(event.connection.dpid), stats)
[%
}
%]

```

#### Simple Firewall Application - GPL (Python)

```

1. from pox.core import core
2. block_ports = set()
3. def block_handler (event):
4. ip = event.parsed.find('ipv4')
5. if not ip: return # Not IP
6. if ip.srcip == '10.0.0.1' and ip.dstip == '192.168.1.1':
7.     event.halt = True
8. def launch ():
9.     core.openflow.addListenerByName("PacketIn", block_handler)

```