

UNIVERSIDADE FEDERAL FLUMINENSE

FRANKLIN JORDAN VENTURA QUICO

**L-PRISM: A SPECIFICATION LANGUAGE  
FOR MULTIMEDIA SERVICE CHAINS  
BASED ON VIRTUALIZATION OF  
SENSORS**

NITERÓI

2023

UNIVERSIDADE FEDERAL FLUMINENSE

FRANKLIN JORDAN VENTURA QUICO

**L-PRISM: A SPECIFICATION LANGUAGE  
FOR MULTIMEDIA SERVICE CHAINS  
BASED ON VIRTUALIZATION OF  
SENSORS**

Master Thesis presented to the Programa de  
Pós-Graduação em Computação of the Uni-  
versidade Federal Fluminense as requirement  
to obtain the Degree of Master in Comput-  
ing. Area: Computer Science

Advisor:

DÉBORA CHRISTINA MUCHALUAT SAADE

Co-Advisor:

FLÁVIA COIMBRA DELICATO

NITERÓI

2023

Ficha catalográfica automática - SDC/BEE  
Gerada com informações fornecidas pelo autor

V4681 Ventura Quico, Franklin Jordan  
L-PRISM: A specification language for Multimedia Service  
Chains based on Virtualization of Sensors / Franklin Jordan  
Ventura Quico. - 2023.  
146 f.

Orientador: Débora Christina Muchaluat Saade.  
Coorientador: Flávia Coimbra Delicato.  
Dissertação (mestrado)-Universidade Federal Fluminense,  
Instituto de Computação, Niterói, 2023.

1. Domain Specific Language. 2. Internet of Things. 3.  
Internet of Media Things. 4. Virtual Multimedia Sensors. 5.  
Produção intelectual. I. Muchaluat Saade, Débora Christina,  
orientadora. II. Coimbra Delicato, Flávia, coorientadora.  
III. Universidade Federal Fluminense. Instituto de  
Computação.IV. Título.

CDD - XXX

Franklin Jordan Ventura Quico

L-PRISM: A specification language for Multimedia Service Chains based on  
Virtualization of Sensors

Master Thesis presented to the Programa de  
Pós-Graduação em Computação of the Uni-  
versidade Federal Fluminense as requirement  
to obtain the Degree of Master in Comput-  
ing. Area: Computer Science

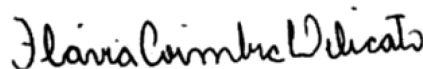
Approved in April 2023

APPROVED BY



---

Profª Débora Christina Muchaluat Saade - Advisor, UFF



---

Profª Flávia Coimbra Delicato - Co-Advisor, UFF



---

Prof. Paulo de Figueiredo Pires, UFF

Documento assinado digitalmente



THAIS VASCONCELOS BATISTA

Data: 31/05/2023 17:37:20-0300

Verifique em <https://validar.iti.gov.br>

---

Profª Thaís Vasconcelos Batista

Niterói

2023

# Acknowledgements

I want to thank God for giving me the strength to reach another life goal.

I appreciate the unconditional support of my dear mother, Florencia, who was always there to encourage me to follow my dreams and for being one of the greatest examples in life.

I want to thank my advisor, Professor Débora, who had supported me since before I even met her, even before I arrived in this beautiful country; thank you very much for the time dedicated to this research and for all the patience you have with me!

I thank my co-advisor, Professor Flavia, for agreeing to advise me, for all the time she dedicated to this research, and for her patience with me!

I also thank my friend Anselmo Battisti for his support in this and other projects. I also thank my friends Julio and Jose for taking the time to participate in face-to-face training in the experiments of this thesis.

Finally, I thank all my friends and family for the love and friendship. Last but not least, I thank everyone at the UFF Institute of Computing for welcoming me so well.

# Resumo

A virtualização é uma tecnologia amplamente utilizada que pode reduzir a complexidade derivada da heterogeneidade em ambientes IoT. Uma vez que os sensores multimídia são uma importante fonte de dados em IoT, surge o paradigma *Internet of Media Things* (IoMT). Com base na virtualização e na IoMT, adotou-se o conceito de virtualização de sensores multimídia (VMS), que são a representação virtualizada dos dispositivos pertencentes à IoMT. Em muitos cenários, vários processos devem ser aplicados a fluxos multimídia em uma sequência predefinida, criando assim cadeias de VMS, ou seja, cadeias de serviços multimídia. Existem poucos esforços na literatura para criar uma linguagem de descrição para apoiar a definição de cadeias de VMS. Para preencher esta lacuna, propomos uma Linguagem Específica de Domínio (DSL) denominada L-PRISM. Esta DSL pode ser utilizada como base conceitual para desenvolvedores implementarem e virtualizarem aplicações multimídia utilizando o conceito de VMS. Também apresentamos uma prova de conceito (PoC) usando L-PRISM para executar cadeias de serviços multimídia baseadas em VMS. Nosso DSL e PoC foram avaliados por desenvolvedores de software, e os resultados mostram que a adoção do L-PRISM facilita a definição e implantação de cadeias de serviços multimídia baseadas em VMS.

**Palavras-Chaves:** Linguagem de Domínio Específico, Internet das Coisas, Internet das Coisas Multimídia, Sensor Virtual Multimídia.

# Abstract

Virtualization is a widely used technology that can reduce the complexity derived from heterogeneity in IoT environments. Once multimedia sensors are an important data source in IoT, the Internet of Media Things (IoMT) paradigm emerges. Based on virtualization and IoMT, the concept of virtualization multimedia sensors (VMS) has been adopted, which are the virtualized representation of the devices belonging to IoMT. In many scenarios, multiple processes must be applied to multimedia streams in a predefined sequence, thus creating chains of VMS, i.e., multimedia service chains. There are few efforts in the literature to create a description language to support the definition of chains of VMS. In order to fill this gap, we propose a Domain Specific Language (DSL) called L-PRISM. This DSL can be used as a conceptual base for developers to implement and virtualize multimedia applications using the VMS concept. We also present a proof of concept (PoC) using L-PRISM to run multimedia service chains based on VMS. Our DSL and PoC were evaluated by software developers, and the results show that adopting L-PRISM facilitates the definition and deployment of multimedia service chains based on VMS.

**Keywords:** Domain Specific Language, Internet of Things, Internet of Media Things, Virtual Multimedia Sensors.

# List of Figures

2.1	V-PRISM three-tier architecture overview [42]. . . . .	31
2.2	VMS Hierarchical Categorization [42]. . . . .	32
2.3	VMS creation sequence diagram [42]. . . . .	36
3.1	Example of extended TOSCA template for VM description (the extended security attributes are in bold) [32] . . . . .	49
4.1	components of a Multimedia Service Chain based on VMS . . . . .	55
4.2	Class diagram of the L-PRISM metamodel . . . . .	57
5.1	Class diagram of the L-PRISM metamodel based on TOSCA-NFV. . . . .	62
5.2	Simple multimedia service chain with L-PRISM . . . . .	102
5.3	Diagram and output of a complex multimedia service chain. . . . .	102
5.4	Specification of virtual devices with the metamodel of L-PRISM . . . . .	103
5.5	Specification of virtual multimedia sensors with the metamodel of L-PRISM	104
5.6	Specification of virtual link with the metamodel of L-PRISM- Part 1 . . .	106
5.7	Specification of virtual link with the metamodel of L-PRISM- Part 2 . . .	107
5.8	Specification of virtual link with the metamodel of L-PRISM- Part 3 . . .	108
6.1	Interface to upload YAML file. . . . .	113
6.2	Interface to upload YAML file - List of VMS. . . . .	113
6.3	Interface to upload YAML file - List of Edge Node and list of VD. . . . .	114
6.4	Interface for register type VMS ( <i>swImage</i> ) . . . . .	114
6.5	Interface for view detail of type VMS ( <i>swImage</i> ) . . . . .	114
6.6	Interface to create a <i>VMS</i> , through the intuitive interface. . . . .	115
6.7	Interface to select a <i>VD</i> , through the intuitive interface. . . . .	116



6.8	Intuitive interface to create multimedia service chain based on VMS. . . . .	116
6.9	General example of a multimedia service chain created in <i>prototype</i> . . . . .	117
7.1	View of the website of main elements of L-PRISM for creating a multimedia service chain based on VMS . . . . .	127
7.2	View the examples of multimedia service chains created with L-PRISM on the website. . . . .	127
7.3	Details of the examples of multimedia service chains created with L-PRISM on the website. . . . .	127
7.4	View the examples of multimedia service chains created with V-PRISM on the website. . . . .	127
7.5	Multimedia service chain that transforms a color video to grayscale - Task 1.	128
7.6	Multimedia service chain that bundles two video streams - Task 2. . . . .	128
7.7	Complex multimedia service chain - Task 3. . . . .	129
7.8	Reuse of multimedia service chains - Task 4. . . . .	129
7.9	Number of subjects. . . . .	131
7.10	Academic level. . . . .	131
7.11	Experience in the use of XML, JSON, and YAML languages. . . . .	131
7.12	Time needed per task in the two methods ( <i>L-PRISM</i> and <i>Classic method (V-PRISM)</i> ) . . . . .	132
7.13	Summary of responses to G1 questions Q2-Q5 . . . . .	133
7.14	Summary of responses to G2 questions . . . . .	135
7.15	Results of question Q3 of objective G2 . . . . .	137

# List of Tables

2.1	Comparison of scalar and multimedia IoT data [29]	27
3.1	Compares languages, templates, and data models	50
4.1	Variables for VNF creation	54
5.1	Properties of <i>l3AddressData</i>	63
5.2	Properties of <i>addressData</i>	65
5.3	Properties of <i>connectivityType</i>	66
5.4	Properties of <i>virtualCpu</i>	68
5.5	Properties of <i>virtualMemory</i>	69
5.6	Properties of <i>virtualStorage</i>	71
5.7	Properties of <i>virtualGraphicsCard</i>	72
5.8	Properties of <i>configurableProperties</i>	73
5.9	Properties of <i>port</i>	74
5.10	Properties of <i>metric</i>	76
5.11	Properties of <i>virtualCompute</i>	77
5.12	Properties of <i>swImage</i>	80
5.13	Properties of <i>host</i>	84
5.14	Properties of <i>device</i>	86
5.15	Properties of <i>source</i>	87
5.16	Properties of <i>destination</i>	90
5.17	Properties of <i>vms</i>	92
5.18	Properties of <i>virtualLink</i>	97
5.19	Properties of <i>chainModel</i>	99

---

7.1	Goals of the experiment . . . . .	119
7.2	Questions for the goal $G1$ . . . . .	119
7.3	Questions for the goal $G2$ . . . . .	123
7.4	Tasks of the experiment . . . . .	129
7.5	Tasks used for Goal $G1$ . . . . .	130
7.6	Tasks used for Goal $G2$ . . . . .	130

# List of Acronyms

API	:	Application Programming Interface;
CDN	:	Cognitive Dimensions of Notations;
CCO	:	CPU Consumed;
CoT	:	Cloud of Things;
DSL	:	Domain Specific Language;
EC	:	Edge Computing;
EF	:	Edge Function;
ENM	:	Edge Node Manager;
ETSI	:	European Telecommunications Standards Institute;
GQM	:	Goal Question Metric;
IoMT	:	Internet of Media Things;
IoT	:	Internet of Things;
IP	:	Internet Protocol;
JSON	:	JavaScript Object Notation;
L-PRISM	:	Language for Programming IoT Sensors for Multimedia;
LXC	:	Linux Containers;
MBT	:	Megabits Transferred;
MEC	:	Multi-access Edge Computing;
NFV	:	Network Function Virtualization;
OS	:	Operating System;
PoC	:	Proof-of-Concept;
QoE	:	Quality of Experience;
QoS	:	Quality of Service;
RAM	:	Resource Allocation Manager;
REST	:	Representational State Transfer;
ROI	:	Return On Investment;
TAM	:	Technology Acceptance Model;
V-PRISM	:	Virtual Programmable IoT Sensor for Multimedia;
VD	:	Virtual Device;

---

VMS	:	Virtual Multimedia Sensor;
VS	:	Virtual Sensor;
YAML	:	YAML Ain't Markup Language;

# Contents

<b>1</b>	<b>Introduction</b>	<b>20</b>
1.1	Research Question . . . . .	21
1.2	Goals . . . . .	24
1.3	Main Contributions . . . . .	25
1.4	Dissertation Organization . . . . .	25
<b>2</b>	<b>Background</b>	<b>26</b>
2.1	Internet of Media Things - IoMT . . . . .	26
2.2	Virtualization Applied to IoMT . . . . .	28
2.3	V-PRISM and ALFA . . . . .	30
2.3.1	V-PRISM . . . . .	30
2.3.1.1	VMS Categorization . . . . .	31
2.3.1.2	V-PRISM Logic Components . . . . .	33
2.3.1.3	V-PRISM Deployment . . . . .	35
2.3.1.4	Initialization of a VMS . . . . .	35
2.3.2	ALFA . . . . .	36
2.3.2.1	ALFA Main Technologies . . . . .	36
2.3.2.2	Implemented Virtual Devices . . . . .	39
2.3.2.3	Implemented Virtual Multimedia Sensors . . . . .	39
2.4	Domain Specific Language (DSL) . . . . .	39
2.4.1	Development of a DSL . . . . .	40
2.4.1.1	Implementation of a DSL . . . . .	41

<b>3</b>	<b>Related Work</b>	<b>45</b>
3.1	TOSCA-NFV . . . . .	45
3.2	Other languages . . . . .	47
<b>4</b>	<b>L-PRISM Proposal and Development Process</b>	<b>51</b>
4.1	Domain analysis . . . . .	52
4.2	L-PRISM Design . . . . .	54
4.3	L-PRISM Implementation . . . . .	56
4.4	L-PRISM evaluation . . . . .	59
4.5	L-PRISM Maintenance . . . . .	60
<b>5</b>	<b>L-PRISM Metamodel</b>	<b>61</b>
5.1	Data Types . . . . .	63
5.1.1	lPrism.datatype.vms.l3AddressData . . . . .	63
5.1.1.1	Definition . . . . .	64
5.1.1.2	Example . . . . .	64
5.1.2	lPrism.datatype.vms.addressData . . . . .	65
5.1.2.1	Definition . . . . .	65
5.1.2.2	Example . . . . .	66
5.1.3	lPrism.datatype.vms.connectivityType . . . . .	66
5.1.3.1	Definition . . . . .	66
5.1.3.2	Example . . . . .	67
5.1.4	lPrism.datatype.vms.virtualCpu . . . . .	67
5.1.4.1	Definition . . . . .	68
5.1.4.2	Example . . . . .	69
5.1.5	lPrism.datatype.vms.virtualMemory . . . . .	69
5.1.5.1	Definition . . . . .	70

---

5.1.5.2	Example . . . . .	70
5.1.6	IPrism.datatype.vms.virtualStorage . . . . .	70
5.1.6.1	Definition . . . . .	71
5.1.6.2	Example . . . . .	71
5.1.7	IPrism.datatype.vms.virtualGraphicsCard . . . . .	72
5.1.7.1	Definition . . . . .	72
5.1.7.2	Examples . . . . .	73
5.1.8	IPrism.datatype.vms.configurableProperties . . . . .	73
5.1.8.1	Definition . . . . .	74
5.1.8.2	Examples . . . . .	74
5.1.9	IPrism.datatype.vms.port . . . . .	74
5.1.9.1	Definition . . . . .	75
5.1.9.2	Example . . . . .	76
5.2	Capabilities Types . . . . .	76
5.2.1	IPrism.capabilities.vms.metric . . . . .	76
5.2.1.1	Definition . . . . .	76
5.2.1.2	Example . . . . .	77
5.2.2	IPrism.capabilities.vms.virtualCompute . . . . .	77
5.2.2.1	Definition . . . . .	78
5.2.2.2	Example . . . . .	79
5.3	Artifact Types . . . . .	79
5.3.1	IPrism.artifacts.vms.swImage . . . . .	79
5.3.1.1	Definition . . . . .	82
5.3.1.2	Example . . . . .	83
5.3.2	IPrism.artifacts.vms.host . . . . .	84
5.3.2.1	Definition . . . . .	84



5.3.2.2	Example . . . . .	85
5.3.3	lPrism.artifacts.vms.device . . . . .	85
5.3.3.1	Definition . . . . .	86
5.3.3.2	Example . . . . .	87
5.4	Relationship Types . . . . .	87
5.4.1	lPrism.relationship.vms.source . . . . .	87
5.4.1.1	Definition . . . . .	88
5.4.1.2	Example . . . . .	89
5.4.2	lPrism.relationship.vms.destination . . . . .	89
5.4.2.1	Definition . . . . .	90
5.4.2.2	Example . . . . .	91
5.5	Node Types . . . . .	91
5.5.1	lPrism.nodes.vms.VDU.vms . . . . .	91
5.5.1.1	Definition . . . . .	94
5.5.1.2	Example . . . . .	96
5.5.2	lPrism.nodes.vms.virtualLink . . . . .	97
5.5.2.1	Definition . . . . .	97
5.5.2.2	Example . . . . .	98
5.6	Chain Type . . . . .	98
5.6.1	lPrism.chain.vms.chainModel . . . . .	98
5.6.1.1	Definition . . . . .	99
5.6.1.2	Example . . . . .	100
5.7	Example of the L-PRISM Metamodel . . . . .	102
<b>6</b>	<b>ALFA 2.0: Integration of L-PRISM in ALFA</b>	<b>110</b>
6.1	Database . . . . .	110
6.2	API . . . . .	111

6.3	Web interface . . . . .	112
6.3.1	Web interface for L-PRISM . . . . .	112
6.3.2	Prototype web interface . . . . .	114
6.4	Differences between ALFA and ALFA 2.0 . . . . .	117
<b>7</b>	<b>Evaluation</b>	<b>118</b>
7.1	Experiment Goals . . . . .	118
7.1.1	G1 - Questions and metrics . . . . .	118
7.1.1.1	G1 - Questions . . . . .	119
7.1.1.2	G1 - Hypotheses, Variables, and Constructions . . . . .	119
7.1.1.3	G1 - Metrics . . . . .	121
7.1.2	G2 - Questions and metrics . . . . .	122
7.1.2.1	G2 - Questions . . . . .	122
7.1.2.2	G2 - Hypotheses, Variables, and Constructions . . . . .	123
7.1.2.3	G2 - Metrics . . . . .	125
7.2	Experiment Tasks . . . . .	126
7.2.1	Training phase . . . . .	126
7.2.2	Execution phase . . . . .	127
7.2.3	Tasks . . . . .	128
7.3	Subjects . . . . .	129
7.4	Results . . . . .	131
7.4.1	G1 - Evaluation . . . . .	131
7.4.2	G2 - Evaluation . . . . .	134
7.4.3	Threats to Validity . . . . .	136
7.4.4	Final considerations . . . . .	138
<b>8</b>	<b>Conclusion</b>	<b>139</b>

Contents	xvi
8.1 Contributions . . . . .	141
8.2 Future work . . . . .	141
<b>References</b>	<b>143</b>

# Chapter 1

## Introduction

In recent years, virtualization technology has gained significant relevance in different research areas. One of these areas is that of networks, where the introduction of the concept of Network Function Virtualization (NFV) seeks to decouple network functions from physical devices through software-based implementation [32]. The introduction of the NFV concept has significantly impacted different fields of research and the industry. This impact is mainly because NFV reduces capital costs and operating (CapEx, OpEx), increases the network's efficiency and agility, provides a shorter time to implementation, and improves scalability in the reuse of resources [32].

The Internet of Things (IoT) is another concept that is having a significant impact on the network area. The main objective of IoT is to have a more connected world where users can connect to their devices (sensors) faster and intelligently. A sub-area of IoT is the Internet of Media Things (IoMT) [20], which focuses on working with multimedia-type sensors and applications. One of the main limitations of IoT sensors is that they have limited resources for their operation, this limitation is inherited to IoMT sensors, but it has a more impact on these since IoMT sensors produce and process complex data, compared to traditional sensors that produce and process discrete data. The virtualization technology applied to IoMT partially addresses the resource limitation problems in this type of device since the processing and sending of data to end users would be inherited to the virtual sensor [42]. The research carried out by [42] defines the V-PRISM architecture, where virtualization technology is applied to IoMT and creates the concept of Virtualized Multimedia Sensor (VMS). This concept encompasses the virtualization of sensors belonging to IoMT. The VMSs bring with them several benefits, such as reducing the heterogeneity of the physical sensors, reduction of costs since the VMSs make it possible that it is not necessary to acquire physical sensors, greater Return on Investment (ROI)

since the VMSs can be reused without any additional cost and finally increase of fault tolerance.

There are scenarios where complex applications must process data streams by different functions in a defined sequence, thus creating a chain of functions or services. We define these chains of functions or services in IoMT as a Multimedia Service Chain, which aims to transfer and process multimedia data through different functions or services distributed within a network. In a virtualization-based environment based on the VMS concept, the functions that are part of a multimedia service chain would be represented by VMSs, where each VMS will perform operations on multimedia data streams and transmit this processed data through the network.

The architecture proposed by [42] enables the technologies for creating multimedia service chains based on VMS. Multimedia service chains based on VMS could be managed more practically and simply. This would allow of the Quality of Service (QoS) [22] control in the components and the entire service chain to be more accessible. Another aspect to consider is that, with the virtualization of multimedia sensors, the Quality of Experience (QoE) can be addressed, a metric that up to now can only be evaluated at the application level, but with virtualization, it is possible to monitor information related to QoE within the network [4, 5].

A common issue in research in the multimedia area is that different researchers have proposed and developed many solutions in the form of software tools that use multimedia content as a base. These multimedia data can be transformed or processed by algorithms whose final objective is to deliver one or several responses. One issue with these tools (software modules), which we call functions, is that most of them are implemented and tested in closed environments, and their use requires the manual intervention of an expert. To use each of these functions or services, an exhaustive analysis must be made about its operation and behavior, to understand what it does, how it works, and what its results are. That is why most of these functions are little used in practice.

## 1.1 Research Question

Over time, different applications using virtualization concepts have been developed. Most of these applications are created without following a specification or pattern in their development. This implies that external users who wish to use these applications must have an intermediate to advanced level of knowledge in developing said applications, that

is, in the technologies used during their creation. Consequently, this generates a significant limitation of the community's access to this type of technology.

One of the main advantages of multimedia applications is their ability to support multimedia streams with different characteristics, such as different types of formats, resolutions, frame rates, bit rates, and compression codecs, allowing greater flexibility in the management and playback of multimedia content. These advantages can be a double-edged sword for VMSs since the variation of any of these characteristics can affect the behavior and operation of VMS. In a virtualized environment, where the multimedia streams processed by VMSs will have different characteristics, a static configuration of computing and network resources in a VMS may be inappropriate.

As previously mentioned, one of the problems we face is that to make correct use of virtualized multimedia applications (VMS) one must have a medium to advanced knowledge of the technologies used both in the development of said applications and the technologies used for their virtualization.

An alternative to address this problem is to create a specification for the virtualization of multimedia sensors, which would encourage users to want to use these virtualized applications. This would allow users interested in using these virtualized applications not to need advanced knowledge about the technologies used in their development. However, even with a specification for the virtualization of multimedia applications, users should have basic to advanced knowledge of the virtualization technologies used.

The level of knowledge of users who want to use virtualized multimedia applications (VMS) will depend on how complex the IoMT application they want to create will be. For example, suppose they only want to use a VMS. In that case, they should only know how to deploy a VMS, but in the case of a complex IoMT application, where different VMSs must process the multimedia stream(s) in a distributed environment, users should have knowledge about deploying the applications, and connect them, which translates into knowing networks, virtualization, and distributed systems.

A study that we take as a base in this work is the architecture presented by [42]. This architecture addresses the concepts of virtualization in Edge Computing environments, which addresses the problems mentioned earlier in a certain way. [42] presents a proof of concept (PoC) of his architecture called ALFA and allows the creation of IoMT application based on VMS through an intuitive web interface. The problem with ALFA is that it only allows you to create simple IoMT applications or simple multimedia service chains based on VMS. To create complex multimedia service chains based on VMS, you need

to have advanced knowledge about the virtualization technology used in ALFA. The communication between VMSs in ALFA is based on IP addresses and ports, which means that in order to create complex multimedia service chains, it would be necessary to have knowledge about the available IP addresses and ports of each VMS and to have access to this information. It is necessary to have relatively advanced knowledge about the virtualization technology used in ALFA, which in this case is Docker. It is also necessary to know how a multimedia service chain works; for example, the correct sequence of creation of each multimedia service chain based on VMS should be. Another limitation of ALFA is that it only allows you to configure aspects at the application level in the VMS. Hence, the configuration of the allocation of computational and network resources is not possible. Furthermore, the most significant limitation of ALFA is that it limits us to the use of the VMS developed by the author, which makes it difficult if we want to use a virtualized multimedia application from another author, since [42] does not present a formal specification to virtualize and integrate virtualized multimedia applications to ALFA.

One of the solutions that address some of ALFA problems is to extend ALFA so that it allows creating and configuring the allocation of resources to the VMSs of a multimedia service chain through its web application. However, even with this, we are still limited to using the VMSs developed by [42].

Lastly, in real scenarios, developers of complex multimedia applications tend to create these applications very frequently and some of these applications are usually very similar. ALFA proposed extension would allow these applications to be created, but they would have to be developed each time from scratch, and depending on the complexity of the application, the time required to develop the same application several times would be considerable.

Using a web application to develop multimedia service chains based on VMS is not always the most appropriate solution. A web application can be beneficial for users who are testing the operation of this type of application due to its intuitiveness. However, if we talk about users who constantly work developing multimedia service chains based on VMS, using a web application is not the most appropriate since other solutions can increase their productivity. Additionally, when it comes to complex multimedia service chains, it is necessary to have an overview of them, so using a web interface becomes a bit difficult.

Based on the problems and solutions mentioned previously, this work investigates the

use of Domain Specific Languages (DSL) to create virtualized multimedia applications. Therefore our main research question is:

**RQ:** *Does using a Domain Specific Language (DSL) facilitate the deployment of multimedia service chains based on VMS?*

We investigate if the definition of a domain language (DLS) that supports the use of multimedia applications based on virtualization, such as Virtualized Network Functions (VNF) or Virtualized Multimedia Sensors (VMS), will offer the community the possibility to deploy multimedia applications developed by third parties without the need for prior or advanced knowledge about its implementation, since a DSL can be used as a conceptual basis for the virtualization of applications and also for the correct use of these applications. A DSL also gives us an overview of the implemented solutions, in our case, multimedia service chains based on VMS. Also, the use of a DSL reduces the implementation time of already developed solutions since the solutions can be stored in a file where the solutions are described, and the reuse of these solutions would be very simple; with this, it would be possible to increase the productivity of the users. Additionally, solutions developed with a DSL can serve as a knowledge base for other users since DSLs have the characteristic of being more expressive and close to the vocabulary of a domain, which facilitates the understanding and writing of new solutions.

## 1.2 Goals

The main goal of this work is to propose a Domain Specific Language (DSL) for creating multimedia service chains based on VMS. This DSL is called L-PRISM (Language for Programming IoT Sensors for Multimedia) and is based on YAML (YAML Ain't Markup Language) [39]. L-PRISM can also be used as a conceptual base for developers to implement and virtualize multimedia applications using the VMS concept. These virtualized applications could be published, aiming that other developers can use these simply using L-PRISM. Our proposal will use the concepts of virtualization, Edge Computing (EC), Internet of Media Things (IoMT), and DSL.

L-PRISM addresses the essential components and attributes of multimedia service chains. It should be noted that, as other domain languages, it focuses on creating solutions based on already developed tools or VMS, so it does not address the VMS implementation process itself, but how VMSs can be used together to build complex solution or multimedia service chain.



## 1.3 Main Contributions

This work provides the following contributions:

- Propose L-PRISM, a DSL designed to create multimedia services chains based on VMS.
- ALFA 2.0: ALFA Extension for the Integration of DSL L-PRISM in ALFA.
- Evaluation of the proposal for creating multimedia service chains based on VMS using L-PRISM with an experiment with software developers.

## 1.4 Dissertation Organization

The remainder of this document is organized as follows. Chapter 2 depicts the background of our research. Chapter 3 presents related work. We present the L-PRISM proposal and development process in Chapter 4. The L-PRISM metamodel is discussed in Chapter 5. Chapter 6 presents ALFA 2.0 and how L-PRISM was integrated into the ALFA platform. In Chapter 7, we present an evaluation of our work. Chapter 8 brings main conclusions and future work.

# Chapter 2

## Background

This chapter presents the main concepts used in this dissertation. Section 2.1 presents how IoMT arises. Section 2.2 presents concepts related to virtualization and how it can be applied in the field of IoMT. Section 2.4 presents the main concepts about domain-specific languages (DSL) and mentions some of the most prominent ones.

### 2.1 Internet of Media Things - IoMT

The core concept of Internet of Things (IoT) is to connect omnipresent objects such as mobile devices, sensors, and actuators through a wired or wireless network [29]. This allows these objects to interact with each other to create or improve systems. The notable growth of devices connected to the Internet and the increasing demand for multimedia traffic have given rise to the Internet of Media Things (IoMT) [20][2].

Generally, sensors are designed to consume the least amount of energy possible, with little storage memory and low processing power. One way to differentiate traditional sensors from multimedia sensors is that traditional sensors produce scalar data, and multimedia sensors produce unstructured data [29]. The transmission of data from multimedia sensors requires higher bandwidth, large amounts of memory, and greater computational power to process this data.

The authors of [29] conducted a detailed study about IoMT. They presented the difference between data from traditional sensors and multimedia sensors, as depicted in Table 2.1.

The characteristics of multimedia data make them bulky and require greater bandwidth in their transmission process. For this reason, over time, more efficient and intel-

Table 2.1: Comparison of scalar and multimedia IoT data [29]

Required Parameter	Scalar IoT Data	Multimedia Data
Data Size (Approximate)	Bytes to Kilobytes	Megabytes to Gigabytes
Memory	Kilobytes to Megabytes	Megabytes to Gigabytes
Processing	Kilohertz to Megahertz	Megahertz to Gigahertz
Storage	Kilobytes to Megabytes	Gigabytes
Bandwidth	Kilobytes per second	Megabytes per second
Delay Sensitivity	Low	High
Power Consumption	Low	High

ligent network solutions have been developed that cover this problem, such as the case of Edge Computing (EC) [18], which provides benefits such as low latency, consumption of bandwidth in the core of the network, better use of resources and perhaps increased security/privacy [18, 9].

Data transmission in traditional multimedia applications can be point-to-point, point-to-multipoint, or multipoint-to-multipoint. Instead, IoMT applications would require immense resources to perform point-to-multipoint or multipoint-to-multipoint transmission, which is challenging for IoMT [40].

With the arrival of 5G networks, the increase in network bandwidth, and the more multimedia services offered, the demand for higher-quality multimedia content has increased rapidly [3]. This, in turn, has caused networks to adopt new technologies, such as Software Defined Network (SDN) [17], to facilitate the management of resources within the network in a scalable, flexible, and dynamic way. Another technology adopted is Network Function Virtualization (NFV) [32], which broke with the traditional monolithic software and hardware approach.

SDN and NFV allow complex applications to be created on distributed virtual platforms, executing network functions as if they were an application but in a virtual machine (VM). Thus, their administration and management become more straightforward. SDN and NFV are two enabling technologies of 5G networks [17] that offer the possibility of creating QoE-compliant value-added multimedia services, such as high-quality, low-latency 3D/4K/8K video streaming.

Traditional applications aim to comply with Quality of Service (QoS), unlike multimedia applications that must also focus on Quality of Experience (QoE). According to [10], QoE is “*the degree of pleasure or annoyance to an application or service user. It results from the fulfillment of their expectations regarding the usefulness and/or enjoyment of the application or service in light of the personality and current state of the user.*”, which makes it difficult to define whether an application is QoE compliant, unlike the QoS that is responsible for measuring the performance of an application, in the case of IoMT the performance from the perspective of the network.

## 2.2 Virtualization Applied to IoMT

Virtualization is the logical abstraction of physical devices within a network through the implementation of software [1], which allows modification, management, and updating to be carried out more easily. Virtualization applied to sensors is not a new idea but emerging as a viable solution for IoT environments. There are different reasons why adopting virtual sensors is an alternative to face the challenges of IoT. Some of these are presented by [42] and are listed as follows.

- **Heterogeneity of physical sensors:** The great variety of sensor providers means they have different characteristics, so in physical environments, the coexistence of sensors from different providers is complex or difficult to maintain [42, 23]. With sensor virtualization, the lower-level interface is decoupled.
- **Impossibility to build a physical device:** The costs of physical sensors can be an obstacle in the system development process. Sensor virtualization is a much more accessible alternative in terms of costs [42, 24].
- **Reuse physical sensors to increase ROI (Return Of Investment):** Virtual sensors can be used in different services; unlike their physical peers, which can only work in one system, virtual sensors can be part of different systems using the existing fabric [42, 24].
- **Increase fault tolerance:** Monitoring physical sensors is a challenge, either due to the specific characteristics of each provider or the complexity of accessing their monitoring services. Sensor virtualization is one alternative that can help make sensor-based system monitoring less complex and more accessible to data [42].

The use of virtualization in IoT environments makes these more agile, robust, and profitable since it would reduce the number of physical devices needed, the network could be easily segmented, and security policies could be more easily applied to virtual sensors [28].

One technology that has greatly impacted networks is Network Function Virtualization (NFV), a mechanism for abstracting network functions such as firewalls, load balancing, route calculation, Etc [1]. On-demand NFV deployment improves scalability and elasticity to build self-contained and cost-effective applications. NFV optimizes computing capabilities such as memory, storage, and network, unlike their physical peers who have limited these resources.

Implementing IoMT applications in the cloud or edge and their distribution within the network nodes have presented challenges, either due to the particular characteristics of these flows or because they require more resources for their correct processing. One way to deal with these challenges is to use virtualization. However, traditional virtualization or virtualization based on hypervisor is not the most appropriate method since this virtualization method requires more computing resources. The virtualization based on hypervisor installs a complete operating system in a virtual machine, which generates a greater use of computational resources, and the images used for virtualization are big.

Lightweight virtualization in recent years has positively impacted IoT environments, as it brings benefits such as fast startup, low deployment costs, and energy efficiency [33]. Virtualization based on containers is a lightweight alternative to virtualization based on hypervisor. The concept of "containerization" is not new in virtualization, but it has become very relevant in recent years. Docker [15] is a tool that uses containers, where processes or applications can run simultaneously within a container, it can also be used to create services in different containers and these interact with each other.

The introduction of Edge computing brought with it advantages such as latency reduction, traffic reduction, and context awareness [9]. It also brought new challenges, such as resource limitations in edge nodes. Lightweight virtualization based on container is an alternative to meet these challenges, as it offers the following advantages [9].

- Quick creation and initialization of virtualized instances [9].
- High density of applications, thanks to the fact that the images used are smaller than virtualization based on hypervisors [9].
- Reduced overhead, virtualization based on container runs its processes on the same

kernel, which reduces power consumption on the host machine [12].

As mentioned above, the idea of virtualization is not recent, and there are different areas where this technology is used. Virtualization applied to multimedia sensors brings with it all the advantages of IoT sensor virtualization. Additionally, different approaches can be applied to address IoMT challenges.

Adopting virtual sensors in IoMT is an excellent strategy to address the challenges it brings. A software component emulating a multimedia device is called a Virtual Multimedia Sensor (VMS). Virtualization based on container helps make VMS development more accessible, as it allows the best-suited technology and requirements to be used to each VMS [42].

## 2.3 V-PRISM and ALFA

### 2.3.1 V-PRISM

V-PRISM is an architecture proposed by [42], aimed at managing and orchestrating virtualized multimedia applications in an Edge Computing (EC) environment. V-PRISM is divided into three levels; cloud, edge, and things as illustrated in Figure 2.1. The cloud tier is big data centers that have almost unlimited computing and network resources. These data centers are located a long distance from things, which causes higher latency between the cloud and things. Elements that are hosted at this tier are IoMT applications. The Edge level is the intermediate layer between things and the cloud, where the computing and network resources are smaller compared to the cloud, but the distance between this layer and things is smaller, so latency decreases considerably. The elements that are hosted at this level are mostly Edge Nodes, Virtual Devices (VD), Virtual Multimedia Sensors (VMS), V-PRISM Manager and in some cases IoMT applications. Multimedia devices are located in the things level. Generally, the elements at this level have limited computing and network resources, so it is difficult for them to process and store data. The objective of this level is to produce data streams.

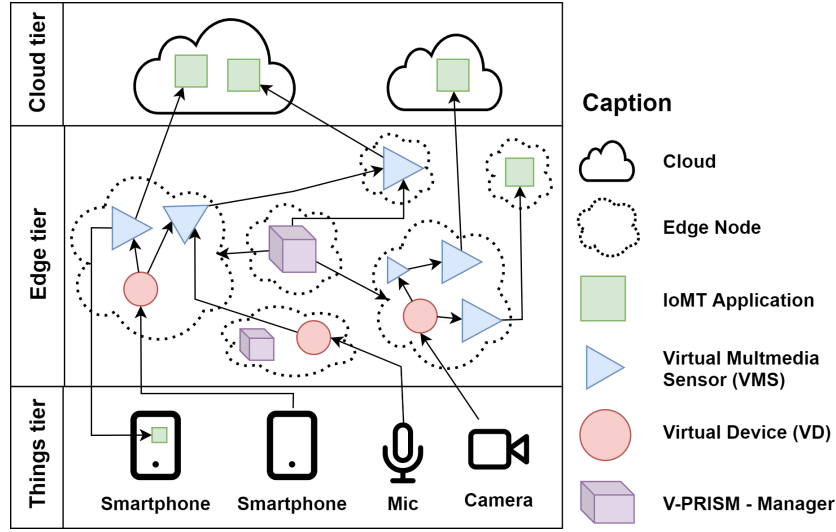


Figure 2.1: V-PRISM three-tier architecture overview [42].

To improve the understanding of this architecture, we point out that a VMS and a VD are virtualized applications that will process a multimedia stream. The virtualization of these applications follows the concept of Network Functions Virtualization (NFV) but is oriented only to multimedia applications; for what we could say that a VMS is a Virtualized Network Function of multimedia types (VNF - Multimedia).

In Figure 2.1, the process of a multimedia flow produced by a multimedia device in the things tier is sent to a VD, which will be in charge of retransmitting the multimedia flow to the VMS that requests it. The VMS will commonly perform some operation on the multimedia flow and finally will send this multimedia stream to another VMS or some IoMT application.

### 2.3.1.1 VMS Categorization

V-PRISM categorizes virtualized multimedia applications to make it easier to organize and understand these applications. The categorization depends on the abstraction of the type of multimedia stream the virtualized multimedia application offers. To carry out this categorization [42] coined two new terms *Types of VMS* and *Types of VD*, where Type of VMS refers to a class and VMS to an object. The categorization follows two purposes, to categorize the virtualized applications according to the level of complexity (from the simplest to the most complex), so it also allows directing this type of categorization to the amount of computing and network resources to instantiate these virtualized applications. The second purpose is to provide templates for each type of virtualized application (VMS)

to facilitate developers' use of these VMS.

Figure 2.2 categorizes the types of virtualized multimedia applications, or as [42] calls VMS, into three categories. VMS as *thing*, when VMS is in charge of forwarding media streams. VMS as *Process*, when the VMS, apart from forwarding the multimedia streams, also applies some function on this stream. VMS as *Service*, when the VMS applies high-level abstract functions to media streams.

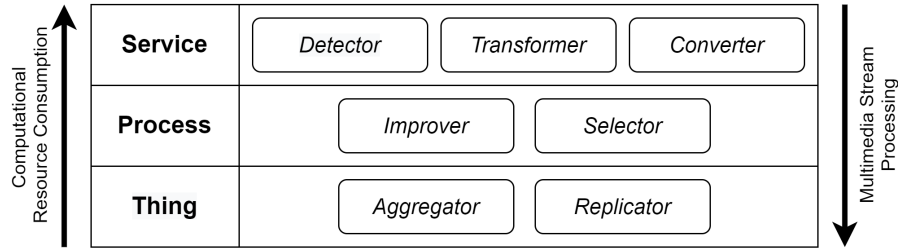


Figure 2.2: VMS Hierarchical Categorization [42].

Figure 2.2 also presents a second level of categorization, where virtualized multimedia applications can be categorized as:

- *Replicator*: VMS in this category abstracts all the characteristics and behavior of physical multimedia devices. These VMS do not change the media streams since their primary function is to forward multimedia streams. The computational resources required to run this type of VMS are low and can be run on edge nodes with low capacities.
- *Improver*: VMS in this category fulfill the function of adding some type of functionality to IoMT applications, such as; for example, security for multimedia streams produced by a physical device.
- *Converter*: VMS in this category address one of the main challenges of IoT, which is the heterogeneity of things since these VMS are responsible for transforming multimedia streams according to the needs of multimedia applications.
- *Selector*: The VMS in this category receives multiple media streams and returns a single multimedia stream. A characteristic of these VMS is that the different multimedia streams it receives must be the same type.
- *Aggregator*: A VMS in this category fulfills the function of receiving multiple multimedia drawings and returning a single multimedia stream. Multimedia streams



receiving this type of VMS must be of the same type.

- *Detector*: The VMS in this category detects a specific event in the multimedia stream. It is common for VMS of this type to use Artificial Intelligence techniques to process the multimedia streams.
- *Transformer*: The VMS in this category fulfills the function of changing the nature of the multimedia stream it receives. An example of this type of VMS is the video-to-text stream converter.

### 2.3.1.2 V-PRISM Logic Components

V-PRISM follows the ETSI [13] meta-specification and, as a general specification, addresses the challenges of IoMT in Edge Computing environments. The main components of V-PRISM fulfill the function of processing multimedia streams produced by multimedia devices and consumed by IoMT applications. As described above, multimedia applications are generally implemented in the cloud, and the physical devices that produce the multimedia streams are at the lowest layer of this architecture (things). The processing of multimedia streams produced by physical devices is performed by Virtual Multimedia Sensors (VMS).

The V-PRISM architecture presents a set of components, which we briefly describe below:

- *Virtual multimedia sensor*: A Virtual Multimedia Sensor (VMS) is the architectural component in charge of processing multimedia streams from Physical Devices (multimedia things). Two types of entities can use the data produced by a VMS. The first is the IoMT applications, and the second is another or other VMS. Therefore, VMS can form a VMS chain, which we call multimedia service chains based on VMS in this work.
- *Virtual Device*: A Virtual Device (VD) is the architectural component in charge of receiving the flow or multimedia flows produced by the physical devices; only a VD can be connected to a physical device. VD is independent of the type of multimedia stream since it is only responsible for transporting the multimedia streams. The relationship between the VD and VMS is one-to-many since the VD also plays the role of replicating multimedia streams and sending these streams to different VMSs.

- *VMS Registry and VMS Request Manager*: The VMS Registry component is used to manage and provide a list of descriptors for the VMS types available on each Edge Node. The VMS Request Manager (VRM) component is responsible for receiving requests to create or delete a VMS.
- *Master*: The master component has an overview of the computing and network resources of the edge nodes. Maestro was specified based on the multi-access edge orchestrator component proposed in [16]. Maestro is the first component executed after VRM.
- *Resource Allocation Manager (RAM)*: It is the component in charge of defining which Edge Node a VMS will be installed. This component manages information such as the amount of CPU, memory, and storage of the Edge Nodes. One of the premises of this component is that Edge Nodes have limited resources and must be well managed to guarantee the Quality of Service (QoS) of IoMT applications.
- *Virtualization engine*: This component is in charge of providing interfaces for the virtualization of the main components (VD and VMS). Virtual Engine is created to address the heterogeneity of the technologies used to develop VMS. It also advocates the use of lightweight virtualization systems, such as containers, since one of the characteristics of V-PRISM is the limitation of computing and network resources on Edge Nodes.
- *Edge Node Manager (ENM)*: This component is in charge of managing all Edge Nodes. This component manages information about how many and which edge nodes are available at any given time. ENM's responsibilities are: receiving failure of virtualized resources, performance measurements of the virtualization infrastructure, providing management elements of core V-PRISM components, and preparing the infrastructure to run the VMS images.
- *Environment Monitor & VN Monitor*: This component provides statistical data about the VMS and VD.
- *VMS Administrator*: This component is responsible for creating, deleting, and managing VMS. It also provides monitoring information, such as metadata, to *VN monitor*. This component must be run on each edge of V-PRISM.
- *VD Manager (VDM)*: This component is responsible for creating, deleting, and managing VD. Like VMS, VD provides monitoring information that *VN monitor*

will use. One of the requirements to create a VD is that the Edge Node where the VD is created must have access to the physical device. This component must be run on each edge of V-PRISM.

- *Stream Sharer (SS)*: This component allows a VMS to send a multimedia stream to different destinations.

### 2.3.1.3 V-PRISM Deployment

As seen so far, V-PRISM has different components which can be implemented in different Edge Nodes. V-PRISM defines that each Edge Node must have at least the Virtualization Engine component installed and one or more VMS or VD types.

To enter the V-PRISM environment, the Edge Nodes must have registered in the *Edge Node Manager*; with this, the Edge Nodes can be identified with a unique ID later to use that ID in the VMS registration and VD.

One of the requirements of V-PRISM is that the Edge Nodes must have some network communication with each other in order to make the architecture more secure.

### 2.3.1.4 Initialization of a VMS

The initialization of a VMS has two phases. In the first phase, the VRM component defines whether the request is valid. In the second phase, if the request is valid, the Master component creates the VMS. Figure 2.3 shows the sequence diagram of creating a VMS, where it can be seen that the Service Registry component sends a request to Maestro to create the VMS. Maestro requests the Edge Node Administrator to retrieve the list of Edge Nodes available and that have already executed the requested VMS, having the list of candidate Edge Nodes and the characteristics of the VMS is sent to the Resource Manager and through the use of a resource allocation algorithm, it will choose which Edge Node VMS will run. Finally, this information is taken to the VMS manager, who will call VM to create the VMS.

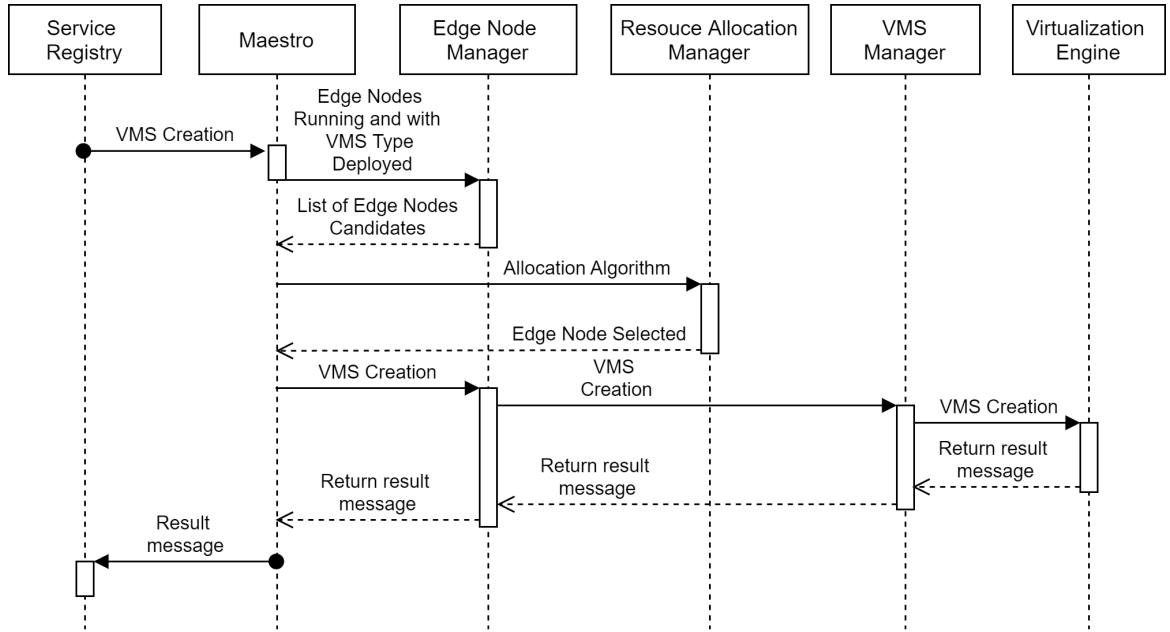


Figure 2.3: VMS creation sequence diagram [42].

## 2.3.2 ALFA

ALFA is the implementation of V-PRISM, presented as proof of concept (PoC) by [42]. ALFA instantiates all the components presented in the V-PRISM three-tier architecture. The source code, as well as additional information for the use of ALFA, is published on GitHub<sup>1</sup>.

### 2.3.2.1 ALFA Main Technologies

The main technologies used to implement ALFA aim to represent the interaction of the technologies of the V-PRISM components to generate an environment where VMS and VD can work in Edge Computing environments.

As mentioned above, V-PRISM does not restrict the technologies used to develop VMS or VD. However, in the implementation, ALFA highlights that GStreamer and OpenCV were mainly used to implement VMS and VD.

Each VMS or VD is a software instance that is executed within a Docker container, and its objective is to provide a multimedia flow as if it were a microservice.

The communication protocol used for the implementation of the VMS and VD is UDP, therefore, the physical devices on the level of things send their multimedia streams

<sup>1</sup><https://github.com/midiacom/alfa>

to VDs using UDP.

### 1. *Edge Nodes*

In ALFA, the Edge Nodes are devices with storage, memory, and CPU capable of running Docker, which are accessible through a TCP/IP network. ALFA allows the registration of different Edge Nodes so that they can perform all V-PRISM tasks. However, for the Edge Nodes to be able to communicate, it is necessary to use Docker Swarm. Docker Swarm is used since it offers load balancing, scaling, multi-host networking, and security.

After an Edge Node has been registered in ALFA, it will be necessary to install the different VMS types and VD types that will be used in this Edge Node. Not all VMS and VD need Internet access, only VMS that communicates with IoMT applications in the cloud.

### 2. *Docker*

ALFA uses Docker as a lightweight virtualization tool since it virtualizes solutions using fewer computational resources than other virtualization tools, such as hypervisor-based virtualization. A Docker container represents the virtualization of each of the VMS or VD, so it is possible to manage the life cycle of the containers using the Docker API.

Docker also has a mechanism to capture the monitoring data of each edge node and container, so it is possible to use the different components of V-PRISM; for example, the resource allocation component.

### 3. *VD and VMS in Containers*

VMS and VD are run in Docker containers. As mentioned above, V-PRISM does not restrict the types of technology used for VMS or VD development. ALFA uses GStreamer as the main library to develop different VMS and VD since it offers robust tools to manipulate multimedia transmission.

To start a Docker container, a container image must first be created. This image contains a Dockerfile, which contains all the packages needed for the installation of this image. Once the Dockerfile is created, it will be used to create the Docker image using the Docker CLI. ALFA provides Script to install the images available in ALFA easily.

### 4. *Main APIs used*

ALFA uses two different APIs. The first is the Docker API, which is used for container orchestration and lifecycle, that is, containers' orchestration (create, delete, monitor, and configure). The second API is ALFA REST API, and was developed by the authors to enable ALFA for users. A web application was also developed which uses the ALFA REST API. This web application is presented in more detail below.

5. **Web interface application** ALFA web application was developed to manage ALFA infrastructure; this application communicates with the APIs implemented in the server through an HTTP connection. The main modules of the ALFA web application are:

- *VMS*: Lists all VMS created on all Edge Nodes.
- *Virtual Device*: Lists the VDs created in all the Edge Nodes, which are independent of each IoMT application and can be used as a multimedia data source.
- *Edge Nodes*: Lists the registered Edge Nodes, their status, and the number of instances running within this Edge Node.
- *Locations*: Handles all places where a physical device can be deployed.
- *VMS types*: This shows the list of images (VMS types) available to be used. It also allows the registration of new VMS-type Docker images.
- *Device Types*: This shows the list of images (Device types) available to be used. It also allows the registration of new VD-type Docker images.

As mentioned, each VMS and VD are created in a container, and the communication between them is done through a publish/subscribe service. To perform this action, ALFA uses the MQTT<sup>2</sup> protocol within each VD, which will start or stop the multimedia transmission.

6. **Resource allocation management** Different methods for resource allocation are adapted to the V-PRISM architecture. In ALFA, the Round-Robin algorithm is used for the allocation of resources. This method is one of the least complex and is capable of running in machines with few resources.

---

<sup>2</sup><http://mqtt.org/>

### 2.3.2.2 Implemented Virtual Devices

Initially, each VD is a Docker image that contains the source code of the VD at the software level. When executing these images, we create containers that work as an available microservice. There are two types of communication between the VD and VMS. The first is the media stream (received by UDP) which flows in only one direction. The second is the control data (received by TCP), which can flow in both directions, such as text messages sent to an MQTT server. Among the VDs implemented in ALFA, the VD *RTSP to UDP video*, *USB Camera*, *Mic* and *SmartPhone RTSP Server* stand out.

### 2.3.2.3 Implemented Virtual Multimedia Sensors

Like VDs, VMSs are encapsulated in Docker images containing the source code that will process the multimedia stream and deliver the results to another VMS or an IoMT application.

Among the VMS implemented in ALFA, there are *Video Greyscale*, *Video Crop*, *Video QR code detection*, *Noise detector*, *face counter*, and *Video Mosaic*.

In this work, we use V-PRISM and ALFA for different stages of our proposal. We also present an extension of ALFA, oriented to the administration of multimedia service chains. This extension is called ALFA 2.0 and is presented in more detail in Chapter 6.

## 2.4 Domain Specific Language (DSL)

The development of computer tools for specific domains is increasingly complex, either because of each domain different characteristics or the different fields of knowledge that each domain needs. For example, different fields must be considered to develop a modern web system, such as usability, security, persistence, and the different business rules [21].

An alternative to developing these tools is the adoption of a Domain Specific Language (DSL), which allows implementation solutions based on already developed technologies, where users of a DSL would not have to worry about the encoding technologies with which these base technologies have been developed.

A DSL allows for better abstraction compared to commonly used programming languages. However, developing a DSL is difficult since different tools must be provided to support the DSL development and maintenance process, ensuring its consistency, evalu-

ation, and maintainability [21].

In [21], the authors present six categories, which address the characteristics a DSL must have, as detailed below.

- **Notation:** Determines if the models or programs developed with the DSL are presented in the textual, graphical, or tabular form [21].
- **Semantics:** Refers to the meaning of the models developed with the DSL; the semantics can be of two types, translational and interpretive. Translational semantics generate a program in a language based on the model. Interpretive semantics executes a model without any previous translation [21].
- **Editor support:** It refers to the editing mode of the models developed with the DSL. There are two editing modes, free-form editing, where the user can freely write and edit their model, and projection editing, where the user has to work with pre-defined layouts to write and edit their models [21].
- **Validation:** The DSL can identify structural, syntactic, and semantic disagreements and inconsistencies in the models developed by the users. There are two types of validation: structural and semantic validation [21].
- **Tests:** The tests aim to debug the language; for this, you can perform semantic and syntactic tests [21].
- **Compatibility:** It refers to the fact that the metamodels of a DSL can include structural characteristics of other metamodels [21].

### 2.4.1 Development of a DSL

The DSL can be developed in two ways, both internal and external: Internal DSLs are designed to work with the base language and are limited by the grammar of the host language, which limits the editor's functionality since the editor does not know the grammar of the host language. Unlike the internal DSL, the external DSL is implemented independently of the host language, so it has its grammar and editor; this makes the DSL more flexible, and the editor can help programmers develop solutions for the DSL domain. This dissertation uses the term DSL when referring to an external DSL.

The DSL development process consists of five stages [30].



- **Domain analysis:** In this stage, the domain for which the DSL is being developed is analyzed. To perform this task, it is necessary to obtain and understand the concepts and relationships of the domain, and it is also important to define the limit or limits of the domain.
- **Design:** In this stage, the design of the DSL is defined. To carry out this task, it will be necessary to use the domain problems found in the previous stage, and each problem will be evaluated and coded.
- **Implementation:** In this stage, the language is implemented based on the design defined in the previous stage. To perform this task, tools available for DSL implementation can be used.
- **Evaluation:** In this stage, the new DSL is evaluated in order to know if the DSL meets the business needs.
- **Maintenance:** It is a stage after the startup; this stage is carried out continuously to update the new DSL to the continuous changes of the business.

The DSL analysis and design stages depend directly on the study of the domain for which the DSL is proposed. Therefore, a specific process is not defined for the analysis and design stages since they vary according to the specific domain.

#### 2.4.1.1 Implementation of a DSL

There are two approaches to implementing a DSL, the compiler approach and the interpreter approach. In the compilation approach, the compiler translates programs written with the DSL into a low-level language for later execution by the computer. In the interpretation approach, programs written with the DSL will be interpreted by a compiler designed for the DSL. The interpretation approach leads to a complex task since it will be necessary to implement a compiler, which requires a lot of effort and time. In addition, it should be noted that DSLs are lightweight languages; therefore, their implementation must be fast and with as little effort and knowledge as possible.

According to [30], the implementation stage of a new DSL must comply with three work components: 1) The abstract representation that will include the definition of the language structure, 2) The editor, which will allow the user to implement, manipulate the abstract representation and 3) The generator, which will transform the abstract representations into a coded executable.

## 1. Aspects of language structure

To define the structure of a DSL, there are mainly two approaches.

- The first approach is the *grammar-based* approach, where the language will be defined directly by the grammar of the language; usually, Context Free Grammar (CFG) is used for the formal definition of the language syntax; the metalanguages of the CFG are modified to fit appropriately with the new DSL definition.
- The second approach is *model-based*; this approach does not use grammar rules; on the other hand, this approach uses metamodels to define the structure of the language. Programs written with the model-based approach conform to the metamodel defined by the language developer; Furthermore, the structure of the metamodel of the language is used to describe the abstract syntax and not the concrete syntax. One of the main advantages of using this technique is that the model-based approach provides a higher level of abstraction of the domain than the grammar-based technique.

In this work, we use the model-based approach to define the structure of our DSL. This approach will help us better represent the concepts, attributes, and relationships between the components of a multimedia service chain based on virtual multimedia sensors.

## 2. Aspects of the Language Editor

To define the DSL editor, the structure defined in the previous stage must be considered. Based on this structure, it is defined how users of the new language will edit solutions based on the new DSL. Users will be able to edit their solutions, either grammar-based or model-based. Two approaches can approach the definition of the editor of a DSL:

- The first approach is *parser-based editing*. In this type of editing, users interact with the concrete syntax; the process consists of users inputting sequences of characters to a text buffer, then a parser matches this sequence with the language's grammar, thus building an Abstract Syntax Tree (AST). This approach is commonly used when the DSL structure is defined following the grammar-based approach.
- The second approach is *projection editing (i.e., parser-less editing)*, also called parser-less editing; since you do not need a parser to build the AST,

there is no need to transform the concrete syntax into the abstract syntax. This type of edition depends on the abstract representation of the domain and can be represented in textual, graphical, or tabular form. This approach is commonly used when the DSL structure is defined following the model-based approach.

In this work, we define the editor of our DSL using the projection-based approach since language modularity, notation freedom, and representation flexibility of solutions based on the new DSL can be easily achieved using this approach.

### 3. Aspects of language semantics

The semantics of the language define step by step how the computer executes the solutions implemented with the language of the new DSL, and there are two ways to define the semantics of a DSL:

- ***Translational semantics*** or translation semantics define the meaning of the language and how it will be translated into a different language; commonly, DSL languages are translated into a general-purpose language such as Java, JavaScript, C, Python, Etc. An example of translational semantics is compilers, which translate a high-level language into a low-level language.

The translation of a language can be implemented in two ways:

- (a) *Model transformation* (i.e., from model to model) consists of translating the DSL language into a different language independent of the specific syntax of both languages. Model transformation can be done using two approaches. The first approach is the classic one, which consists of building a destination AST while traversing the source AST. The second approach builds a relationship between the source AST and the destination AST.
  - (b) *Code generation* (that is, Model-to-Text) consists of directly transforming the source language into the source code of the target language. The source language must have embedded source code from the target language to implement this translation.
- ***Interpretive semantics***, unlike translation semantics, directly translates the solutions implemented by the DSL language, or in other words, executes the source language directly without translating it into another language. The semantic actions are executed while the AST is traversed to carry out this process.

In this work, we define the language semantics of our DSL using translational semantics; additionally, we use model transformation using the classical approach for language translation.

In short, DSLs are language specifications that describe in detail the features, syntax, and behavior of systems in a specific field [27]. Over the years, different domain languages have been proposed to help develop network systems and architectures, such as YANG [34], and TOSCA-NFV [36], as will be described in the following chapter.

# Chapter 3

## Related Work

This chapter presents the works related to using DSL, templates, metamodels, etc., to develop IT solutions. Our primary focus is whether any of these works suits our requirements. Additionally, they help us better understand why, how, and for what these tools were designed, and we also mention some works that use some of these tools in their solutions.

TOSCA-NFV is a language that we take as the basis for our proposal; for this reason, we detail this language more deeply.

### 3.1 TOSCA-NFV

The TOSCA-NFV profile is a specific data model for Network Function Virtualization (NFV) based on the TOSCA language. This profile aims to capture in a template the requirements for the implementation and behavior of Network Services (NS) based on Virtualized Network Functions (VNF) to later create the instances based on the specification [36].

The TOSCA metamodel uses different templates to describe cloud network services workloads; The main templates are the node template, responsible for modeling the service components, and the relationship template, responsible for modeling the relationships between the service components. Additionally, TOSCA provides templates for node types and relationship types; these templates help define a service's life cycle operations. Finally, a service template, which groups the different templates described above, will be processed by an orchestration engine.

TOSCA-NFV uses the concepts of NFV and mentions that end-to-end services in NFV

require software tools for their management and orchestration and that most of these tools use Network Service Descriptors (NSD) to capture the requirements of implementation and operational behavior of network services (NS).

In order to describe the network services based on NFV, TOSCA implements the NFV template, which describes the attributes and requirements necessary to implement this service. It also presents an NFV orchestrator (NFVO), which is in charge of managing the life cycle of the network service; a VNF Manager (VNFM), which manages the life cycle of a VNF; and finally VIM, which is in charge of managing the resources virtualized.

The template in charge of describing a network service is called Network Service Descriptor (NSD); this describes the relationship that exists between the VNF and possibly the Physical Network Functions (PNF) and their connections through Virtual Links (VL).

Below are the different TOSCA-NFV templates [36]:

- VNF Descriptor (VNFD) is a template describing a VNF in terms of requirements and behavior. It also contains requirements for connectivity, interface, and virtualized resources such as CPU, RAM, and disks.
- VNF Forwarding Graphical Descriptor (VNFFGD) is a template that describes the or part of the network service topology, referring to the VNFs, PNFs, and VLS.
- Virtual Link Descriptor (VLD) is a template that describes the links between VNFs, PNFs, and network service endpoints.
- Physical Network Function Descriptor (PNFD) is a template that describes the connectivity, interface, and KPI requirements between the physical device and VL.
- Network Service Descriptor (NSD) is a template NFVO uses to manage the NS life cycle. Within the NSD structure, one can find zero or many VNFDs, zero or many PNFs, zero or many VLDs, and finally, zero or many nested NSDs.

The templates proposed by TOSCA-NFV are based on the TOSCA-simple-Profile-YAML specification; they use this specification based on how a VNF is implemented. For example, they describe that the VNFD template is similar to the specification of a Virtual Deployment Units (VDU), and for this reason, the VDU can be considered a subsystem of a VNF and thus be part of the VNFD template.

The description of the different templates of the TOSCA-NFV metamodel is composed of a series of elements that are grouped into seven main groups [36].

- Data Types
- Artifact Types
- Capability Types
- Requirements Types
- Relationships Types
- Interface Types
- Node Types

These groups describe different elements that will be part of the TOSCA-NFV templates. These elements can be part of other elements as an aggregation function.

In this work, we use the TOSCA-NFV metamodel as a reference for implementing our DSL. We will use the elements partially or totally; in some cases, we will add new attributes to the existing elements. This is shown in more detail in Chapter 5, where we compare the TOSCA-NFV metamodel and the metamodel of our proposal.

## 3.2 Other languages

A domain language related to the network area is YANG [34], a data modeling language used to describe network configurations and telecommunications services. YANG can be used for NETCONF-based operations, including configuration, data state, remote procedure calls (RPCs), and notifications. YANG models can be translated into XML syntax, allowing applications using XML parsers to operate and manipulate YANG [7] models. However, the YANG language does not have the features required to describe virtual media sensors.

OpenStack developed a project for the orchestration of applications in the cloud. That project is called HEAT [31]. Heat is a template that allows the configuration of multiple composite applications in the cloud. HEAT additionally has an orchestration engine to launch applications [31]. HEAT presents its different components through the YAML language, with which you can configure instantiation, networking, storage, and pretty much everything that OpenStack offers [31].

The work of [38] proposes an architecture for federated network environments based on OpenStack. This presents OpenStack Federation Flow Manager (OSFFM), an orchestration agent layered to parse a template HOT [31]. The authors extend HOT intending to be able to implement distributed edge computing services. For this, it creates two types of templates: Microservices template that is fully compatible with HOT and distributed services template compatible with the proposed HOT extension. It should be noted that these templates are based on the YAML serialization language, as it is interoperable, open, and human-friendly.

Mininet-NFV [11] proposes a framework for NFV orchestration to implement and operate NFV or network services on Mininet. This work takes as a reference that Mininet is an excellent tool for agile experimentation of networks/SDN/NFV and that different works use Mininet for prototyping, testing, or implementation of NFV-based solutions. However, it stands out that none of these works uses TOSCA-NFV as a data model to create templates or descriptors of their solutions [11]. The primary operations of Mininet-NFV for the use of TOSCA-NFV are that it takes an inventory of the allocation of virtual and physical resources, it uses a VNF descriptor (VNFD) that defines the behavior information and deployment of VNF and deployment of end-to-end network services in templates using decomposed VNFs [11]. Mininet-NFV supports parameterized TOSCA-NFV templates and network definitions via virtual link descriptors.

In [32], the authors propose an extension of the TOSCA-NFV model, allowing basic security functions to be organized effectively, allowing clients to use these functions as a service, and establishing high-level security policies for VNFs. That study concludes that open-source VNF orchestrators do not allow establishing security policies in the life cycle of NFV services. [32] extends the TOSCA data model, incorporating a set of security-related attributes such as security level, tenant domain, members, and security groups, as shown in Figure 3.1. He also develops a security orchestrator prototype that uses a TOSCA analyzer and software-defined tenant access control. They also validated this security orchestrator prototype.

In contrast to the languages, templates, and data models presented, our L-PRISM proposal focuses on using lightweight virtualization to create IoMT applications (create VMS) to deploy VMS chains (Multimedia Service Chains) in Edge Computing environments. L-PRISM abstracts most of the variables that influence the behavior of VMS and its interaction within the network, so L-PRISM allows:

- Register VMS in architectures that have integrated L-PRISM more simply.



```

Example
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0
description: TOSCA-based security model for VM description
topology_template:
  --node_templates:
    --VM1:
      --type: tosca.nodes.Compute
      --capabilities:
        num_cpus: 2
        mem_size: 4096 MB
        disk_size: 2 GB
      --properties:
        image: cirros-0.3.5-x86_64-disk
        availability_zone: nova
        vm_security_level: low
        tenant_domain: tenant1
        members: {(user1, high), (user2, medium), ..., (user5, low)}
    --VM2:
      --type: tosca.nodes.Compute
      --capabilities:
        num_cpus: 2
        mem_size: 4096 MB
        disk_size: 2 GB
      --properties:
        image: cirros-0.3.5-x86_64-disk
        availability_zone: nova
        vm_security_level: medium
        tenant_domain: tenant1
        members: {(user1, high), (user2, medium), ..., (user5, low)}

```

Figure 3.1: Example of extended TOSCA template for VM description (the extended security attributes are in bold) [32]

- Present VMS clearly and the summarized way so that third parties can use these VMS without needing advanced knowledge about their implementation.
- Create multimedia service chains using VMS.
- Manage multimedia service chains, as well as their components.

Table 3.1 compares some languages, templates, and data models. The last row of the table presents our L-PRISM proposal. Each column in the table means.

- **IoMT:** Analyzes some of the languages, templates, and data models in IoMT applications.
- **Edge Computing:** Analyzes if some languages, templates, and data models work on edge computing environments.
- **Light Virtualization:** Analyzes if some languages, templates, and data models were designed to work with some light virtualization method to virtualize its components.
- **Service Chain:** Analyzes if some of the languages, templates, and data models support the definition of service chains in their models.

Table 3.1: Compares languages, templates, and data models

<b>Papers</b>	<b>IoMT</b>	<b>Edge Computing</b>	<b>Light Virtualization</b>	<b>Service Chain</b>
YANG [34]	-	-	-	✓
HEAT [31]	-	-	-	✓
OSFFM [38]	-	✓	-	✓
TOSCA-NFV [36]	-	✓	-	✓
Mininet-NFV [11]	-	✓	✓	✓
TOSCA-NFV Security [32]	-	✓	-	✓
<b>L-PRISM</b>	✓	✓	✓	✓

# Chapter 4

## L-PRISM Proposal and Development Process

This chapter introduces L-PRISM, a Domain Specific Language (DSL) [21] that describes multimedia service chains based on virtual media sensors (VMS). As mentioned in previous chapters, L-PRISM is based on serialization language TOSCA-NFV [36] but focuses on lightweight virtualization technologies and Edge Computing. L-PRISM is based on the YAML serialization format for its simple, readable, and friendly syntax.

The main advantages of L-PRISM are:

- L-PRISM allows describing, implementing, and managing multimedia service chains based on VMS.
- In general, L-PRISM facilitates the use of virtualized multimedia applications or VMS developed by third parties, so developers of VMS-based solutions do not need to have advanced knowledge about the technologies or tools used in developing the components of a multimedia service chain.
- L-PRISM also enables interoperability and portability of multimedia service chains across different platforms or architectures that have L-PRISM embedded. This advantage was not tested in this work, so it is proposed for future work.

To develop our Domain Specific Language (DSL) L-PRISM, we follow the implementation model of an external DSL since our focus is to represent how a multimedia service chain and its components work. Additionally, external DSLs are more flexible compared to internal DSLs; this means that by using the external DSL implementation method, we do not depend directly on the base language that will compile the solutions, with this

we address one of our secondary objectives, which is to that our DSL can be used as a theoretical basis for the implementation of future solutions that use the virtualization of multimedia sensors as a basis.

To achieve the five stages of DSL development, we follow the steps defined in [30], which are described in the following sections.

## 4.1 Domain analysis

In this stage, an analysis was carried out on multimedia applications in virtualized environments to understand and abstract the main characteristics that influence the correct functioning of multimedia applications. It should be noted that the abstracted features do not address the internal implementation of the applications, but only configurable characteristics, non-functional requirements, such as allocation of computing and network resources, communication requirements, etc.

The work carried out by [19] mentions that service chains or chaining of network functions allow the automatic connection of services in a wired or wireless network. These service chains can serve different objectives, which makes their components or functions differ within each type of service chain. The authors of [19] also mention that the Network Functions Virtualization (NFV) and Software Defined Networks (SDN) enable the possibility that the connection of virtualized network services can be made automatically. In turn, this would allow the creation of service chains based on Virtualized Network Functions (VNF).

For our study, virtualization technology allows us to divide multimedia services chains into levels of virtualized multimedia functions or, as it is called in [42], Virtual Multimedia Sensors (VMS). The technology used for this division is known as service segmentation [19]. These virtualized functions can be administered and managed by different applications; for example, in [3] and [19], they use OpenStack to instantiate the physical part and create a VNF. This allows for managing computational resources such as CPU, RAM, and GPU for each VNF.

The architecture proposed by [42] divides the necessary resources for the proper functioning of the VNF into two parts. In the first part, it describes the necessary resources for sending data; These resources know the nodes that are part of the cloud edge present within the network, the number of VNFs that can be executed in each cloud edge node, the latency that exists between the nodes and the application server, the CPU usage

level, amount of free memory, battery level, resource usage history, Etc. The second part focuses on resources at the application level, as is the case of CPU, RAM, GPU, and bandwidth necessary for an application to run correctly.

In the different investigations carried out, various service chains have been observed; in turn, it can be seen that the components that are part of these service chains are very similar. For example, [3] presents a teleimmersive multimedia application that aims to communicate with two users (players) and viewers interacting in a 3D environment. For this application to work, it is necessary to use a Transformer multimedia 3D (T3D), which both players will use, and a Replay to allow viewers to connect. In a typical scenario, one server with all three physical devices (two T3Ds and one Replay) would need to be used to create this media service chain. However, using virtualization technology, in this case, NFV, the T3D (vT3D - VNF), and the Replay (vReplay - VNF) could be virtualized. This chain of multimedia services based on NFV would be composed of two Physical Network Functions (PNF) that will have to be instantiated, three VNFs, and Virtual Links (VL) that will be in charge of connecting the entire chain of multimedia services.

At [3], they also present a chain of multimedia services that aims to broadcast live-streaming events. This chain of multimedia services is composed of a digital signal transformer (SDI) to Internet Protocol (IP), a media processor (MPE), a compression engine (CE), a speech-to-text (S2T) engine, and a network service (NS). In that example, MPE, CE, S2T, and NS can be virtualized network functions and services, making the multimedia service chain much more efficient and cost-effective. This virtualization-based multimedia service chain comprises a PNF, different VNFs, and VLs that connect this entire multimedia service chain.

The variables found in the different studies are presented in Table 4.1, these variables were described directly or indirectly since some articles mention them as problems to be solved, and others evaluate their experiments with these variables.

Another analysis that was carried out is the scenarios where different applications have to interact with each other as if it were a directed graph. During our study, we saw that complex multimedia applications need to process one or many multimedia streams through different processes. In our proposal, we call these complex applications as multimedia service chains, and those in charge of processing the multimedia streams are the Virtual Multimedia Sensors (VMS).

Table 4.1: Variables for VNF creation

	Network bandwidth	Latency	RAM	CPUs	Energy
Battisti [42]	X	X	X	X	X
Alvarez [3]	X	X	X	X	
Imagane [19]		X	X	X	
Zikria [41]	X	X	X	X	X

## 4.2 L-PRISM Design

To carry out this stage, we remember that our main objective is to create multimedia service chains based on virtual multimedia sensors (VMS). In addition to the fact that we use as a base the architecture proposed by [42], where multimedia service chains can be implemented in distributed environments, and the virtualization model is lightweight virtualization or virtualization based on containers. The design process of our DSL has the following steps.

1. **Defining the components of the multimedia service chain:** In this step, we identify the different services and components that will be part of different types of multimedia service chains.

To better understand the components that are part of a multimedia service chain, we present Figure 4.1, in which it can be seen that a multimedia service chain is made up of different elements, each of these is presented below:

- (a) **Virtual Device (VD):** The VDs can be part of one or many multimedia service chains; consequently, when a multimedia service chain is created, it does not create VDs; the multimedia service chains subscribe to the VDs.

This may include services for data capture, processing, storage, transmission, and playback of multimedia content.

- (b) **Virtual Multimedia Sensor (VMS):** The VMS is in charge of processing the multimedia streams coming from the VD, one of the main characteristics of the VMS is that the amount of computational resources necessary for its correct

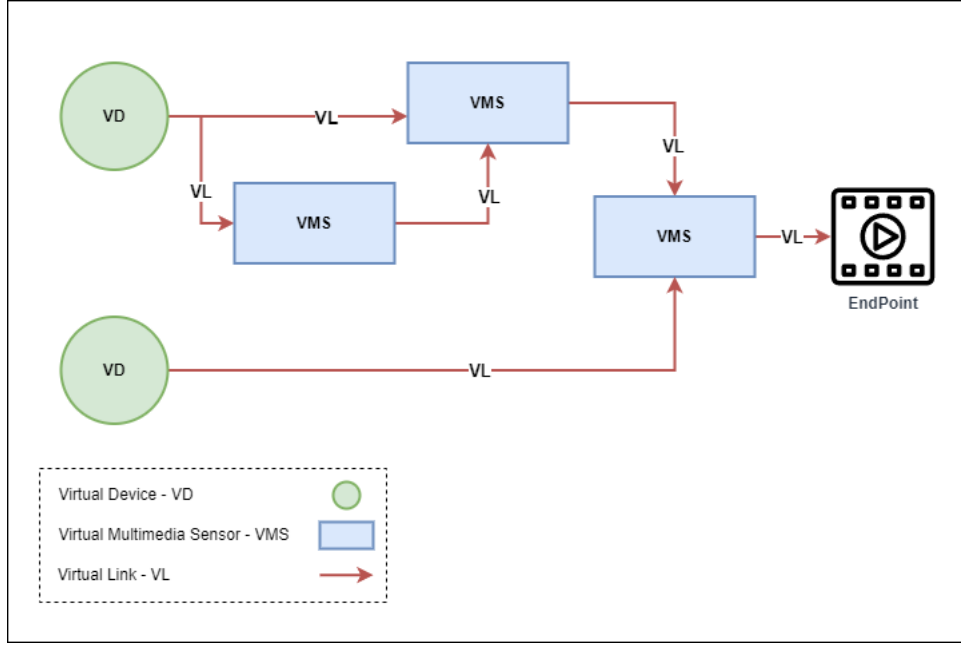


Figure 4.1: components of a Multimedia Service Chain based on VMS

operation is not the same for all solutions, the allocation of these resources will depend on the multimedia stream to be processed. Also, unlike VDs, VMSs are created with the multimedia service chain.

- (c) **Virtual Link (VL):** The VLs are responsible for creating communication from VD to VMS, VMS to VMS, and VMS to an endpoint. It should be noted that since the lightweight virtualization model is being used, we define the VLs in the simplest way possible and do not focus on the aspect of network configuration and management.

2. **Container design:** In this step, we define the containers that will house the different services and components of the chain. Taking into account that each service or component will be encapsulated in an independent container, which will facilitate its deployment, management, and scalability.

The container design is defined based on the architecture proposed by [42]; since we use docker as a lightweight virtualization tool, based on docker and the characteristics of the components of a multimedia service chain, we define the services necessary that we will use from docker to fulfill our primary objective.

3. **Design of the communication between the containers:** In this step, we establish the communication and coordination mechanisms between the containers that are part of the multimedia services chain. For this, we will use as a base the architecture proposed by [42], where the standard communication protocol MQTT is

used, and communication interface definition based on docker, where we will create a swarm where we can register each of the nodes (servers) distributed within a network and in turn, each container belonging to this swarm will be able to communicate and exchange data with each other.

As we mentioned before, the communication design between the components of the multimedia service chain was defined based on the V-PRISM architecture and its ALFA proof of concept, proposed by [42].

4. **Container orchestration definition:** The container orchestration will be done by using the docker API. Multimedia service chains are applications with different components, so at the time of their creation, there are some requirements, such as. The VDs are components of the service chain that are already created, so they are containers with an IP address and an identifier by which we can communicate. In the case of VMS, since they are not created, it must be taken into account that to communicate with a VMS, it is necessary to have its IP address; following this line, the multimedia service chain must be created from the final element backward. Additionally, the endpoints where the multimedia streams resulting from a multimedia service chain will be reproduced must have the necessary programs to capture and reproduce these streams.
5. **Resource management design:** We leave resource management for future work; these are presented in greater detail in Section 8.2.
6. **Validation and tests:** Different tests of different types of multimedia service chains were carried out. To carry out these tests, virtualized multimedia applications developed by [42] were used, and their deployment was initially carried out using docker through command lines in a terminal, later an interface was implemented within version 2.0 of ALFA. This interface allows to implement complex multimedia service chains intuitively. ALFA 2.0 is presented in more detail in Section 6.3.2.

## 4.3 L-PRISM Implementation

To carry out the implementation stage of our DSL, we use the compilation approach since we focus on the definition of a high-level language which will be translated into a low-level language in order to be compiled. To complete this stage, we had to comply with the three components defined by [30], which we detail below.



1. **Language structure aspect** To define the language structure of our DSL, we followed the model-based approach, which was defined based on the analysis and design described in Sections 4.1, and 4.2 respectively. As a result, we present the metamodel of our language summarized in Figure 4.2. We detail this metamodel in Chapter 5. As shown in Figure 4.2, we present a class diagram describing the attributes and relationships that are part of a multimedia service chain. In turn, we separate these elements by color and describe them below:

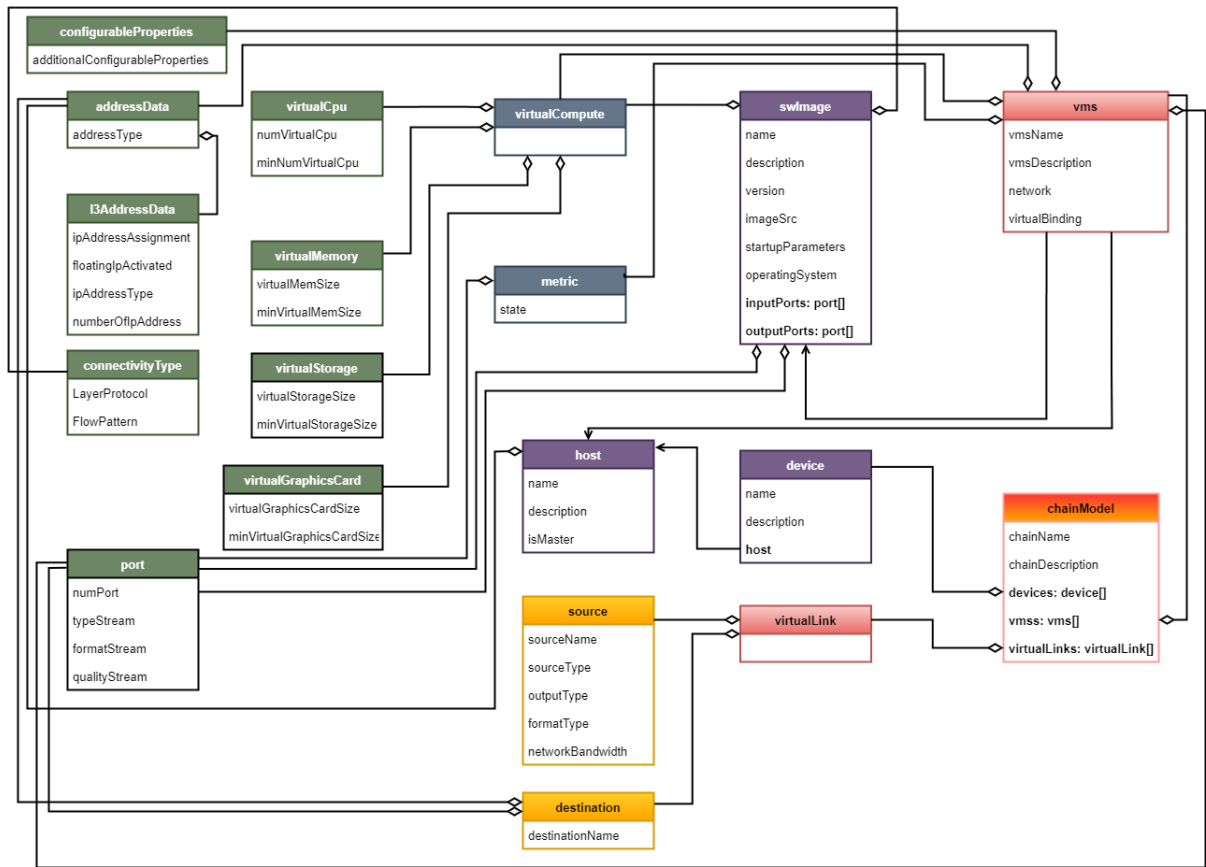


Figure 4.2: Class diagram of the L-PRISM metamodel

- Elements in green color are the data types; these will form part of the more complex structures of our metamodel. We represent this process as an aggregation function.
- Elements in lead color represent the types of capabilities that components of a multimedia service chain might have; for example, the *virtualCompute* element groups together the computational capabilities of a virtualized component.
- Elements in purple color represent the artifacts that can be used in a multimedia service chain; for example, based on our requirements, multimedia service

chains can be implemented in distributed environments, so their components could be implemented on different servers within a network (host), so when describing a VMS within a multimedia service chain, it must be defined in which node this VMS will be implemented.

- Elements in yellow color represent the necessary elements so that the different components of a multimedia service chain can communicate; for example, the *source* element presents the necessary attributes to define from where the sending of a multimedia stream will start, in the same way, the *destination* element helps to define the configuration of the destination of the media stream.
- Elements in pink color represent the elements to be created in a multimedia service chain; for our domain, we define two main elements, *vms* which represents a container, and *virtualLink* which represents the network configuration between two containers.
- Finally, the orange element is our main element, which describes a multimedia service chain, this element groups all the elements described above and is the element to test in future stages.

The metamodel elements of our DSL are presented in more detail in Chapter 5. Additionally, we present the formal definition and examples of how these elements would be used with the base language that will be discussed later.

## 2. Language editor aspects

To define the language editor of our DSL, we follow the projection editing approach, also called parser-less editor. Our language uses the YAML language as a base since it offers different advantages [39]:

- **Readability:** YAML is a data serialization format noted for its readability and ease of understanding by humans and machines. Its simple and structured syntax makes interpreting and writing YAML code easier than other configuration or data exchange formats.
- **Flexibility:** YAML is a flexible format that supports a wide variety of data structures, such as objects, lists, and dictionaries. This allows us to efficiently model and represent different aspects of our specific domain, adapting to the requirements defined in the previous stages.
- **Cross-platform support:** YAML is a widely recognized standard supported by various programming languages and platforms. This means that we can use

this language in our DSL. Additionally, using YAML will allow our DSL or parts of it to work in different environments and systems, facilitating interoperability and adoption by other developers.

- ***Integration with existing Tools:*** YAML has broad compatibility with various tools and frameworks, allowing you to take advantage of the utilities and libraries already available in the YAML ecosystem. This includes editors, validators, documentation generators, and more, which can improve the workflow for developers using our DSL.
- ***Fast learning curve:*** YAML is a simple and readable syntax that makes it easy for developers and users to quickly become familiar with our DSL. This can speed up the adoption process and facilitate collaboration on projects that reference our YAML-based DSL.

In general, the choice of YAML as the base language for our DSL is based on its qualities of readability, flexibility, cross-platform support, integration with existing tools, and fast learning curve. These features will make our DSL intuitive and easy to use, and in turn, YAML is tailored to our specific needs and makes our DSL widely accepted by developers.

### 3. Aspects of language semantics

We follow the translational semantics approach to define the language semantics of our DSL. Our language, as previously specified, is based on the YAML language and based on the V-PRISM architecture and its ALFA proof of concept proposed by [42], where the language to be compiled is JavaScript executed on a nodeJS server.

The solutions implemented with our DSL will be implemented in a file in YAML format, which will be sent as a nodeJS server (backend), which will be in charge of transforming our model to the JavaScript language and using the classic approach where an AST will be built in JavaScript while traversing our YAML file.

Aspects of the language semantics of our DSL are based on the architecture and proof of concept proposed by [42], for which ALFA was extended to a version that we call ALFA 2.0. This extension is presented in more detail in Chapter 6.

## 4.4 L-PRISM evaluation

The evaluation of our DSL is presented in detail in Chapter 7.

## 4.5 L-PRISM Maintenance

We leave the maintenance stage of our DSL for future work.

# Chapter 5

## L-PRISM Metamodel

This chapter presents in detail the L-PRISM metamodel. We describe the data types used throughout the description in Section 5.1 and their elements are presented in Figure 5.1 in dark green. The structures that represent the different capabilities that a component can have are described in Section 5.2 and their elements are presented in Figure 5.1 in lead color. The different artifacts that represent global elements that developers can use is described in Section 5.3 and their elements are presented in Figure 5.1 in purple color. The elements that are used in the connections of the different elements in a multimedia service chain are described in Section 5.4 and their elements are presented in Figure 5.1 in yellow color. The types of nodes that represent the virtualized components are described in Section 5.5 and their elements are presented in Figure 5.1 in pink color, and finally in Section 5.6 we present the main *chainModel* structure that describes a multimedia service chain based on virtual multimedia sensors and there is presented in Figure 5.1 in orange color.

As mentioned above, the language of our DSL is based on the TOSCA-NFV metamodel. For this reason, we use TOSCA-NFV as the primary reference for developing our metamodel.

The TOSCA-NFV metamodel has different elements used to implement its templates. TOSCA-NFV presents these elements in groups, as mentioned in Section 3.1, in the same way, our metamodel takes these elements as a base for our specification. Figure 5.1 presents the L-PRISM metamodel based on the TOSCA-NFV metamodel. The elements marked in light green are elements or attributes added to the TOSCA-NFV metamodel. These elements or attributes are described in greater detail in the subsequent sections.

It should be clarified that L-PRISM metamodel is not an extension of the TOSCA-

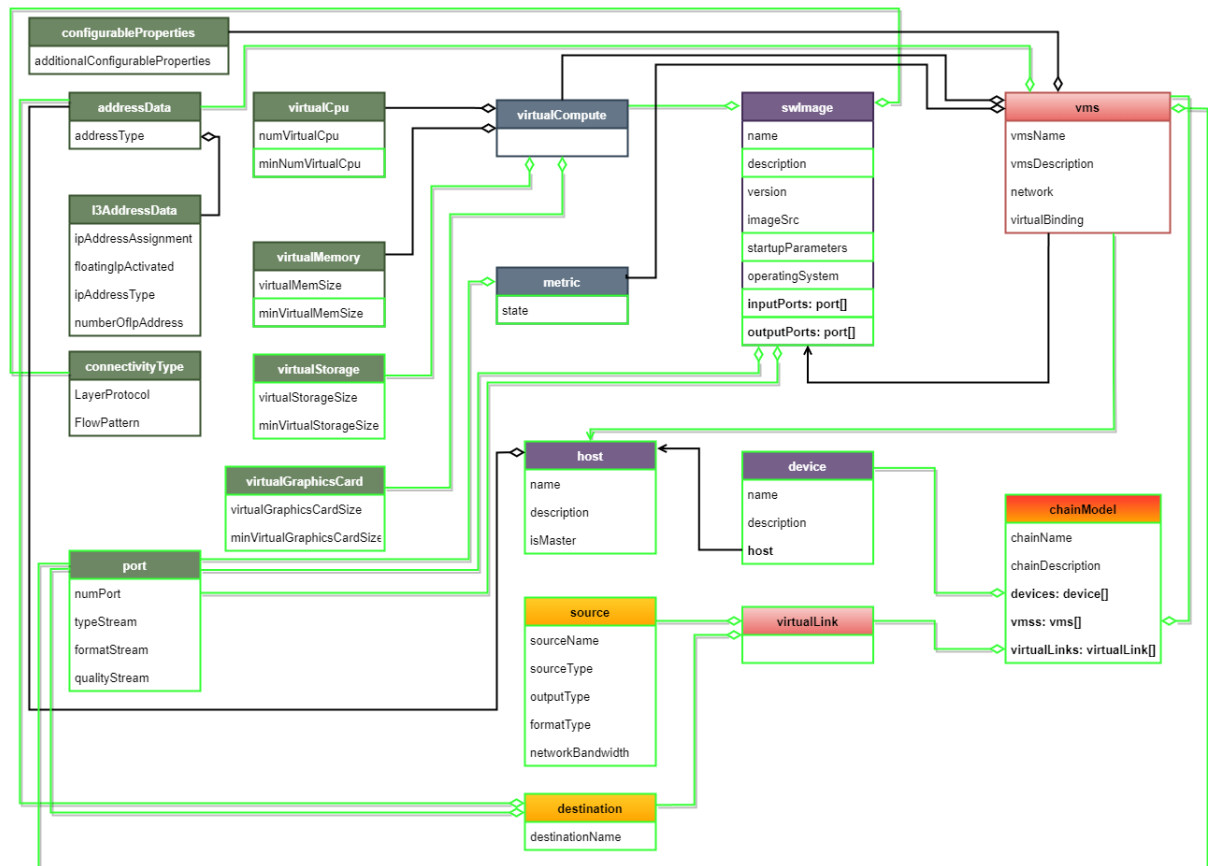


Figure 5.1: Class diagram of the L-PRISM metamodel based on TOSCA-NFV.

NFV metamodel since our work addresses concepts and technologies that TOSCA-NFV does not address.

## 5.1 Data Types

### 5.1.1 lPrism.datatype.vms.l3AddressData

The data type *l3AddressData* is defined by TOSCA-NFV [37], and ETSI-NFV [13]; its function is to inform about the configuration of the *IP* addresses within a subnet, information such as whether the *IP* addresses will be assigned manually or automatically by the orchestrator.

The *l3AddressData* data type is specified for network and subnet management, allowing developers to configure and manage *IP* addresses within the network. L-PRISM supports the *l3AddressData* data type. However, our PoC will only allow the configuration of two of its properties since both ALFA and ALFA 2.0 do not contemplate network management.

The properties of *l3AddressData* are described in detail in Table 5.1.

Table 5.1: Properties of *l3AddressData*

Name	Required	Type	Description
ipAddress-Assignment	Yes	Boolean	Specifies whether the <i>IP</i> address will be assigned manually by the user or automatically by the orchestrator [37]. The default value is True (Automatically assigned by the orchestrator)
floatingIpActivated	No	Boolean	
ipAddressType	Yes	String	Specifies the type of address; this type of address must be compatible with the properties of the architecture where the network will work [37]. Valid values: ipv4, ipv6
Continued on next page			

Table Continued from Previous Page

Name	Required	Type	Description
numberOfIp-Address	No	Integer	Minimum number of IP addresses to assign [37].

#### 5.1.1.1 Definition

Source Code 5.1 shows the formal definition of *l3AddressData* proposed in L-PRISM.

Source Code 5.1: Definition of *l3AddressData* in L-PRISM

```
lPrism.datatypes.vms.l3AddressData:
  properties:
    ipAddressAssignment:
      type: Boolean
      required: true
    floatingIpActivated:
      type: Boolean
      required: false
    ipAddressType:
      type: string
      required: false
      constraints:
        - valid_values: [ipv4, ipv6]
    numberOfIpAddress:
      type: Integer
      required: false
```

#### 5.1.1.2 Example

Source Code 5.2 shows an example of using the *l3AddressData* data type in L-PRISM.

Source Code 5.2: Example *l3AddressData* in L-PRISM

```
l3AddressData:
  ipAddressAssignment: true
  floatingIpActivated: false
  ipAddressType: ipv4
```



```
numberOfIpAddress: 1
```

### 5.1.2 lPrism.datatype.vms.addressData

It is a complex data type that is used by TOSCA-NFV [37] and ETSI-NFV [13]. This data type provides information about the address assigned to the connection point. The *addressData* is a structure with *addressType* and *l3AddressData* as attributes specified in Table 5.2.

Table 5.2: Properties of *addressData*

Name	Required	Type	Description
addressType	Yes	String	Describe the address that was assigned to the connection point. This address can be assigned by the orchestrator or by the user; this will depend on the state of the apAddressAssignment attribute of <i>l3AddressData</i> [37].
l3AddressData	No	<i>lPrism.data</i> – <i>type.vms.</i> – <i>l3Address</i> – <i>Data</i>	Provide information on the address assigned to the connection point.

#### 5.1.2.1 Definition

Source Code 5.3 shows the formal definition of *addressData* data type proposed in L-PRISM.

Source Code 5.3: Definition of *addressData* in L-PRISM

```
lPrism.datatypes.vms.addressData:
  properties:
    addressType:
      type: String
      required: true
    constraints:
```

```

- valid_values: [mac_address , ip_address ]
l3AddressData:
  type: lPrism.datatypes.vms.L3AddressData
  required: false

```

### 5.1.2.2 Example

Source Code 5.4 shows an example of using the *addressData* data type in L-PRISM.

Source Code 5.4: Example *addressData* in L-PRISM

```

addressData:
  addressType: IP address
  l3AddressData:
    ipAddressAssignment: true
    ipAddressType: IPv4

```

### 5.1.3 lPrism.datatype.vms.connectivityType

It is a complex data type that uses TOSCA-NFV [37] and ETSI-NFV [13], to describe the type of connection. The structure of *connectivityType* is composed of the attributes described in the table 5.3.

Table 5.3: Properties of *connectivityType*

Name	Required	Type	Description
layerProtocol	Yes	String	Identifies the connection protocol through which we can connect (ethernet, mpls, odu3, ipv4, ipv6, pseudo_Wire).
flowPattern	No	String	Identifies the connection mode (Line, Tree, Mesh)

#### 5.1.3.1 Definition

Source Code 5.5 shows the formal definition of *connectivityType* data type proposed in L-PRISM.

Source Code 5.5: Definition of *connectivityType* in L-PRISM

```
lPrism.datatypes.vms.connectivityType:
  properties:
    layerProtocol:
      type: string
      required: true
      constraints:
        - valid_values: [ ethernet , ipv4 , ipv6 ]
    flowPattern:
      type: string
      required: true
      constraints:
        - valid_values: [ Line , Tree , Mesh ]
```

Unlike TOSCA-VNF, which uses the *connectivityType* to create VirtualLinks, L-PRISM uses the *connectivityType* to create the virtual machine image (swImage - VMS), so that by creating a chain of multimedia services, end users can know the type of connection that will exist between the connection points.

### 5.1.3.2 Example

Source Code 5.6 shows an example of using the *connectivityType* data type in L-PRISM.

Source Code 5.6: Example *connectivityType* in L-PRISM

```
connectivityType:
  layerProtocol: ipv4
  flowPattern: Line
```

### 5.1.4 lPrism.datatype.vms.virtualCpu

It is a data type defined in TOSCA-NFV [37] that describes the properties of the CPU at each connection point. *virtualCpu* is a structure that is made up of different properties, which will allow the manipulation of CPU properties. The properties and the attributes are described in Table 5.4.

Table 5.4: Properties of *virtualCpu*

Name	Required	Type	Description
numVirtualCpu	No	Integer	It is a property that defines the number of processors assigned to the connection point (VMS); this property is of integer type that, by default, will have the same value as minNumVirtualCpu and that can be modified when creating the multimedia service chain.
minNumVirtualCpu	Yes	Integer	It is a property that will allow defining the minimum amount of CPU number that the connection point (VMS) will need; this property will have to be defined in the creation of the image (swImage).

The *minNumVirtualCpu* property is proposed by L-PRISM, as an additional property to the structure proposed by TOSCA-NFV.

#### 5.1.4.1 Definition

Source Code 5.7 shows the formal definition of *virtualCpu* data type proposed in L-PRISM.

Source Code 5.7: Definition of *virtualCpu* in L-PRISM

```
lPrism.datatypes.vms.virtualCpu:
  properties:
    numVirtualCpu:
      type: integer
      required: true
    minNumVirtualCpu:
      type: integer
      required: true
```

#### 5.1.4.2 Example

Source Code 5.8 shows an example of using the *virtualCpu* data type in L-PRISM.

Source Code 5.8: Example *virtualCpu* in L-PRISM

```
virtualCpu:
  numVirtualCpu: 2
  minNumVirtualCpu: 1
```

#### 5.1.5 lPrism.datatype.vms.virtualMemory

It is a data type used by TOSCA-NFV [37] to describe the properties of memory used by the virtual machine. The properties and the attributes are described in Table 5.5.

Table 5.5: Properties of *virtualMemory*

Name	Required	Type	Description
virtualMemSize	Yes	Integer	Defines the amount of memory in megabytes (MB) assigned to the connection point (VMS). This property is of integer type that, by default, will have the same value as minVirtualMemSize, and that can be modified when creating a multimedia service chain.
minVirtualMem-Size	Yes	Integer	This property will allow defining the minimum amount of memory that must be assigned to the connection point. This property must be defined when creating the image (swImage).

The *minVirtualMemSize* property is proposed by L-PRISM, as an additional property to the structure proposed by TOSCA.

### 5.1.5.1 Definition

Source Code 5.9 shows the formal definition of *virtualMemory* data type proposed in L-PRISM.

Source Code 5.9: Definition of *virtualMemory* in L-PRISM

```
lPrism.datatypes.vms.virtualMemory:
  properties:
    virtualMemSize:
      type: scalar-unit.size # Number
      required: true
    minVirtualMemSize:
      type: scalar-unit.size # Number
      required: true
```

### 5.1.5.2 Example

Source Code 5.10 shows an example of using the *virtualMemory* data type in L-PRISM.

Source Code 5.10: Example *virtualMemory* in L-PRISM

```
virtualMemory:
  virtualMemSize: 1024
  minVirtualMemSize: 512
```

## 5.1.6 lPrism.datatype.vms.virtualStorage

TOSCA-NFV does not describe a data type related to storage, but it does consider storage in the virtual machine deployment. In this specification, we define the data type *virtualStorage* since virtualized multimedia applications require sufficient storage space to store the multimedia data to be processed, the application data (code, files, etc.), and the associated temporary files.

The properties and the attributes of *virtualStorage* are described in Table 5.6.

Table 5.6: Properties of *virtualStorage*

Name	Required	Type	Description
virtualStorage-Size	No	Integer	Defines the amount of storage in megabytes (MB) assigned to the connection point (VMS). This property is of integer type that, by default, will have the same value as minVirtualVirtualStorageSize, and that can be modified when creating a multimedia service chain.
minVirtualStorageSize	No	Integer	This property will allow defining the minimum amount of storage that must be assigned to the connection point. This property must be defined when creating the image (swImage).

#### 5.1.6.1 Definition

Source Code 5.11 shows the formal definition of *virtualStorage* data type proposed in L-PRISM.

Source Code 5.11: Definition of *virtualStorage* in L-PRISM

```
lPrism.datatypes.vms.virtualStorage:
  properties:
    virtualStorageSize:
      type: scalar-unit.size # Number
      required: false
    minVirtualStorageSize:
      type: scalar-unit.size # Number
      required: false
```

#### 5.1.6.2 Example

Source Code 5.12 shows an example of using the *virtualStorage* data type in L-PRISM.

Source Code 5.12: Example *virtualStorage* in L-PRISM

```

virtualStorage:
    virtualStorageSize: 10000
    minVirtualStorageSize: 5000

```

### 5.1.7 lPrism.datatype.vms.virtualGraphicsCard

TOSCA-NFV does not describe a data type related to graphics card. In this specification, we define the data type *virtualGraphicsCard* since Some virtualized multimedia applications require a powerful graphics card to handle the graphical demands of the application and ensure the application QoS.

The properties and the attributes of *virtualGraphicsCard* are described in Table 5.7.

Table 5.7: Properties of *virtualGraphicsCard*

Name	Required	Type	Description
virtualGraphics-CardSize	No	Integer	Defines the amount of memory of the graphics card in megabytes (MB) assigned to the connection point (VMS). This property is of integer type that, by default, will have the same value as minVirtualGraphicsCardSize, and that can be modified when creating a multimedia service chain.
minVirtual-GraphicsCard-Size	No	Integer	This property will allow defining the minimum amount of graphic memory (VRAM) that must be assigned to the connection point. This property must be defined when creating the image (swImage).

#### 5.1.7.1 Definition

Source Code 5.13 shows the formal definition of *virtualGraphicsCard* data type proposed in L-PRISM.



Source Code 5.13: Definition of *virtualGraphicsCard* in L-PRISM

```
lPrism.datatypes.vms.virtualGraphicsCard:
  properties:
    virtualGraphicsCardSize:
      type: scalar-unit.size # Number
      required: false
    minVirtualGraphicsCardSize:
      type: scalar-unit.size # Number
      required: false
```

### 5.1.7.2 Examples

Source Code 5.14 shows an example of using the *virtualGraphicsCard* data type in L-PRISM.

Source Code 5.14: Example *virtualGraphicsCard* in L-PRISM

```
virtualGraphicsCard:
  virtualGraphicsCardSize: 1024
  minVirtualGraphicsCardSize: 512
```

### 5.1.8 lPrism.datatype.vms.configurableProperties

It is a data type that allows to describe the configurable properties of an element. In the case of L-PRISM, the virtual device (VD) or VMS can have properties that can be configured before its initialization.

The properties and the attributes of *configurableProperties* are described in Table 5.8.

Table 5.8: Properties of *configurableProperties*

Name	Required	Type	Description
additionalConfigurableProperties	Yes	String	Defines a variable that can be set. This data type is commonly used in the description of the software image.

### 5.1.8.1 Definition

Source Code 5.15 shows the formal definition of *configurableProperties* data type proposed in L-PRISM.

Source Code 5.15: Definition of *configurableProperties* in L-PRISM

```
lPrism.datatypes.vms.configurableProperties:
    properties:
        additionalConfigurableProperties:
            type: String
            required: true
```

### 5.1.8.2 Examples

Source Code 5.16 shows an example of using the *configurableProperties* data type in L-PRISM.

Source Code 5.16: Example *configurableProperties* in L-PRISM

```
configurableProperties:
    additionalConfigurableProperties: "13"
```

## 5.1.9 lPrism.datatype.vms.port

The *port* data type describes an input or output port enabled on a VMS. In L-PRISM, this port has characteristics oriented to the treatment of multimedia streams. Therefore, in addition to the port number, the properties of the type, format, and quality of the multimedia stream that will enter through the specified port are added.

The properties and the attributes of *port* are described in Table 5.9.

Table 5.9: Properties of *port*

Name	Required	Type	Description
numPort	Yes	integer	Port number has enable.
Continued on next page			

Table Continued from Previous Page

Name	Required	Type	Description
typeStream	Yes	String	Multimedia stream type to use this port. This attribute can receive four values (video, audio, image, or text)
formatStream	No	list	Multimedia stream formats to be used by this port.
qualityStream	No	list	Multimedia stream formats to be used by this port.

#### 5.1.9.1 Definition

Source Code 5.17 shows the formal definition of *port* data type proposed in L-PRISM.

Source Code 5.17: Definition of *port* in L-PRISM

```
lPrism.datatypes.vms.port:
  properties:
    numPort:
      type: String
      required: true
    typeStream:
      type: String
      required: true
      constraints:
        -valid_values: [video, audio, image, text]
    formatStream:
      type: list
      entrySchema:
        type: String
      required: false
    qualityStream:
      type: list
      entrySchema:
        type: String
      required: false
```

### 5.1.9.2 Example

Source Code 5.18 shows an example of using the *port* data type in L-PRISM.

Source Code 5.18: Example *port* in L-PRISM

```
port:
  numPort: 5000
  typeStream: video
  formatStream: #not define
  qualityStream: #not define
```

## 5.2 Capabilities Types

### 5.2.1 lPrism.capabilities.vms.metric

Metric is a type of particular capacity that is used to monitor a virtualized element. This capacity is significant in case a VMS sends reports on its operation. The capacity Metric was modified to fit the V-PRISM architecture, but it still works as defined by TOSCA-NFV [37].

The properties and the attributes of *metric* are described in Table 5.10.

Table 5.10: Properties of *metric*

Name	Required	Type	Description
state	Yes	Boolean	—
listPorts	No	list	—.

#### 5.2.1.1 Definition

Source Code 5.19 shows the formal definition of *metric* capabilities type proposed in L-PRISM.

Source Code 5.19: Definition of *metric* in L-PRISM

```
lPrism.capabilities.vms.metric:
  properties:
    state:
```

```

    type: Boolean
    required: true
  listPorts:
    type: list
    entrySchema:
      type: lPrism.datatypes.vms.port
      required: false

```

### 5.2.1.2 Example

Source Code 5.20 shows an example of using the *metric* capabilities type in L-PRISM.

Source Code 5.20: Example *host* in L-PRISM

```

metric:
  state: True
  listPorts:
  —
    numPort: 15001
    typeStream: text

```

## 5.2.2 lPrism.capabilities.vms.virtualCompute

*virtualCompute* encompasses the computational resources that can be assigned to a virtualized component. These resources can be configured in the VMS or preconfigured in the software image registry.

The properties and the attributes of *virtualCompute* are described in Table 5.11.

Table 5.11: Properties of *virtualCompute*

Name	Required	Type	Description
VirtualMemory	Yes	lPrism.data-type.vms.virtualMemory	Describes the minimum amount of RAM required by software images and the amount allocated to the virtualized component (VMS, VD).
Continued on next page			

Table Continued from Previous Page

Name	Required	Type	Description
VirtualCpu	Yes	lPrism.data-type.vms.virtualCpu	Describes the minimum number of CPUs required by software images and the number allocated to the virtualized component (VMS, VD).
VirtualStorage	Yes	lPrism.data-type.vms.virtualStorage	Describes the minimum amount of storage required by software images and the amount allocated to the virtualized component (VMS, VD).
VirtualGraphics-Card	No	lPrism.data-type.vms.virtualGraphics-Card	Describes the minimum amount of graphic memory required (if required) by the software image and the amount allocated to the virtualized component.

### 5.2.2.1 Definition

Source Code 5.21 shows the formal definition of *virtualCompute* capabilities type proposed in L-PRISM.

Source Code 5.21: Definition of *virtualCompute* in L-PRISM

```

lPrism.capabilities.vms.virtualCompute:
  properties:
    VirtualMemory:
      type: lPrism.datatype.vms.virtualMemory
      required: true
    VirtualCpu:
      type: lPrism.data-type.vms.virtualCpu
      required: true
    VirtualStorage:
      type: lPrism.datatype.vms.virtualStorage
      required: true
    VirtualGraphicsCard:
      type: lPrism.datatype.vms.virtualGraphicsCard
      required: false

```

### 5.2.2.2 Example

Source Code 5.22 shows an example of using the *virtualCompute* capabilities type in L-PRISM.

Source Code 5.22: Example *virtualCompute* in L-PRISM

```
virtualCompute:
  VirtualMemory:
    virtualMemSize: 1024
    minVirtualMemSize: 512
  VirtualCpu:
    numVirtualCpu: 2
    minNumVirtualCpu: 1
  VirtualStorage:
    virtualStorageSize: 10000
    minVirtualStorageSize: 5000
  VirtualGraphicsCard:
    virtualGraphicsCardSize: 1024
    minVirtualGraphicsCardSize: 512
```

## 5.3 Artifact Types

The artifacts in L-PRISM are considered tools that are available to developers. Within these tools, we have the software images *swImage*, the host *host* where different virtualized components can be mounted, and the virtualized devices *vd*. The latter becomes a virtualized component but does not work only for a developer. *VDs* are available to all developers.

### 5.3.1 lPrism.artifacts.vms.swImage

It is an artifact defined by TOSCA-NFV [37] that presents a data model that describes a system image that will be used to virtualize multimedia sensors VMS.

It is used to register software images developed by the community. This description does not display any component. It is a registry that will provide the necessary information to whoever wishes to use this software image.

*swImage* uses a data structure that is described in detail in Table 5.12.

Table 5.12: Properties of *swImage*

Name	Required	Type	Description
name	Yes	String	Name of image
description	No	String	Description of image
version	Yes	String	Version of image
ImageSrc	Yes	String	A reference to where the image is registered within the files, each image has an exact file address where it was created.
startupParameters	No	lPrism.data-type.vms.-configurable-Properties	They are the initialization variables required by the image. This variable is defined at the discretion of the software image author and information server for the developer who uses this image.
operatingSystem	Yes	String	Operating system used by the software image.
virtualCompute	Yes	lPrism.capabilities.vms-.virtual-Compute	Describes the minimum computational properties that must be assigned to the software image. As mentioned in this study, computing capabilities may vary according to the multimedia stream type, format, and quality. However, a minimum capacity of computing resources is also required for its operation.
Continued on next page			



Table Continued from Previous Page

Name	Required	Type	Description
inputPorts	No	list	<p>It offers information about which ports can receive multimedia streams; additionally, it offers information about what type of stream is accepted by the port and the formats and qualities that the software image can process.</p> <p>In case the author of the software image leaves this variable null, it will have to be configurable in startupParameters. We recommend placing this property so that developers using this software image can understand how the image works.</p>
outputPorts	No	list	<p>It offers information about which ports are enabled to send information flows. This variable is available if the image offers data streams specifically for a port; this is commonly used when the software image offers information about the performance metrics of the software image put into production. This information is helpful for developers using this software image.</p>
connectivityType	Yes	lPrism.datatypes.vms-connectivity-Type	Describes the behavior of the software image in terms of the type of output connection it will have.
End of Table <i>swImage</i>			

### 5.3.1.1 Definition

Source Code 5.23 shows the formal definition of *SwImage* artifact proposed in L-PRISM.

Source Code 5.23: Definition of *SwImage* in L-PRISM

```
lPrism . artifacts . vms . SwImage:
  derived_from: lPrism . artifacts . Deployment . Image
  properties:
    name:
      type: String
      required: true
    description:
      type: String
      required: false
    version:
      type: String
      required: true
    operatingSystem:
      type: String
      required: false
    ImageSrc:
      type: String
      required: true
    startupParameters:
      type: map
      entrySchema:
        type: .. datatype . vms . configurableProperties
        required: false
    virtualCompute:
      type: .. capabilities . vms . virtualCompute
      required: true
    inputPorts:
      type: list
      entrySchema:
        type: .. datatypes . vms . port
        required: false
```

```

outputPorts:
  type: list
  entrySchema:
    type: .. datatypes.vms.port
  required: false
connectivityType:
  type: .. datatypes.vms.connectivityType
  required: true

```

### 5.3.1.2 Example

Source Code 5.24 shows an example of using the *SwImage* artifact in L-PRISM.

Source Code 5.24: Definition of *swImage* in L-PRISM

```

lPrism.artifacts.vms.SwImage:
  name: Example register software image
  description: Description of software image
  version: 1.0
  operatingSystem: linux
  ImageSrc: device/blackAndWhite
  startupParameters: Null
  virtualCompute:
    VirtualMemory:
      virtualMemSize: Null
      minVirtualMemSize: 512
    VirtualCpu:
      numVirtualCpu: Null
      minNumVirtualCpu: 1
    VirtualStorage:
      virtualStorageSize: Null
      minVirtualStorageSize: 5000
    VirtualGraphicsCard: Null
  inputPorts:
    —
    numPort: 5000
    typeStream: video

```

```

outputPorts: Null
connectivityType:
    layerProtocol: ipv4
    flowPatter: Line

```

### 5.3.2 lPrism.artifacts.vms.host

A *host* can be characterized as an entity intended to provide computing resources to applications implemented within them. TOSCA-NFV does not define or specify this element in detail. In L-PRISM, a host works as an element at the service of the developers, which means that it is within the category of artifacts. According to [42], a *host* can be defined as a small to medium-sized computing entity that aims to provide computing, storage, and network resources to the components deployed within this host.

The architecture on which L-PRISM is based is V-PRISM, and in this, it is defined that a *host* is controlled by a *masterhost*, which manages the available hosts within the network.

The structure of the *host* artifact is detailed in Table 5.13.

Table 5.13: Properties of *host*

Name	Required	Type	Description
name	Yes	String	Name of host
description	No	String	Description of host
addressIp	Yes	lPrism.data-type.vms.-addressData	Describes the host's IP address, <i>addressIp</i> serves as the host identifier, so developers can easily describe where their peers will be created.
isMaster	Yes	Bool	Defines if the node that is being registered will be master or not.

#### 5.3.2.1 Definition

Source Code 5.25 shows the formal definition of *host* artifact proposed in L-PRISM.

Source Code 5.25: Definition of *host* in L-PRISM

```
lPrism.artifacts.vms.host:
  properties:
    name:
      type: String
      required: true
    description:
      type: String
      required: false
    addressIp:
      type: lPrism.datatype.vms.addressData
      required: true
    isMaster:
      type: Bool
      required: true
```

### 5.3.2.2 Example

Source Code 5.26 shows an example of using the *host* artifact in L-PRISM.

Source Code 5.26: Example *host* in L-PRISM

```
lPrism.artifacts.vms.host:
  name: host master Midiacom
  description: This is host master in Midiacom
  addressIp:
    addressType: 192.168.0.101
  isMaster: true
```

### 5.3.3 lPrism.artifacts.vms.device

To fully understand virtual devices (*device*), whose attributes are presented in Table 5.14, it is necessary to recall one of the main reasons why they were proposed to virtualize multimedia sensors. One of the main advantages of virtualized physical sensors is that a media stream can be replicated multiple times and sent to different destinations, unlike the physical sensor, which can only send its stream to one destination. L-PRISM treats this element as if it were already created and working, so it would only require an identifier

to subscribe to this element in order for it to replicate and send its media stream to the new subscriber.

Table 5.14: Properties of *device*

Name	Required	Type	Description
name	Yes	String	name that was given to the virtualized physical device ( <i>device</i> ). This name only serves as a guide for the user.
description	No	String	describes the virtualized sensor. This field is for users to understand the <i>device</i> better.
host	Yes	String	Specifies on which of the hosts within the network the device will work. Hosts are identified by their address (IP, MAC), which must have been previously registered using of <i>lPrism.artifacts.vms.host</i>

### 5.3.3.1 Definition

Source Code 5.27 shows the formal definition of *device* artifact proposed in L-PRISM.

Source Code 5.27: Definition of *device* in L-PRISM

```
lPrism.artifacts.vms.device:
  properties:
    name:
      type: String
      required: true
    description:
      type: String
      required: false
    host:
      address:
        type: lPrism.artifacts.vms.host
        required: true
```

### 5.3.3.2 Example

Source Code 5.28 shows an example of using the *device* artifact in L-PRISM.

Source Code 5.28: Example *device* in L-PRISM

```
lPrism.artifacts.vms.host:
  name: Device color to bw
  description: Device transform video color
  host: 192.168.1.101
```

## 5.4 Relationship Types

*Relationship types* presents the different elements used to model how the different components of a multimedia service chain relate to each other. These elements are defined by specifying properties and attributes.

L-PRISM presents two elements in *relationship types*, *source* that defines the properties where the sending of the multimedia stream will start, and *destination* that defines the properties where the multimedia stream will be received.

### 5.4.1 lPrism.relationship.vms.source

The structure of the *source* relationship type is detailed in Table 5.15.

Table 5.15: Properties of *source*

Name	Required	Type	Description
sourceName	Yes	String	describes from which component of the service chain the transmission of the media stream is initiated. This data must be identical to that configured either in a deviceName (Section 5.6.1) or a vmsName (Section 5.5.1).
Continued on next page			

Table Continued from Previous Page

Name	Required	Type	Description
sourceType	Yes	String	describes the type of component that will initiate the transfer of the multimedia stream; this data type has conditions, for now only (vms device).
outputType	Yes	String	describes the type of media stream that will be sent. Multimedia applications typically process media streams of one type but do not always produce the output of the same input stream type.
formatType	No	String	describes the format of a media stream type. The format of a multimedia stream influences the application behavior. An example is that it is different to process a 360px video than a 4K video, most likely that more network bandwidth is needed for the second case.
networkBandwidth	No	lPrism.data-type.vms.-networkBandwidth	defines the bandwidth that will be assigned to this connection source. The connection between virtualized components in a host does not need to configure this attribute. Still, for connections between different network hosts, it is necessary to consider defining this value.

#### 5.4.1.1 Definition

Source Code 5.29 shows the formal definition of *source* relationship type proposed in L-PRISM.

Source Code 5.29: Definition of *source* in L-PRISM

```
lPrism.relationship.vms.source:
```



```

properties:
  sourceName:
    type: String
    required: true
  sourceType:
    type: String
    required: true
  outputType:
    type: String
    required: true
  formatType:
    type: String
    required: false
  networkBandwidth:
    type: lPrism.datatype.vms.networkBandwidth
    required: false

```

#### 5.4.1.2 Example

Source Code 5.30 shows an example of using the *source* relationship type in L-PRISM.

Source Code 5.30: Example *source* in L-PRISM

```

lPrism.relationship.vms.source:
  sourceName: VD1
  sourceType: device
  outputType: video
  formatType: Null
  networkBandwidth: Null

```

## 5.4.2 lPrism.relationship.vms.destination

The structure of the *destination* relationship type is detailed in Table 5.16.

Table 5.16: Properties of *destination*

Name	Required	Type	Description
destinationName	Yes	String	describes which component of the service chain will receive the media stream. This data must be identical to the one configured in a <i>vmsName</i> of one of the <i>vms</i> (Section 5.5.1) used in the same multimedia service chain.
destinationIp	No	lPrism.data-type.vms.-addressData	complex data type initially defined by TOSCA-NFV and adapted in L-PRISM, which describes a network address (IP, MAC).
destinationPort	Yes	lPrism.data-type.vms.-port	describes a UDP port and type of multimedia stream that the destination of the connection accepts.

#### 5.4.2.1 Definition

Source Code 5.31 shows the formal definition of *destination* relationship type proposed in L-PRISM.

Source Code 5.31: Definition of *destination* in L-PRISM

```
lPrism.relationship.vms.destination:
  properties:
    destinationName:
      type: String
      required: true
    destinationIp:
      type: String
      required: false
    destinationPort:
      type: lPrism.datatype.vms.port
      required: true
```

### 5.4.2.2 Example

Source Code 5.32 shows an example of using the *destination* relationship type in L-PRISM.

Source Code 5.32: Example *destination* in L-PRISM

```
lPrism.relationship.vms.destination:
    destinationName: VMS1
    destinationIp: Null
    destinationPort:
        numPort: 5000
        typeStream: video
```

## 5.5 Node Types

### 5.5.1 lPrism.nodes.vms.VDU.vms

A *vms* within a multimedia service chain is used for processing one or several multimedia streams and generate one or more output results. The result of a *vms* can be directed to one or more destination VMS(s), or can be provided for an end user application. Characteristics such as type, format, and quality of a multimedia stream make multimedia applications stand out. These characteristics influence the amount of computational resources that a multimedia application will need for its proper functioning, resources such as CPU, RAM, storage, and in some cases, the graphics card will vary according to the characteristics of the multimedia flow that a multimedia application will process. In the case of virtualized multimedia applications, another resource that must be considered is network bandwidth since, in many cases, multimedia streams tend to occupy a high bandwidth.

The structure of the *vms* node type is detailed in Table 5.17.

Table 5.17: Properties of *vms*

Name	Required	Type	Description
vmsName	Yes	String	The VMS name assigned by the developer of the multimedia service chain. This data type works as an <i>ID</i> in a <i>virtualLink</i> . The name of a <i>vms</i> must be unique in a multimedia service chain.
vmsDescription	No	String	The description of the VMS in a multimedia service chain.
configurable-Properties	No	IPrism.data-type.vms-configurable-Properties	Describes the properties that can be set on this component. It will depend directly if the selected image has configurable properties.
address	No	IPrism.data-type.vms-addressData	Describes the address of the virtualized component ( <i>IP, MAC</i> ). In L-PRISM, we do not consider the address configuration in the description, since network management would be a complex task when working with lightweight virtualization technology. We leave this description so that it can be used in future work.  This variable is not considered in the description, but in the orchestration, each virtualized component has an address ( <i>IP, MAC</i> ) assigned by the orchestrator.
Continued on next page			

Table Continued from Previous Page

Name	Required	Type	Description
network	No	String	Describes the network to which the virtualized component will belong. L-PRISM does not consider this data in the description since this component will be virtualized in a network pre-configured by the host to which it is assigned.
ports	No	lPrism.data-type.vms-ports	<p>Describes the ports that will be made available by the virtualized component. L-PRISM does not consider this data in the description since this information comes from the image (<i>swImage</i>), which can be configured in <i>configurableProperties</i> or, failing that, they will already be static variables that the author of the image will make available.</p> <p>This variable is not considered in the description, but in the orchestration, the orchestrator will enable the pre-configured gates.</p>
virtualCompute	No	lPrism.capabilities.vms-virtualCompute	<p>Describes the storage, memory, processing, and graphics card requirements that will be allocated to the virtualized component.</p> <p>If this description is not included, the default description defined when creating the image for this component will be used.</p>
Continued on next page			

Table Continued from Previous Page

Name	Required	Type	Description
monitorignParameter	No	lPrism.capabilities.vms.-metric	It describes whether the component to be virtualized offers data to be able to monitor its operation.  The monitoring parameters can be memory consumption, CPU usage, bandwidth consumption, activity time, latency, etc.
virtualBinding	No	lPrism.capabilities.vms.-virtualBindable	Describes whether the virtualized component functions as a subscription-based service, similar to an API. L-PRISM uses this property for the VDs since the operation of these will depend on the VMS subscribed to the VD. This capability is mandatory on VDs and can be used by VMSs that are implemented to act as services.
vmsType	Yes	lPrism.artifacts.vms.-swImage	Describes the software image that will be used; this image brings different properties.
host	Yes	lPrism.artifacts.vms.-host	Describes the host where the container will be created. L-PRISM considers that within a cloud/edge platform, there may be different hosts with different capacities; within each host, different virtualized components (containers) may be hosted.

#### 5.5.1.1 Definition

Source Code 5.33 shows the formal definition of *vms* node type proposed in L-PRISM.

Source Code 5.33: Definition of *vms* in L-PRISM

```
lPrism.nodes.vms.VDU.vms:
```

```
properties:
  vmsName:
    type: String
    required: true
  vmsDescription:
    type: String
    required: false
  configurableProperties:
    type: map
    entrySchema:
      type: .. datatype.vms.configurableProperties
    required: false
attributes:
  address:
    type: lPrism.datatype.vms.addressData
    required: false
  network:
    type: String
    required: false
  ports:
    type: map
    entrysSchema:
      type: lPrism.datatype.vms.ports
    required: false
requirements:
  virtualCompute:
    type: lPrism.capabilities.vms.virtualCompute
    required: false
capabilities:
  monitorignParameter:
    type: lPrism.capabilities.vms.metric
    required: false
  virtualBinding:
    type: lPrism.capabilities.vms.virtualBindable
    required: false
```

```
artifacts:
  vmsType:
    file:
    type: Prism.artifacts.vms.swImage
    required: true
  host:
    address:
    type: lPrism.artifacts.vms.host
    required: true
```

### 5.5.1.2 Example

Source Code 5.34 shows an example of using the *vms* node type in L-PRISM.

Source Code 5.34: Example *vms* in L-PRISM

```
vms:
  vmsName: My VMS
  vmsDescription: Description my VMS
  configurableProperties: #not define
  address: #not define
  network: #not define
  ports: #not define
  virtualCompute:
    virtualMemory:
      virtualMemSize: 1024
      minVirtualMemSize: 512
    virtualCpu:
      numVirtualCpu: 2
      minNumVirtualCpu: 1
    virtualStorage:
      virtualStorageSize: 10000
      minVirtualStorageSize: 5000
    virtualGraphicsCard: Null
  monitorignParameter: #not define
  virtualBinding: #not define
  vmsType: 638e70b10a1fbd0026fccae8
```



```
host: 192.168.0.117
```

### 5.5.2 lPrism.nodes.vms.virtualLink

The interconnections between the elements of a multimedia service chain are described by *virtualLink*. This element provides information on two other components, the source point (*source*) where the stream is sent from and the destination point (*destination*) where the stream will be received.

To better describe this element, Section 5.4.1 presents the *source* element attributes examples, and Section 5.4.2 presents the *destination* element attributes and examples.

The structure of the *virtualLink* node type is detailed in Table 5.18.

Table 5.18: Properties of *virtualLink*

Name	Required	Type	Description
source	Yes	lPrism.relationship.vms.source	Describes the point from which the multimedia stream is being sent.
destination	Yes	lPrism.relationship.vms.destination	Describes the destination point, which will receive the multimedia stream.

#### 5.5.2.1 Definition

Source Code 5.35 shows the formal definition of *virtualLink* node type proposed in L-PRISM.

Source Code 5.35: Definition of *virtualLink* in L-PRISM

```
lPrism.nodes.vms.virtualLink:
  properties:
    source:
      type: lPrism.relationship.vms.source
      required: true
    destination:
      type: lPrism.relationship.vms.destination
```

<b>required: true</b>
-----------------------

### 5.5.2.2 Example

Source Code 5.36 shows an example of using the *virtualLink* node type in L-PRISM.

Source Code 5.36: Example *virtualLink* in L-PRISM

```

virtualLink:
  source:
    sourceName: VD1
    sourceType: device
    outputType: video
    formatType: #not define
    networkBandwidth: #not define
  destination:
    destinationName: VMS1
    destinationIP: #not define
    destinationPort:
      numPort: 5000
      typeStream: video

```

## 5.6 Chain Type

### 5.6.1 lPrism.chain.vms.chainModel

L-PRISM represents a multimedia service chain as a language element named *chainModel*. A *chainModel* has three main components: Virtual Device (*devices*), Virtual Multimedia Sensors (*vmss*), and their connections (*virtualLinks*).

The structure of the *chainModel* is detailed in Table 5.19.

Table 5.19: Properties of *chainModel*

Name	Required	Type	Description
chainName	Yes	String	the name of the multimedia service chain.
chainDescription	No	String	the description of a multimedia service chain.
devices	Yes	list	list of virtual devices that compose the multimedia service chain.
vmss	Yes	list	list of VMSs that compose the multimedia service chain
virtualLinks	Yes	list	list of virtual connections between virtual devices and VMSs.

#### 5.6.1.1 Definition

Source Code 5.37 shows the formal definition of *chainModel* proposed in L-PRISM.

Source Code 5.37: Definition of *chainModel* in L-PRISM

```

lPrism.chain.vms.chainModel:
  properties:
    chainName:
      type: String
      required: true
    chainDescription:
      type: String
      required: false
    devices:
      type: list
      entrySchema:
        deviceName:
          type: String
          required: true
        deviceId:
          index:
            type: lPrism.artifacts.vms.device

```

```

        required: true
    required: true
    vmss:
        type: list
        entrySchema:
            type: lPrism.nodes.vms.VDU.vms
            required: true
    virtualLinks:
        type: list
        entrySchema:
            type: lPrism.nodes.vms.virtualLink
            required: true

```

### 5.6.1.2 Example

Source Code 5.38 shows an example of using the *chainModel* in L-PRISM.

Source Code 5.38: Example *chainModel* in L-PRISM

```

chainModel:
    chainName: VMS Chain
    chainDescription: Video Flux
    devices:
    —
        deviceName: example1 video ball
        deviceId: 638e70b10a1fbd0026fccaf4
    vmss:
    —
        vmsName: vms1 example 1
        vmsDescription: My VMS description
        vmsType: 638e70b10a1fbd0026fccae4
        startupParameters: # Null
        host: 192.168.0.117 # IP
        virtualCompute:
            virtualMemory:
                virtualMemorySize: 1024
            virtualCPU:

```

```

        numVirtualCpu: 2
    virtualStorage:
        virtualStorageSize: 10000
virtualLinks:
- #virtualLink 1
    source:
        sourceName: example1 video ball
        sourceType: device
        outputType: video
        formatType: # not define
        networkBandwidth: # not define
    destination:
        destinationName: vms1 example 1
        destinationIp: #define for orchestrator
        destinationPort:
            numPort: 5000
            typeStream: video
- #virtualLink 2
    source:
        sourceName: vms1 example 1
        sourceType: vms
        outputType: video
        formatType: # not define
        networkBandwidth: # not define
    destination:
        destinationName: 192.168.0.101 #My PC
        destinationIp: 192.168.0.101 #My IP
        destinationPort:
            numPort: 20002 # my port
            typeStream: video

```

Figure 5.2 shows the example of the multimedia service chain implemented with L-PRISM. This chain receives a multimedia stream from the device (*video ball*); for this example, we assume that this device has as its identifier *deviceId: 638e70b10a1fbd0026fccaf4*, the video stream sent by (*video ball*) is processed by a multimedia application (*VMS1*) that has the function of transporting UDP-UDP data (it only retransmits the multimedia

stream). Finally, *VMS1* transmits the media stream to address *192.168.0.101* on port 20002. The detailed implementation of this example can be found on the website<sup>1</sup>.

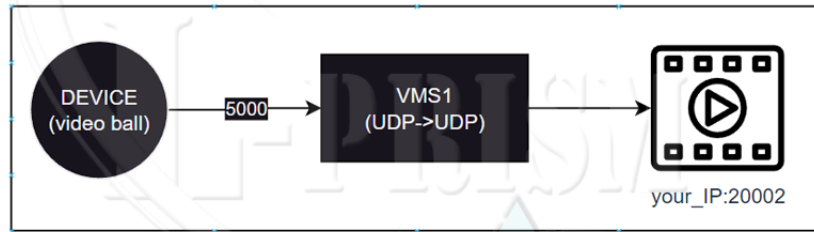


Figure 5.2: Simple multimedia service chain with L-PRISM

## 5.7 Example of the L-PRISM Metamodel

This Section presents an example of specifying a multimedia service chain based on VMS using our metamodel.

For this example, we use a complex multimedia service chain, which we see on the left in Figure 5.3. In this multimedia service chain, two multimedia streams from two different Virtual Devices (VD) are processed by different Virtual Multimedia Sensors (VMS). The result of this multimedia service chain is seen on the right-hand side of Figure 5.3.

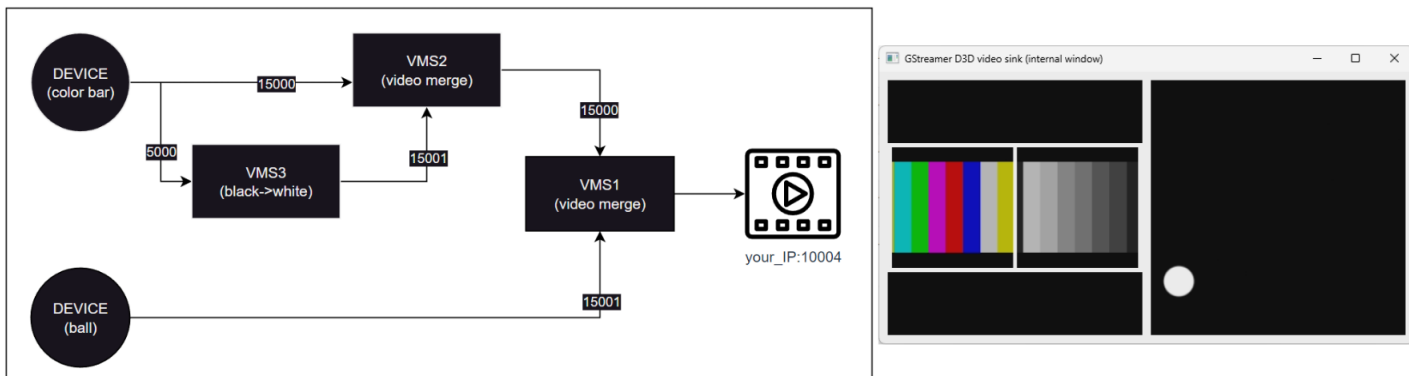


Figure 5.3: Diagram and output of a complex multimedia service chain.

<sup>1</sup><https://fventuraq.github.io/taks1-details.html>

For a better understanding, we separate this description into three parts, the description of the virtual devices (VD), the description of the virtual multimedia sensors (VMS), and finally, the description of the virtual link (VL).

### 1. Description of the virtual devices (VD)

Figure 5.4 presents how the specification of a multimedia service chain is started using our metamodel. As mentioned above, our metamodel uses YAML as the base language, and line 1 is how all YAML files should be initialized. Then in line 2, we call our main element *chainModel*, which within its structure, will store the essential elements of a multimedia service chain; VD, VMS, and VL. Additionally, *chainModel* requests a name and description for the multimedia service chain, as shown in lines 3 and 4.



Figure 5.4: Specification of virtual devices with the metamodel of L-PRISM

As the first main element of a multimedia service chain, we present the list of VD described from lines 6 to 12. *devices* store a list of VD. We use two VDs in this example.

The first VD we call *colorBar*, as shown in line 8, and in line 9, we place the VD identifier (*deviceId*) to give us the architecture where this VD is working. Let us remember that VDs are considered artifacts already working within the architecture, and what we do is request the multimedia stream from a VD.

In the same way, the second VD is specified from lines 10 to 12. We call this VD *Ball*, and we place its identifier, which, if we look closely, is different from the previous VD.

We clarify that lines 7 and 10 are the way YAML represents an element inside a list and remember that *devices* stores elements of type VD.

## 2. Description of the virtual multimedia sensors (VMS)

In Figure 5.5, we present the specification of the list of VMS used in this example. In line 14, *vmss* is a list of VMS. In this example, we specify that we will create three VMS.

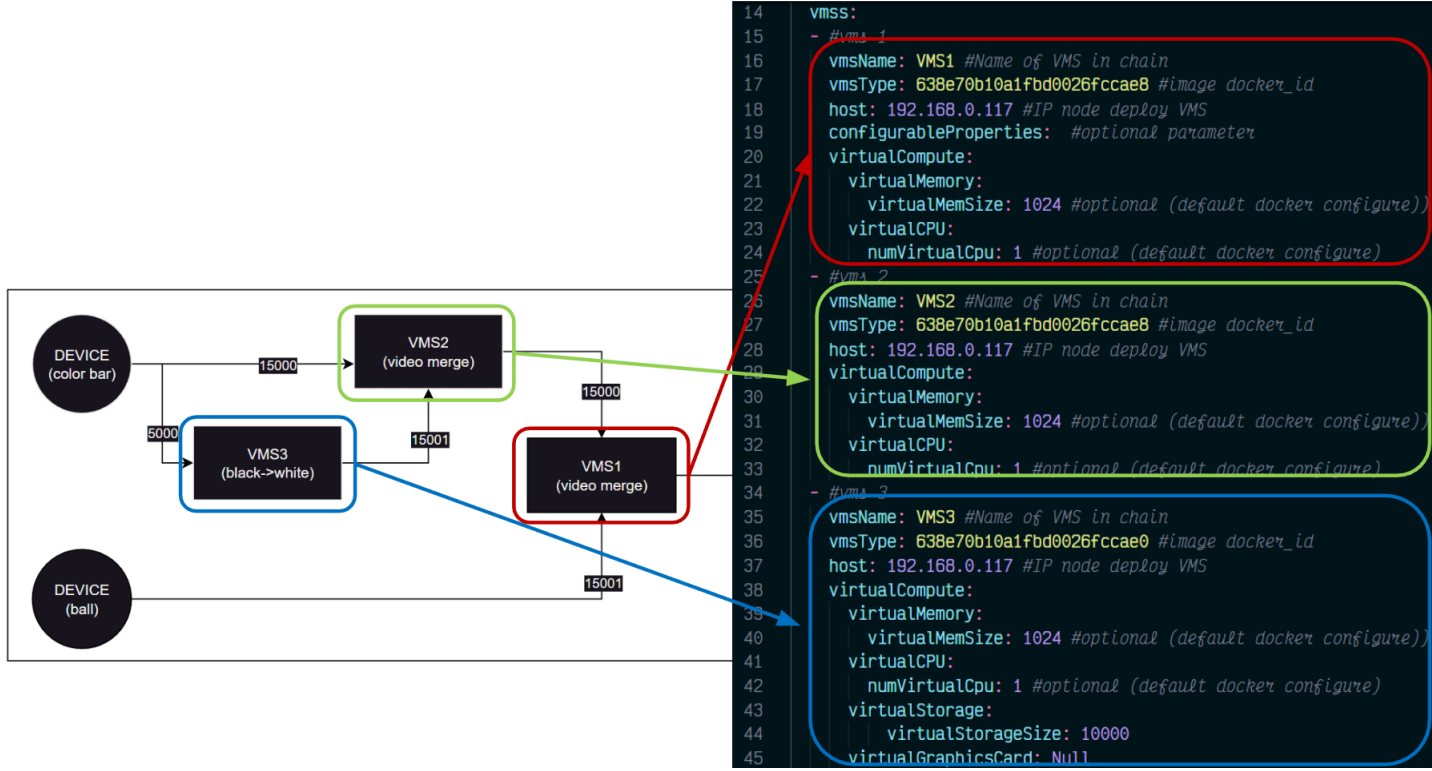


Figure 5.5: Specification of virtual multimedia sensors with the metamodel of L-PRISM

The first VMS is marked in the red box in Figure 5.5. This VMS will have the name (*vmsName*) *VMS1* defined in line 16. In line 17, we place the type of VMS (*vmsType*), a code that refers to a docker image. For this example, the code placed belongs to a VMS that joins two multimedia streams (*vms merge*). In line 18 (*host*), we define in which Edge Node this VMS will be orchestrated; the list of *host* must also be provided to be able to do this configuration. Finally, from lines 20 to 24, we define the computational resources assigned (*virtualCompute*) to this VMS, as seen in the formal specification of *virtualCompute* in previous sections, the elements that are part of it are not required. That is why for this VMS, we only use *virtualMemory* and *virtualCPU*, where we allocate 1024MB of RAM and 1 CPU for this VMS.

The second VMS is the one that is marked in the green box in Figure 5.5. This VMS will have the name (*vmsName*) *VMS2* defined in line 26. In line 17, like the previous VMS, we place the VMS type (*vmsType*), a code that references a docker



image. Looking closely at that figure, you can see that the *vmsType* of VMS1 and VMS2 are identical since we used the same image to display two containers that will receive different media streams. Like VMS1, the specification of the allocation of computational resources is made from lines 29 to 33.

The third VMS is the one that is marked in the blue box in Figure 5.5. This VMS will have the name (*vmsName*) *VMS3* defined in line 35. In line 36, like the previous VMS, we place the type of VMS (*vmsType*), a code that refers to a docker image. For this example, the code attached to this VMS belongs to a VMS image that transforms a multimedia stream of color video to a black-and-white video (*vms black-white*). If you take a closer look at the image, you can see that the *vmsType* of VMS3 differs from the previous VMSs, since we used a different image to unfold the container. The specification of the allocation of computational resources is made from lines 38 to 45, and you can see two new elements, which specify the allocation of storage resources (*1000MB*) and GPU allocation (*null*).

### 3. Description of the virtual link (VL)

Figures 5.6, 5.7, and 5.8 present the specification of the VL list used in this example. In line 47, *virtualLinks* is a list of VLs. In this example, we specify that we will create six VLs.

The first VL is marked in the red box in Figure 5.6. This VL represents the connection between the VD *ColorBar* and the VMS *VMS2*. As mentioned in the formal specification of a VL, it has two elements in its structure, *source*, and *destination*. *source* specifies the properties from which the multimedia stream will be sent, for this VL *sourceName* is the VD *ColorBar*, and since the source is a VD *sourceType* will be *device*, we also set the type of multimedia stream will be sent from this component, in this VL *outputType* will be *video*. For this example, we do not specify the format of the multimedia stream; it is for this reason that we do not put anything in *formatType*. *destination* specifies the properties of who will receive the multimedia stream; for this VL the destination *destinationName* is the VMS *VMS2*. We do not put any data in *destinationIp* because *VMS2* does not exist yet, and therefore there is not yet an IP address attributed to *VMS2*. Finally, specify the properties about which port *destinationPort* of *VMS2* will receive the multimedia stream sent by *ColorBar*; for this VL, the port in charge of receiving the multimedia stream will be 15000, and it will be of type *video*. The author of the VMS commonly defines the information about the port enabled to receive the multimedia stream, so those

who use these applications must have this information to configure the VLs.

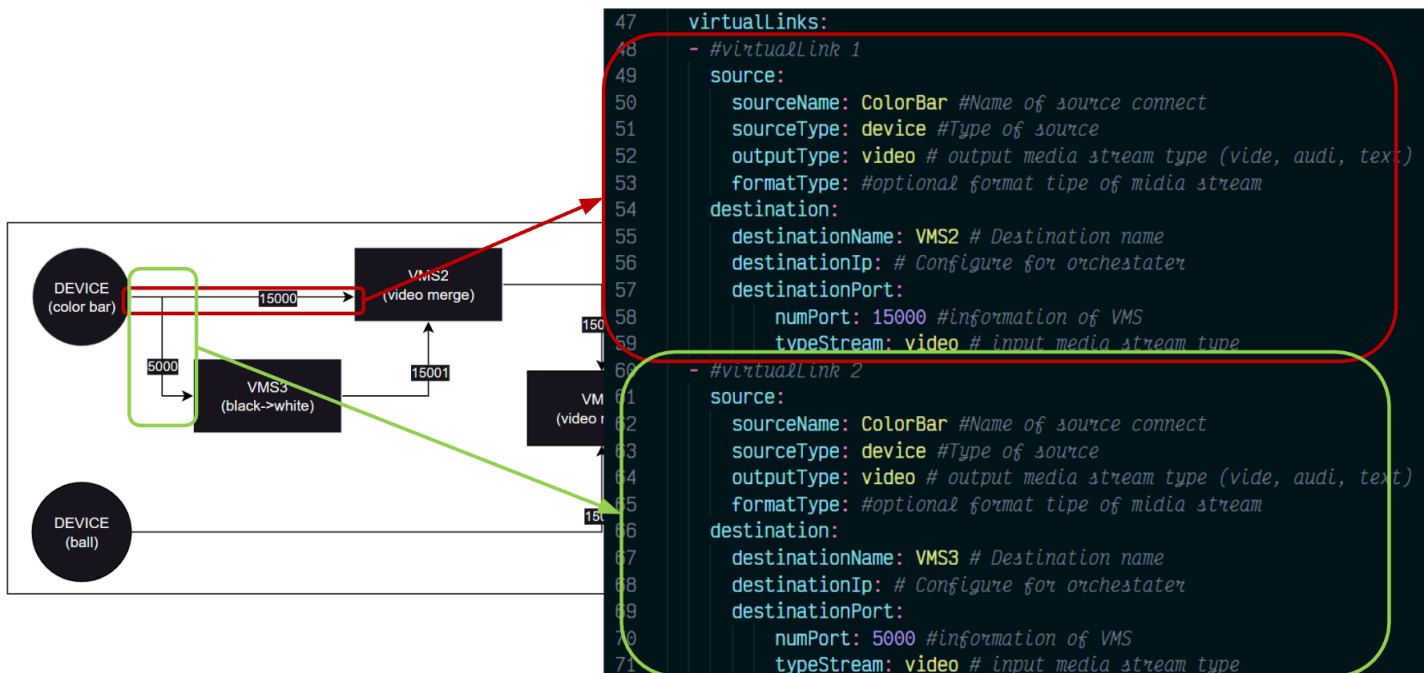


Figure 5.6: Specification of virtual link with the metamodel of L-PRISM- Part 1

The second VL is the one that is marked in the green box in Figure 5.6. This VL represents the connection between the VD *ColorBar* and the VMS *VMS3*. For this VL, *sourceName* is the VD *ColorBar*, and since the source is a VD, *sourceType* will be *device*, we also place the type of multimedia stream that will be sent from this component in this VL *outputType* will be *video*. For this example, we do not specify the format of the multimedia stream; it is for this reason that we do not put anything in *formatType*. *destination* specifies the properties of who will receive the multimedia stream; for this VL the destination *destinationName* is the VMS *VMS3*. We do not put any data in *destinationIp* because *VMS3* does not exist yet, and therefore there is not yet an IP address attributed to *VMS3*. Finally, specify the properties about which port *destinationPort* of *VMS3* will receive the multimedia stream sent by *ColorBar*; for this VL, the port in charge of receiving the multimedia stream will be 5000, and it will be of type *video*. The author of the VMS commonly defines the information about the port enabled to receive the multimedia stream, so those who use these applications must have this information to configure the VLs.

The third VL is the one marked in the red box in Figure 5.7. This VL represents the connection between the VMS *VMS3* and the VMS *VMS2*. For this VL, *sourceName* is the VMS *VMS3*, and as the source is a VMS, *sourceType* will be

*vms*, we also place the type of multimedia stream that will be sent from this component; in this VL *outputType* will be *video*. For this example, we do not specify the format of the multimedia stream; it is for this reason that we do not put anything in *formatType*. *destination* specifies the properties of who will receive the multimedia stream; for this VL the destination *destinationName* is the VMS *VMS2*. We do not put any data in *destinationIp* because *VMS2* does not exist yet, and therefore there is not yet an IP address attributed to *VMS2*. Finally, specify the properties about which port *destinationPort* of *VMS2* will receive the multimedia stream from *VMS3*; for this VL, the port in charge of receiving the multimedia stream will be 15001, and it will be of type *video*. The author of the VMS commonly defines the information about the port enabled to receive the multimedia stream, so those who use these applications must have this information to configure the VLs.

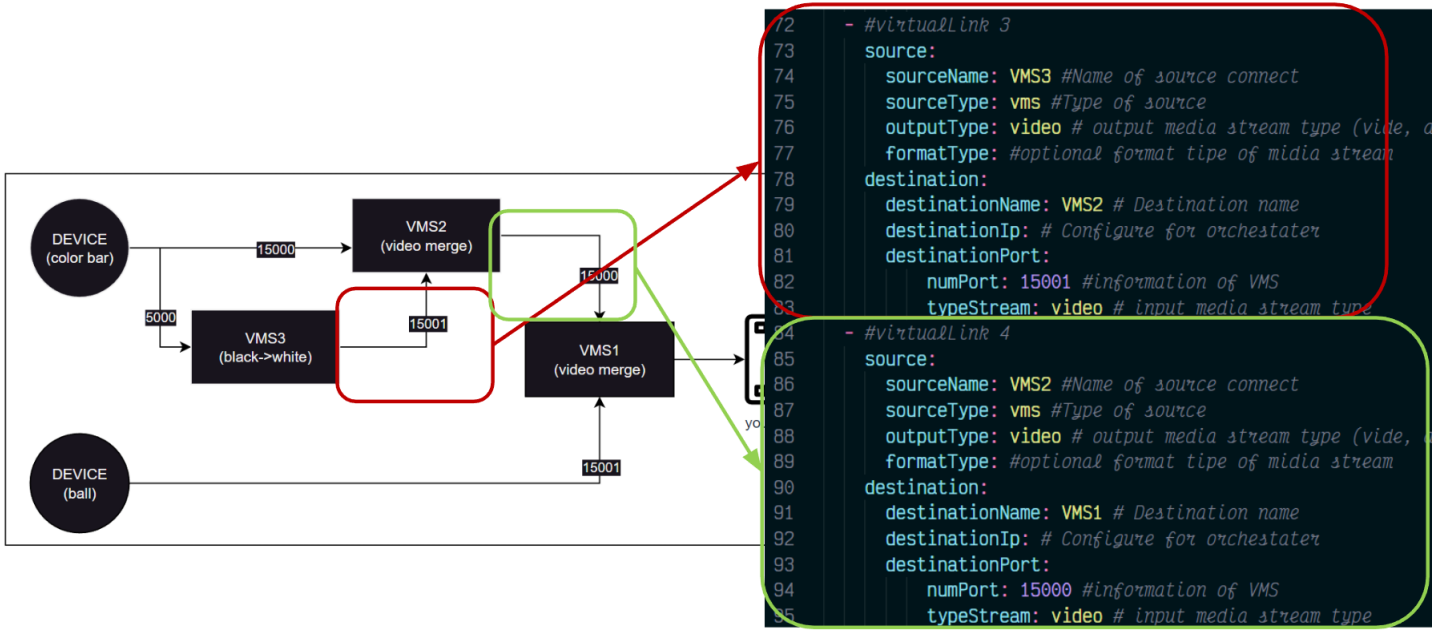


Figure 5.7: Specification of virtual link with the metamodel of L-PRISM- Part 2

The fourth VL is the one marked in the green box in Figure 5.7. This VL represents the connection between the VMS *VMS2* and the VMS *VMS1*. For this VL, *sourceName* is the VMS *VMS2*, and since the source is a VMS, *sourceType* will be *vms*, we also place the type of multimedia stream that will be sent from this component in this VL *outputType* will be *video*. For this example, we do not specify the format of the multimedia stream; it is for this reason that we do not put anything in *formatType*. *destination* specifies the properties of who will receive the multimedia stream; for this VL the destination *destinationName* is the VMS *VMS1*. We do not put any data in *destinationIp* because *VMS1* does not exist

yet, and therefore there is not yet an IP address attributed to *VMS1*. Finally, specify the properties about which port *destinationPort* of *VMS1* will receive the multimedia stream from *VMS2*; for this VL, the port in charge of receiving the multimedia stream will be 15000, and it will be of type *video*. The author of the VMS commonly defines the information about the port enabled to receive the multimedia stream, so those who use these applications must have this information to configure the VLs.

The fifth VL is the one marked in the red box in Figure 5.8. This VL represents the connection between the VD *Ball* and the VMS *VMS1*. For this VL *sourceName* is the VD *Ball*, and since the source is a VD *sourceType* will be *device*, we also place the type of multimedia stream that will be sent from this component in this VL *outputType* will be *video*. For this example, we do not specify the format of the multimedia stream; it is for this reason that we do not put anything in *formatType*. *destination* specifies the properties of who will receive the multimedia stream; for this VL the destination *destinationName* is the VMS *VMS1*. We do not put any data in *destinationIp* because *VMS1* does not exist yet, and therefore there is not yet an IP address attributed to *VMS1*. Finally, specify the properties about which port *destinationPort* of *VMS1* will receive the multimedia stream from *Ball*; for this VL, the port in charge of receiving the multimedia stream will be 15001, and it will be of type *video*. The author of the VMS commonly defines the information about the port enabled to receive the multimedia stream, so those who use these applications must have this information to configure the VLs.

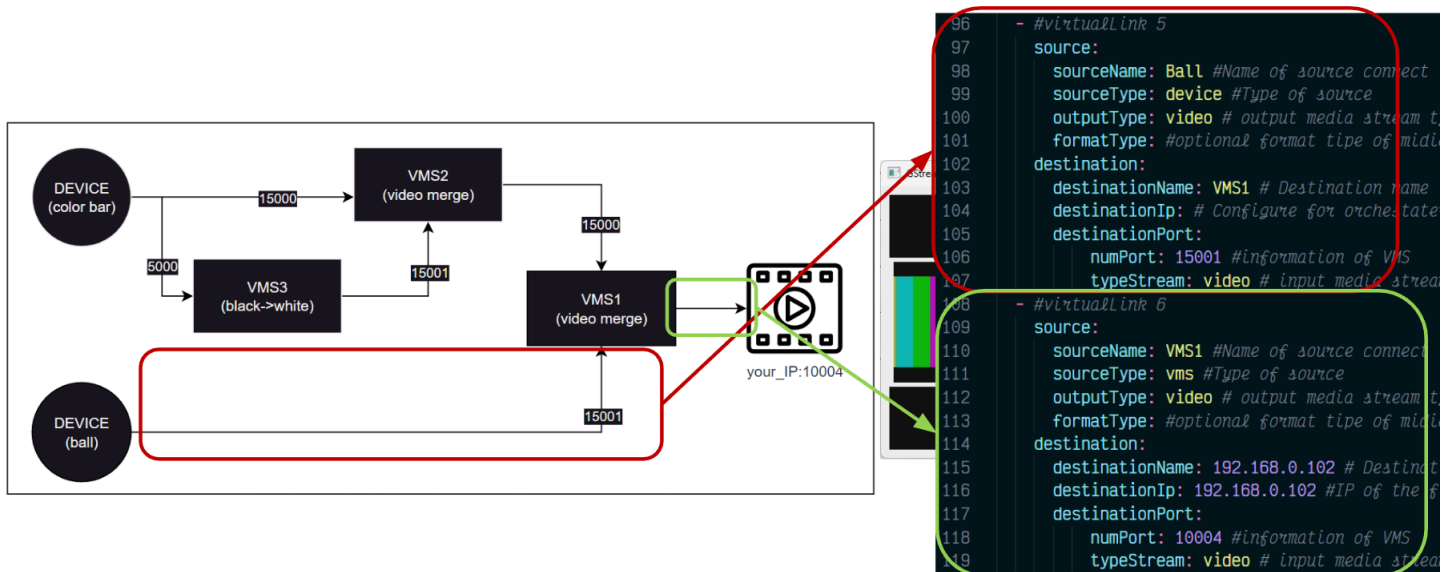


Figure 5.8: Specification of virtual link with the metamodel of L-PRISM- Part 3

Finally, the sixth VL is the one that is marked in the green box in Figure 5.8; this VL represents the final connection between the VMS *VMS1* and the final point. For this VL, *sourceName* is the VMS *VMS1*, and as the source is a VMS, *sourceType* will be *vms*, we also place the type of multimedia stream that will be sent from this component; in this VL, *outputType* will be *video*. For this example, we do not specify the format of the multimedia stream; it is for this reason that we do not put anything in *formatType*. *destination* specifies the properties of who will receive the multimedia stream. For this VL, the destination *destinationName* is a computer within the network, so the destination's IP address will represent the destination's name. In this case, the component that will receive the multimedia stream is a computer or device that already exists and has an IP address. In *destinationIp*, we place the IP address 192.168.0.102, which for our example, is a computer within our network. Finally, specifying the properties about which port *destinationPort* of the computer 192.168.0.102 will receive the multimedia flow coming from *VMS1*, for this VL, the port in charge of receiving the multimedia flow will be 10004 and will be of type *video*. The port information will depend on which available ports can be used on the computer or device that is used to deploy the multimedia service chain.

In this chapter, we presented L-PRISM metamodel, our proposed domain language for creating multimedia service chains based on VMS. In Chapter 6, we will present ALFA 2.0 and how L-PRISM was integrated into the ALFA [42] tool.

# Chapter 6

## ALFA 2.0: Integration of L-PRISM in ALFA

This chapter describes the integration of L-PRISM into the V-PRISM architecture [6], described in Section 2.3. As L-PRISM is a DSL for creating multimedia service chains based on VMS, the V-PRISM architecture had to be extended to support solutions developed with L-PRISM.

Besides the architecture, we also extended V-PRISM implementation (ALFA) in a new version called ALFA 2.0.

Since ALFA uses Docker containers to run VMS and virtual devices (VDs), any application can run regardless of the underlying technologies used. One of the premises of L-PRISM is that it can be used as a guide to develop and virtualize multimedia applications (VMS). ALFA 2.0 aims to facilitate the registration, compression, and use of the different elements that are part of L-PRISM within the V-PRISM architecture.

To meet the objective of ALFA 2.0, different changes were made to the modules *database*, *API*, and the *webapplication*., This is detailed in the following sections.

### 6.1 Database

ALFA uses MongoDB as a database; this database allows storing information about VMS images (type VMS), running VMS, virtual devices (VD), host (Nodes), and with our extension also multimedia service chains. For the database to store information about the multimedia service chains created on ALFA, *chainModel* was implemented. This model stores the list of VDs used, list of VMS created, list of virtual links (VL), and additional

information such as name and description of the multimedia service chain created. The implementation of *chainModel* is based on the definition of *vms.chainModel* described in detail in Section 5.6.1.

For the ALFA database to fully adapt to L-PRISM, the models *vmsTypeModel*, *vmsModel*, and *deviceModel* also had to be extended. In the case of *vmsTypeModel*, it was adapted according to the definition of the *swImage* artifact described in Section 5.3.1. In the case of *vmsModel*, this model was extended to support the definition of the *VDU.vms* node type described in Section 5.5.1. Finally, *deviceModel* has its model extended to support the definition of the *device* artifact described in Section 5.3.3.

## 6.2 API

ALFA implementation already has an API used to manage a VMS's life cycle. Within the ALFA API services, no aspect of VMS chains was considered, so new services were added, which allow users to create multimedia service chains based on VMS. The main service that was implemented enables users to upload from the front-end a YAML file where the multimedia service chain is described using L-PRISM. Internally this service processes each of the elements following the following rules:

- The lists of VD, VMS, and VL are separated.
- The VMS at the end of the multimedia service chain is searched for since it contains complete information about its operation. For example, the final components of a multimedia service chain contain information about the address (IPv4) where the processed multimedia stream will be sent.
- Every time an element of the VMSs list is executed and transformed into a container, information is captured to help complete the information of other elements of the multimedia service chain. For example, for a VMS to send a multimedia stream to another VMS, it would need the IP address of the destination VMS. The list of VMSs in the YAML file will be transformed into containers, and when this happens, Docker will assign them an IP address. This IP address will be used so that other elements of the multimedia service chain can communicate with the created VMS (container). For this reason, each time a VMS is created, we capture some data, such as IP, container id, and others, to complete information on other elements within the multimedia service chain (VMS, VD, VL).

- When a VMS is created, one looks for which of all the connections (VL) has *destine* to this VMS as its final element. Once the VL related to the created VMS is found, the VMS that is specified in *origin* is created of this VL.
- When the entire list of VMSs has been created, the connection between the VDs and VMSs is created. This process consists of subscribing the VMS to the VD flows.
- Once this process is finished and the data is completed, the multimedia service chain is stored in the database, and a response is sent to the front-end.

*List4Chain* in other implemented service is one that is in charge of bringing the list of VMS types (*VMStype*), the list of virtual devices (*VD*), and the list of nodes (*host*). Figure 6.2 and Figure 6.3 show how these lists present the necessary information that the users will be required to create multimedia service chains using L-PRISM.

Additionally, some minor modifications were made to other services to support the CRUD (create, read, update, delete) of the elements described in the database.

## 6.3 Web interface

### 6.3.1 Web interface for L-PRISM

A new component has been added to the ALFA web application, which allows uploading a YAML file and sending it to the backend through the API described in Section 6.2. Figure 6.1 shows the implemented interface, in which it can be seen that in the upper right part, there is a button to load the YAML file. Additionally, the information on the VMSs, virtual devices (VDs), and Nodes (Host), available so that users can use them to create multimedia service chains, is shown.

We also present the list of available *VMS*, as shown in Figure 6.2. The list of *EdgeNodes*, with the state in which it is located. And finally, the list of available Virtual Devices (VDs) presents additional information, such as whether the VD is running and ready to send its multimedia stream, as shown in Figure 6.3.

L-PRISM not only allows creating multimedia service chains but also allows registering new elements that can be used in the multimedia service chain. Figure 6.4 shows the modified VMS type registration interface (*swImage*) for L-PRISM, allowing easy registration of new VMS types. Figure 6.5 shows the interface that allows seeing in detail the



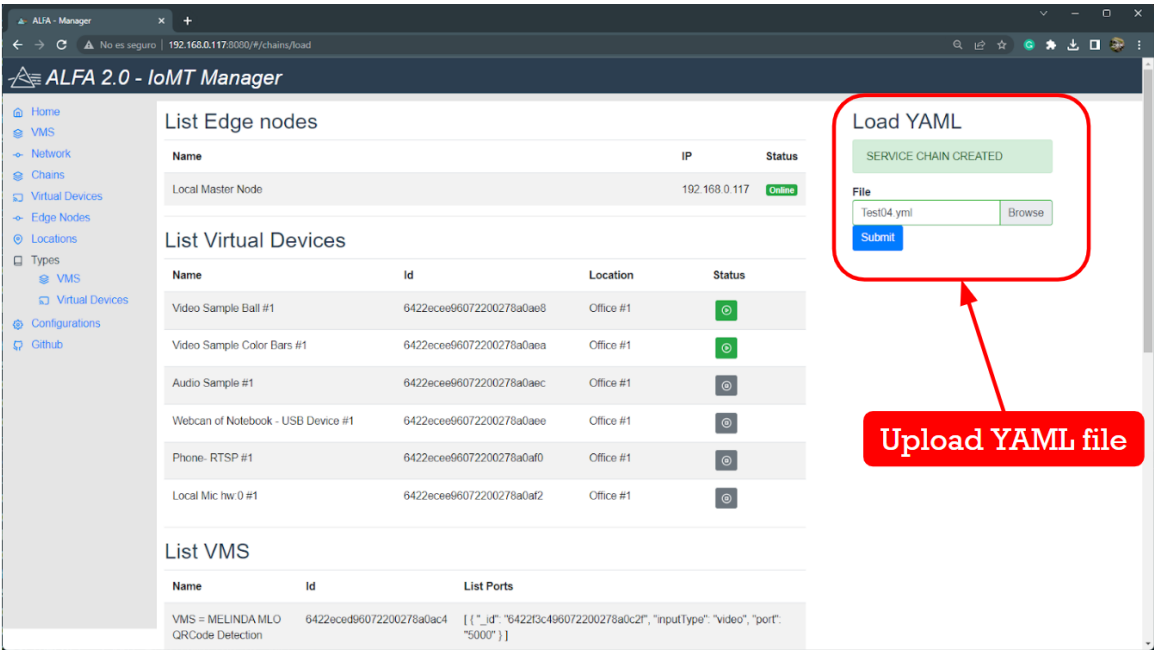


Figure 6.1: Interface to upload YAML file.

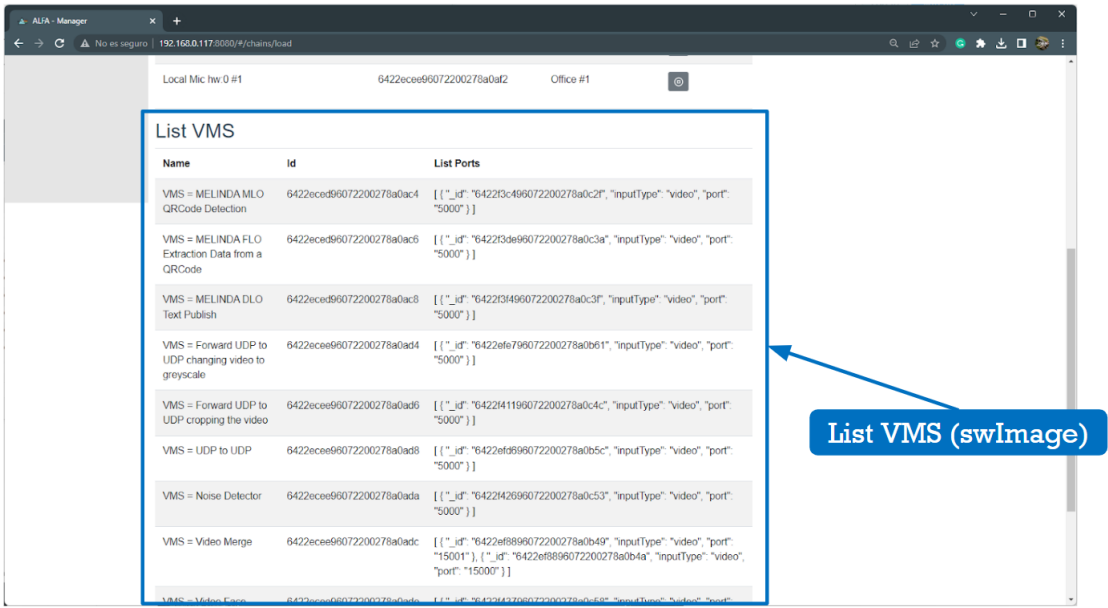


Figure 6.2: Interface to upload YAML file - List of VMS.

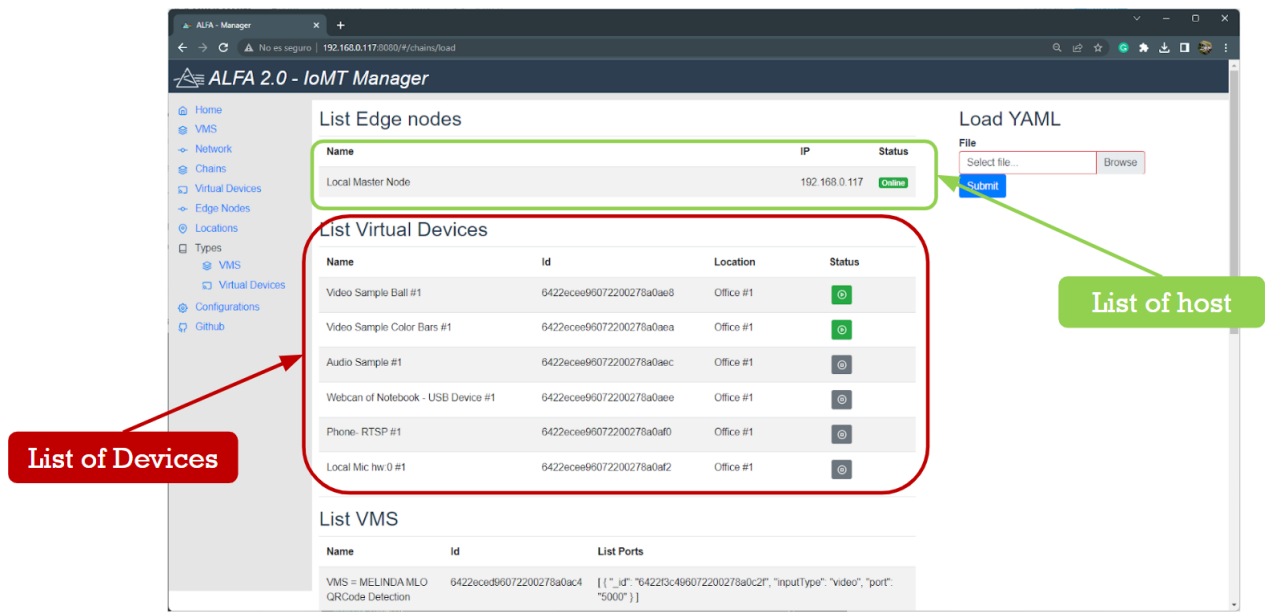


Figure 6.3: Interface to upload YAML file - List of Edge Node and list of VD.

types of registered VMS. This interface will help users better understand the behavior of the elements they need to create chains of multimedia services.

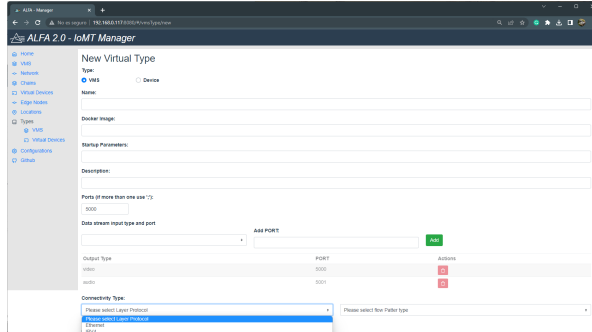


Figure 6.4: Interface for register type VMS

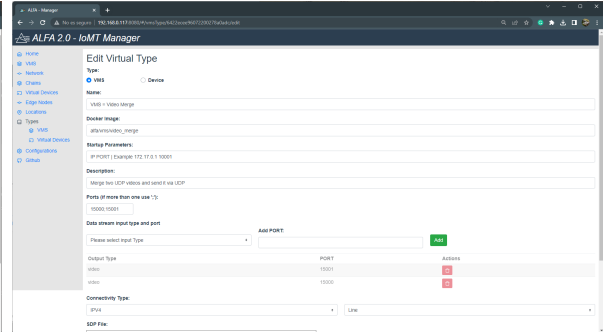


Figure 6.5: Interface for view detail of type VMS (swImage)

### 6.3.2 Prototype web interface

An interface was developed that allows the implementation of multimedia service chains through an intuitive interface; this interface can be seen in Figure 6.8. To create a chain of multimedia services based on VMS using a traditional method (command lines in a terminal) the elements must be created separately and mainly, the order of creation must be from the end (endpoint), backwards. This is because the multimedia service chains work as a directed acyclic graph (DAG) [17], where the initial nodes of the DAG would be the VDs and the internal nodes would be the VMS. To create the communication between

$VD$  to  $VMS$ ,  $VMS$  to  $VMS$ , and  $VMS$  to a *final – destination*, some information about the destination where the multimedia flow will be sent must be known. In ALFA 2.0, this destination is represented by the  $IP$  address. To create a multimedia service chain traditionally, you must create the final element first since you have the necessary information for its execution.

The ALFA 2.0 interface follows the model of implementing multimedia service chains traditionally, which means that a specific order must be followed to create a multimedia service chain. First, the  $VMS$ s are created (from the end to the back), as shown in Figure 6.6, and at the end, the  $VD$ s are selected, as shown in Figure 6.7, that will send media streams to this media service chain. Clicking on *Save* will use a service designed to process the data from the multimedia service chain, defined through the intuitive interface.

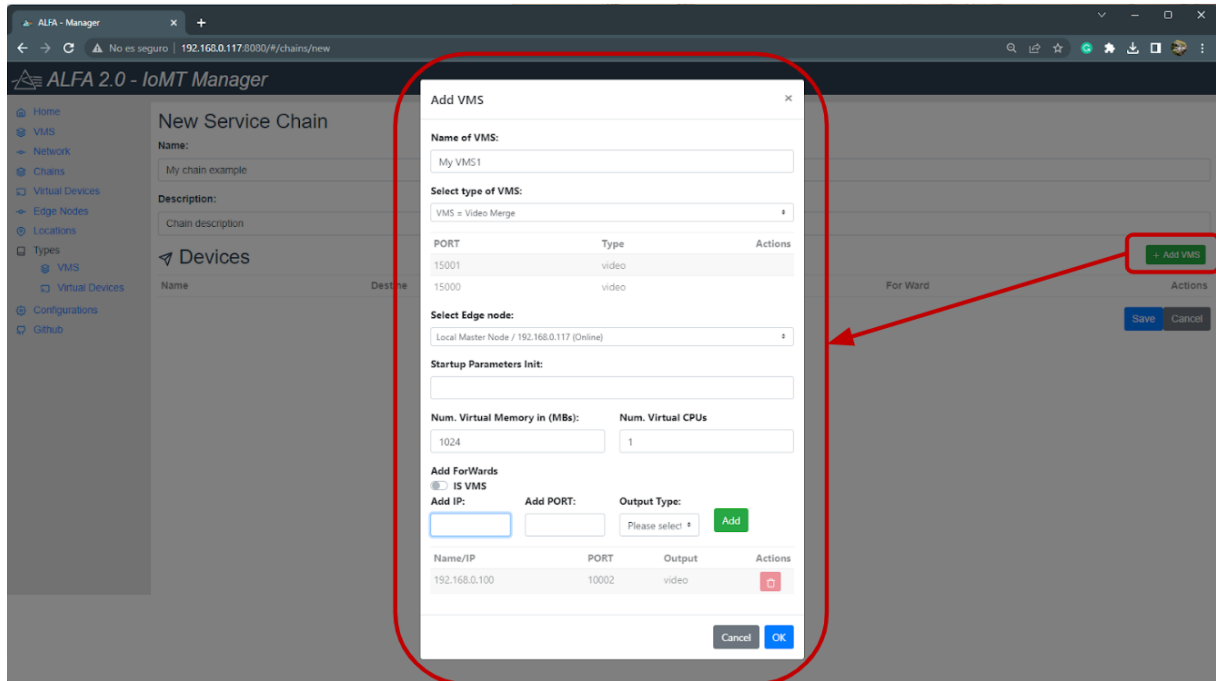


Figure 6.6: Interface to create a  $VMS$ , through the intuitive interface.

Figure 6.9 shows how a multimedia service chain would be implemented with the *prototype* interface. This figure shows the service chain to be created, the components registered in the interface, and the expected result.

One of the bases to develop this interface is that in future work, a YAML file can be loaded where a multimedia service chain has been implemented using L-PRISM, and this chain can be displayed graphically to facilitate the understanding of the operation of multimedia service chain based on  $VMS$ .

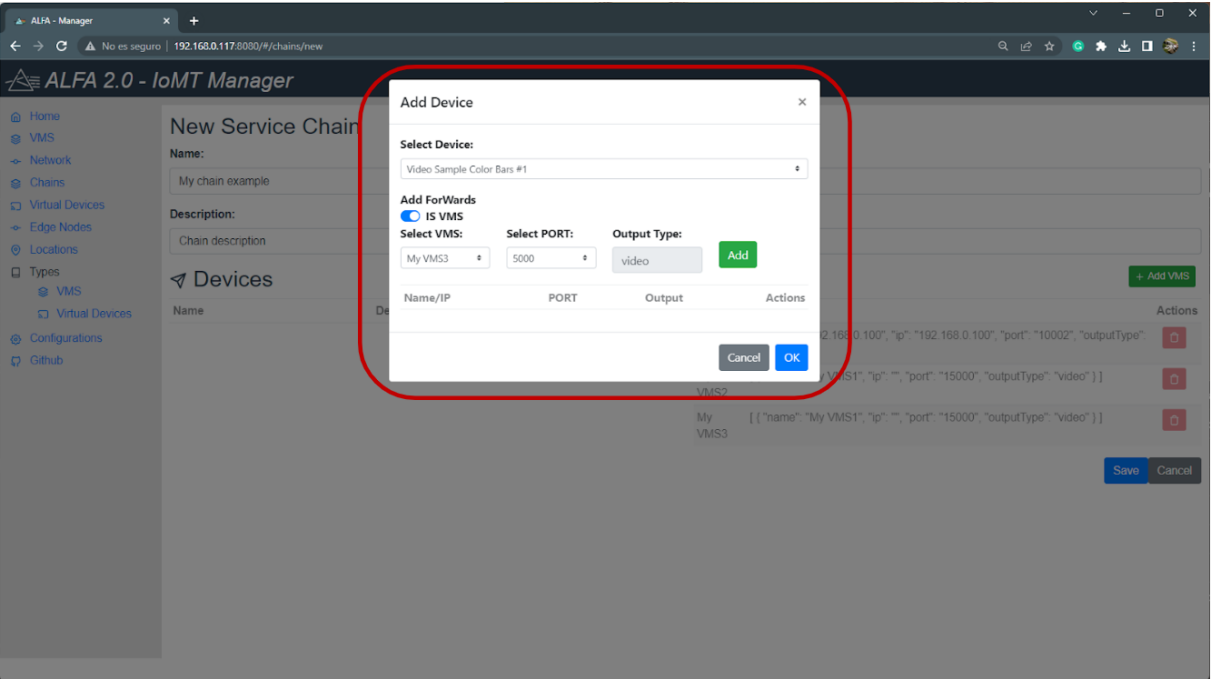


Figure 6.7: Interface to select a *VD*, through the intuitive interface.

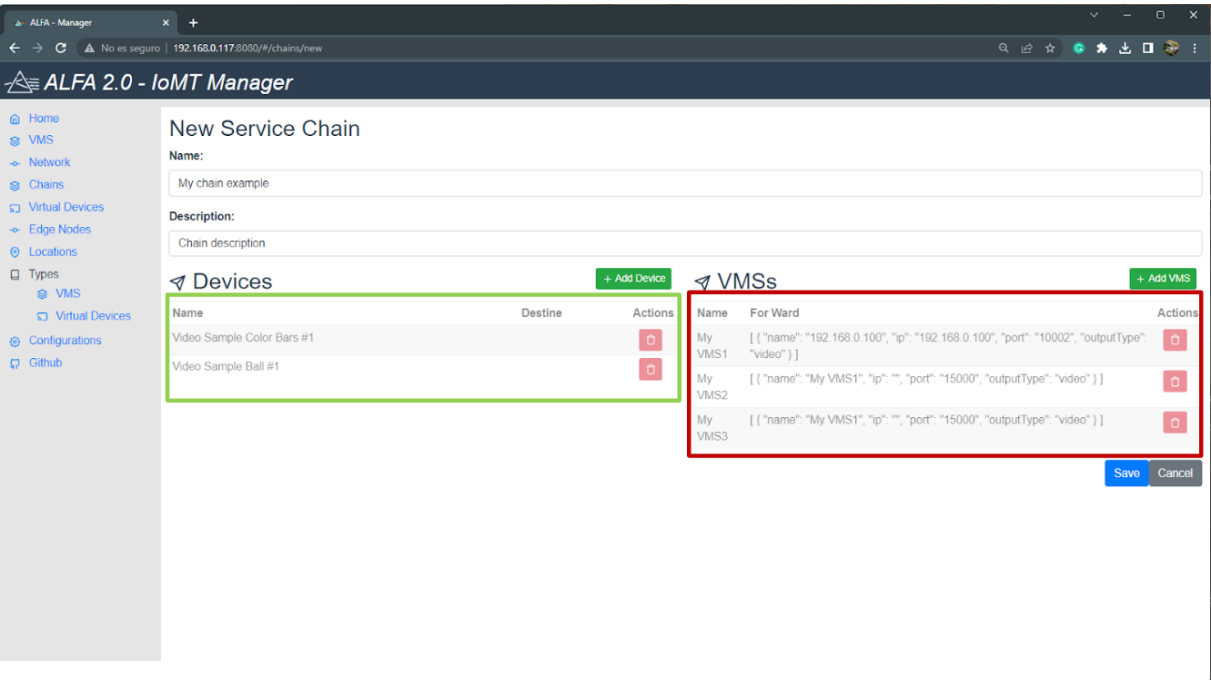


Figure 6.8: Intuitive interface to create multimedia service chain based on VMS.

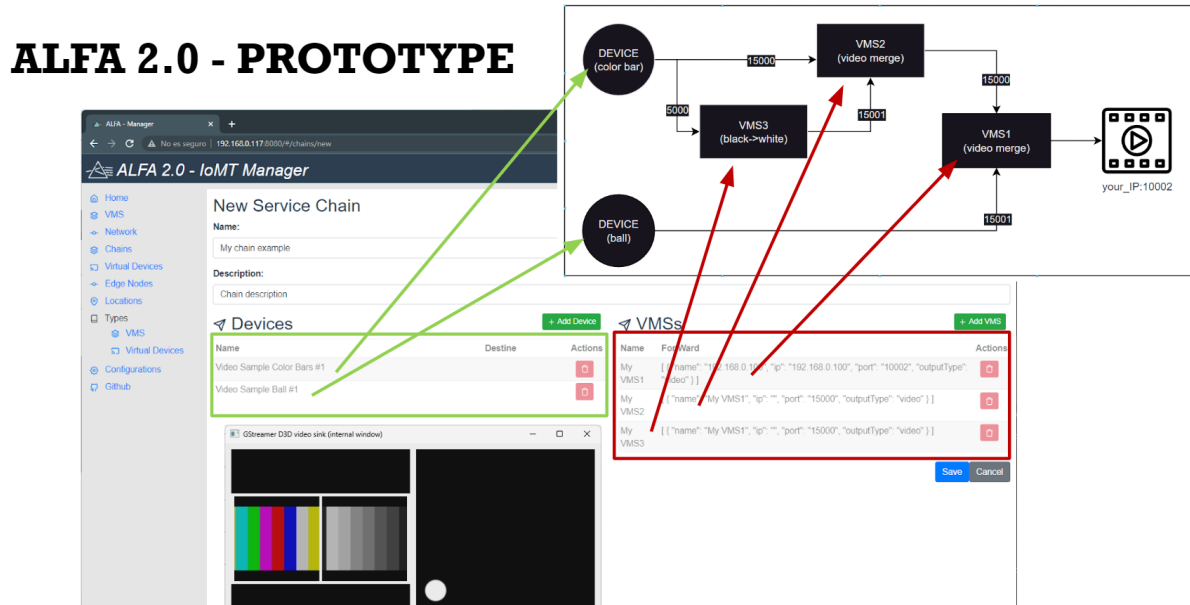


Figure 6.9: General example of a multimedia service chain created in *prototype*.

## 6.4 Differences between ALFA and ALFA 2.0

ALFA 2.0, as such, is not just a new version of ALFA, it is more than a simple extension. The main difference between ALFA and ALFA 2.0 is that the functionality to create multimedia service chains based on VMS has been added to ALFA 2.0. To add this functionality, new elements were included in ALFA existing modules, and the *Chain* module was designed and developed from scratch.

# Chapter 7

## Evaluation

This chapter describes the experiments performed for evaluating L-PRISM and answering our research question "Does using a Domain Specific Language (DSL) facilitate the deployment of multimedia service chains based on VMS?".

The Goal Question Metric (GQM) [25] approach was adopted to carry out the experiments. The GQM method has a hierarchical structure of three levels (Goals, Questions, and Metrics) that refines the results and makes them provide reliable information. The goals present the phenomenon to be analyzed, each objective can be represented with one or more questions, and one or more metrics are used to answer each question.

Section 7.1 presents the evaluation process, following the GQM method [25], where we detail how to analyze the two main goals of our experiments. Section 7.2 explains the tasks proposed to participants. Section 7.3 describes the subjects that participated in our experiments and Section 7.4 provides results and discussion.

### 7.1 Experiment Goals

The goals of this experiment are shown in Table 7.1. Goal *G1* analyzes the application engineering process with and without L-PRISM to evaluate the efficiency and productivity of developing multimedia service chains. Goal *G2* evaluates the comprehensibility of L-PRISM to analyze if variables, attributes, and structures are understandable for subjects.

#### 7.1.1 G1 - Questions and metrics

The questions defined for *G1* are based on some of the Technology Acceptance Model (TAM) [35]. This approach will help to validate L-PRISM from a quantitative point of

Table 7.1: Goals of the experiment

Goal	Description	Perspective
<b>G1</b>	Analyze the application engineering process with and without L-PRISM to evaluate the efficiency and productivity of developing multimedia service chains.	Efficiency and Productivity
<b>G2</b>	Evaluate the comprehensibility of L-PRISM to analyze if variables, attributes, and structures are understandable for subjects.	Usability

view.

#### 7.1.1.1 G1 - Questions

Questions  $Q1 - Q6$  shown in Table 7.2 are related to the first goal, focusing on efficiency and productivity in developing multimedia service chains based on VMS. The answers to these questions will be obtained through the metrics defined for  $G1$  in the controlled experiment.

Table 7.2: Questions for the goal  $G1$ 

Question	Description
<b>Q1</b>	Is the application engineering process using L-PRISM effective in terms of time for developing multimedia service chains based on VMS, compared to the traditional approach (V-PRISM)?
<b>Q2</b>	Does the developer claim that using L-PRISM makes it easier to understand the functional and non-functional requirements of the multimedia service chain based on VMS.?
<b>Q3</b>	Does the developer claim that using L-PRISM facilitates creating multimedia service chains based on VMS.?
<b>Q4</b>	Does the developer claim that L-PRISM is useful to create multimedia service chains based on VMS?
<b>Q5</b>	Does the developer claim that using L-PRISM makes it easier to reuse multimedia service chains created with L-PRISM to create new multimedia service chains?
<b>Q6</b>	Is the process of modifying multimedia service chains based on VMS faster with L-PRISM? Compared with the traditional method (V-PRISM).

#### 7.1.1.2 G1 - Hypotheses, Variables, and Constructions

For goal  $G1$  of this experiment, it will be necessary to derive the null hypotheses, called  $H0_{ij}$ , and their corresponding alternative hypotheses called  $H1_{ij}$ , where  $i$  corresponds to

the goal; in this case,  $G1(1)$  and  $j$  is a counter, in case there is more than one hypothesis for this goal.

Null hypotheses:

- $H0_{11}$  The use of L-PRISM is equivalent to the traditional development of multimedia service chains based on VMS in terms of development time.
- $H0_{12}$  Compression of functional and non-functional requirements of multimedia service chains with L-PRISM is the same as the traditional method (V-PRISM).
- $H0_{13}$  The use of L-PRISM is just as productive in developing VMS-based multimedia service chains compared to the traditional method (V-PRISM).
- $H0_{14}$  L-PRISM is just as beneficial for developing multimedia service chains based on VMS as the alternative method.
- $H0_{15}$  The use of L-PRISM maintains the same degree of complexity in reusing other multimedia service chains based on VMS compared to the traditional method (V-PRISM).
- $H0_{16}$  The time required to modify a multimedia service chain with L-PRISM and the alternative method is equivalent.

Alternative hypotheses:

- $H1_{11}$  L-PRISM is better than the traditional development of multimedia service chains based on VMS in terms of development time.
- $H1_{12}$  L-PRISM makes it easier to understand the functional and non-functional requirements of multimedia service chains based on VMS.
- $H1_{13}$  L-PRISM is more productive than the traditional development of multimedia service chains based on VMS.
- $H1_{14}$  L-PRISM is more helpful for developing multimedia service chains based on VMS than the alternative method.
- $H1_{15}$  The use of L-PRISM makes the reuse of multimedia service chains already developed less complex compared to the traditional method (V-PRISM).



- $H1_{16}$  The time required to modify a multimedia service chain with L-PRISM is less than the alternative method.

To answer the questions related to  $G1$ , as well as to check if its formulated hypotheses are true or false, the following constructs are defined:

- Development effort.
- Understanding of requirements.
- Perceived ease of use.
- Perceived utility.
- Perceived reuse.
- Reuse effort.

These constructs represent the properties we want to evaluate for  $G1$ . Additionally, these will be measured according to the metrics defined in Section 7.1.1.3.

#### 7.1.1.3 $G1$ - Metrics

Since there is no standard metrics that can evaluate all the questions defined for  $G1$ , we use some of the metrics defined in the Technology Acceptance Model (TAM) [35]. Additionally, we specify our metrics based on the constructs proposed in Section 7.1.1.2. The goal of the proposed metrics is to quantitatively evaluate the development process of multimedia service chains based on VMS using L-PRISM and the traditional method (V-PRISM).

**M1** is defined to evaluate the development effort construct: This metric records the total time required to perform the proposed tasks considering both approaches (*L-PRISM* and *TRADITIONAL(V-PRISM)*). The objective of  $M1$  is to measure the time it takes each participant to develop different multimedia service chains based on VMS using the two approaches. This metric will help answer  $Q1$  of  $G1$ .

**M2** is defined to assess the design and understanding of the requirements: The objective of  $M2$  is to assess the participant's understanding of the functional and non-functional requirements of the four tasks. Considering that a clear understanding of the requirements

positively affects the development of the different tasks, this metric helps to answer *Q2* of *G1*. This metric is measured using a questionnaire based on the Likert scale [26].

**M3** is defined to evaluate the construct of perceived ease of use: The objective of *M3* is to measure the opinion of the participant about usability in both approaches; this metric captures the degree to which a person believes that the use of a particular system or model is better [14]. This metric will help to answer *Q3* of *G1*. This metric is measured using a questionnaire based on the Likert scale [26].

**M4** is defined to evaluate the perceived usefulness construct: The objective of *M4* is to measure the participant's opinion regarding the usefulness of L-PRISM for developing multimedia service chains based on VMS. This metric indicates the degree to which a person believes that a particular system or method helps develop a specific task [14]. This metric will answer *Q4* of *G1*. This metric is measured using a questionnaire based on the Likert scale [26].

**M5** is defined to evaluate the construct of perceived reuse: The objective of *M5* is to evaluate the opinion of the participant regarding code reuse using L-PRISM in the development process of multimedia service chains based on VMS. This metric indicates how a person can reuse existing code to create new applications. This metric will help to answer *Q5* of *G1*. This metric is measured using a questionnaire based on the Likert scale [26].

**M6** is defined to evaluate the reuse effort construct: The objective of *M6* is to measure the time required to modify or add new functionality to an application previously developed with L-PRISM. This metric measures the average time to modify a previously developed application. This metric will help answer questions *Q6* of *G1*.

## 7.1.2 G2 - Questions and metrics

The questions defined for *G2* are based on the Cognitive Dimensions of Notations (CDN) [8]. This approach will help to validate L-PRISM from a usability point of view.

### 7.1.2.1 G2 - Questions

Questions *Q1* – *Q7* shown in Table 7.3 are related to the second goal, focusing on usability in developing multimedia service chains based on VMS.

The answers to these questions will be obtained through the metrics defined for *G2*

in the controlled experiment.

Table 7.3: Questions for the goal  $G2$

Question	Description
<b>Q1</b>	How easy is it to visualize or find the various components of L-PRISM while creating or changing a multimedia application?
<b>Q2</b>	How easy is modifying a multimedia service chain with L-PRISM?
<b>Q3</b>	Is the L-PRISM language too verbose to specify a multimedia services chain?
<b>Q4</b>	In general, do the elements and attributes of L-PRISM represent well a multimedia service chain?
<b>Q5</b>	Is it easy to understand the data types and structures in L-PRISM?
<b>Q6</b>	There are structures and data types in L-PRISM that can be closely related, and changes to one can affect the other. Are those dependencies visible?
<b>Q7</b>	Does L-PRISM generally seem easy or difficult to understand (for example, when changing different elements of a multimedia service chain)?

#### 7.1.2.2 $G2$ - Hypotheses, Variables, and Constructions

For goal  $G2$  of this experiment, it will also be necessary to derive null hypotheses, called  $H0_{ij}$ , and their corresponding alternative hypotheses called  $H1_{ij}$ , where  $i$  corresponds to the goal; in this case,  $G2(2)$  and  $j$  is a counter, in case there is more than one hypothesis for this purpose.

Null hypotheses:

- $H0_{21}$  L-PRISM components are hard to find when creating or modifying a multimedia application.
- $H0_{22}$  Changes to existing applications are difficult to make.
- $H0_{23}$  The data types and structures in L-PRISM are too extensive to describe.
- $H0_{24}$  L-PRISM elements have a strange description.
- $H0_{25}$  Data types and structures are difficult to understand.
- $H0_{26}$  Dependencies exist between components or data types in L-PRISM and are not visible.
- $H0_{27}$  L-PRISM at a general level is difficult to understand.

Alternative hypotheses:

- $H_{121}$  L-PRISM components are easy to find when creating or modifying a multimedia application.
- $H_{122}$  Changes to existing applications are easy to make.
- $H_{123}$  The data types and structures in L-PRISM are reasonably brief to describe.
- $H_{124}$  The elements of L-PRISM have a simple description.
- $H_{125}$  Data types and structures are easy to understand.
- $H_{126}$  The dependencies between the components or data types in L-PRISM are easy to recognize.
- $H_{127}$  L-PRISM at a general level is easy to understand.

To answer the questions related to  $G2$ , as well as to check if its formulated hypotheses are true or false, the following constructs are defined:

- Visibility
- Viscosity
- Diffuseness
- Closeness of Mapping
- Role Expressiveness
- Hidden dependencies
- Hard mental operations

These constructs represent the properties that we want to evaluate for  $G2$ . These will be measured according to the metrics defined in Section 7.1.2.3.

### 7.1.2.3 G2 - Metrics

As mentioned previously, the cognitive dimensions metric will be used to assess the *G2* questions. The objective of using these metrics is to evaluate L-PRISM usability.

**M1** is defined to evaluate the visibility construct: This metric records how easy it is to see the different components of L-PRISM. The objective of *M1* is to evaluate the complexity that the participants have when visualizing the components. This metric will help answer *Q1* of *G2* through a questionnaire based on the Likert scale [26].

**M2** is defined to evaluate the Viscosity construct: This metric records the resistance to changes of applications made with L-PRISM. The objective of *M2* is to evaluate how difficult it is to modify an application made with L-PRISM. This metric helps answer *Q2* of *G2* through a questionnaire based on the Likert scale [26].

**M3** is defined to assess the diffuseness construct: This metric records the level of understanding and usefulness of L-PRISM components. The objective of *M3* is to test whether participants understand the behavior of L-PRISM data types and structures. This metric helps answer *Q3* of *G2* through a questionnaire based on the Likert scale [26].

**M4** is defined to evaluate the Closeness of Mapping construct: This metric records the proximity of L-PRISM to the performance of multimedia applications. The objective of *M4* is to perceive if the participants relate L-PRISM with the operation or behavior of multimedia applications. This metric helps to answer *Q4* of *G2* through a questionnaire based on the Likert scale [26].

**M5** is defined to evaluate the Expressiveness construct: This metric records the expressiveness of the L-PRISM components. The objective of *M5* is to assess whether participants can understand what the L-PRISM elements are for just by reading them. This metric helps answer *Q5* of *G2* through a questionnaire based on the Likert scale [26].

**M6** is defined to evaluate the Hidden Dependencies construct: this metric registers if the relationships between the different components are not visible. The objective of *M6* is to assess whether the participants can easily recognize the relationships between L-PRISM components. This metric helps answer *Q6* of *G2* through a questionnaire based on the Likert scale [26].

**M7** is defined to assess the Hard Mental Operations construct: This metric records the overall complexity of L-PRISM. The objective of *M7* is to evaluate if L-PRISM is challenging to understand at a general level. This metric helps answer *Q7* of *G2* through

a questionnaire based on the Likert scale [26].

## 7.2 Experiment Tasks

The experiment was planned to occur in two phases, the training phase and the execution phase.

### 7.2.1 Training phase

In the first or training phase, the participants are trained in two topics:

- Understanding and development of multimedia service chains based on VMS. For this, it is necessary to carry out the manual deployment of a multimedia service chain. In this stage, the help of the V-PRISM architecture presented by [42] (*model 1*) is used. With this, we want to simulate the development of multimedia service chains based on VMS using the traditional method (manually executing each of the components of the multimedia service chain (V-PRISM)).
- Development of multimedia service chains based on VMS using L-PRISM (*model 2*), for which a module was implemented within V-PRISM, which allows the loading of an application made with L-PRISM (File in YAML format).

The material delivered to the subjects for the training stage is available on the website<sup>1</sup>. This website presents the following content:

- Information about L-PRISM and the researchers involved in its development.
- Summary of L-PRISM and the main elements used to create a multimedia service chain based on VMS. Figure 7.1 shows how the main elements of a multimedia service chain based on VMS are presented on the website. You can also see the example code of each element and its details.
- Examples of multimedia service chains created with L-PRISM as shown in Figure 7.2. Additionally, each example is detailed with a brief description of the multimedia service chain created, the code using L-PRISM in YAML format, a reference image of how the multimedia service chain is structured, a video showing the implementation

---

<sup>1</sup><https://fventuraq.github.io/lprism.html>

process in detail, the command line to display the result and finally an image of how the result should be when displaying this example. Figure 7.3 shows a view of an example on the website.

- Examples of multimedia service chains created with the traditional method (V-PRISM) are presented in videos, as shown in Figure 7.4. These videos detail creating multimedia service chains based on VMS using V-PRISM.

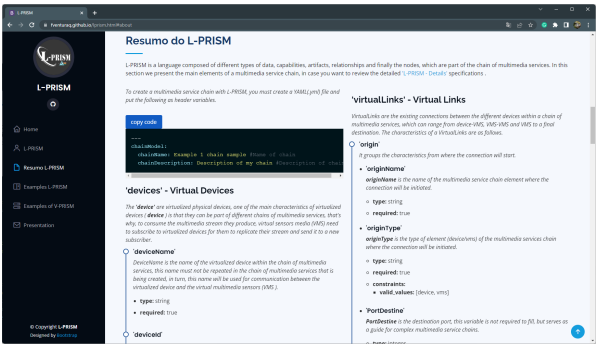


Figure 7.1: View of the website of main elements of L-PRISM for creating a multimedia service chain based on VMS

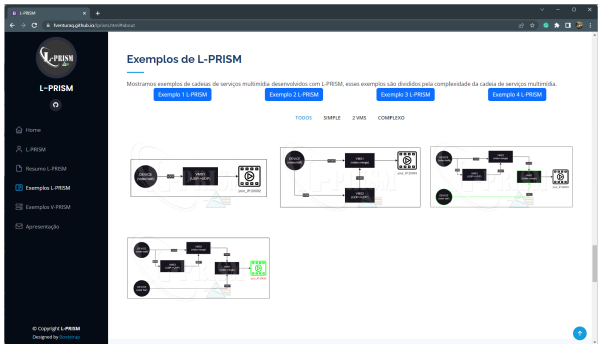


Figure 7.2: View the examples of multimedia service chains created with L-PRISM on the website.

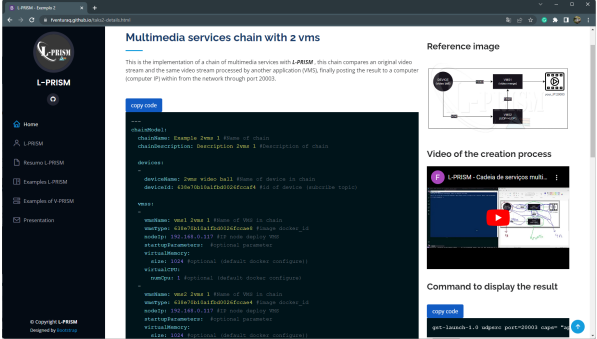


Figure 7.3: Details of the examples of multimedia service chains created with L-PRISM on the website.

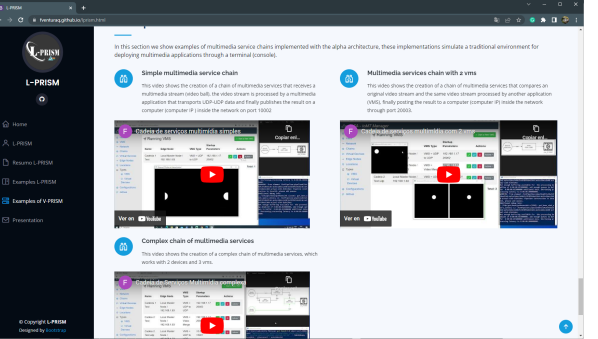


Figure 7.4: View the examples of multimedia service chains created with V-PRISM on the website.

Additionally, the participants will answer a characterization questionnaire which will provide information about professional experience and software development skills.

### 7.2.2 Execution phase

In the second or execution phase, each participant will be randomly assigned which model 1 or 2 to start. After they know which model they will start the experiment with, they will be given the tasks described in Section 7.2.3.

### 7.2.3 Tasks

- **Task 1:** Create a multimedia service chain based on VMS with and without L-PRISM, which receives a multimedia stream (color video), transforms the video to grayscale, and finally publishes this video stream on a computer within the network by port 10002; refer to Figure 7.5.

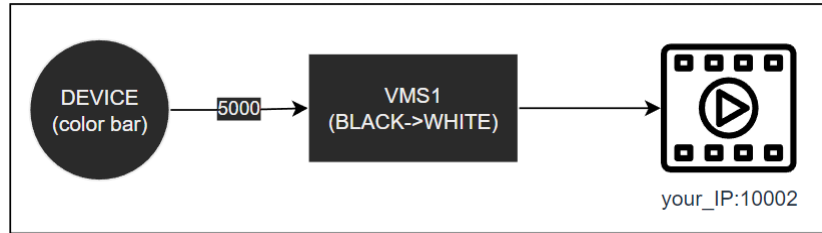


Figure 7.5: Multimedia service chain that transforms a color video to grayscale - Task 1.

- **Task 2:** Create a multimedia service chain based on VMS with and without L-PRISM, comparing a raw video stream and the same greyscale-transformed video stream, finally publishing this video stream on a computer within the network through port 10003; refer to Figure 7.6.

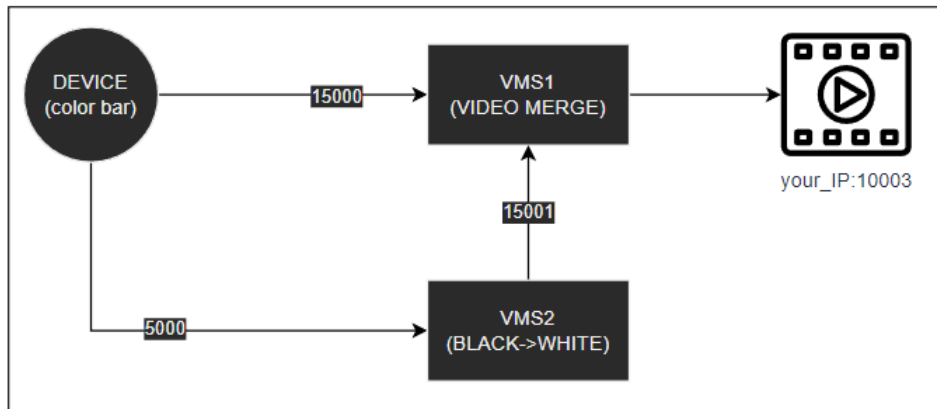


Figure 7.6: Multimedia service chain that bundles two video streams - Task 2.

- **Task 3:** Add to task 2 a different video stream (video ball), group with the result of task 2, and finally publish this video stream on a computer within the network through port 10004; refer to Figure 7.7.
- **Task 4:** Replicate task 3 and publish this video stream on a computer within the network through port 10005; refer to Figure 7.8.

The tasks are organized as shown in Table 7.4 for two participants PA and PB.



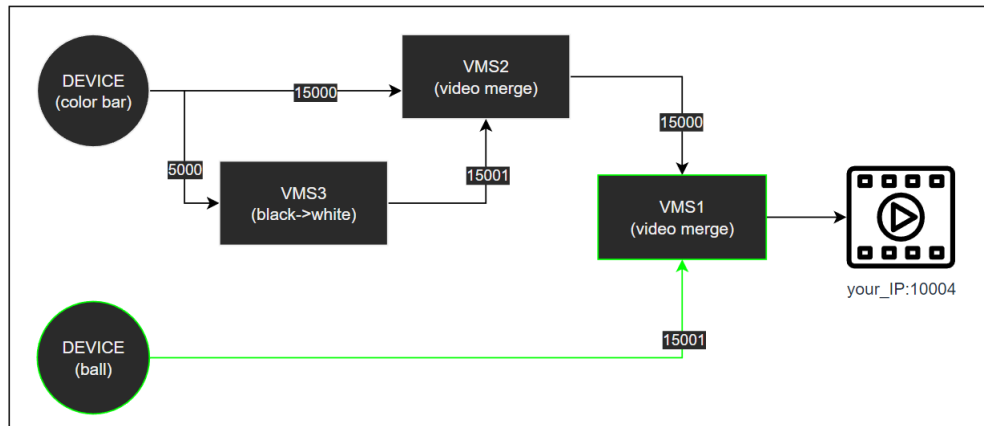


Figure 7.7: Complex multimedia service chain - Task 3.

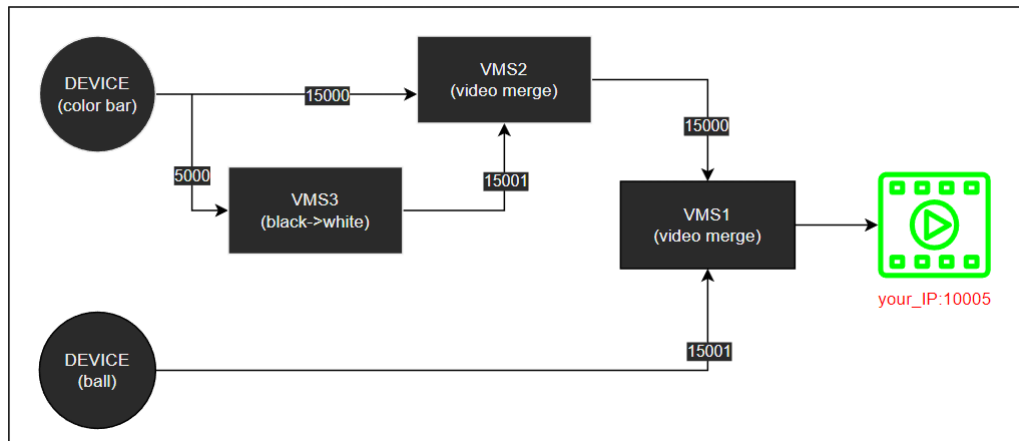


Figure 7.8: Reuse of multimedia service chains - Task 4.

Table 7.4: Tasks of the experiment

<i>Participants</i>	<i>FIRST</i>		<i>LAST</i>	
	L-PRISM	<i>TRADITIONAL</i> / <i>V-PRISM</i>	L-PRISM	<i>TRADITIONAL</i> / <i>V-PRISM</i>
PA	T1, T2, T3, T4	-	-	T1, T2, T3, T4
PB	-	T1, T2, T3, T4	T1, T2, T3, T4	-

Different tasks were used to answer G1 and G2 questions, as depicted in Tables 7.5 and 7.6.

## 7.3 Subjects

The profile of the participants involved in the experiment includes (i) educational level; (ii) level of knowledge in the field of networks, multimedia systems, virtualization tools,

Table 7.5: Tasks used for Goal G1

Goals	Question	Metrics	Tasks
<b>G1</b>	Q1	M1	T1, T2, T3, T4
	Q2	M2	T1, T2, T3, T4
	Q3	M3	T1, T2, T3, T4
	Q4	M4	T1, T2, T3, T4
	Q5	M5	T3, T4
	Q6	M6	T2, T3, T4

Table 7.6: Tasks used for Goal G2

Goals	Question	Metrics	Tasks
<b>G2</b>	Q1	M1	T1, T2, T3, T4
	Q2	M2	T3, T4
	Q3	M3	T1, T2, T3, T4
	Q4	M4	T1, T2, T3, T4
	Q5	M5	T1, T2, T3, T4
	Q6	M6	T1, T2, T3, T4
	Q7	M7	T1, T2, T3, T4

and distributed systems; and (iii) level of knowledge in programming languages.

The participants are graduate and postgraduate students in computer science or related to the computing area, organized as mentioned above; they have experience from introductory to advanced in the profile mentioned above. Additionally, all participants were trained with previously developed materials and signed a confidentiality and consent form that explains the experiment procedure.

Ten subjects participated in the experiments. They were computer science developers aged 26-39 years, 9 were men, and 1 was a woman, as shown in Figure 7.9 (a). Their academic degree was 20% undergraduate, 30% graduate, and 50% post-graduate, as shown in Figure 7.10. About your level of experience with the languages in XML, JSON and YAML (from 1 (no experience) to 5 (a lot of experience)). Median answers were 4 for XML, 5 for JSON, and 3 for YAML, as shown in Figure 7.11. We built a testbed with edge nodes in the lab to run the experiments.

Three subjects did it physically at the lab and seven subjects did it remotely. All of them were observed all the time during the experiment, even the remote ones.

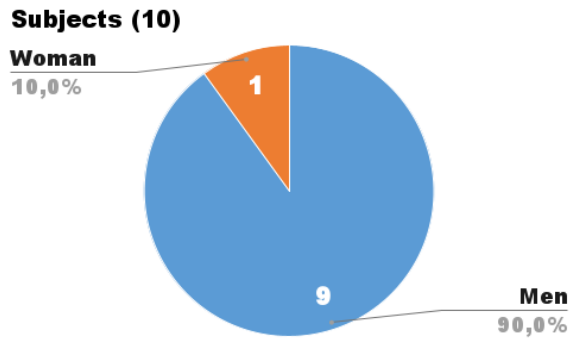


Figure 7.9: Number of subjects.

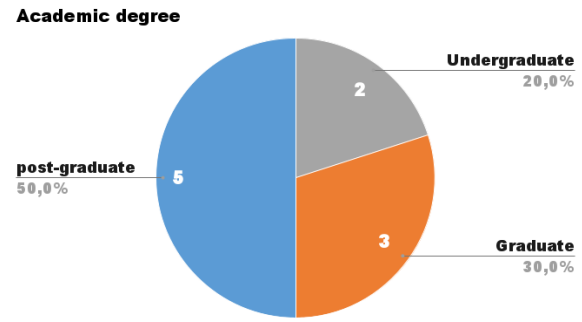


Figure 7.10: Academic level.

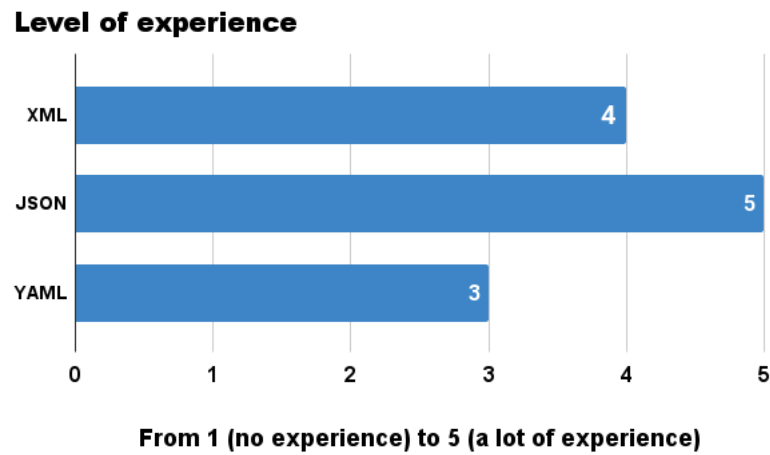


Figure 7.11: Experience in the use of XML, JSON, and YAML languages.

## 7.4 Results

### 7.4.1 G1 - Evaluation

Figure 7.12 will help us answer questions  $Q1$  and  $Q6$  of  $G1$ ; the metrics used for these questions are related to time. As shown in Figure 7.12, the average time in minutes with a confidence interval of 90% to perform the tasks with L-PRISM is presented by blue bars. The average times in minutes with a confidence interval of 90% to carry out the tasks with the traditional method (V-PRISM) are presented by orange bars. Table 7.5 presents the relationship of the questions, metrics, and tasks for  $G1$ , together with Figure 7.12, provides us with the necessary information to analyze the responses to questions  $Q1$  and  $Q6$ .

- For question  $Q1$  of  $G1$ , the *development effort* metric was used, and the results for this response depend on four Tasks 1, 2, 3 and 4. As seen in Figure 7.12, the times in

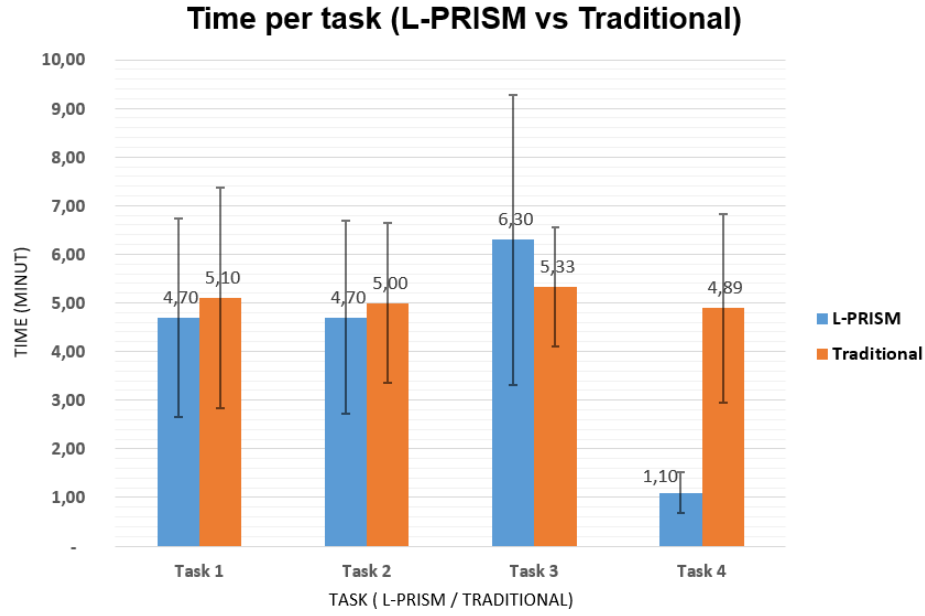


Figure 7.12: Time needed per task in the two methods (*L-PRISM* and *Classic method (V-PRISM)*)

minutes to perform the tasks with L-PRISM are similar to those of the traditional method (V-PRISM), so it can be concluded that both methods are equivalent in the development effort. This result validates the null hypothesis of Q2 defined in Section 7.1.1.2; this means that "The use of L-PRISM is equivalent to the traditional development (V-PRISM) of VMS-based multimedia service chains in terms of development time."

- For question Q6 of G1, the perceived reuse metric was used, and the results for this answer depend on Task4. As shown in Figure 7.12, the time in minutes to perform task 4 with L-PRISM (average 1.10 minutes) is notably smaller than the traditional method (V-PRISM) (average 4.89 minutes), so it can be concluded that L-PRISM is superior in relation to perceived reuse. This result validates the alternative hypothesis of Q2 defined in Section 7.1.1.2; this means that "The time required to modify a chain of multimedia services with L-PRISM is less than the traditional method (V-PRISM)."

In Figure 7.13, each colored region represents the percentage of subjects who chose the option indicated at the bottom of this figure. For a better analysis of the results, the types of responses for questions Q2 to Q5 of G1 will be evaluated by values of 1 to 5, where responses range from "StronglyDisagree = 1" to "StronglyAgree = 5". Additionally, Table 7.5 presents the relationship between the questions and metrics of G1; this, related

to the results of the answers presented in Figure 7.13 leads us to a brief discussion for each of the questions mentioned above.

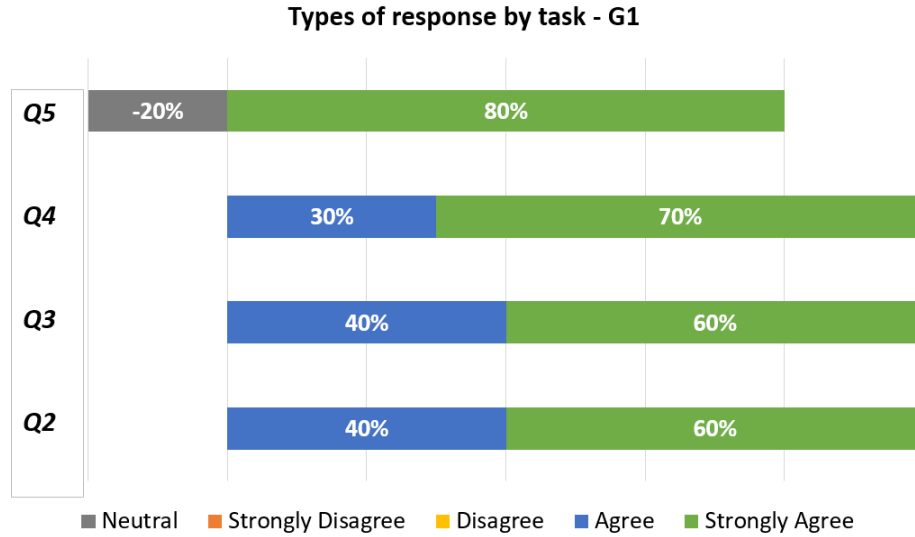


Figure 7.13: Summary of responses to G1 questions Q2-Q5

- For question  $Q2$  of  $G1$ , it can be seen that 60% of the subjects Strongly Agree (5) and 40% Agree (4). The value of the mode for  $Q2$  is ( $M_O = 5$ ); this means that most of the subjects Strongly Agree with this question. The value of the median is ( $M_d = 5$ ), which means that the subjects Strongly Agree with  $Q2$ . The results of  $Q2$  lead us to validate the alternative hypothesis of  $Q2$  defined in Section 7.1.1.2; this means that *"L-PRISM facilitates the understanding of the functional and non-functional requirements of VMS-based multimedia service chains."*
- For question  $Q3$  of  $G1$ , it can be seen that 60% of the subjects Strongly Agree (5) and 40% Agree (4). The value of the mode for  $Q3$  is ( $M_O = 5$ ); this means that most of the subjects Strongly Agree with this question. The value of the median is ( $M_d = 5$ ), which means that the subjects fully agree with  $Q3$ . The results of  $Q3$  lead us to validate the alternative hypothesis of  $Q3$  defined in Section 7.1.1.2; this means that *"L-PRISM is more productive than the traditional V-PRISM method, to develop chains of multimedia services based on VMS."*
- For question  $Q4$  of  $G1$ , it can be seen that 70% of the subjects Strongly Agree (5) and 30% Agree (4). The value of the mode for  $Q4$  is ( $M_O = 5$ ); this means that most of the subjects Strongly Agree with this question. The value of the median is ( $M_d = 5$ ), which means that the subjects fully agree with  $Q4$ . The results of  $Q4$  lead us to validate the alternative hypothesis of  $Q4$  defined in Section 7.1.1.2; this

means that *"L-PRISM is more useful than the traditional method (V-PRISM) to develop chains of multimedia services based on VMS."*

- For question Q5 of G1, it can be seen that 80% of the subjects Strongly Agree (5) and 20% are Neutral (3). The value of the mode for Q5 is ( $M_O = 5$ ); this means that most of the subjects Strongly Agree with this question. The median value for Q5 is ( $M_d = 5$ ), which means that the subjects Strongly Agree with Q5. The results of Q5 lead us to validate the alternative hypothesis of Q5 defined in Section 7.1.1.2; this means that *"The reuse of multimedia service chains already developed with L-PRISM is less complex compared to the traditional method (V-PRISM)."*

Based on the results obtained for questions Q1 and Q6 of G1, which are directly related to the *productivity* perspective, it can be seen that our L-PRISM proposal is slightly better than the traditional method when it comes to creating simple multimedia service chains for the first time. When it comes to creating complex multimedia service chains for the first time, it is a bit worse. However, in creating a complex multimedia service chain for the second or more time L-PRISM is utterly superior to the traditional method. So we can conclude that L-PRISM is more productive the more it is used.

Regarding the results obtained through the metrics defined for questions Q2 to Q5 related to the *efficiency* of L-PRISM, a median of 5 ( $M_d = 5$ ) was obtained in all these cases, which shows that L-PRISM is excellent from the point of view of efficiency. Then it is possible to conclude that the goal G1 was achieved.

### 7.4.2 G2 - Evaluation

In Figure 7.14, each colored region represents the percentage of subjects who chose the option indicated at the bottom of this figure. For a better analysis of the results, the types of responses for questions Q1 to Q7 of G2 will be evaluated by values of 1 to 5, where responses range from *StronglyDisagree* = 1 to *StronglyAgree* = 5. Additionally, Table 7.6 presents the relationship between the questions and G2 metrics; this, related to the results of the answers presented in Figure 7.14 leads us to a brief discussion for each of the questions mentioned above.

- For question Q1 of G2, it can be seen that 20% of the subjects Strongly Agree (5), 70% Agree (4), and 10% Neutral (3). The value of the mode for Q1 of G2 is ( $M_O = 4$ ); this means that most of the subjects Agree with this question. The

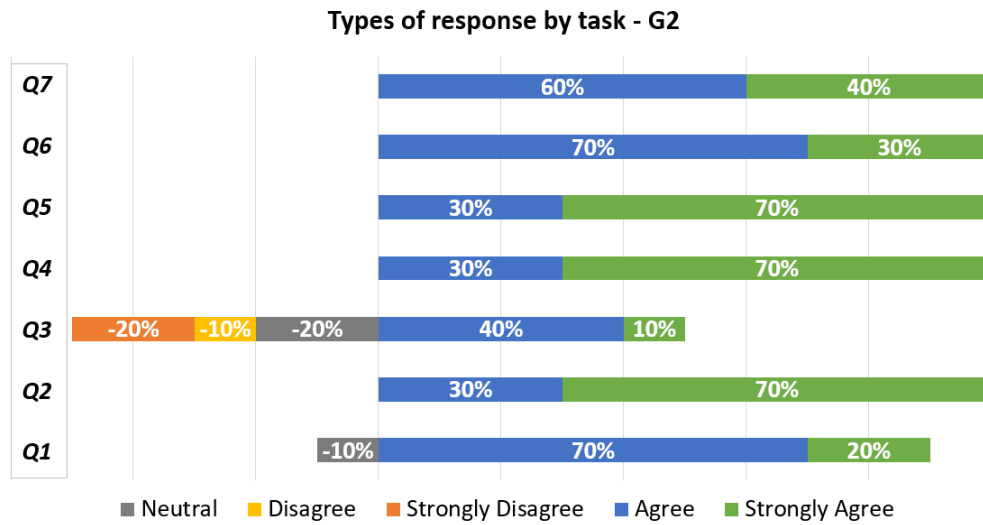


Figure 7.14: Summary of responses to G2 questions

median value for  $Q1$  of  $G2$  is ( $M_d = 4$ ), which means that the subjects Agree with  $Q1$ . The results of  $Q1$  lead us to validate the alternative hypothesis of  $Q1$  defined in Section 7.1.2.2; this means that *"L-PRISM components are easy to find when creating or modifying a multimedia application."*

- For question  $Q2$  of  $G2$ , it can be seen that 70% of the subjects Strongly Agree (5) and 30% Agree (4). The value of the mode for  $Q2$  of  $G2$  is ( $M_O = 5$ ); this means that the majority of the subjects Strongly Agree with this question. The median value for  $Q2$  of  $G2$  is ( $M_d = 5$ ), which means that the subjects Strongly Agree with  $Q2$ . The results of  $Q2$  lead us to validate the alternative hypothesis of  $Q2$  defined in Section 7.1.2.2; this means that *"Changes to existing applications made with L-PRISM are easy to make."*
- For question  $Q3$  of  $G2$ , it can be seen that 10% of the subjects Strongly Agree (5), 40% Agree (4), 20% Neutral (3), 10% Disagree (2) and 20% Strongly Disagree (1). The value of the mode for  $Q3$  of  $G2$  is ( $M_O = 4$ ); this means that most of the subjects Agree with this question. The median value for  $Q3$  of  $G2$  is ( $M_d = 3$ ), which means that the subjects are neutral concerning  $Q3$ . The results of  $Q3$  lead us to validate the null hypothesis of  $Q3$  defined in Section 7.1.2.2; this means that *"The data types and structures in L-PRISM are too extensive to describe."*
- For question  $Q4$  of  $G2$ , it can be seen that 70% of the subjects Strongly Agree (5) and 30% Agree (4). The value of the mode for  $Q4$  of  $G2$  is ( $M_O = 5$ ); this means that the majority of the subjects Strongly Agree with this question. The

median value for  $Q4$  of  $G2$  is ( $M_d = 5$ ), which means that the subjects Strongly Agree with  $Q4$ . The results of  $Q4$  lead us to validate the alternative hypothesis of  $Q4$  defined in Section 7.1.2.2; this means that *"The elements of L-PRISM have a simple description."*

- For question  $Q5$  of  $G2$ , it can be seen that 70% of the subjects Strongly Agree (5) and 30% Agree (4). The value of the mode for  $Q5$  of  $G2$  is ( $M_O = 5$ ); this means that the majority of the subjects Strongly Agree with this question. The median value for  $Q5$  of  $G2$  is ( $M_d = 5$ ), which means that the subjects Strongly Agree with  $Q5$ . The results of  $Q5$  lead us to validate the alternative hypothesis of  $Q5$  defined in Section Section 7.1.2.2; this means that *"The data types and structures in L-PRISM are easy to understand."*
- For question  $Q6$  of  $G2$ , it can be seen that 30% of the subjects Strongly Agree (5) and 70% Agree (4). The mode value for  $Q6$  of  $G2$  is ( $M_O = 4$ ); this means that most subjects Agree with this question. The median value for  $Q6$  of  $G2$  is ( $M_d = 4$ ), which means that the subjects Agree with  $Q6$ . The results of  $Q6$  lead us to validate the alternative hypothesis of  $Q6$  defined in Section 7.1.2.2; this means that *"The dependencies between components or data types in L-PRISM are easy to recognize."*
- For question  $Q7$  of  $G2$ , it can be seen that 40% of the subjects Strongly Agree (5) and 60% Agree (4). The mode value for  $Q7$  of  $G2$  is ( $M_O = 4$ ); this means that most subjects Agree with this question. The median value for  $Q7$  of  $G2$  is ( $M_d = 4$ ), which means that the subjects Agree with  $Q7$ . The results of  $Q7$  lead us to validate the alternative hypothesis of  $Q7$  defined in Section 7.1.2.2; this means that *"L-PRISM at a general level is easy to understand."*

About the results obtained through the metrics defined for the  $G2$  questions, related to the *usability* of L-PRISM, median results were obtained ( $Q1 = 4$ ,  $Q2 = 5$ ,  $Q3 = 3$ ,  $Q4 = 5$ ,  $Q5 = 5$ ,  $Q6 = 4$ ,  $Q7 = 4$ ), with the mode of the medians equal to 4, 5 and the median of the medians equal to 4. This shows that L-PRISM is between good and excellent from the *usability* point of view, then it is possible to conclude that the goal  $G2$  was also achieved.

### 7.4.3 Threats to Validity

Within our results, we can observe that  $Q3$  of  $G2$  did not have the results we expected, but it was also observed that the results are not inclined to accept or reject, as shown



in Figure 7.15. This led us to conduct a small investigation with participants about this question.

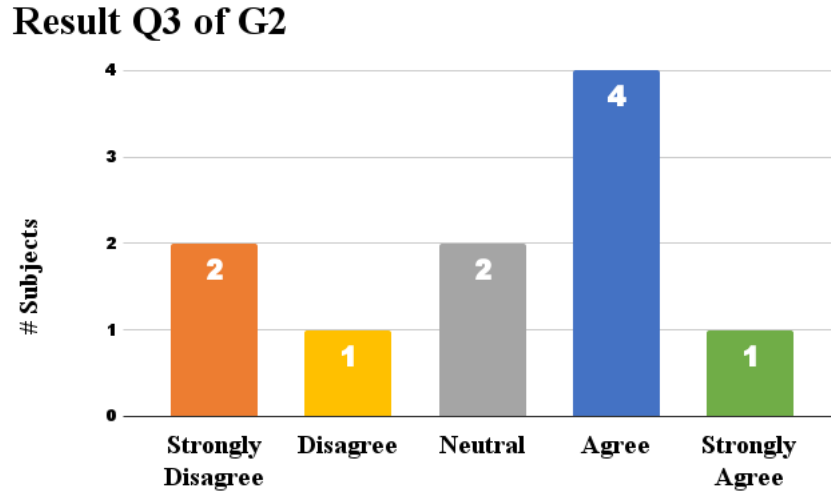


Figure 7.15: Results of question Q3 of objective G2

The results are due to two reasons. The first is that the question was poorly posed; what was really wanted to ask is:

- Is the L-PRISM language verbose to specify a multimedia services chain?

However, what was asked was:

- Is the L-PRISM language **too** verbose to specify a multimedia services chain?

The second reason is that the questionnaire was carried out in Portuguese, and the participating subjects have Spanish and Portuguese as their mother tongue. For those who have Portuguese as their mother tongue, they did not have any doubts when answering the questionnaire, but those whose mother tongue is Spanish asked what *verbose* means in the question:

- *Q3 of G2: A linguagem L-PRISM é muito verbosa para especificar uma cadeia de serviços multimídia?*

Which it was answered that the question in English was:

- Is the L-PRISM language verbose to specify a multimedia services chain?

Some other observations on our validation are mentioned below.

- The L-PRISM evaluation included the participation of 10 subjects, which could be considered a small sample size. Therefore, the results obtained may not be fully supported by the sample evaluated.
- It is important to note that the majority of the participants in the evaluation of L-PRISM were friends of the researcher, which could have introduced bias into the results. It is possible that the researcher's friends share similar opinions, which may have influenced their evaluations of L-PRISM and led to more favorable results than would have been obtained with a more diverse and representative sample.

#### 7.4.4 Final considerations

As can be seen, the results for the questions that expect positive responses are as expected, and for  $Q3$  of  $G2$  the results are somewhat ambiguous; a subsequent consultation was made with the participants asking about this result, and the conclusion was that the question was not very clear, hence the disparity in the responses.

This chapter presented the methodology used to validate L-PRISM. The next chapter presents the conclusions, main contributions, limitations, and future work.

# Chapter 8

## Conclusion

Unlike general programming languages, such as Java, C++, or Python, which can be used to develop various applications, Domain Specific Languages (DSL) are designed to solve specific problems or tasks within a framework in a particular domain. Templates are another tool that offers similar solutions to DSL since they allow instantiating applications simply using a pre-defined schema. Usually, a template describes the application components and the relationships between them. Another tool similar to DSL and templates is data models, which are used to describe the structure and relationships between data in an application or system. Data models facilitate the design of applications or systems.

In general, there is little effort in the literature to help create multimedia service chains based on VMS. None of the DSL, templates, and data models found in the literature address the concepts that are used for the specification and orchestration of multimedia service chains based on VMS. One of the languages that most resemble our objective is TOSCA-NFV.

TOSCA-NFV is a declarative modeling language that allows describing cloud applications and services in a standardized way. TOSCA-NFV extends the functionalities of the TOSCA specification to allow the description and orchestration of services based on NFV. TOSCA-NFV was modeled to work in the cloud, and although it could be used in Edge Computing environments, it would not be the most appropriate since its specification is based on virtualization based on hypervisors, and this type of virtualization requires a complete operating system to be installed inside a virtual machine, which generates a higher use of computational resources. TOSCA-NFV also bases its specification to create services on NFV, and its specification focuses on the behavior of network functions and does not address multimedia applications or services.

This thesis introduces L-PRISM, a DSL based on YAML, for creating multimedia service chains based on VMS. Our DSL describes the main data types, capabilities, artifacts, and relationships that are part of the structure of the multimedia service chain based on VMS. L-PRISM bases its specification on the concepts of Internet of Media Thing (IoMT), lightweight virtualization (containerization), and Edge computing, aiming to specify and orchestrate multimedia service chains based on VMS.

In order to validate L-PRISM, the ALFA implementation was extended, which we call ALFA 2.0. The main objective of this extension is to allow YAML files that contain the specification of a multimedia service chain based on our L-PRISM, to be loaded by a web interface and sent through an API to a service that will be in charge of orchestrating the elements of this multimedia service chain (VD, VMS, VL), to register this multimedia service chain within a database finally.

ALFA 2.0 is released under an open-source license and can be accessed on GitHub<sup>1</sup>. Additionally, other features were implemented in ALFA 2.0, as:

- Adaptation of the main elements of ALFA (VD, typeVMS, and VMS) so that developers who use L-PRISM can use these elements without any problem.
- A service that retrieves from the database the necessary information to specify a multimedia service chain (list of VMS types (type VMS), list of host (Nodes), and list of virtual devices (VD)).
- A module that creates multimedia service chains through an intuitive interface was used as proof of concept in the early stages to validate that the V-PRISM architecture would support L-PRISM.

The Goal Question Metric (GQM) method was used to validate L-PRISM. Firstly, two goals were defined ( $G1$  and  $G2$ ), and for each goal, a group of questions related to a group of metrics was proposed. The questions for  $G1$  use Technology Acceptance Model (TAM) metrics, and the questions for  $G2$  use Cognitive Dimensions of Notations (CDN) metrics.

A group of developers carried out the experiment stage; this stage was divided into two phases. The first phase is the training phase, where the group of developers had to be trained in how to create multimedia service chains based on VMS, with L-PRISM

---

<sup>1</sup><https://github.com/fventuraq/alfa>

and a traditional method (simulating the deployment of each VMS by console using V-PRISM), in order to be able to train, a website<sup>2</sup> was set up that contains information about L-PRISM, implementation examples with L-PRISM and V-PRISM. The second phase is to perform four tasks. Each task is to create a multimedia service chain based on VMS with L-PRISM and the traditional method (V-PRISM). At the end of the tasks, the participants had to complete a questionnaire containing the questions *G1* and *G2*.

The results obtained for both objectives were promising. For *G1*, it can be concluded that L-PRISM is excellent from the point of view of *efficiency*, and in the case of *productivity*, it was shown that L-PRISM is slightly superior in most cases. For *G2*, it could be concluded that L-PRISM is good from the *usability* point of view.

## 8.1 Contributions

In summary, the main contributions achieved in this thesis were:

- Design and develop L-PRISM language for creating multimedia service chains based on VMS.
- Integrate L-PRISM into the V-PRISM architecture with the extension of ALFA to ALFA 2.0.
- Evaluate L-PRISM comparing to a traditional method for creating multimedia service chains with V-PRISM.

## 8.2 Future work

One future work is developing a framework that allows viewing multimedia service chains created with L-PRISM. This framework could help to make the solutions implemented with our proposal more understandable. Additionally, it could also be used to design multimedia service chains and export them in our language.

One of the characteristics of using virtualization is that access to virtualized application monitoring information is not very complex, which would allow capturing different types of information. As future work, we propose to create an API that allows capturing

---

<sup>2</sup><https://fventuraq.github.io/lprism.html>

and storing the operational data of multimedia service chains and their components, intending to create a database that can be used for investigations related to the operation or behavior of multimedia applications in environments virtualized.

One of the characteristics of virtualized multimedia applications is that the amount of resources assigned to them is not always the most appropriate for their operation. The use of machine learning techniques can be very useful in environments where virtualized multimedia applications work since the number of resources assigned to these applications could be self-managed by an AI, thanks to the fact that virtualization makes access to monitoring data easier. We also intend to integrate L-PRISM to the 5G architecture and extend the L-PRISM specification to support a different application domain, besides IoMT.

# References

- [1] ALAM, I.; SHARIF, K.; LI, F.; LATIF, Z.; KARIM, M. M.; BISWAS, S.; NOUR, B.; WANG, Y. A Survey of Network Virtualization Techniques for Internet of Things Using SDN and NFV. *ACM Computing Surveys* 53, 2 (Mar. 2021), 1–40.
- [2] ALLOUCHE, M.; MITREA, M.; MOREAUX, A.; KIM, S.-K. Automatic smart contract generation for internet of media things. *ICT Express* 7, 3 (2021), 274–277.
- [3] ALVAREZ, F.; BREITGAND, D.; GRIFFIN, D.; ANDRIANI, P.; RIZOU, S.; ZIOULIS, N.; MOSCATELLI, F.; SERRANO, J.; KELTSCH, M.; TRAKADAS, P., ET AL. An edge-to-cloud virtualized multimedia service platform for 5g networks. *IEEE Transactions on Broadcasting* 65, 2 (2019), 369–380.
- [4] BARAKABITZE, A. A.; AHMAD, A.; MIJUMBI, R.; HINES, A. 5g network slicing using sdn and nfv: A survey of taxonomy, architectures and future challenges. *Computer Networks* 167 (2020), 106984.
- [5] BARAKABITZE, A. A.; BARMAN, N.; AHMAD, A.; ZADTOOTAGHAJ, S.; SUN, L.; MARTINI, M. G.; ATZORI, L. Qoe management of multimedia streaming services in future networks: a tutorial and survey. *IEEE Communications Surveys & Tutorials* 22, 1 (2019), 526–565.
- [6] BATTISTI, A.; MUCHALUAT-SAADE, D. C.; DELICATO, F. C. V-PRISM: An edge-based iot architecture to virtualize multimedia sensors. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)* (2020).
- [7] BJÖRKLUND, M. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020, Oct. 2010.
- [8] BLACKWELL, A. F.; GREEN, T. R. A cognitive dimensions questionnaire optimised for users. In *PPIG* (2000), vol. 13, Citeseer.
- [9] BOUDI, A.; FARRIS, I.; BAGAA, M.; TALEB, T. Assessing lightweight virtualization for security-as-a-service at the network edge. *IEICE Transactions on Communications* 102, 5 (2019), 970–977.
- [10] BRUNNSTRÖM, K.; BEKER, S. A.; DE MOOR, K.; DOOMS, A.; EGGER, S.; GARCIA, M.-N.; HOSSFELD, T.; JUMISKO-PYYKKÖ, S.; KEIMEL, C.; LARABI, M.-C., ET AL. Qualinet white paper on definitions of quality of experience.
- [11] CASTILLO-LEMA, J.; VENÂNCIO NETO, A.; DE OLIVEIRA, F.; TAKEO KOFUJI, S. Mininet-nfv: Evolving mininet with oasis toasca nvf profiles towards reproducible nvf prototyping. In *2019 IEEE Conference on Network Softwarization (NetSoft)* (2019), pp. 506–512.

- [12] CELESTI, A.; MOLFARI, D.; GALLETTA, A.; FAZIO, M.; CARNEVALE, L.; VILLARI, M. A study on container virtualization for guarantee quality of service in Cloud-of-Things. *Future Generation Computer Systems* 99 (2019), 356–364.
- [13] CHIOSI, M.; CLARKE, D.; WILLIS, P.; REID, A.; FEGER, J.; BUGENHAGEN, M.; KHAN, W.; FARGANO, M.; CUI, C.; DENG, H., ET AL. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow world congress* (2012), vol. 48, sn, pp. 1–16.
- [14] DAVIS, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly* (1989), 319–340.
- [15] DOCKER. What is a Container?, 2020. <https://www.docker.com/resources/what-container>, Accessed: Mar 25, 2023.
- [16] ETSI. MEC 003 - V2.1.1 - Multi-access Edge Computing (MEC); Framework and Reference Architecture. Tech. rep., ETSI, Valbonne/França, 2019.
- [17] GRIGORIOU, E.; BARAKABITZE, A. A.; ATZORI, L.; SUN, L.; PILLONI, V. An sdn-approach for qoe management of multimedia services using resource allocation. In *2017 IEEE International Conference on Communications (ICC)* (2017), pp. 1–7.
- [18] IEEE COMMUNICATIONS SOCIETY. *IEEE Std 1934-2018 : IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing*. IEEE, 2018.
- [19] IMAGANE, K.; KANAI, K.; KATTO, J.; TSUDA, T.; NAKAZATO, H. Performance evaluations of multimedia service function chaining in edge clouds. In *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)* (2018), IEEE, pp. 1–4.
- [20] ISO. Iso/iec 23093-1:2022 information technology — internet of media things — part 1: Architecture, 2022. Accessed: Mar 25, 2023.
- [21] IUNG, A.; CARBONELL, J.; MARCHEZAN, L.; RODRIGUES, E.; BERNARDINO, M.; BASSO, F. P.; MEDEIROS, B. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering* 25 (2020), 4205–4249.
- [22] JAN, M. A.; USMAN, M.; HE, X.; UR REHMAN, A. SAMS: A seamless and authorized multimedia streaming framework for WMSN-Based IoMT. *IEEE Internet of Things Journal* 6, 2 (2019), 1576–1583.
- [23] KARAAGAC, A.; DALIPI, E.; CROMBEZ, P.; DE POORTER, E.; HOEBEKE, J. Light-weight streaming protocol for the Internet of Multimedia Things: Voice streaming over NB-IoT. *Pervasive and Mobile Computing* 59 (2019), 101044.
- [24] KIM-HUNG, L.; DATTA, S. K.; BONNET, C.; HAMON, F.; BOUDONNE, A. A scalable IoT framework to design logical data flow using virtual sensor. *International Conference on Wireless and Mobile Computing, Networking and Communications 2017-October*, ii (2017).
- [25] KOZIOLEK, H. Goal, question, metric. In *Dependability metrics*. Springer, 2008, pp. 39–42.



- [26] LIKERT, R. A technique for the measurement of attitudes. *Archives of psychology* (1932).
- [27] MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.
- [28] MOURADIAN, C.; EBRAHIMNEZHAD, F.; JEBBAR, Y.; AHLUWALIA, J. K.; AFRASIABI, S. N.; GLITHO, R. H.; MOGHE, A. An IoT Platform-as-a-Service for NFV Based-Hybrid Cloud/Fog Systems. *IEEE Internet of Things Journal* 4662, c (2020), 1–1.
- [29] NAUMAN, A.; QADRI, Y. A.; AMJAD, M.; ZIKRIA, Y. B.; AFZAL, M. K.; KIM, S. W. Multimedia internet of things: A comprehensive survey. *IEEE Access* 8 (2020), 8202–8250.
- [30] NEGM, E.; MAKADY, S.; SALAH, A. Survey on domain specific languages implementation aspects. *International Journal of Advanced Computer Science and Applications* 10, 11 (2019).
- [31] OPENSTACK. Welcome to the heat documentation! — openstack-heat 19.1.0.dev30 documentation. url <https://docs.openstack.org/heat/latest/>, May 2021.
- [32] PATTARANANTAKUL, M.; HE, R.; ZHANG, Z.; MEDDAHI, A.; WANG, P. Leveraging network functions virtualization orchestrators to achieve software-defined access control in the clouds. *IEEE Transactions on Dependable and Secure Computing* 18, 1 (2021), 372–383.
- [33] PETROLO, R.; MORABITO, R.; LOSCRÌ, V.; MITTON, N. The design of the gateway for the cloud of things. *Annals of Telecommunications* 72 (2017), 31–40.
- [34] SCHÖNWÄLDER, J.; BJÖRKLUND, M.; SHAFER, P. Network configuration management using netconf and yang. *IEEE communications magazine* 48, 9 (2010), 166–173.
- [35] SURENDRAN, P., ET AL. Technology acceptance model: A survey of literature. *International Journal of Business and Social Research* 2, 4 (2012), 175–178.
- [36] TOSCA, O. Tosca simple profile for network functions virtualization (nfv) version 1.0, committee specification draft 04, 2017.
- [37] TOSCA, O. Tosca simple profile for network functions virtualization, 2020.
- [38] VILLARI, M.; CELESTI, A.; TRICOMI, G.; GALLETTA, A.; FAZIO, M. Deployment orchestration of microservices with geographical constraints for edge computing. In *2017 IEEE Symposium on Computers and Communications (ISCC)* (2017), pp. 633–638.
- [39] YAML. Yaml 1.2. especificación completa. <https://yaml.org/spec/1.2/spec.html>. Accessed: Mar 25, 2023.
- [40] ZIKRIA, Y. B.; AFZAL, M. K.; KIM, S. W. Internet of Multimedia Things (IoMT): Opportunities, Challenges and Solutions. *Sensors (Basel, Switzerland)* 20, 8 (2020), 1–8.

- 
- [41] ZIKRIA, Y. B.; AFZAL, M. K.; KIM, S. W. Internet of multimedia things (iomt): Opportunities, challenges and solutions, 2020.
  - [42] ÉDEN BATTISTI, A. L. *V-PRISM: An Edge-Based Architecture to Virtualize Multimedia sensors in the Internet of Media Things*. Tese de Doutorado, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Julho 2020.