Politecnico di Torino

Master of Science in Cybersecurity

A.a. 2024/2025

Graduation Session December 2025

A NILE-to-VEREFOO Translator for Intent-Based Network Security Automation

Supervisors:

Prof. Riccardo Sisto

Prof. Fulvio Valenza

Prof. Daniele Bringhenti

Candidate:

Mikhael Russo

This thesis was done in collaboration with IMT Atlantique University, under the supervision of professors Guillaume Doyen, Pierre Alain, and Fabien Autrel.

Abstract

The rapid evolution of network infrastructures, driven by paradigms such as Software-Defined Networking (SDN), Network Function Virtualization (NFV), cloud computing, and the Internet of Things (IoT), has significantly increased the complexity of configuration and management tasks. In this context, Intent-Based Networking (IBN) introduces a paradigm shift: it allows operators to express high-level objectives, intents, without specifying their technical implementation. However, a key challenge remains the correct and verifiable translation of these intents into enforceable configurations, especially in the security domain.

This thesis investigates this translation problem by focusing on the interoperability between NILE (Network Intent LanguagE), an intermediate and human-readable intent language, and Verefoo, a framework developed at Politecnico di Torino for the automation and formal verification of network security policies. After a comparative analysis of the semantics, input, and output models of the two systems, a translation model is proposed to map NILE constructs into Verefoo's XML-based representation. A prototype translator has been implemented to automate this process, enabling the conversion of NILE intents into valid Verefoo input files. The tool has been validated through a series of case studies on different network topologies.

The results confirm the feasibility of semantic translation as a bridge between intent-based specification and automated verification frameworks. This work represents a step toward the development of intent-driven network security systems that are more accessible, reliable, and less dependent on specialized expertise, paving the way for future research on AI-assisted intent interpretation and adaptive security automation.

Acknowledgements

First of all, I would like to express my gratitude to my supervisors, Professor Riccardo Sisto, Professor Fulvio Valenza, and Professor Daniele Bringhenti, for their invaluable guidance, constructive advice, and constant support throughout the development of this thesis. I am also grateful to Professors Guillaume Doyen, Pierre Alain, and Fabien Autrel from IMT Atlantique for their support and valuable advice.

A special thanks goes to Professor Andrea Nardin, who gave us the great opportunity and support to write our first research paper, to me and my colleagues Cristian Carallo and Riccardo Tommasi. Writing this paper was a fundamental experience that allowed me to apply the knowledge I had acquired in a practical way and to engage directly with the world of scientific research, learning the importance of collaboration, critical review, and effective communication of results. Moreover, Prof. Nardin not only gave me the chance to write the paper, but also allowed me to present it and participate in an international conference, interacting with experts in the field, an invaluable experience that greatly enriched my academic growth.

I would also like to warmly thank my colleagues and friends from the master's program: Cristian Carallo, Carlo Cimino, Simone D'Addio, Simone Di Nucci, Lorenzo Ferretti, Alessio Mantineo, Francesco Marrapodi, Alessio Palermo, Roberto Previtali, Ming Su, and Riccardo Tommasi, for their collaboration, stimulating discussions, and practical help, which made this journey both enriching and inspiring.

I wish to express my deepest and most sincere gratitude to my family, who have been my pillar throughout these years. In particular, I owe everything to my mother: without her, I would not be here today. She has supported and tolerated me through every challenge and every stage of my academic journey, offering unconditional love, encouragement, and patience. Her guidance and belief in me have been fundamental, and I am forever grateful. To my entire family, thank you for sharing every moment of joy and difficulty with me; your presence made this achievement possible.

A special thanks also goes to Alessandro Rossini and his sister Lucrezia Rossini for their help and support: they stood by me through the highs and lows of this

journey, always lifting my spirits. Affectionate thoughts also go to long-time friends such as Carlo Vergano, who, even when I had disappeared off the radar, never gave up and kept inviting me out, keeping our friendship alive.

I am deeply grateful to the Politecnico, which demanded much but gave even more in return: invaluable experiences, opportunities, and the chance to grow both personally and professionally. Among these, joining the Politecnico Public Speaking team was particularly meaningful. I would like to thank my teammates Martina Mastroianni, Giuliano Agostini, Edoardo Bona, Lucy Luparelli, and Milica Djoric for their collaboration, support, and for making this experience so memorable. I am also grateful to all the other members of the team, whose presence and dedication contributed greatly to a wonderful and enriching experience.

Finally, I would like to thank all the people who, in different ways, contributed to the realization of this work: without their support and presence, reaching this goal would not have been possible.

To myself, I would like to remember the words of Worf from Star Trek: The Next Generation, in the episode "Coming of Age": *"Thinking about what you can't control will only waste energy and create its own enemy."* May this reminder inspire me to focus on action rather than worry, to manage my mental and physical resources wisely, and to accept uncertainty with courage and determination.

Table of Contents

List of Figures VII

1 Introduction 1
- 1.1 Thesis Objective . . . 2
- 1.2 Thesis Structure . . . 3

2 Intent-Based Networking: Concepts, Architecture, and Language Solutions 4
- 2.1 Introduction . . . 4
- 2.2 Intent based networking . . . 6
  - 2.2.1 Core Components . . . 6
  - 2.2.2 Role of AI in IBN . . . 7
  - 2.2.3 Intent Translation . . . 7
- 2.3 Intent-Based Languages . . . 9
  - 2.3.1 NILE . . . 10
- 2.4 Case Studies: IBN Architectures and Tools . . . 13
  - 2.4.1 Indira . . . 13
  - 2.4.2 Inspire . . . 15
  - 2.4.3 Polanco . . . 16
- 2.5 Conclusion . . . 17

3 Network Automation 19
- 3.1 Automation for Network Security Configuration . . . 19
- 3.2 VEREFOO . . . 21
  - 3.2.1 Purpose and Capabilities . . . 21
- 3.3 Conclusion . . . 23

4 Semantics, Inputs, and Outputs of NILE and VEREFOO 25
- 4.1 Overview . . . 25
- 4.2 NILE Semantics and Structure . . . 25
  - 4.2.1 Syntactic Definition of an Intent . . . 26

4.3 Operations 26
4.3.1 Mandatory 27
4.3.2 Optional 27
4.3.3 Targets 29
4.4 Considerations 30
4.5 VEREFOO Semantics and Architecture 30
4.5.1 Overview 30
4.5.2 Input 31
4.5.3 Output 34
4.5.4 Input-to-Output Processing in VEREFOO 36
4.6 Conclusion 37

5 NILE to VEREFOO Translation **38**
5.1 Introduction 38
5.2 Analysis of the Translation Requirements 39
5.2.1 Structure of NILE Intents 39
5.2.2 Requirements of VEREFOO 40
5.3 Challenges of Translation 42
5.3.1 Conflict-free 42
5.3.2 Information Gap 43
5.3.3 Grammar Modifications 44
5.3.4 Translation Rules 46
5.4 Translator Implementation 47
5.4.1 Translation Workflow 47
5.4.2 The `main.py` Module 48
5.4.3 The `transformer.py` Module 50
5.4.4 The `grammar.py` Module 50
5.4.5 The `enrich.py` Module 51
5.4.6 The `constructor.py` Module 52
5.4.7 The `topology_manager.py` Module 52
5.4.8 The `property.py` Module 53
5.4.9 The `utils.py` Module 54
5.4.10 The `xml_converter.py` Module 54
5.4.11 The `create_db.py` Module 55
5.5 Conclusions 56

6 Validation of the Translation between NILE and VEREFOO **57**
6.1 Introduction 57
6.1.1 Network Topology 58
6.2 Use Case 1: Validation of the `from/to` Construct 59
6.3 Use Case 2: Validation of the `for` Construct 62

6.4 Use Case 3: Validation with Multiple Intents Across Two Service Graphs . . . 68
6.5 Conclusions . . . 71

7 Conclusions and Future Works **73**
7.1 Future works . . . 74
7.1.1 Extension of the supported NILE grammar . . . 74
7.1.2 Automated enrichment and dynamic data retrieval . . . 74
7.1.3 Integration with AI-based intent interpretation . . . 75
7.2 Final Remarks . . . 76

Bibliography **77**

List of Figures

5.1 Structure of a NILE intent 39
5.2 VEREFOO requirements schema 40
5.3 VEREFOO NSR 41
5.4 Mapping between NILE and VEREFOO requirements 44
5.5 Translator workflow from NILE intent to XML input for VEREFOO. 49

6.1 Network topology used for the first and second validation use cases. 58
6.2 Extended network topology used for the third validation use case. . 58
6.3 Excerpt of the recognized and enriched intent in JSON 60
6.4 Excerpt of the enriched NILE intents. 64

Chapter 1

Introduction

Traditionally, network infrastructures were configured manually, with administrators directly setting parameters on individual devices. While this approach could be effective in small-scale environments, the emergence of modern technologies such as network function virtualization (NFV), software-defined networking (SDN), cloud computing, and the Internet of Things (IoT) has made it increasingly difficult. These paradigms have introduced unprecedented flexibility but also a dramatic rise in complexity, making manual configuration both error-prone and difficult to scale. Administrators, confronted with large and sophisticated systems, often struggle to maintain a complete overview of the network's operational state, thereby increasing the likelihood of misconfigurations that can lead to security vulnerabilities and critical operational failures.

In response to these challenges, Intent-Based Networking (IBN) has emerged as a promising paradigm. IBN shifts the focus from low-level, prescriptive configuration protocols to higher-level declarative specifications, known as *intents*. An intent represents the desired outcome of the network's behavior, often expressed in natural language or through domain-specific abstractions, without prescribing the exact steps needed for implementation. By decoupling high-level goals from low-level configurations, IBN simplifies network management, reduces configuration errors, and improves policy enforcement, ultimately lowering administrative costs and enhancing efficiency.

However, there are still several challenges. The first is the precise definition of intents, especially seeing the heterogeneity of network operators with different levels of experience. A second challenge is the conflict resolution among several, possibly overlapping, intents. Finally, even when intents are well-formed and consistent, automating their correct translation into enforceable configurations remains difficult, particularly given the limitations of the existing topologies and resources [1]. This last point is especially critical in the security domain, where errors in policy translation or enforcement can have severe consequences.

This thesis addresses one specific aspect of the broader IBN problem: the translation of intents expressed in NILE, a domain-specific language for network policy specification, into configurations enforceable by VEREFOO, a network verification and enforcement mechanism. By analyzing the correspondence between NILE's abstractions and semantics and the operational constraints of VEREFOO, this work identifies the mechanisms required to faithfully preserve the correctness of intents during translation.

The study presented here represents a foundational step toward more accessible and reliable intent-based security systems. While future research may explore the integration of AI-assisted tools for automated intent specification and enforcement, the current work establishes the groundwork by formalizing the NILE-to-VEREFOO translation process. This provides a rigorous basis for future extensions aimed at intelligent, scalable, and user-friendly intent-based security management.

1.1 Thesis Objective

The primary goal of this thesis is to address the problem of translating and refining intents expressed through the `NILE` language into configurations that can be applied and verified within `VEREFOO`, a framework for network policy automation and validation.

NILE is designed as an intermediate language that combines ease of understanding with expressiveness, aiming to serve as a tool usable both by network operators with limited knowledge of policy management and by experienced users. However, for an intent expressed in NILE to be effectively applied in a real network, an additional step is required: translation into tools capable of enforcing and executing these specifications. In this context, VEREFOO represents an advanced system that, through formal verification and optimization techniques, can generate correct and optimal configurations from declarative security requirements.

The objective of this thesis is therefore to define a structured method for mapping the semantic constructs of NILE to the operational mechanisms of VEREFOO, analyzing the similarities, differences, and potential ambiguities that arise during the translation process. In particular, this involves:

- Investigating which syntactic and semantic elements of NILE can be directly translated into equivalent constructs in VEREFOO;

- Identifying cases where such correspondence is not straightforward, requiring a refinement or transformation process;

- Assessing the proposed model's ability to preserve the semantic correctness of the original intent, avoiding information loss or misleading interpretations;

- Considering the potential benefits this translation offers in terms of reducing configuration errors, enhancing security, and simplifying network management processes.

This work thus aims to bridge the gap between high-level intent languages and network automation tools, laying the foundation for the development of intent-driven management systems that are accessible, reliable, and less dependent on advanced specialized expertise.

1.2 Thesis Structure

The structure of this thesis reflects a progressive path, starting from the analysis of fundamental theoretical concepts and leading up to the experimentation and validation of the proposed translation process.

- **Chapter 2 - Intent-Based Networking: Concepts, Architecture, and Language Solutions**: introduces Intent-Based Networking, its key components, intent translation, intent languages, and reviews some of the solutions used to study intent translation.
- **Chapter 3 - Network Automation**: analyzes the role of automation tools in the configuration and security management of modern networks. Special attention is given to VEREFOO, exploring its capabilities, input/output methods, and verification and allocation models.
- **Chapter 4 - Semantics, Input, and Output of NILE and VEREFOO**: compares the two systems, providing a detailed analysis of their semantic structures, input requirements, and output results. This comparison highlights the conceptual gap that must be addressed in the translation process.
- **Chapter 5 - NILE to VEREFOO Translation**: describes the methodology used to translate an intent defined in NILE into configurations interpretable by VEREFOO. The translation implementation, conceptual mapping steps, and main challenges encountered are illustrated, along with proposed solutions.
- **Chapter 6 - Validation of the Translation Process**: presents the verification and evaluation phase of the developed method. The objectives of the validation, adopted methodology, test cases, and obtained results are discussed, without neglecting limitations and prospects for future improvement.
- **Chapter 7 - Conclusions and Future Works**: summarizes the final considerations on the work carried out, highlighting the main contributions of the thesis and outlining possible future developments.

Chapter 2

Intent-Based Networking: Concepts, Architecture, and Language Solutions

2.1 Introduction

Recently, as already mentioned in the introduction chapter, the world of computer networks has undergone continuous transformation. The growing complexity of infrastructures, the increase in data traffic, and the need to ensure flexibility, security, and high performance have highlighted the inadequacy of traditional manual configuration methods.

Several fundamental technological changes have contributed to this evolution. Among them are *Software-Defined Networking* (SDN), which separates the control plane from the data plane by introducing new levels of abstraction; *Network Function Virtualization* (NFV), which makes it possible to replace physical devices with more agile and scalable virtualized functions; the paradigm of *cloud computing*, which has revolutionized service delivery through on-demand access to distributed resources; the concept of *liquid computing*, an emerging paradigm that enables dynamic and transparent resource sharing across different tenants and administrative domains [2]. These innovations have made networks more dynamic and flexible, but at the same time more complex to manage and secure.

Software-Defined Networking The paradigm of *Software-Defined Networking* (SDN) represents one of the most significant innovations in the evolution of network architectures. Its main feature consists in the separation of the control plane from the data plane: traffic decisions are centralized in a logical controller, while network devices are limited to forwarding packets according to the received rules. This approach introduces a high level of programmability and abstraction, allowing the network to be dynamically reconfigured in response to traffic variations or specific service requirements. Thanks to SDN, networks become more flexible, automatable, and adaptable, but at the same time they pose new challenges related to the scalability and security of the centralized control infrastructure.

Network Function Virtualization *Network Function Virtualization* (NFV) have introduced a radical change compared to the traditional model based on dedicated hardware appliances. Network functions, such as firewalls, load balancers, or intrusion detection systems, are implemented as software components running on generic platforms, eliminating dependence on specialized devices. This approach makes it possible to reduce infrastructure costs, accelerate service deployment, and improve scalability, enabling operators to quickly adapt resources to traffic demands. However, virtualization introduces new complexities related to orchestration, monitoring, and performance assurance, requiring advanced management mechanisms.

Cloud computing and liquid computing *Cloud computing* has revolutionized the way services and applications are delivered, based on the principle of on-demand access to a shared pool of computing, networking, and storage resources. Thanks to this paradigm, it is possible to build highly scalable and efficient distributed systems capable of supporting applications at a global level. The consumption-based model reduces initial investment costs and fosters high elasticity, enabling organizations to quickly adapt to peaks or drops in demand. However, static allocation of cloud resources can lead to inefficiencies, with significant portions of capacity remaining unused, thus raising new challenges in terms of optimization and management.

To address such limitations, the concept of *liquid computing* has been recently proposed, an emerging paradigm that enables dynamic and transparent resource sharing across different tenants and administrative domains [2]. In this scenario, resources are no longer rigidly bound to a single user or specific infrastructure, but can be “lent” or “acquired” fluidly, creating a distributed computational continuum. This approach ensures greater flexibility and cost optimization, but also introduces new challenges, especially in terms of security and isolation. In *multi-domain* and *multi-tenant* contexts, in fact, the same physical node can be shared by users belonging to different administrative domains, expanding the attack surface and making the management of security boundaries more complex, as they become dynamic and adaptable [2].

2.2 Intent based networking

The technological developments discussed in the previous sections have increased the programmability, elasticity, and automation potential of modern networks. At the same time, however, they have introduced a growing level of complexity: infrastructures are now highly dynamic, heterogeneous, and continuously reconfigurable, making manual management increasingly impractical.

In this context, *Intent-Based Networking* (IBN) has emerged as a paradigm capable of bridging the gap between high-level operational goals and the low-level configurations required to implement them. IBN is based on the idea that network policies can be defined starting from a level of abstraction closer to human language, rather than through low-level specific configurations. Thanks to this approach, IBN makes it possible to reduce human errors, accelerate service provisioning, and maintain constant alignment between business needs and the actual behavior of the network. Moreover, the declarative setting facilitates integration with automation and formal validation tools, ensuring that the generated configurations not only meet the expressed intents, but are also consistent and secure with respect to the operating conditions of the network.

2.2.1 Core Components

As indicated in the survey by Aris Leivadeas et al. [1], IBN is a relatively new paradigm that has recently received considerable attention. Although still emerging, several efforts have been made to properly define and standardize it. These efforts have led to the identification of five fundamental components: Intent Profiling, Intent Translation, Intent Resolution, Intent Activation, and Intent Assurance. Based on the classification provided in the survey [1], these components can be described as follows:

1. **Intent Profiling (Intent Expression):** The user expresses the intent through natural language, a GUI, or other interfaces. The system may also guide the user to ensure that the intent is meaningful and interpretable.

2. **Intent Translation:** Since the intent cannot be directly executed, it must be translated into concrete network policies and low-level configurations that network devices can understand.

3. **Intent Resolution:** This component detects and manages conflicts among multiple intents (e.g., incompatible policies, users with different priorities). It may propose solutions or notify the administrator.

4. **Intent Activation:** Once compatibility with other intents is verified, the

system activates the requested service by configuring the network according to the user's requirements.

5. **Intent Assurance:** Because network environments are dynamic, assurance mechanisms ensure that the intent remains satisfied over time, intervening proactively or reactively (self-healing and continuous adaptation).

2.2.2 Role of AI in IBN

In recent years, a research trend has emerged in which Artificial Intelligence is increasingly used to fully automate the entire IBN pipeline. Several works [3, 4, 5, 6] propose AI-driven solutions capable of covering, to varying degrees, all five components identified in [1]: from intent profiling through natural-language understanding, to translation via supervised or reinforcement learning, to conflict resolution and autonomous activation, and finally to predictive, self-correcting assurance mechanisms. These AI-centric approaches hold the promise of highly adaptive and self-optimizing networks. However, current AI-based solutions still exhibit several limitations associated with the freshness of the technology and the evolving state of the art; for these reasons, despite the growing interest in end-to-end AI-based systems, this thesis adopts a more controlled and verifiable approach based on a structured intent language.

2.2.3 Intent Translation

Among the core components of an Intent-Based Networking System, *Intent Translation* plays a pivotal role in bridging the gap between high-level operator intents and low-level, deployable network configurations. In the context of this thesis, which focuses on translating NILE intents into VEREFOO configurations, the discussion naturally centers on this component. As highlighted in the survey by Aris Leivadeas et al. [1], each IBNS component presents specific technical challenges; the work presented here addresses two of the main challenges associated with Intent Translation.

The first challenge is the intrinsic complexity of translating high-level intents into concrete and device-understandable network policies. As highlighted in [1], even simple intents may expand into multiple interdependent policies, and an incorrect translation can lead to severe misconfigurations and degraded network performance. Effective intent translation therefore requires a hierarchical refinement process capable of progressively enriching the intent with the missing technical details while identifying policy dependencies. The solution to this challenge follows the approach outlined above and will be discussed in detail in the following chapters.

The second challenge instead concerns *vendor agnosticism.* An intent should remain independent of vendor-specific technologies, implying the need for interoperable configuration languages or tools capable of managing heterogeneous infrastructures. To address this issue, the approach adopted in this thesis relies on the NILE intent language, designed to be vendor-neutral, and on the VEREFOO framework, which can generate configurations for diverse network environments without requiring the user to provide vendor-specific information.

Intent Translation mechanisms

Beyond the challenges described above, the survey [1] identifies several mechanisms commonly employed to translate high-level intents into network policies:

- **Template or Blueprint-Based Translation:** Predefined configuration blocks are selected according to the intent's requirements. Simple and effective for unambiguous intents, but relies on prior expertise to create templates.
- **Mapping:** The high-level intent is decomposed into semantically meaningful components, which are associated with specific network services or technical requirements. Useful for abstract intents and detecting conflicts among multiple intents.
- **Refinement:** Hierarchical, stepwise enrichment of the intent adds missing technical details. Especially useful when mapping alone cannot fully capture the intent.
- **Network Service Descriptors (NSDs):** Used in NFV scenarios such as service chaining or network slicing. NSDs serve as deployment templates containing VNFs, links, and forwarding graphs, either mapped from a catalogue or inferred from the intent.
- **Inference-Based Translation:** Deduce implicit requirements or missing details by reasoning over correlations between intent elements or using knowledge from past designs.
- **Keyword-Based Translation:** Keywords extracted from natural language intents are associated with specific services or policies. Can be combined with inference to resolve interdependencies and generate ECA rules.

The survey also discusses more advanced mechanisms, such as *machine-learning-based translation*, which uses classification or reinforcement learning models to infer suitable policies like [7], and *semantic techniques* leveraging RDF or OWL graphs to formally represent relationships among requirements, services, and deployment rules.

Finally, *state-machine-based approaches* translate intents into Event-Condition-Action policies by modeling intent elements as transitions in deterministic finite automata.

Among the mechanisms presented in [1], the method adopted in this thesis combines semantic analysis, mapping, and refinement techniques. By analyzing the semantic structure of the NILE language and the requirements imposed by the VEREFOO framework, it is possible to construct a precise correspondence between the two representations. This mapping enables a clear and accurate translation of NILE intents into the low-level, vendor-agnostic configurations required by VEREFOO, as will be detailed in the following chapters.

2.3 Intent-Based Languages

As mentioned before the work presented in this thesis adopts the use of a structured intent language, callled NILE, instead of AI. The *intent-based languages* are a key element of IBN, they are formal languages that allow the description, in an abstract and high-level manner, of what the network must achieve. They represent a middle ground between natural language and machine language: on the one hand, they maintain a level of expressiveness accessible to the user; on the other hand, they provide a rigorous structure interpretable by orchestration systems. In this way, the removal of the need to directly express technical specifications makes their use simpler and more immediate, even for non-expert users, broadening the audience of those who can interact effectively with the network. Introducing an intent language as an intermediate representation also makes it possible to separate policy extraction from its implementation and enforcement, ensuring greater flexibility and facilitating human feedback before the deployment phase. The adoption of such languages offers numerous advantages: reduction of configuration errors, faster provisioning, dynamic adaptation to changing contexts, and greater alignment between business objectives and technological capabilities.

Several intent-based languages have been proposed in the literature, such as *INTPOL* [8], *Nemo* [9], *Polanco* [10], *Lai* [11] and *NILE* [12, 13]. However, languages such as Nemo, Polanco, Lai, and INTPOL were not selected for this work due to their intrinsic limitations with respect to the objectives of this thesis. In particular, Nemo requires highly detailed and complex specifications, which reduces usability and generality; Polanco is strongly oriented toward academic contexts and tailored to translating policies from PWC-style structures, making it less flexible and overly close to human language, with still several aspects to refine; Lai is designed mainly for managing Alibaba's ACLs and is therefore too technical and domain-specific; and INTPOL, although formally rigorous, relies on a highly structured specification approach that lacks accessibility for non-expert users. NILE was instead chosen because it offers a more flexible, general, and user-friendly intent expression model, capable of bridging natural-language inputs and network policies without being tied to a specific domain or representation format.

2.3.1 NILE

NILE (Network Intent LanguagE) is an intent language proposed by Arthur S. Jacobs et al. [13], with the goal of providing an intermediate level of abstraction between the definition of network policies and their actual implementation. The idea originates in the context of research on *self-driving networks*, where the need for a tool capable of translating operator's natural language expressions into precise and verifiable configuration rules was observed.

In the work of Arthur S. Jacobs et al. [13], is possible to find an initial version of NILE's grammar (Listing 2.1); in this work, the authors introduce a process of intent refinement based on machine learning and operator feedback. The key insight was the introduction of an intermediate representation that, while resembling natural language, is sufficiently structured to allow automatic translation into network rules (for example for SDN or NFV). In this scenario, NILE acts as a bridge language: on the one hand, it is readable and confirmable by the operator, while on the other it ensures the precision required for compilation into formal policies and the subsequent verification of conflicts or errors.

```

<intent> ::= 'define intent'intent_name':' <commands>

<commands> ::= <command> {'\n' <command> }

<command> ::= (<middleboxes> | <qos> | <rules>)+[ <optional> ]

<middleboxes> ::= 'add' <middlebox> {(','|',\n') <middlebox> }

<middlebox> ::= 'middlebox('middlebox_id')'

<qos> ::= 'with' <metrics>

<metrics> ::= <metric> {(','|',\n') <metric> }

<metric> ::= <metric_id> '(' <constraint> ','value')' | <metric_id
    > '(none)'

<metric_id> ::= latency | jitter | loss | throughput

<constraint> ::= 'less [or equal]' | 'more [or equal]' | 'equal' |
     'different'

<rules> ::= <rule> {'\n' <rule> }
```

```

<rule> ::= (allow | block) <traffic>

<optional> ::= <targets> | <locations> | <interval>

<targets> ::= 'for' <target> {(','|',\n') <target> }

<target> ::= 'client('client_id')'| <traffic>

<locations> ::= 'from' <endpoint> 'to' <endpoint>

<endpoint> ::= 'endpoint('endpoint_id')'

<interval> ::= 'start' <date_time> '\n''end' <date_time>

<traffic> ::= 'traffic('traffic_id')' | 'flow('[<five_tuple>]+')'

<five_tuple> ::= 'protocol:' v | 'src_port:' v | 'src_ip:' v | '
    dest_port:' v | 'dest_ip:' v

<date_time> ::= 'datetime('datetime')' | 'date('date')' | 'hour('
    hour')'
```

Listing 2.1: First grammar of NILE [13]

In the following years, the language was extended (Listing 2.2) and used as an integral part of the LUMI chatbot system [12]. LUMI is a platform that allows operators to converse in natural language directly with the network, expressing intents, such as *"Inspect the traffic for the dormitory"*. The system exploits Named Entity Recognition (NER) algorithms and a chatbot interface to extract key entities, build an intent in NILE, and present it to the operator for confirmation. Afterwards, the intent is compiled into configuration languages such as Merlin and applied to the network. In this context, NILE represents the fundamental abstraction between natural inputs and low-level commands, extending the initial functionalities (for example, beyond service chaining, it now supports QoS constraints, bandwidth limits, usage quotas, and temporal constraints).

```

<intent> ::= 'define intent' <term>':' <operations>

<operations> ::= <path> <operation> { ' ' <operation> }

<path> ::= [<from_to> | <targets>]

<from_to> ::= 'from' <endpoint> 'to' <endpoint>

<operation> ::= (<mboxes> | <qos> | <rules>)+ [ <interval> ]

```

```
<mboxes> ::= ['add' | 'remove'] <middlebox> { (',' | ',\n') <
    middlebox> }

<middlebox> ::= 'middlebox('<term>')'

<qos> ::= ['set' | 'unset'] <metrics>

<metrics> ::= <metric> { (',' | ', \n') <metric>}

<metric> ::= [ <bandwidth> | <quota> ]

<rules> ::= ['allow' | 'block'] <matches> [ <matches> ]

<targets> ::= 'for' <target> { (',' | ', \n') <target> }

<target> ::= [ <group> | <service> | <endpoint> | <traffic> ]

<matches> ::= [ <service> | <traffic> | <protocol> ]

<endpoint> ::= 'endpoint('<term>')'

<group> ::= 'group('<term>')'

<service> ::= 'service('<term>')'

<traffic> ::= 'traffic('<term>')'

<protocol> ::= 'protocol('<term>')'

<bandwidth> ::= 'bandwidth('['max' | 'min' ]', '<term>', <bw_unit
    >)'

<bw_unit> ::= [ 'bps' | 'kbps' | 'mbps' | 'gbps' ]

<quota> ::= 'quota('['download' | 'upload' | 'any' ]', '<term>', <
    q_unit>)'

<q_unit> ::= [ 'mb/d' | 'mb/wk' | 'gb/d' | 'gb/wk' | 'gb/mth' ]

<interval> ::= 'start' <datetime> 'end' <datetime>

<datetime> ::= 'datetime('<term>')' | 'date('<term>')' | 'hour('<
    term>')'

<term> ::= [a-z0-9]+
```

Listing 2.2: NILE grammar [12]

Thanks to this evolution, NILE has proven capable of balancing readability and expressiveness; in fact it is understandable even by non-expert operators, who can confirm or correct the generated intent, and also it provides formal constructs able to describe complex network policies. This dual nature distinguishes it from languages that are too close to natural language (often ambiguous) and from excessively formal languages (hardly accessible). Moreover, these characteristics have led to NILE being adopted in various research works, where its combination of clarity and expressive power has been leveraged to support intent-based network management and verification tasks [14, 15].

NILE was chosen as the reference language for the development of this thesis precisely for these reasons. It allows the construction of powerful yet clear intent, as shown in Listing 2.3, in which a complex policy is expressed in a readable and structured form. In this way, the language serves both as an abstraction tool with respect to the underlying policy mechanisms and as a concrete bridge between the expressiveness of natural language and the precision required by execution in the network.

```
define intent one:
    from endpoint(NodeWS2) to endpoint(NodeWC5)
        allow protocol(FTP)
        block service(XBOXLive) protocol(POP3)
```

Listing 2.3: Example of a NILE intent

In the following chapters the characteristics of NILE will be analized in more detail, focusing in particular on its semantics and on the challenges it poses to be translated and integrated into automatic policy verification tools, such as *VEREFOO*.

2.4 Case Studies: IBN Architectures and Tools

To gain an initial understanding of the starting point for the translation work in this thesis, several solutions in the field of Intent-Based Networking (IBN), with a specific focus on intent languages and mechanisms for translating high-level specifications into operational configurations, were analyzed; among these were Indira, INSpIRE, and Polanco.

2.4.1 Indira

In the work of Mariam Kiran et al. [16], *Indira* (Intelligent Network Deployment Intent Renderer Application) is presented, a platform for enabling intent-based *networking*, designed to interact with the *north-bound* interfaces of the SDN networks of the Energy Sciences Network (ESnet). The main goal of *Indira* is to provide a simple, reliable, and technology-independent communication channel between users and the network infrastructure, reducing operational complexity and making advanced features accessible even to non-expert operators. The system was developed in the context of data-intensive scientific collaborations, where it is crucial to ensure secure and efficient transfers of large datasets, latency management, and reliable network performance. In scenarios such as high-energy physics or computational climatology, users need to transfer massive datasets, comply with strict latency constraints, and ensure service reliability; however, the manual management of such requirements through traditional configuration tools proves to be complex and poorly scalable.

To address these needs, *Indira* adopts the IBN paradigm, allowing requests to be expressed through a high-level descriptive language (for example: connect site A with site B), which is then automatically translated into prescriptive instructions up to the generation of operational rules (for example OpenFlow), avoiding the need for the user to specify technical parameters such as VLANs, IP addresses, or routing protocols. A distinctive aspect of the framework is the use of natural language processing and ontological engineering techniques: user requests are interpreted and converted into RDF semantic graphs, which formally represent the requested services and associated conditions. These graphs allow *Indira* to maintain semantic consistency, perform constraint checks (for example QoS or policy), and translate intents into concrete commands for provisioning tools such as NSI (for multi-domain circuit creation) and Globus (for secure file transfers).

The architecture of *Indira* therefore integrates four main phases: first, the intent expressed by the user is acquired through a command-line interface or a dialogic interaction in natural language. Subsequently, a parsing module extracts the key information and converts it into an RDF graph, enriched with metadata from external files containing user profiles and topology information. At this point, the system performs a consistency check, resolving any conflicts and verifying the availability of the requested resources. Finally, the validated intent is translated into concrete commands, ready to be sent to provisioning tools such as NSI (Network Service Interface) for the setup of multi-domain circuits or Globus for the secure and reliable transfer of files. This pipeline makes it possible to automate the entire intent lifecycle, from its abstract formulation to its practical execution, drastically reducing the cognitive load on the user and increasing the responsiveness of the network. In this way, the system automates the entire cycle, from the requirement expressed in natural language to the operational configuration, ensuring interoperability between heterogeneous tools and the possibility of automatic reasoning on application requirements.

Indira represents a concrete case of application of the IBN paradigm in complex scientific scenarios: thanks to the combination of semantic abstraction, linguistic interaction, and provisioning automation, it demonstrates how semantic intelligence can make Intent-Based Networking more accessible, flexible, and suitable for contexts characterized by high heterogeneity and data intensity[16]. Although it is not specifically focused on network security, its approach to intent translation through ontologies provides useful insights and parallels with the objectives of this thesis.

2.4.2 Inspire

INSpIRE (Integrated NFV-based Intent Refinement Environment), developed by Eder J. Scheid et al. [17], represents one of the first solutions that integrates the paradigm of Intent-Based Networks (IBN) with Network Function Virtualization (NFV), proposing an intent refinement mechanism capable of generating concrete configurations starting from high-level specifications.

The main objective of INSpIRE is to bridge the gap between the abstraction offered by intents and the low-level configurations required for the operation of the network. To this end, the system takes as input intents described through a Controlled Natural Language (CNL) and translates them into Service Function Chains (SFC). This process ensures that the requirements expressed by administrators are met regardless of the complexity of the underlying infrastructure.

In particular, INSpIRE is characterized by three fundamental aspects:

- Identifies the Virtual Network Functions (VNF) necessary to satisfy the specified intent;

- Concatenates the VNFs according to their logical and functional dependencies, generating coherent service chains;

- Provides low-level information to the network devices, necessary for routing the traffic through the selected functions.

A distinctive element of the solution is the extension of the VNF descriptor defined in the ETSI MANO framework, which is enriched with metadata useful to support the ordering of functions during the concatenation phase [17]. This allows the system not only to respect functional constraints, but also to take into account non-functional criteria, such as performance requirements and optimization objectives (softgoals).

INSpIRE represents a significant step towards the automation of intent-based network management, demonstrating how refinement can bridge the gap between abstraction and implementation. Although its design focuses primarily on the composition of service chains using VNFs and physical middleboxes, concentrating on the management of network functions, INSpIRE has provided valuable inspiration for the work presented in this thesis.

2.4.3 Polanco

Among the intent languages cited in this thesis there is *Polanco*, presented by Sergio Rivera et al. [10] and developed to support *Policy Writing Committees (PWCs)* in the specific context of university campus networks. Campus networks represent complex and dynamic environments, characterized by a high heterogeneity of devices, services, and access requirements, and for this reason, PWCs are formed to discuss and decide their configuration. The members of PWCs are usually administrators who have a clear understanding of network management needs, but do not possess the technical skills typical of system engineers. These committees generally draft documents in natural language, such as *Acceptable Use Policies* (AUPs), which define security rules, usage constraints, and operational guidelines. However, these documents must then be manually translated by network experts into detailed technical configurations. This process largely depends on the experience of the individual administrator and is therefore subject to ambiguous interpretations and frequent errors, especially when network changes compromise policies that were previously correctly enforced.

To address this problem, Sergio Rivera et al. [10] proposed **Polanco** (*POlicy LANguage for Campus Operations*), an intermediate language designed to bridge the gap between the definition of network policies by PWCs, expressed in natural language, and their technical implementation. Polanco approaches the language used by the committees, simplifying the translation phase and reducing the risk of errors compared to direct conversion into network configurations. While remaining human-readable, it allows policies to be expressed with sufficient precision to be automatically transformed into rules and actions for SDN networks or into low-level configuration files, ensuring that defined policies are correctly enforced. Polanco also integrates information about the network context (e.g., topology or device variations), allowing configuration rules to adapt dynamically and ensuring that policies remain correctly enforced even in the presence of infrastructural changes.

Polanco is based on several key principles [10]:

- **Use of high-level identifiers**: roles and attributes (e.g., device type, user group, or traffic type) replace MAC or IP addresses, increasing abstraction and readability.
- **Common abstractions**: concepts such as servers, firewalls, or secure channels are expressed uniformly, while the generation of low-level configurations is delegated to automatic translation mechanisms.
- **Context awareness**: policies can dynamically adapt to network changes, such as the introduction of new equipment or emerging threats.
- **Explicit exception management**: the ability to clearly specify exceptions simplifies automation processes and reduces the costs of manual management.

Regarding the collection of necessary information, Polanco combines two main sources. On one hand, the *alias file*, which associates low-level identifiers (MAC, IP, VLAN, port numbers) with more intuitive high-level labels for operators. On the other hand, the *topology discovery* features provided by SDN controllers, which use protocols such as LLDP, BDDP, and SNMP to detect routers and switches, and packet inspection techniques (ARP, DHCP, ND) to identify end hosts. The integration of these sources allows for the construction of an initial network map, enriched with additional information from traditional management protocols or dedicated discovery systems. This dataset forms the basis for defining Polanco policies that are precise, readable, and automatically translatable into operational configurations.

Polanco was specifically developed for *Policy Writing Committees* in the context of university campus networks, thus being tied to a well-defined usage scenario. For the objectives of this thesis, however, a more general-purpose intent language is required, capable of adapting to different contexts and not limited to the academic domain. Furthermore, the use of Polanco still assumes the intervention of a network expert to translate policies written in natural language into the corresponding Polanco specifications. All of these reasons have led to the choice of NILE over Polanco. Nevertheless, Polanco provided valuable inspiration, especially in showing how intents can be enriched with detailed specifications.

2.5 Conclusion

This chapter has provided an overview of the evolution of modern network architectures and the emergence of Intent-Based Networking as a paradigm capable of addressing the increasing complexity of network management. Key enabling technologies, such as SDN, NFV, cloud computing, and liquid computing, have been examined, highlighting both their contributions to flexibility, scalability, and automation, and the new challenges they introduce in terms of orchestration, security, and operational complexity.

Intent-Based Networking has been presented as a solution that abstracts high-level operational goals into concrete network configurations, structured around the core components. The discussion then focused on Intent Translation, one of the core components of IBN, analyzing its main challenges as well as the mechanisms commonly used to perform this translation. Following this, intent-based languages were introduced, with particular attention to NILE as a structured, readable, and flexible tool for expressing network intents. Finally, several IBN solutions and frameworks, including Indira, INSpIRE, and Polanco, were examined to illustrate practical approaches and provide inspiration for the methodology adopted in this thesis.

The next chapter will introduce VEREFOO, the framework used in this work for verifying and generating low-level network configurations from high-level NILE intents.

Chapter 3

Network Automation

Network automation is the use of software-driven approaches to configure, manage, and optimize network devices and services with minimal human intervention. It improves efficiency, reduces errors, and enables scalable management of complex infrastructures.

This chapter focuses on Automation for Network Security Configuration, examining how automation can address the limitations of manual practices in security functions. In this context, particular attention is given to VEREFOO, a framework developed at Politecnico di Torino, which serves as the main case study of this thesis.

3.1 Automation for Network Security Configuration

The growing complexity of modern network infrastructures has made security management an increasingly critical and difficult task to sustain with manual approaches. In recent years, the emergence of paradigms such as the Internet of Things, virtualization, and the massive adoption of cloud services has expanded the attack surface and multiplied the number and variety of devices and protocols to protect. In this scenario, the manual configuration of network security functions, such as firewalls, intrusion detection and prevention systems, or VPN gateways, has proven to be not only slow and burdensome, but also highly error-prone. Several studies have highlighted how misconfigurations are among the main causes of security incidents, making the development of tools capable of minimizing human intervention urgent.

As reported by Daniele Bringhenti et al. [18] in the survey on the world of Automation for Network Security Configuration, the management of network security policies, in particular those of packet-filtering firewalls, has in recent years become an increasingly complex and critical task. The evolution of modern networks, characterized by growing size, heterogeneity, and variety of services, makes traditional approaches based on manual configurations inadequate. In fact, such practices prove to be error-prone, inefficient, and costly in terms of time, contributing to a growing number of anomalies in security policies, which include both conflicts and under-optimizations [18].

Confirming the issues related to the manual management of security configurations highlighted in Bringhenti et al. survey [18], Verizon's *Data Breach Investigations Reports* from 2013 to 2024 [19] show that errors classified under the "miscellaneous" macro-category are among the main causes of security incidents, with misconfigurations accounting for the majority of these cases. In recent years, anomalies related to the configuration of *Network Security Functions* (NSF) have steadily increased underlining that manual prevention and mitigation of misconfigurations is now unsustainable in complex networks, where even a single error can lead to large-scale cyberattacks.

Although the 2025 Verizon report [19] shows an apparent decrease in incidents attributed to the miscellaneous errors category, this is mainly due to a change in the composition of the partners contributing to data collection and not to a real reduction in human errors. Therefore, the probability that human errors will continue to cause violations remains high, underlining the urgency of automated tools for network security configuration and management. As the survey [18] and the evidence suggest, only approaches based on automation and centralized controls can significantly reduce the risk associated with configuration errors, improving both operational efficiency and the overall resilience of networks.

The main reasons for this problem lie in structural and organizational factors, including the separation of roles between security managers and network administrators, the growth in the size and complexity of infrastructures, the high heterogeneity of network functions, and configuration practices based on trial and error. In addition, the increasingly significant impact of breaches can cause not only direct financial damage but also indirect consequences on user trust and the overall stability of the system [18].

In this context, automation emerges as a key paradigm for addressing these issues; in fact it makes possible to translate high-level security requirements into concrete and optimized configurations of network functions, reducing the risks of inconsistencies and improving the overall efficiency of the system. Two technological factors have been decisive for this evolution: on the one hand, the softwarization of networks, with the affirmation of Software-Defined Networking (SDN) and Network Functions Virtualization (NFV), on the other, the development of Policy-Based Management (PBM) models. SDN allows the separation of the control plane from the data plane and the centralized orchestration of device behavior, while NFV allows replacing traditional hardware appliances with more agile and easily reconfigurable virtualized functions. PBM, finally, provides a methodological framework to automatically derive configurations starting from policies expressed in high-level languages, progressively translating them into executable rules for the various devices [18]. These innovations have made possible the development of fully automated workflows for security configuration, which include phases of policy specification, automatic composition of security services, and generation of network function configurations.

Automation is not limited to replicating what was previously performed manually, but introduces substantial benefits in terms of scalability, responsiveness, and optimization. In a context in which cyber threats evolve rapidly, the ability to reconfigure security services in near real time becomes essential to ensure adequate resilience. Moreover, through formal verification and optimization techniques, it is possible not only to avoid errors but also to reduce resource consumption and improve the quality of the service provided.

Despite significant progress, the automation of network security remains a lively and continuously evolving research field. Current solutions have already demonstrated how it is possible to drastically reduce dependence on manual operations, but challenges remain open related to the management of highly heterogeneous networks, the need to adapt to dynamic conditions, and transparency in human-machine interaction. In this perspective, several tools and frameworks have been developed to systematically address the generation and verification of configurations. Among these, one of the most significant contributions is represented by VEREFOO, which stands out as an innovative tool to make network security configuration automation concrete and effective in modern networks.

3.2 VEREFOO

VEREFOO [2, 20, 18, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] is a network automation tool, developed at the Politecnico di Torino by the NetGroup [31], which supports the definition, analysis, and automated optimization of packet filtering firewall policies. The goal of VEREFOO is to reduce the impact of human errors, improve operational efficiency, and ensure a higher level of security, establishing itself as an advanced solution in the context of modern network management [20].

3.2.1 Purpose and Capabilities

Nowadays, network protection is a highly relevant and, at the same time, challenging issue to address. Thanks to the rapid spread of technologies such as Network Function Virtualization (NFV), Software-Defined Networking (SDN), the Internet of Things (IoT), and the cloud computing paradigm, the evolution of network architectures has grown exponentially and, consequently, the complexity of the architecture, the traffic, and the amount of data flowing through the infrastructures have increased considerably. One of the defenses traditionally adopted is the classic packet filtering firewall which, thanks to its versatility and simplicity of implementation, makes it possible to keep the network safe from unwanted or malicious traffic, both internal and external [20].

Packet filtering firewalls use a set of rules that specify which packets should be discarded and which, instead, can proceed to the intended destination; this set is called a policy and is usually configured manually by the network administrator. However, as discussed previously, with the increasing complexity of networks, the manual approach is no longer sustainable or, more precisely, no longer feasible in large organizations such as companies or universities, which often have extensive and complex networks. In such cases, the manual configuration of packet filtering firewall policies can lead to configuration errors, which in turn generate security flaws and incidents. In fact, it is easy for those who define policies to overlook sub-optimizations or conflicts among the numerous rules; precisely to address this problem, network automation tools have been developed, including VEREFOO [20].

VEREFOO acts as an intermediary between high-level security policies and their implementation with packet-filtering firewalls. Instead of requiring administrators to manually configure each firewall or security function, VEREFOO takes as input the network topology and the intended security objectives; it then analyzes the network, interprets these objectives, and generates a consistent set of rules and configurations for packet-filtering firewalls, optimally placing the new firewalls. This process is driven by a formal MaxSMT-based methodology that guarantees correctness-by-construction and optimization, ensuring that all security requirements are satisfied while minimizing the number of firewalls and filtering rules [25, 26]. In this way, VEREFOO, not only automates repetitive and error-prone tasks, but also ensures that the resulting network behavior aligns with the specified policies. By abstracting the complexity of individual devices and security functions, VEREFOO allows network operators to focus on policy design rather than low-level configuration, significantly reducing the risk of misconfigurations and enhancing overall network security.

As can be seen in the work of Daniele Bringhenti et al. [21], VEREFOO is not limited to the automatic configuration of firewalls, but represents a general framework for the orchestration of various virtualized security functions. With the introduction of the concept of projection, in fact, Network Security Policies (NSPs) are translated into abstract operations independent of the type of Virtual Network Function (VNF), allowing the inclusion in the process not only of traditional packet filters, but also functions such as URL filters, HTTP filters, VPN gateways, logging systems, intrusion/anomaly detection, anti-virus, anti-malware, content filters, monitoring, and anonymization proxies. In this way, VEREFOO establishes itself as a flexible and comprehensive platform for the automatic configuration of security in virtualized networks, going well beyond the sole scope of packet filtering firewalls.

The work of Daniele Bringhenti et al. [23] provides a concrete example of the versatility of VEREFOO, showing how this tool can support the management of VPN configurations and Communication Protection Systems (CPS). VPNs are now fundamental in many contexts, as they ensure the protection of network traffic from inspections and unwanted modifications. The traffic passing through a VPN is normally encrypted and safeguarded by CPS; however, overall security depends on the correct configuration of the latter, which, if set incorrectly, can pose significant risks. The manual configuration of CPS is notoriously complex; in fact, it requires deciding where to place them, selecting appropriate encryption algorithms from a wide range of options, and correctly setting the device rules; with the increasing complexity of modern networks, these tasks have become even more difficult. The work [23] formalizes this problem as a MaxSMT problem, enabling VEREFOO to automatically identify correct and optimal CPS configurations, also integrating network efficiency objectives to provide secure and high-performing solutions.

VEREFOO has also been the foundation for the GreenShield project [22], in which an innovative automation system for firewall configuration and optimization is developed, designed to reduce environmental impact by lowering energy consumption without compromising the security level. In fact, both cybersecurity and computer networks represent a significant source of resource consumption: just think of data centers, routers, switches, and security devices such as firewalls, which require large amounts of energy to operate, resulting in higher maintenance costs and a growing contribution to pollution. As reported in Daniele Bringhenti et al. [22], GreenShield is one of the few projects addressing the issue of sustainability related to the automatic configuration of distributed packet filtering firewalls and, thanks to the use of VEREFOO, it is able to activate firewalls in a way that reduces consumption and optimizes filtering, blocking communications as close as possible to the source.

3.3 Conclusion

In this chapter, we have highlighted the growing complexity of modern networks and the challenges associated with manual configuration of network security functions. Automation emerges as a fundamental solution, enabling consistent, efficient, and reliable management while reducing human error and improving resilience.

VEREFOO has been introduced as a representative tool for network security automation. Its flexible framework, capable of managing not only packet filtering firewalls but also a wide range of virtualized security functions, exemplifies the benefits and potential of automated network management.

The next chapter will provide a detailed analysis of VEREFOO, focusing on its semantic structures, input and output methods, and verification models, laying the groundwork for the comparison and translation processes presented in the subsequent chapters.

Chapter 4

Semantics, Inputs, and Outputs of NILE and VEREFOO

4.1 Overview

This chapter provides a comparative analysis of the semantics, inputs, and outputs of the two core systems analyzed in this thesis: the NILE intent language and the VEREFOO verification and deployment framework. Understanding their respective information models is essential to bridge the abstraction gap between them.

4.2 NILE Semantics and Structure

Network Intent Language (NILE) is a near-natural language, designed to express security intents in an intermediate form between human language and typical network configuration language. In this way, the user can formulate comprehensible security objectives without needing to learn specific languages, while NILE can be automatically translated into the configurations required by the different policy mechanisms.

Based on NILE's requirements, Jacobs et al. [13] identified three fundamental requirements for designing the grammar of the intent language:

- **High comprehensibility:** allows operators, even non-experts, to understand and verify the correctness of an intent;
- **Full expressiveness:** enables faithful representation of the operator's intention;

- **Writing efficiency:** facilitates quick and easy modifications to generated intents.

Based on these three fundamental requirements, Jacobs et al. [13] designed an initial NILE grammar, expressed in EBNF notation, aimed at satisfying comprehensibility, expressiveness, and writability needs (Listing 2.1). The grammar shown in Listing 2.2 represents the second version, improved from the original, presented in the work for the Lumi chatbot [32] and adopted for the project of this thesis and analyzed in this chapter.

4.2.1 Syntactic Definition of an Intent

An *<intent>* represents the user's intent, i.e., the high-level goal to be achieved in the network, as discussed in the previous chapter (Chapter 2). It constitutes an abstraction that allows expressing *what* is to be achieved without directly specifying *how* to implement it. From a syntactic point of view, an intent in NILE consists of *<operations>*, which define the concrete actions that the network orchestrator must translate into actual configurations.

The basic construction of an *intent* is defined as follows:

```
<intent> ::= 'define intent' <term> ':' <operations>
```

Each intent is identified by a unique term (`<term>`), which can be any alphanumeric string. This term associates a name with the intent and links it to a sequence of operations to apply. The choice of a declarative approach allows separating the *what* from the *how*, enabling the network orchestrator to interpret the intent and translate it into the necessary configurations.

4.3 Operations

As shown in [12, 32], *<operations>* represent the semantic core of the grammar and constitute the mechanisms through which a NILE intent expresses the user's objective. They are organized into two main categories: *mandatory*, required to delimit the scope of the intent, and *optional*, which enrich the expressiveness of the language with access constraints, quality of service parameters, middlebox management, and temporal conditions.

The mandatory operations consist of *from/to*, which define the involved endpoints, and *for*, which defines the target of the intent, that can be a group, an endpoint, a service, and a traffic; these mandatory operations apply to all network policy types.

One of the optional operations is *allow/block*, which operates on traffic, services, and protocols and is for ACL. Another optional operation is *set/unset* which manages the quotas or bandwidth limits for Quality-of-service (QoS). Also the management of service chaining via *add/remove*, which allows the insertion or deletion of middleboxes is an optional operation. The last optional operation is *start/end*, which accepts hours, dates, or full datetime specifications and allows the definition of time-bounded intents.

4.3.1 Mandatory

Path Definition

The operands `from/to` and `for` are mandatory, as they define the fundamental elements of the intent's application domain. In particular, `from/to` indicates the *path* on which the policy will be applied, identifying the relationship between two specific endpoints; `for` defines the policy's *targets*, allowing reference to broader categories such as groups, services, or traffic types. It is possible to use either `from/to` or `for` within an intent, depending on the desired level of specificity, but both operands must never appear together in the same intent.

```
<from_to> ::= 'from' <endpoint> 'to' <endpoint>
<targets> ::= 'for' <target> { (',' | ',\n') <target> }
```

Examples:

```
from endpoint(a) to endpoint(b)
for group(admin), service(http)
```

4.3.2 Optional

ACL

Operands of type `allow` and `block` correspond to control policies aimed at permitting or denying specific traffic, service, or protocol.

```
<rules> ::= ['allow' | 'block'] <matches> [ <matches> ]
<matches> ::= [ <service> | <traffic> | <protocol> ]
```

Example:

```
block service(ssh) protocol(tcp)
```

Middlebox Management (Service Chaining)

Middlebox chaining, expressed through the operands `add` and `remove`, allows dynamically modifying the network topology by inserting functional devices such as firewalls, proxies, or intrusion detection systems. Importantly, the grammar does not impose constraints on the type of `middlebox` that can be used.

```
<mboxes>::=['add'|'remove']<middlebox>{(','|',\n')<middlebox>}
<middlebox> ::= 'middlebox('<term>')'
<term> ::= [a-z0-9]+
```

Example:

```
add middlebox(firewall), middlebox(ids)
```

From this definition, a `middlebox` is characterized solely by a `<term>`, i.e., an alphanumeric string, and can therefore represent any device (e.g., firewall, nat, etc.), also devices that are not commonly defined as middleboxes.

QoS

Operands `set` and `unset` allow defining *Quality of Service (QoS)* parameters. Metrics include bandwidth, specified as minimum or maximum, and traffic quotas, defined with respect to direction (upload/download) and temporal granularity.

```
<qos> ::= ['set' | 'unset'] <metrics>
<metrics> ::= <metric> { (',' | ', \n') <metric>}
<metric> ::= [ <bandwidth> | <quota> ]
<bandwidth>::='bandwidth('['max'|'min']','<term>',<bw_unit>)'
<bw_unit> ::= [ 'bps' | 'kbps' | 'mbps' | 'gbps' ]
<quota>::='quota('['download'|'upload'|'any']','<term>',<q_unit>)'
<q_unit> ::= [ 'mb/d' | 'mb/wk' | 'gb/d' | 'gb/wk' | 'gb/mth' ]
```

Examples:

```
set bandwidth(min, 100, mbps)
unset quota(download, 500, mb/wk)
```

Temporal Aspects

Operands `start` and `end` introduce a temporal constraint, allowing an intent's validity to be limited to specific time windows. This functionality is crucial in dynamic resource allocation scenarios.

```
<interval> ::= 'start' <datetime> 'end' <datetime>
<datetime>::='datetime('<term>')'|'date('<term>')'|'hour('<term>')'
```

Example:

```
start date(2025-01-01) end date(2025-01-31)
```

4.3.3 Targets

A central aspect of the grammar is the ability to abstract network entities as *targets* of the intent. Targets represent the logical objects to which policies are applied and constitute, together with the path defined by `from/to`, the semantic domain of the intent.

The syntactic rule describing targets is as follows:

```
<target>   ::= [ <group> | <service> | <endpoint> | <traffic> ]
<endpoint> ::= 'endpoint('<term>')'
<group>    ::= 'group('<term>')'
<service>  ::= 'service('<term>')'
<traffic>  ::= 'traffic('<term>')'
```

From this definition, a target can refer to different types of abstractions:

- **Endpoint**: represents a single network entity, such as a virtual machine, server, or identified host. Example: `endpoint(db01)`.
- **Group**: identifies a set of homogeneous entities grouped according to logical or organizational criteria. Useful for applying common policies to multiple users or services, e.g., `group(admin)`.
- **Service**: allows reference to a specific application service (e.g., HTTP, DNS, SSH). This abstraction is crucial for application-level policies. Example: `service(http)`.
- **Traffic**: describes a flow or type of traffic, distinguishing, for instance, between video, VoIP, or bulk data traffic. Example: `traffic(voip)`.

This modeling offers a dual advantage: it allows expressing very granular policies, such as those concerning a single endpoint or service, and it allows extending the scope to broader categories, such as groups of users or traffic types.

Finally, the use of the generic construct `<term>`, defined as an alphanumeric string, provides the language with a high degree of extensibility. In this way, the set of targets is not constrained by a predefined taxonomy but can be expanded and adapted based on specific operational needs and application contexts.

4.4 Considerations

The proposed grammar defines a declarative language capable of expressing high-level network management policies. Specifically, it allows to:

- specify paths or sets of targets;
- modify the network topology by adding or removing devices;
- impose *Quality of Service* constraints;
- apply filtering rules;
- associate temporal conditions with operations.

The strength of the formalism proposed by the NILE grammar lies in its ability to represent network intents clearly and unambiguously, thus facilitating automatic translation into concrete configurations. In this sense, the grammar constitutes an intermediate layer between the abstract expression of user policies and their technical realization on underlying network devices.

4.5 VEREFOO Semantics and Architecture

In the previous chapter 3, VEREFOO was introduced as a representative framework for the automation of network security policies, emphasizing its capacity to translate high-level objectives into concrete configurations and to support a wide range of security functions beyond traditional packet filtering. Having outlined its motivations, scope, and practical applications, it is now necessary to examine in detail the internal mechanisms that enable such functionality.

4.5.1 Overview

The particularity of VEREFOO is that it does not operate according to an empirical approach (based on trial and error), but is founded on formal models and propositional logic, ensuring that the solutions identified are correct by construction [26]. To ensure that the solutions are correct by construction, all data must be precise and structured; for this purpose, a set of .XSD files is provided, which rigorously define the format of VEREFOO's inputs [27]. The problem of firewall allocation and the related rules is mathematically modeled as a partially weighted *Maximum Satisfiability Modulo Theories* (MaxSMT) problem. This model includes both hard constraints, which must always be satisfied (e.g., security requirements), and optimization criteria, such as minimizing the number of distributed firewalls and the overall number of rules [26]. VEREFOO's algorithm automatically finds the solution that respects the constraints while simultaneously optimizing the criteria; in this way, the correctness of the configuration is formally guaranteed, without the need for manual testing or empirical adjustments [26].

The approach not only satisfies security requirements but also minimizes the number of firewalls and rules configured on each of them, bringing benefits in terms of network performance and resource costs [26]. Furthermore, the method allows for the automatic enrichment of a Service Graph defined by the service designer: the SG can include different network functions (not only firewalls, but also NAT, load balancers, etc.) [26]. VEREFOO calculates the minimum number of firewalls required and determines the optimal set of rules for each of them, while simultaneously ensuring security and optimization.

This chapter focuses on the semantics and architecture of VEREFOO, providing a formal description of how security requirements are expressed, processed, and enforced within the system. Particular attention is devoted to the representation of inputs and outputs that guide the interpretation of network security policies, and the architectural modules that realize the automation process. By presenting these foundations, we establish the theoretical and structural basis upon which the subsequent analyses and translations are built.

4.5.2 Input

In VEREFOO, the automated analysis and optimization of packet-filtering firewall policies require a `.xsd` file, containing an NFV schema with the following information:

- Full service graph (SG) of the network
- Configurations of the various nodes
- Security requirements

Service Graph

The starting point of VEREFOO's processing is the Service Graph (SG), which is a logical representation of how a virtual network is built. An SG is defined by the designer as a set of network functions (Network Functions, NF) and nodes, connected by edges, to provide an end-to-end service. The SG represents a general and complex structure, which can include parallel or branched paths, thus supporting real-world scenarios where traffic may follow multiple alternative routes. "Low-level" functions (such as switches and routers) are not explicitly included in the SG, although they exist in reality: the SG is an abstraction highlighting only the more complex functions [26].

The definition and characteristics of the Service Graph are contained in the `verigraph.xsd` file [27]:

- **graph**: the main element that encloses the topology.
- **node**: nodes of the graph, which can represent endpoints or Network Functions (e.g., firewall, DPI, etc.).
- **neighbour**: specifies adjacencies and thus the connections between nodes.
- **configuration**: describes parameters and properties associated with nodes.
- **paths** and **path**: identify allowed traffic paths.
- **attributes** such as `source`, `destination`, `protocol`, `src_port`, `dst_port` allow modeling network flows in terms of IP 5-tuples.

Graphs

Root element containing one or more graphs (`graph`), each identifiable by a unique `id` attribute. Represents a network graph. It contains nodes (`node`) and can be marked as a Service Graph using the `serviceGraph` attribute.

```
<graphs>
    <graph id="1" serviceGraph="true"> ... </graph>
</graphs>
```

Node

Represents a network node, which can be an endpoint or end host (ENDHOST, MAILCLIENT, WEBCLIENT, etc.) or a service function (FIREWALL, DPI, LOADBALANCER, etc.). Each node can have:

- a unique `id` and `name`;
- a functional type (`functional_type`);
- neighbours (`neighbour`), i.e., adjacent nodes;
- a specific configuration (`configuration`) detailing the behavior of the network function.

Main data types

Within the XSD schema, several data types are defined to limit and standardize the values allowed in specific fields. This ensures semantic consistency in the graph description and facilitates validation checks on XML files.

- `functionalTypes`: node types such as FIREWALL, DPI, ENDHOST, WEBSERVER, etc.
- `protocolTypes`: supported protocols (HTTP REQUEST, POP3 RESPONSE, etc.).
- `ActionTypes`: security actions (ALLOW, DENY, ALLOW_COND).
- `L4ProtocolTypes`: layer 4 protocols (TCP, UDP, ANY, OTHER).

FunctionalTypes Defines the *functional types* that a node (`node`) can assume. Each value represents a different network function or end role. Some examples are:

- `WEBCLIENT`, `WEBSERVER`, `MAILCLIENT`, `MAILSERVER`: endpoints generating or receiving application traffic.
- `LOADBALANCER`: node distributing traffic to a set of servers.

ProtocolTypes This type defines the allowed application protocols for traffic generation. Possible values are:

- `HTTP_REQUEST`, `HTTP_RESPONSE`
- `POP3_REQUEST`, `POP3_RESPONSE`

ActionTypes This enumeration is used by security nodes (e.g., firewalls) to define the action to apply to a flow. Possible values are:

- `ALLOW`: traffic is permitted.
- `DENY`: traffic is blocked.
- `ALLOW_COND`: traffic is allowed only under certain conditions.

L4ProtocolTypes This category specifies layer 4 (transport) protocols:

- `ANY`: any protocol.
- `TCP`, `UDP`: main protocols in the IP stack.
- `OTHER`: for protocols not standardized in the schema.

In summary, these categories serve as a common vocabulary to describe nodes and security policies within the Service Graph unambiguously. Thanks to them, an XML file conforming to the schema can be uniquely interpreted by verification and simulation tools.

Network Security Requirements

In the NFV schema, `Property` elements declare what should be blocked or permitted in the network, i.e., the network security requirements (NSR) to be implemented and optimized through VEREFOO.

A `Property` is characterized by the following main attributes:

- **name**: indicates the type of property and can take one of the values defined in the `P-Name` type:
  - `IsolationProperty`: specify that a flow between two nodes is blocked.
  - `ReachabilityProperty`: specify that at least a flow between two nodes is permitted.
  - `CompleteReachabilityProperty`: specify that at all flow between two nodes is permitted.
- **graph**: reference to the service graph where the property is defined.
- **src**, **dst**: identify the source and destination nodes involved.
- **lv4proto**: layer 4 protocol (TCP, UDP, or `ANY`).
- **src_port**, **dst_port**: optional ports to refine the conditions.

4.5.3 Output

Successful Outcome

If, after receiving the SG and the NSRs, the problem of automatic security application is solved, VEREFOO generates:

- Firewall allocation scheme
- the Filtering Policy (FP) for each allocated firewall.

Firewall Allocation Scheme In VEREFOO, the firewall allocation scheme serves to define where, within the logical network (Service Graph, SG), the instances of virtual firewalls should be placed in order to satisfy the security requirements (NSRs), using the minimum number of firewalls necessary to reduce resource consumption. The scheme is logical: it does not specify how to physically place the firewall on the physical infrastructure (physical servers, VMs, NICs, data centers). That task is handled by Virtual Network Embedding (VNE), which VEREFOO does not manage but is delegated to further refinement steps performed after using VEREFOO.

Filtering Policy A Filtering Policy (FP) is the set of filtering rules associated with each firewall allocated in the scheme computed by VEREFOO. In other words, it is the logical configuration of the firewall, automatically generated from the NSRs. It is written in an abstract and readable language, independent of the specific languages of different firewalls;

composed of:

- a default action:
  - whitelisting: everything is blocked except what is explicitly allowed;
  - blacklisting: everything is allowed except what is explicitly blocked.
- Set of filtering rules:
  - Rules auto-generated from the NSRs.
  - Free from anomalies (e.g., no conflicts, duplicates, shadowing).
  - Specify how to handle specific types of traffic (for example, based on source/destination address, protocol, port, etc.).

The Filtering Policy is created by VEREFOO with the minimum number of rules, to reduce memory usage and make the firewall faster; as in the allocation scheme, FPs are expressed in a generic and user-friendly language, independent of real firewalls (Cisco, iptables, Palo Alto, etc.), so the translation of the abstract FP into concrete rules must be done later, but not by VEREFOO.

In this way, the solution produced by VEREFOO can be automatically deployed in the virtual network using existing technologies. If a new configuration is needed (e.g., after an attack), it is sufficient to define new NSRs, and VEREFOO automatically computes the new configuration.

Negative Outcome

In case of a negative solvability result, VEREFOO generates a non-applicability report, which explains to the user why it was not possible to satisfy the requirements. A typical reason is that the defined Service Graph does not provide suitable APs where firewalls can be placed, due to overly strict constraints.

4.5.4 Input-to-Output Processing in VEREFOO

VEREFOO takes the SG provided as input and automatically transforms it into an Allocation Graph (AG), which serves as an internal representation. In this AG, for each connection between nodes/functions, an Allocation Place (AP) is created, i.e., a “potential” point where the system can decide to place a firewall. The system automatically positions the firewalls to obtain an optimal configuration and calculates the optimized FP, formulating and solving a weighted partial MaxSMT problem [26]. A security-skilled user can enforce the presence of a firewall in a specific AP, preventing its removal, or forbid a firewall from being placed in a certain AP. This could increase flexibility and reduce computation time, but if the user imposes overly strict constraints, the system might not find a solution, or may find only non-optimal solutions [26].

Traffic that the packet filter firewallls will filter is modeled as classes of packets defined by predicates over the standard IP 5-tuple. Each network function (VNF) in the AG is abstractly represented by two functions: a forwarding function, which determines which packets are denied, and a transformer function, which specifies how packets are modified. For example, firewalls may only filter traffic (denying or allowing packets), whereas NATs or load balancers may alter addresses or redirect flows [26].

VEREFOO computes maximal flows through the network by propagating these transformations along all paths relevant to each network security requirement (NSR). Maximal flows group together packets that traverse the same sequence of nodes and are affected in the same way, reducing the number of cases the solver must consider. This precomputation ensures that the variables representing flows are partially instantiated before optimization, improving efficiency [26].

Finally, the framework formulates a weighted partial MaxSMT problem, where hard clauses enforce strict security requirements and soft clauses encode optimization goals, such as minimizing the number of allocated firewalls and the number of firewall rules. The solver automatically determines which APs should host firewalls, the default actions of each firewall, and the minimal set of additional rules needed to satisfy all NSRs. The result is a correct-by-construction configuration in which every allocated firewall and rule is guaranteed to respect the specified network security policies while optimizing operational objectives [26].

4.6 Conclusion

NILE and VEREFOO serve fundamentally different purposes. NILE allows users to express what they want the network to do, while VEREFOO verifies whether specific configurations satisfy those goals. This mismatch in abstraction and data requirements is at the core of the translation problem discussed in the next chapter.

Chapter 5

NILE to VEREFOO Translation

5.1 Introduction

This chapter analyzes the translation process proposed to bridge the NILE intent-based language with the VEREFOO framework, enabling the automatic generation of valid VEREFOO XML input files from NILE intents. The goal is to ensure that high-level intents, expressed in the NILE language, can be correctly transformed into the low-level specifications required by VEREFOO, preserving their semantics and goals.

The translation between NILE and VEREFOO poses several challenges due to the conceptual gap between the two models. For example, while NILE focuses on abstract and human-readable intent expressions, VEREFOO requires a complete and concrete service graph description, including all nodes, their interconnections, and the associated NSRs represented through IP quintuples (*ip_src*, *ip_dst*, *protocol*, *src_port*, *dst_port*). Moreover, VEREFOO operates under a closed-world assumption, meaning that all entities and addresses involved in an NSR must belong to the defined service graph.

Additional challenges are that VEREFOO requires the NSRs of the input to be conflict-free and that some constructs available in NILE do not have a direct counterpart in VEREFOO, and vice versa, certain VEREFOO constructs cannot be expressed in NILE.

This chapter presents also the solutions for the challenges, and it also outlines the implementation of the translation tool, which automates the entire workflow, from parsing NILE intents, through data enrichment, to generating the final XML configuration.

5.2 Analysis of the Translation Requirements

As previously mentioned, the translation process between NILE and VEREFOO presents several challenges, such as the gap in information accuracy or that some constructs available in NILE are not represented in VEREFOO, and vice versa, certain VEREFOO constructs have no equivalent in NILE. To better understand both the translation process and these challenges, it is first necessary to analyze the information provided by NILE and identify the additional data required for VEREFOO.

5.2.1 Structure of NILE Intents

The Figure 5.1 illustrates the structural composition of a NILE intent. As discussed in the previous chapter, NILE intents are composed of two main parts: the *path* and the *operations*. The *path* defines the communication scope of the intent through the constructs `from/to`, and `for`, which specify the network elements involved. The *operations* section instead expresses the set of actions or policies associated with the intent, such as `add` or `remove` middleboxes, the definition of `QoS` parameters, specific `rules`, and potential `time intervals` governing their applicability.

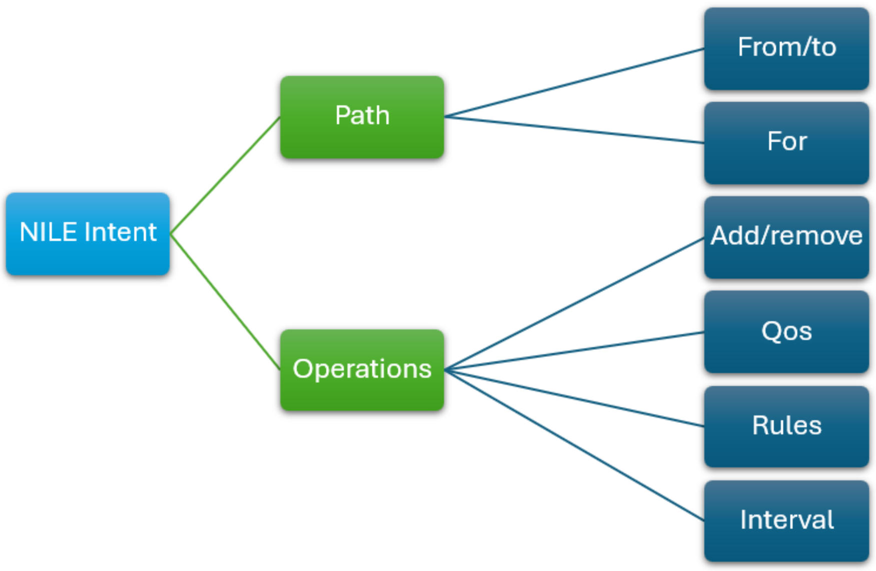


Figure 5.1: Structure of a NILE intent

Building on these theoretical foundations, Listing 5.1 presents a concrete example of an intent expressed in the NILE language. This example demonstrates how communication policies between two network nodes can be declaratively specified:

```
define intent one:
    from endpoint(NodeWS2) to endpoint(NodeWC5)
        allow protocol(FTP)
        block service(XBOXLive) protocol(POP3)
```

Listing 5.1: Example of an intent expressed in NILE

As shown in the example above, a NILE intent specifies only the names of the two nodes or groups according to the *path* construct. It also uses only names to define the services, protocols, traffic types, of the policies that must be enforced.

From this example, it is clear that the intents expressed in NILE are highly abstract and do not include the technical details, such as IP addresses or topology information, required by VEREFOO, which expects input files rich in precise configuration data. It is to bridge this gap that the translation process introduces a dedicated *information enrichment* phase. This phase leverages a database containing all the necessary technical data, provided by the user through the `create_db.py` module, which will be described in detail later in this chapter. It is also possible to notice from the example that the actions of NILE rules are only *allow* and *deny*, which can only map two of the three VEREFOO properties mentioned earlier.

5.2.2 Requirements of VEREFOO

The core requirements of the XML input file of VEREFOO, as illustrated in Figure 5.2, are the *Service Graph* and the *Network Security Requirements (NSRs)*. These two elements together define the logical and functional scope of the verification process carried out by the framework.

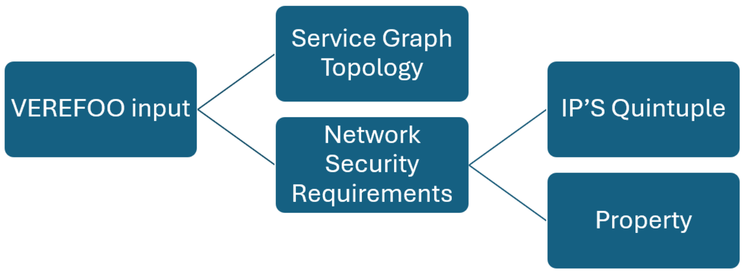


Figure 5.2: VEREFOO requirements schema

Service Graph

The *Service Graph* represents the entire network topology under analysis. It models all the nodes, such as hosts, routers, firewalls, and other network functions, together with their configurations and interconnections. Each node can be associated with specific attributes, including its type, assigned IP addresses, and operational parameters. The Service Graph is therefore a formal abstraction of the network, capturing both its structure and behavior.

VEREFOO operates under the *Closed-World Assumption (CWA)*, according to which all IP addresses and entities that are not explicitly represented within the graph are considered non-existent or unreachable. This assumption ensures a well-defined analysis domain and prevents ambiguity in the verification process. Consequently, every entity referenced in an NSR must be explicitly represented as a node in the Service Graph.

Network Security Requirements (NSRs)

The second requirement concerns the definition of the *Network Security Requirements (NSRs)* (Figure 5.3), which describe the set of properties that must hold for specific packet flows in the network. NSRs specify how a precise packet flow between a source and a destination should behave according to given policies, such as reachability and isolation constraints.

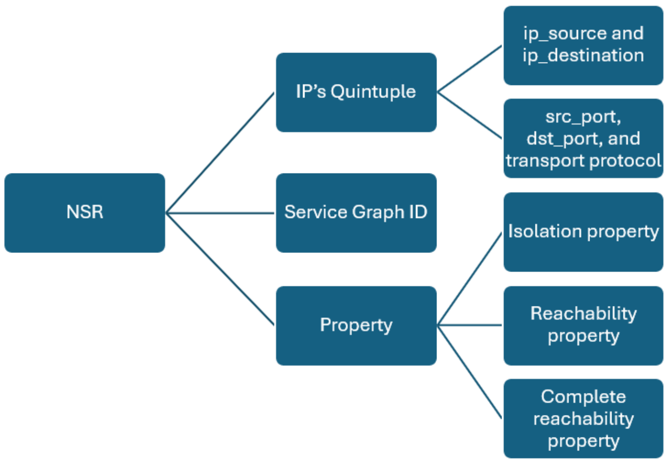

NSR
IP's Quintuple
Service Graph ID
Property
ip_source and ip_destination
src_port, dst_port, and transport protocol
Isolation property
Reachability property
Complete reachability property

Figure 5.3: VEREFOO NSR

Each NSR is composed of three main parts: the identifier of the Service Graph to which it belongs, the *property* that must be verified for the corresponding packet flow, and the IP five-tuple, which consists of the following fields: `ip_source`, `ip_destination`, `source_port`, `destination_port`, and `lvl4proto`. The five-tuple unambiguously identifies a specific class of packets in the network.

VEREFOO assumes that all NSRs provided as input are consistent with one another, meaning that they do not define contradictory requirements. For example, two NSRs should not impose conflicting reachability and isolation properties over the same flow [25]. Furthermore, all IP addresses mentioned within an NSR must refer to nodes contained in the same Service Graph; NSRs cannot span across multiple graphs.

An example of a NSR definition is shown in Listing 5.2. In this case, two reachability properties are specified, indicating that HTTP traffic (`TCP` on port `80`) from the source `130.10.0.1` must be able to reach two different destination nodes within the same Service Graph.

```
<Property name="ReachabilityProperty" graph="1" src="130.10.0.1"
   dst="40.40.41.1" lv4proto="TCP" dst_port="80"/>
<Property name="ReachabilityProperty" graph="1" src="130.10.0.1"
   dst="40.40.42.1" lv4proto="TCP" dst_port="80"/>
```

Listing 5.2: Example of NSRs

As it is possible to notice from the image and the example VEREFOO requires a high level of technical detail in its input, including the complete network topology, the configuration of each node, and all relevant IP addresses. This level of specificity ensures that the verification process can accurately reason about the network behavior and enforce the desired security and functional properties. As highlighted, such detailed information is not provided by NILE, which operates at a higher level of abstraction. Furthermore it is also possible to notice that VEREFOO requirements cannot express some of the operations of NILE like the QoS, middlebox management and temporal management.

5.3 Challenges of Translation

After having analyzed the NILE intents and VEREFOO requirements it is possible to adress the previously mentioned translation's challenges and their solutions.

5.3.1 Conflict-free

One of the challenges of the translation concerns the requirement for *conflict-free* NSRs[25], in fact as previously said VEREFOO assumes that all NSRs provided as input are consistent with one another, meaning that they do not define contradictory requirements. To ensure this, the translator relies on the fact that intents themselves are formulated in a conflict-free manner, therefore delegating the conflict detection to the intent creation phase. In other words, the intents do not express conflicting requirements, so any NSR derived from them will inherently be conflict-free. This design choice is motivated by the interactive nature of NILE, where the system can guide users in identifying potential conflicts or inconsistencies and prompt corrective actions before the translation stage. This approach not only reduces the risk of translation failures but also improves the overall reliability and robustness of the automated translation pipeline.

5.3.2 Information Gap

Another challenge is the fact that intents expressed in NILE are typically highly abstract and often lack the technical details required by VEREFOO, which instead expects input files rich in precise configuration parameters, like for example, the IP addresses associated with the endpoint names specified in the intents. To bridge this information gap, the translation process introduces a dedicated *information enrichment* phase. This phase leverages a database provided by the user through the `create_db.py` module, which will be described in detail later in this chapter.

The database stores all the information related to the network topology and configuration. It includes the definitions of nodes (such as their names, types, identifiers, IP addresses, id of their service graph, and configurations), the descriptions of all available services, protocols, and traffic types, as well as the details of the logical and physical connections between nodes. It also contains the information related to groups, specifying whether a group corresponds to an IP range or to a set of nodes.

During the enrichment phase, both the `path` and `operations` constructs of each intent are enhanced with concrete information derived from the database.

For the `path` constructs, the symbolic names used for endpoints and groups are replaced with their corresponding IP addresses. In the case of groups defined as collections of nodes rather than IP ranges, the enrichment process retrieves and lists the IP addresses of all nodes belonging to that group. During this phase, the translator also retrieves the identifier of the *service graph* to which the nodes involved in the intent belong. This identifier is then inserted into the corresponding field of the generated NSRs, ensuring that each requirement is explicitly associated with the correct service graph within the VEREFOO configuration.

For the `operations` constructs, the names of services, protocols, and traffic types are replaced with their detailed technical descriptions. Through the database, it is possible to determine the Layer 4 protocol and the corresponding source and destination ports associated with each service, protocol, or traffic label specified in the intent. This ensures that each NSR generated by the translator contains the full set of parameters required by VEREFOO.

The database therefore allows intents to be translated accurately without requiring the user to manually specify all missing technical details. Moreover, an experienced user can populate the database in advance, ensuring that even high-level intents can be correctly and consistently translated, while maintaining a clear separation of expertise between intent definition and network configuration.

In summary, the combination of pre-populated technical information and conflict-aware intent formulation enables accurate and efficient translation from NILE to VEREFOO. It minimizes the user's workload while preserving the expressiveness and abstraction level that characterize high-level intent-based network descriptions.

5.3.3 Grammar Modifications

After analyzing the structure of intents in NILE and the input requirements of VEREFOO, it is possible to define and implement a translation method based on the semantic mapping between the two, which can also address the challenge of the different constructs in NILE and VEREFOO. This thesis is based on a specific use case, the automatic configuration of firewall packet filters through VEREFOO. Consequently, it is clear that the translation must exclude certain NILE constructs related to QoS, intervals, and the addition or removal of middleboxes, as these are neither handled nor required by VEREFOO for the intended use case.

Figure 5.4 shows the correspondence between NILE's constructs and VEREFOO's requirements after reducing the NILE's constructs. It is possible to notice that the path elements of NILE intents correspond to the source and destination fields of the VEREFOO's NSRs, while the rules defined within the operations of NILE intents are translated into the properties of the NSRs. As mentioned before, the *rules's* construct of NILE can only address two out of the three properties of VEREFOO's NSRs; to facilitate the mapping, a third action, `complete_allow`, has been added to the rules's actions of NILE to match the remaining property of VEREFOO's NSRs.

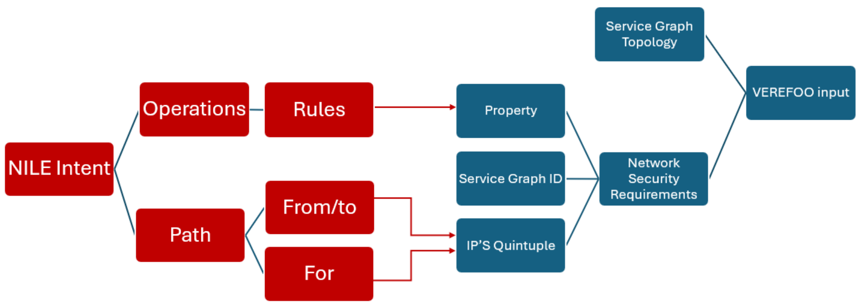


Figure 5.4: Mapping between NILE and VEREFOO requirements

Some modifications have been made to the original NILE grammar presented in [12] and discussed in the previous chapter. These adjustments were necessary because, as mentioned before, the translator is designed for a specific use case, the automatic configuration of firewall packet filters through VEREFOO, and therefore does not support all constructs of the full NILE grammar. Consequently, the translator excludes NILE's constructs related to QoS, intervals, and the addition or removal of middleboxes, as these are neither handled nor required by VEREFOO for the intended use case. Listing 5.3 shows the version of the NILE grammar used by the translator. All operations related to QoS, intervals, and middlebox management have been removed. Additionally, the `for` construct is restricted to use only with `group` or `endpoint`, the `middlebox` construct is eliminated, and a new `action` construct is introduced to group the possible operations. Within *action*, `complete_allow` is added to enable mapping to VEREFOO's `CompleteReachability` property.

```
<start> ::= <intent>

<intent> ::= "define" "intent" <TERM> ":" <operations>

<operations> ::= <path> <operation>+

<path> ::= <from_to> | <targets>

<from_to> ::= "from" <endpoint> "to" <endpoint>

<operation> ::= <rules>

<rules> ::= <ACTION> <matches>+

<ACTION> ::= "allow" | "block" | "complete_allow"

<targets> ::= "for" <target> ("," <target>)*

<target> ::= <group> | <endpoint>

<matches> ::= <service> | <traffic> | <protocol>

<endpoint> ::= "endpoint(" <TERM> ")"

<group> ::= "group(" <TERM> ")"

<service> ::= "service(" <TERM> ")"

<traffic> ::= "traffic(" <TERM> ")"

<protocol> ::= "protocol(" <TERM> ")"

<TERM> ::= /[a-zA-Z0-9_.-]+/
```

Listing 5.3: Updated NILE grammar supported by the translator

5.3.4 Translation Rules

Based on the mapping shown in Figure 5.4 and the adapted grammar, the translation process interprets each NILE's construct as follows.

Path Translation The `from/to` construct accepts only the `endpoint` construct and expresses the directionality of the intent. Therefore, the `endpoint` in `from` is translated as the source, and the `endpoint` in `to` as the destination.

In contrast, the `for` construct accepts only `group` and `endpoint` constructs, and since it does not express directionality, it is translated bidirectionally. This means that two NSRs are created with the same action but with the source and destination inverted. An `endpoint` is interpreted as a single arbitrary node in the network.

A `group`, on the other hand, can be interpreted in two different ways depending on its definition in the database. A group can be defined either through an IP address range or through a name. In the latter case, the individual nodes indicate whether they belong to that group in the databse for the enrichment phase. In the case of an IP range, the group is translated using VEREFOO's `-1` wildcard, which indicates that the NSR applies to all nodes with IP addresses within that range, as shown in Listing 5.4.

```
<Property name="ReachabilityProperty" graph="1" src="130.10.0.-1"
   dst="40.40.41.1" lv4proto="TCP" dst_port="3074"/>
```

Listing 5.4: Generated NSR showing the use of the `-1` wildcard for the IP range.

In the other case, when a group is defined by name, NSRs are created for each node that belongs to the group.

Operations Translation The operations defined within a NILE intent determine the specific properties of the NSRs generated during translation. Each rule's action (`allow`, `block`, or `complete_allow`) guides the selection of the corresponding VEREFOO property.

The constructs `service`, `protocol`, and `traffic` specify the technical parameters that characterize the communication flow. They are used to populate the IP five-tuple fields (Layer 4 protocol, source port, and destination port) thanks to the enrichment phase that is explained in the next part. Services and protocols are intended to represent the classical high level services and protocols. Traffic, instead, is used to identify packet flows that do not have the standard source/destination ports associated with services or protocols.

5.4 Translator Implementation

This section presents the implementation of the translator that bridges the NILE intent language with the VEREFOO verification framework. As introduced in the previous section, the goal of this component is to transform high-level NILE intents, representing user requirements in an abstract and technology-independent form, into concrete NFV XML input files suitable for formal network verification in VEREFOO.

The implementation was developed in Python following a modular architecture to ensure clarity, maintainability, and extensibility. Each module is responsible for a specific aspect of the translation workflow, from syntactic parsing to database enrichment and XML generation, thereby enabling independent testing and future integration of additional functionalities.

The translator is composed of the following main modules:

- main.py
- grammar.py
- transformer.py
- enrich.py
- constructor.py
- utils.py
- topology_manager.py
- property.py
- xml_converter.py
- create_db.py

The following subsections describe the workflow of the translator and detail the role of each module within the overall architecture.

5.4.1 Translation Workflow

As can be inferred from the previous section, the translation from NILE intents to VEREFOO input files involves a series of well-defined steps designed to ensure both accuracy and consistency. The workflow orchestrates parsing, enrichment, property generation, and XML file creation, integrating information from the user-provided database to supply technical details that are not explicitly specified in high-level intents.

Figure 5.5 provides a compact overview of this process, illustrating how each stage contributes to the generation of fully specified NSRs that are ready for verification by VEREFOO.

As illustrated by the workflow, the translation process proceeds as follows: first, the intent is read from the text file and parsed. During parsing, any extraneous information is removed, and the intent is converted into a Python object. The translator then enriches this object by retrieving the technical details required by VEREFOO from the database, including the IP addresses of the nodes, the source and destination ports, and the Layer 4 protocols.

Next, the enriched intent is used to extract the relevant information for generating the NSR properties. Simultaneously, the service graph to which the intent belongs is reconstructed using the database.

Finally, the generated NSRs and the associated service graph are converted into XML format and written into a single output file, which serves as input for VEREFOO.

The translator is also capable of handling multiple intents in a single session. Each intent follows the same workflow independently; however, all converted outputs are aggregated into a single XML file. Importantly, each NSR is associated with its corresponding service graph, ensuring that NSRs do not span multiple service graphs but remain correctly linked to the intended one.

5.4.2 The `main.py` Module

The `main.py` module serves as the entry point of the NILE-to-VEREFOO translator. Its primary responsibility is to orchestrate the complete translation workflow, coordinating the various submodules and ensuring that the input intents are correctly processed and converted into a final XML file suitable for VEREFOO.

The module provides a function, `process_multiple_intents()`, which handles the processing of one or more NILE intent files.

The module also includes a `__main__` block to allow direct execution, which invokes `process_multiple_intents()` on a specified input file and prints a summary of the enriched intents, generated properties, and associated network service graph IDs.

Overall, `main.py` abstracts the orchestration of parsing, enrichment, property construction, and XML generation, providing a single entry point for the translation process and ensuring a modular and maintainable structure.

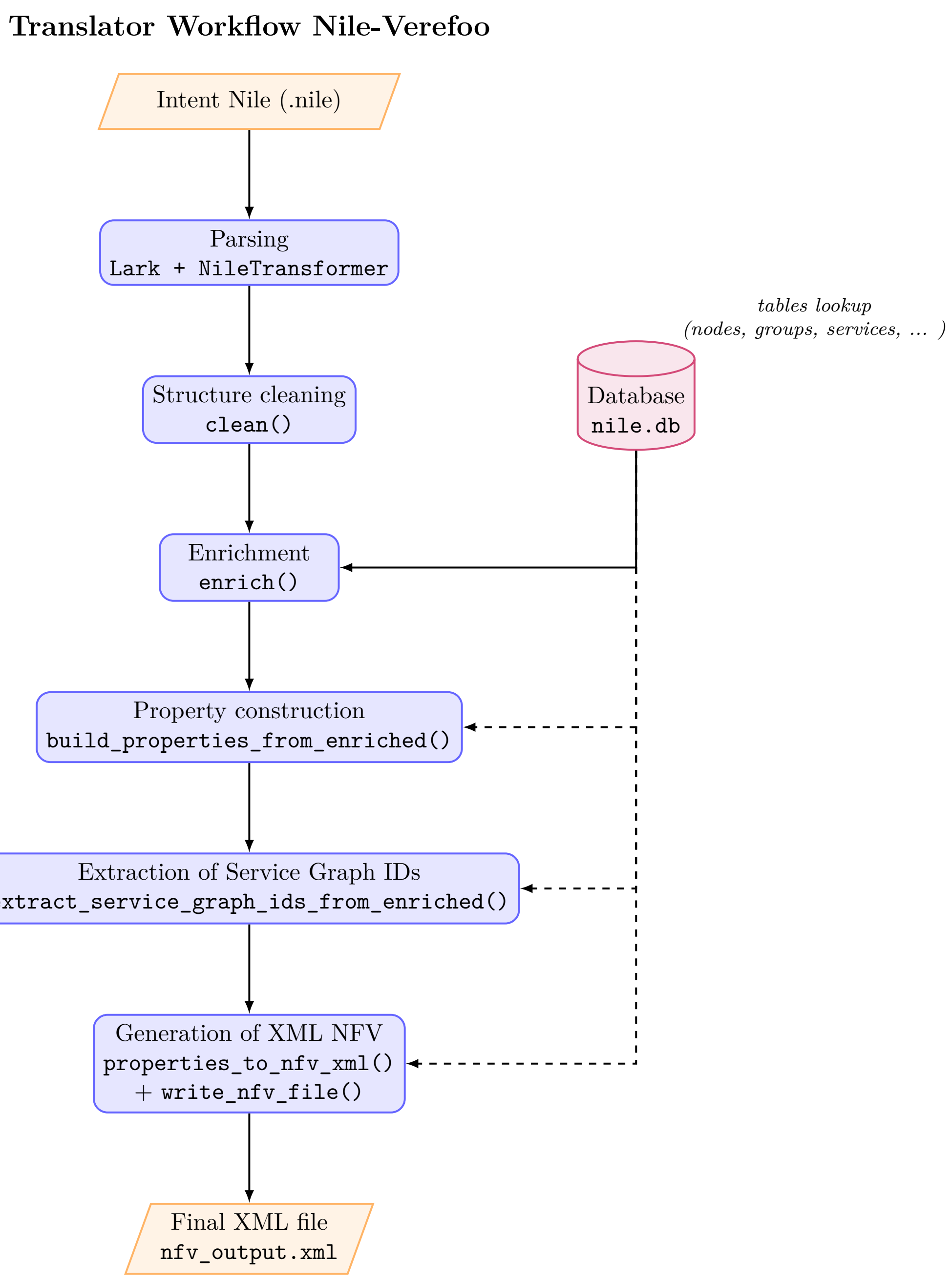


Figure 5.5: Translator workflow from NILE intent to XML input for VEREFOO.

5.4.3 The `transformer.py` Module

The `transformer.py` module defines the `NileTransformer` class, which is a custom transformer used by the Lark parser to convert the abstract syntax tree (AST) of a NILE intent into structured Python data.

`NileTransformer` inherits from `lark.Transformer` and implements methods corresponding to the grammar rules defined in `grammar.py`. Each method receives nodes from the parse tree and returns a Python dictionary or list representing the semantic structure of the intent.

The transformer ensures that the hierarchical structure of the NILE intent is preserved while converting it into a form that can be easily processed by subsequent modules such as `enrich.py` and `property.py`. This abstraction allows the translator to work directly with Python-native structures rather than raw parse trees, simplifying cleaning, enrichment, and property extraction steps.

In essence, `transformer.py` provides the semantic bridge between syntactic parsing and the higher-level processing stages of the translator, making it a central component in the workflow from NILE intents to VEREFOO-compatible XML.

5.4.4 The `grammar.py` Module

The `grammar.py` module defines the updated formal grammar of NILE used by the translator, seen previously. It contains a single variable, `nile_grammar`, which specifies the syntax rules in Lark's EBNF notation.

The grammar describes the structure of NILE intents, including:

- **Intents:** Defined using the keyword `define intent`, each intent has a name and a set of operations.
- **Operations:** Sequences of rules applied either along a path from a source endpoint to a destination endpoint (`from ... to ...`) or targeting specific nodes/groups.
- **Rules:** Each operation specifies an action (`allow`, `block`, or `complete_allow`) and associated match criteria, such as services, traffic types, or protocols.
- **Targets and endpoints:** Grammar elements for endpoints, groups, and service identifiers.

Compared to the NILE grammar presented in the previuos section, the version used in this translator has been slightly modified to simplify parsing and accommodate multi-line target lists. For example, the handling of comma-separated targets allows optional newlines to improve readability in intent files.

The defined grammar is then used by `main.py` in conjunction with the transformer to parse the raw NILE input files into structured Python objects, forming the foundation for the subsequent cleaning, enrichment, and property construction steps.

In summary, `grammar.py` establishes the syntactic rules of the NILE language in a format directly usable by the parser, ensuring a robust and extendable translation workflow.

5.4.5 The `enrich.py` Module

The `enrich.py` module is responsible for enriching the parsed NILE intents with detailed information retrieved from the local database (`nile.db`). Its main function, `enrich()`, takes a dictionary representing a parsed intent and recursively populates it with complete data about endpoints, groups, services, traffic, and protocols.

The enrichment process relies on several key mechanisms:

- **Database lookups:** For each entity (endpoint, group, service, traffic, protocol), the module queries the corresponding database table to retrieve attributes such as IDs, service graph associations, ports, protocols, and other configuration data.
- **Type-specific construction:** Using the `constructors` dictionary from `constructor.py`, the module instantiates objects representing the enriched entities, allowing uniform representation for subsequent processing.
- **Node enrichment:** Endpoints are resolved into specific node types (e.g., FIREWALL, ENDHOST, CACHE, MAILCLIENT) by consulting additional tables for type-specific configuration details. This ensures that the enriched data captures the functional behavior of each network element.
- **Recursive enrichment:** The `enrich()` function applies a recursive procedure to traverse all operations and nested structures of the intent. This allows context propagation (e.g., source, destination, targets) across multiple levels of the operations hierarchy.

By the end of the enrichment process, each intent is transformed into a fully annotated representation containing all necessary information for property construction and eventual conversion into NFV XML. This module thus serves as the semantic bridge between the syntactic representation produced by `NileTransformer` and the formal network properties required by VEREFOO.

In summary, `enrich.py` ensures that every element in a NILE intent is accurately contextualized with real network and service information, providing the foundation for reliable automated verification.

5.4.6 The `constructor.py` Module

The `constructor.py` module provides a collection of functions to build standardized Python objects representing network elements, groups, services, traffic patterns, protocols, and properties. These constructor functions are used throughout the translator to create enriched, structured representations of entities extracted from NILE intents and the database.

Key features of the module include:

- **Element constructors:** Functions such as `build_firewall()` generate objects representing specific types of network nodes, with all necessary attributes.
- **Service and traffic constructors:** Functions that produce standardized representations for network services, traffic flows, and protocol definitions.
- **Group and service graph constructors:** Functions such as `build_group()` and `build_serviceg()` create structured objects representing groups of nodes and their associated service graphs.
- **Property construction:** The function `build_property()` constructs a formal NSR object in line with the XSD schema expected by VEREFOO, including source, destination, protocol, ports, and other optional fields.
- **Centralized registry:** All constructor functions are collected in the *constructors dictionary*, enabling the `enrich.py` module to dynamically build any object type based on its entity type.

By using `constructor.py`, the translator ensures consistency in object creation, simplifies the enrichment process, and provides a clear separation between database retrieval and structured object generation. This design promotes modularity and maintainability while supporting a wide variety of network element types and properties.

In summary, `constructor.py` acts as a factory layer that standardizes the creation of all entities required by the translator, forming the backbone for building enriched intents and generating NFV-compatible XML outputs.

5.4.7 The `topology_manager.py` Module

The `topology_manager.py` module provides functions to retrieve and construct network node objects based on service graphs defined in the database. Its primary purpose is to bridge the gap between abstract service graphs and concrete node configurations, enabling the translator to handle network topology details accurately.

The main function, `get_nodes_by_servicegraph()`, takes a service graph ID and queries the database to retrieve all nodes belonging to that service graph. For each node, the function performs the following steps:

- **Node type detection:** Determines the node's functional type (e.g., FIREWALL, ENDHOST, etc.).
- **Configuration retrieval:** For each node type, the function fetches the corresponding configuration data from the appropriate table in the SQLite database (e.g., firewall rules, endhost settings etc.).
- **Object construction:** Using the constructors dictionary, the function builds a structured object representing the node, fully populated with its configuration data.
- **Node aggregation:** Each node object is collected into a list, along with metadata such as node ID, name, IP, and functional type, ready for use in enrichment and nsr generation.

Additionally, the module provides utility functions such as `get_name_config()`, which maps functional types to their corresponding configuration tables and retrieves configuration names for nodes.

This module encapsulates the logic for transforming database-stored service graph information into fully enriched node objects. This enables the translator to maintain consistency between NILE intents and the actual network topology, providing accurate inputs for NFV property generation and verification in VEREFOO.

5.4.8 The `property.py` Module

The `property.py` module is responsible for generating the `PROPERTY` objects that represent the NSRs of VEREFOO. The module provides two main functions:

- **`build_properties_for_targets()`**: Given two targets (either endpoints or groups), this function resolves their IP addresses and service graph IDs by querying the SQLite database. It then constructs a list of bidirectional NSR for all combinations of source and destination addresses. Optional parameters allow specifying the Layer 4 protocol and source/destination ports.
- **`build_properties_from_enriched()`**: This function processes an enriched NILE intent dictionary and extracts all rules defined in the intent. It handles both simple paths (from/to endpoints) and target lists (for groups or endpoints). For each rule, it determines the type of property (e.g., Reachability, Isolation, CompleteReachability), converts any range IPs into wildcard format, and

uses `constructors["PROPERTY"]` to generate the corresponding objects. The function also performs validation to ensure that targets belong to the same service graph.

Overall, `property.py` encapsulates the logic for translating NILE operations into formal NSRs compatible with VEREFOO. It combines database lookups, IP resolution, and rule interpretation to produce a set of verifiable objects that represent the intended network policies.

5.4.9 The `utils.py` Module

The `utils.py` module provides utility functions that support the parsing, transformation, and data extraction processes required to translate NILE intents into enriched objects and properties. Its main responsibilities include:

- **`clean()`**: Recursively transforms a Lark parse tree into a nested dictionary, list, or string structure. This conversion allows subsequent processing steps to work with standard Python data types instead of Lark objects (`Tree` or `Token`).
- **`get_group_node_ips()`**: Given a group ID, this function queries the database to obtain all IP addresses of the nodes belonging to that group. Any CIDR addresses are converted to wildcard notation using `cidr_to_wildcard()`.
- **`cidr_to_wildcard()`**: Converts IP addresses expressed in CIDR format (e.g., `192.168.1.0/24`) into a wildcard notation by replacing the last octet with `-1` (e.g., `192.168.1.-1`). This is used to normalize IPs for property generation.
- **`extract_service_graph_ids_from_enriched()`**: Extracts all unique service graph IDs from an enriched NILE intent. It recursively traverses paths in the intent, handling both `from/to` endpoints and `for` targets, and retrieves the corresponding service graph IDs from the database. The result is a list of all service graphs involved in the intent.

Overall, `utils.py` abstracts recurring operations such as tree-to-dictionary conversion, IP normalization, and database lookups, allowing the other modules (e.g., `enrich.py` and `property.py`) to focus on higher-level intent processing and property generation.

5.4.10 The `xml_converter.py` Module

The `xml_converter.py` module is responsible for transforming the internal Python representations of nodes and properties into XML files that comply with the NFV schema.

Its main components are:

- **`nodes_to_xml()`**: Converts a list of nodes for a given service graph into an XML representation. Each node element includes its IP, functional type, and configuration details. Neighbors are also retrieved from the database and represented in the XML. Configurations for firewalls, endhosts, endpoints, caches, mail clients/servers, NAT, web clients/servers, field modifiers, forwarders, and load balancers are handled, ensuring that all relevant attributes are properly included.
- **`properties_to_nfv_xml()`**: Transforms a list of PROPERTY objects into XML according to the NFV schema.
- **`write_nfv_file()`**: Integrates the nodes and NSRs into a complete NFV XML file. For each service graph ID, it retrieves the corresponding nodes, converts them into XML, and appends them under the `<graphs>` section. It also appends the NSRs under `<PropertyDefinition>` and creates placeholders for constraints and parsing information.

Overall, `xml_converter.py` acts as the final step in the translation pipeline, producing a structured NFV XML file that represents both the enriched network configuration and the derived NSRs, ready for VEREFOO.

5.4.11 The `create_db.py` Module

The `create_db.py` module is responsible for initializing and populating the SQLite database (`nile.db`) used throughout the translation pipeline. It defines the schema for all network-related entities and inserts initial data to support subsequent processing.

Key components of the module include:

- **Database Schema Creation:** The script creates tables for:
  - `ServiceGraphs`: defines logical network graphs.
  - `groups`: node groups associated with each service graph.
  - `services`, `traffics`, `protocols`: metadata about network services and traffic types.
  - `nodes`: network nodes including functional types and IPs.
  - `neighbours`: node connectivity relationships.
  - `[functional_type]_config`: specific configuration tables for functional nodes such as firewalls, endhosts, endpoints, caches, mail clients/servers, NAT, web clients/servers, field modifiers, forwarders, and load balancers.

- **Data Population:** After creating the tables, the module populates them with example data:
  - Service graphs and groups with IP ranges and descriptions.
  - Nodes with functional types (e.g., webserver, webclient, loadbalancer, NAT) and their IPs.
  - Neighbor relationships to define the network topology.
  - Configuration data for functional nodes, such as NAT sources, web client/server settings, forwarder names, and load balancer pools.

The database serves as the foundational repository for all the processing steps. Thanks to its structure, the enrichment phase can effectively access and manipulate the required data. Moreover, since it is implemented as an independent module, it can be easily extended and improved in future iterations of the framework.

5.5 Conclusions

The implementation of the NILE-to-VEREFOO translator confirms the feasibility of bridging high-level, intent-based network specifications with formal verification frameworks. Throughout the development, several challenges emerged from the conceptual and syntactic gap between the two models. While NILE emphasizes human-readable abstractions, VEREFOO requires a precise and technically complete representation of network topologies and their associated security requirements. This gap was effectively mitigated through the introduction of an enrichment phase, supported by a dedicated database providing the contextual information necessary for accurate translation.

Although the current version of the translator has proven effective for the targeted use case, it still supports only a subset of the NILE grammar. Constructs related to Quality of Service (QoS), time intervals, and dynamic middlebox management remain outside the current implementation, as they are not yet handled by VEREFOO. Nevertheless, the translator has been designed with extensibility in mind, enabling future integration of additional intent constructs and automated data retrieval mechanisms.

The next chapter focuses on the validation of this implementation, assessing the correctness, reliability, and practical applicability of the translator through a set of experimental evaluations.

Chapter 6

Validation of the Translation between NILE and VEREFOO

6.1 Introduction

This chapter presents the validation of the translator developed to convert intents expressed in the NILE language into XML input files compatible with VEREFOO. The objective of the validation is to assess the correctness, flexibility, and robustness of the translation process across a representative set of use cases.

Three distinct scenarios were designed and tested for this purpose. The first use case concerns a simple example illustrating the translation of intents expressed through the `from` construct. The second use case focuses on the translation of the `for` construct, the different ways of expressing `group`, and the handling of multiple intents within the same specification. Both use cases employ the same network topology, which allows a direct comparison between different linguistic forms and their corresponding XML representations. The third validation scenario builds on this structure but introduces a more complex network configuration, enabling the simultaneous processing of multiple intents expressed through both `from/to` and `for` constructs across multiple service graphs. This final test demonstrates the translator's ability to manage multi-graph inputs and to generate consistent VEREFOO XML configurations even in the presence of multiple heterogeneous and interdependent intent specifications.

6.1.1 Network Topology

The network topology adopted in the first two validation scenarios is shown in Figure 6.1. It models a multi-segment infrastructure comprising several functional components: web servers, web clients, load balancers, forwarders, and a NAT device.

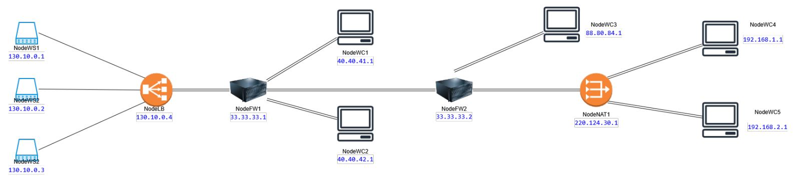


Figure 6.1: Network topology used for the first and second validation use cases.

Three web servers (`NodeWS1-3`) reside in the subnet `130.10.0.0/24`, behind a load balancer (`NodeLB1`, IP `130.10.0.4`). Client traffic directed to the servers traverses two chained forwarding domains, implemented through nodes `NodeFWD1` and `NodeFWD2`.

On the client side, five web clients (`NodeWC1-5`) are distributed across three subnets: `40.40.0.0/16`, which hosts clients `WC1` and `WC2`; `88.80.84.0/24`, which hosts client `WC3`; and `192.168.0.0/16`, which hosts clients `WC4` and `WC5`.

Each connection between nodes is explicitly defined in the topology through a list of neighbor relations, ensuring that the generated XML graph maintains both connectivity and node semantics.

The third validation scenario builds on this structure and extends it with a more complex network configuration, shown in Figure 6.2. This extended topology includes additional subnets and forwarding paths.

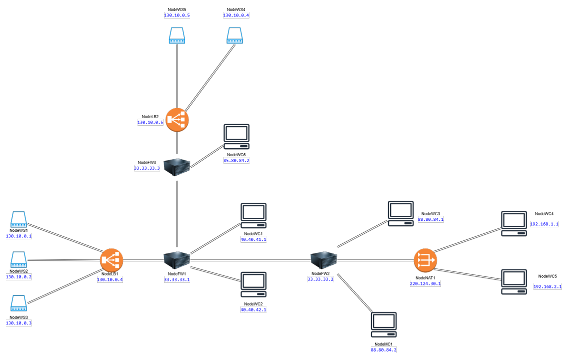


Figure 6.2: Extended network topology used for the third validation use case.

Overall, the validation process aims to demonstrate that the translator accurately captures the semantics of NILE intents and generates coherent VEREFOO XML configurations across different network setups, from simple to complex.

6.2 Use Case 1: Validation of the `from/to` Construct

The first validation scenario focuses on verifying the correct translation of intents defined through the `from/to` construct. This construct specifies the directionality of the intent by identifying a source and a destination endpoint, each corresponding to a single node in the network topology. In this use case, the intent defines traffic rules between two endpoints, namely `NodeWS2` and `NodeWC5`.

```
define intent one:
    from endpoint(NodeWS2) to endpoint(NodeWC5)
      complete_allow service(WebSecService)
      allow protocol(FTP) traffic(Traffic1)
      block service(XBOXLive) protocol(POP3) traffic(Traffic2)
```

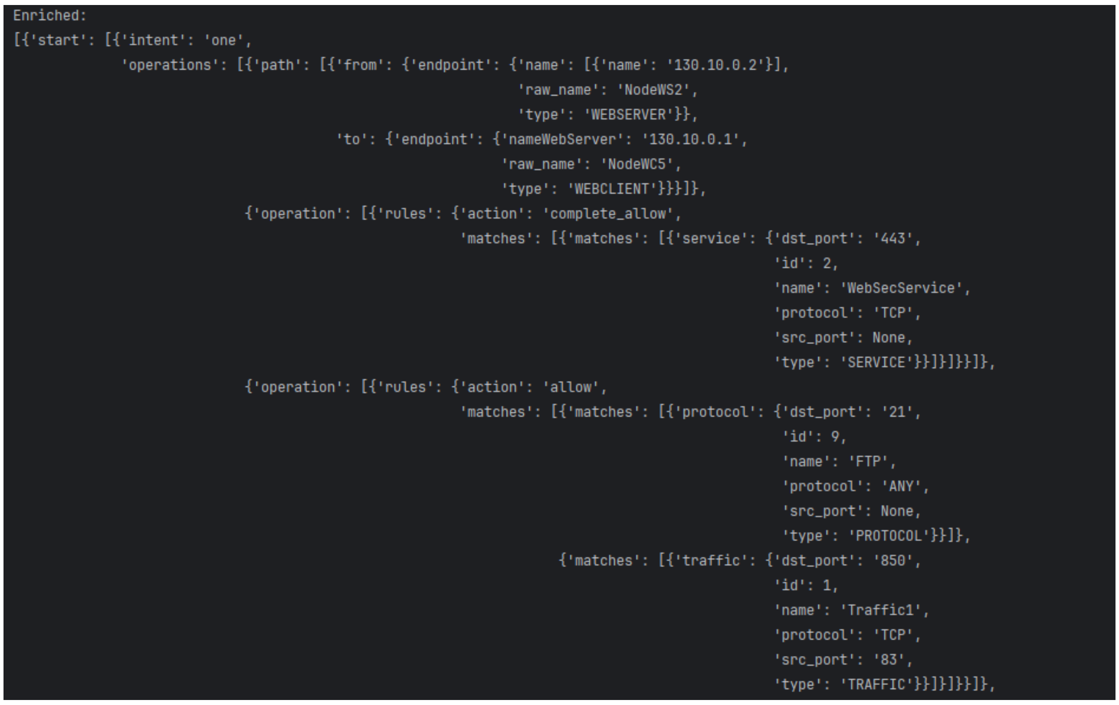
The intent defines three distinct policies. The first one allows all packet flows involving the `WebSecService` service and is translated into a CompleteReachabilityProperty. The second policy permits at least one packet flow using the `FTP` protocol and one packet's flow using the traffic profile `Traffic1`, translating it into the ReachabilityProperty. Finally, the third policy denies all packet flows related to the `XBOXLive` service, the `POP3` protocol, and the traffic profile `Traffic2` translating it into the IsolationProperty.

Before translation, the semantic enrichment step retrieves the definitions of all referenced elements from the database, as summarised below:

- **Service `WebSecService`** is defined as `(id = 2, name = 'WebSecService', protocol = 'TCP', dst_port = '443')`, representing a secure web service accessible through TCP port 443.
- **Service `XBOXLive`** is defined as `(id = 4, name = 'XBOXLive', protocol = 'TCP', dst_port = '3074')`, corresponding to the XBOXLive service.
- **Protocol `POP3`** is defined as `(id = 8, name = 'POP3', protocol = 'TCP', dst_port = '110')`, representing the protocol used for email retrieval.
- **Protocol `FTP`** is defined as `(id = 9, name = 'FTP', protocol = 'ANY', dst_port = '21')`, representing the File Transfer Protocol.

- **Traffic `Traffic1`** is defined as `(id = 1, name = 'Traffic1', protocol = 'TCP', src_port = '83', dst_port = '850')`, describing a specific flow allowed by the intent.
- **Traffic `Traffic2`** is defined as `(id = 2, name = 'Traffic2', protocol = 'TCP', src_port = '25', dst_port = '215')`, describing the specific traffic blocked by the intent.
- **Node**: `NodeWS2 (id=12, IP 130.10.0.2, is a web server)`
- **Node**: `NodeWC5 (id=12, IP 192.168.2.1, is a web client)`

The network topology used in this use case is shown in Figure 6.1. The translator correctly identified the semantics of the `from/to` intent, enriched it with the information stored in the database, and produced the corresponding `nsr` (network service request) elements that reflect the `complete_allow`, the `allow` and `block` rules defined in the intent. The recognized and enriched intent is shown in Figure 6.3, while the generated `nsr` elements are reported in Listing 6.1.



```
Enriched:
[{'start': [{'intent': 'one',
             'operations': [{'path': [{'from': {'endpoint': {'name': [{'name': '130.10.0.2'}],
                                                             'raw_name': 'NodeWS2',
                                                             'type': 'WEBSERVER'}},
                                       'to': {'endpoint': {'nameWebServer': '130.10.0.1',
                                                           'raw_name': 'NodeWC5',
                                                           'type': 'WEBCLIENT'}}}]},
                            {'operation': [{'rules': {'action': 'complete_allow',
                                                      'matches': [{'matches': [{'service': {'dst_port': '443',
                                                                                            'id': 2,
                                                                                            'name': 'WebSecService',
                                                                                            'protocol': 'TCP',
                                                                                            'src_port': None,
                                                                                            'type': 'SERVICE'}}]}]}}]},
                            {'operation': [{'rules': {'action': 'allow',
                                                      'matches': [{'matches': [{'protocol': {'dst_port': '21',
                                                                                             'id': 9,
                                                                                             'name': 'FTP',
                                                                                             'protocol': 'ANY',
                                                                                             'src_port': None,
                                                                                             'type': 'PROTOCOL'}}]},
                                                                  {'matches': [{'traffic': {'dst_port': '850',
                                                                                            'id': 1,
                                                                                            'name': 'Traffic1',
                                                                                            'protocol': 'TCP',
                                                                                            'src_port': '83',
                                                                                            'type': 'TRAFFIC'}}]}]}}]},
```

Figure 6.3: Excerpt of the recognized and enriched intent in JSON

```xml
<PropertyDefinition>
  <Property name="CompleteReachabilityProperty" graph="1" src="
130.10.0.2" dst="192.168.2.1" lv4proto="TCP" dst_port="443"/>
```

```
    <Property name="ReachabilityProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="ANY" dst_port="21"/>
    <Property name="ReachabilityProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" src_port="83" dst_port="850"/>
    <Property name="IsolationProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" dst_port="3074"/>
    <Property name="IsolationProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" dst_port="110"/>
    <Property name="IsolationProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" src_port="25" dst_port="215"/>
  </PropertyDefinition>
```

Listing 6.1: Generated NSR elements reflecting the intent rules

As shown in Listing 6.2, the translator successfully generated the correct VEREFOO XML configuration. Each rule derived from the NILE intent is mapped to an `nsr` entry, representing the corresponding `complete_allow`, `allow`, or `block` operation. This confirms that the `from/to` construct is correctly interpreted, endpoints are correctly associated with individual network nodes, and the translation preserves both the directionality and semantics of the original intent.

```
<?xml version="1.0" ?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="1">
      <node name="130.10.0.1" functional_type="WEBSERVER">
        <neighbour id="4" name="60.0.0.1"/>
        <configuration id="1" name="confB">
          <webserver>
            <name>130.10.0.1</name>
          </webserver>
        </configuration>
      </node>
      ...
      <node name="192.168.2.1" functional_type="WEBCLIENT">
        <neighbour id="21" name="60.0.0.13"/>
        <configuration id="23" name="confB">
          <webclient nameWebServer="130.10.0.1"/>
        </configuration>
      </node>
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints/>
    <LinkConstraints/>
  </Constraints>
  <PropertyDefinition>
```

```
    <Property name="CompleteReachabilityProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" dst_port="443"/>
    <Property name="ReachabilityProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="ANY" dst_port="21"/>
    <Property name="ReachabilityProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" src_port="83" dst_port="850"/>
    <Property name="IsolationProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" dst_port="3074"/>
    <Property name="IsolationProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" dst_port="110"/>
    <Property name="IsolationProperty" graph="1" src="130.10.0.2" dst="192.168.2.1" lv4proto="TCP" src_port="25" dst_port="215"/>
  </PropertyDefinition>
  <ParsingString/>
</NFV>
```

Listing 6.2: Generated VEREFOO XML configuration for the first use case

The results obtained in this first validation scenario confirm that the translator correctly processes intents expressed through the `from/to` construct, preserving their directional semantics and accurately reflecting the relationships between source and destination endpoints.

Each of the three operations defined in the intent, `complete_allow`, `allow`, and `block`,was successfully mapped to the corresponding VEREFOO properties, demonstrating a one-to-one correspondence between NILE semantics and VEREFOO XML representation.

Furthermore, the enrichment step proved to be effective: all references to services, protocols, and traffic profiles were correctly resolved through the database, confirming the translator's ability to integrate external semantic information into the final XML configuration.

Overall, this scenario validates the translator's handling of directional communication intents in simple network contexts, establishing a solid foundation for the following validation scenarios, which introduce non-directional constructs and multi-intent specifications.

6.3 Use Case 2: Validation of the `for` Construct

The second validation scenario focuses on the translation of intents defined using the `for` construct. Unlike `from/to`, this construct does not define a directionality; therefore, each intent is translated into two `nsr` entries to represent bidirectional communication. The `for` construct accepts exactly two arguments, which can be either an endpoint or a group. Groups represent sets of nodes, which can be defined either by an IP range or by associating individual nodes to the group in the database. The network topology used in this use case is shown in Figure 6.1.

Three intents were considered in this scenario:

```
define intent one:
    for endpoint(NodeWS1), group(Group3)
      allow service(WebService)
      block service(WHOIS)

define intent two:
    for group(Group1), endpoint(NodeWC1)
      allow service(XBOXLive)
      block service(AppleTalk)

define intent three:
    for group(Group1), group(Group3)
      allow traffic(Traffic1)
      block service(Steam)
```

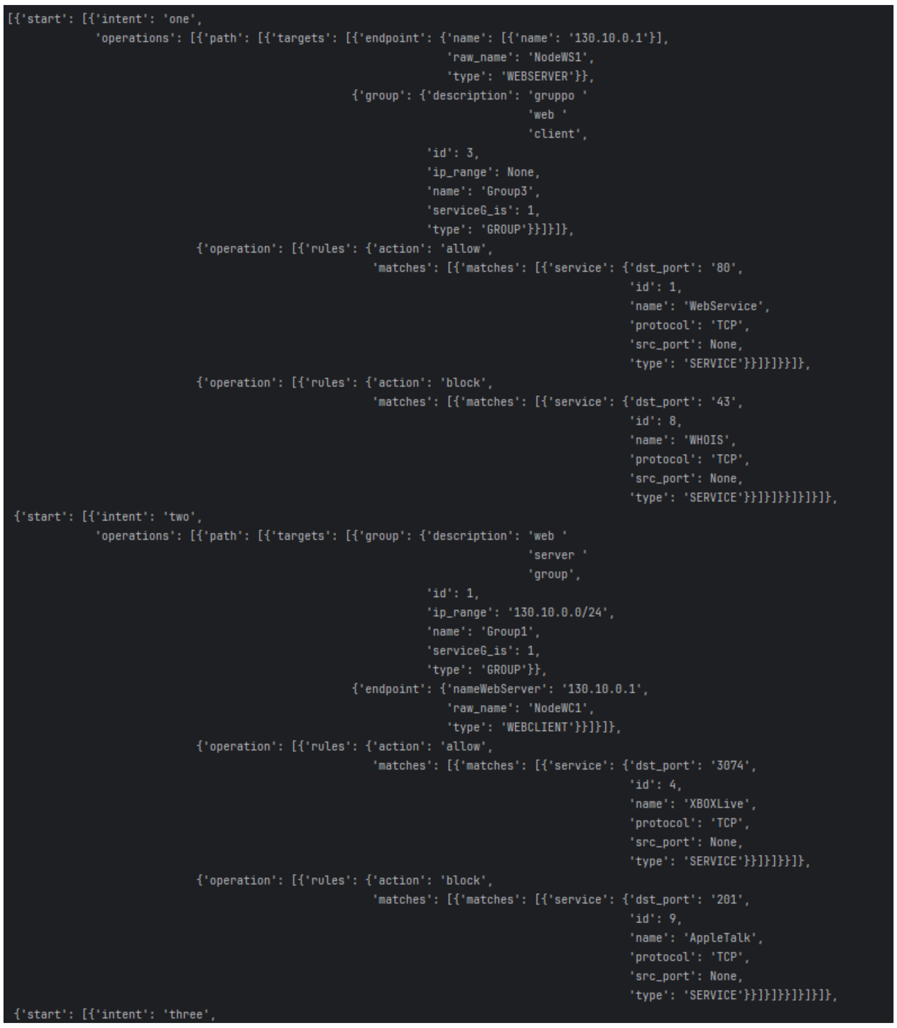
Listing 6.3: NILE intents using the `for` construct.

The first intent illustrates the use of `for` with an endpoint and a group defined by explicitly associating nodes in the database. The second intent uses a group defined via an IP range, while the third intent combines two groups, covering both types of group definitions

Before translation, the semantic enrichment step retrieves the definitions of all referenced elements from the database, as summarised below:

- **Service**: `WebService (id=1, TCP, dst_port=80)`
- **Service**: `WHOIS (id=10, TCP, dst_port=43)`
- **Service**: `XBOXLive (id=6, TCP, dst_port=3074)`
- **Service**: `AppleTalk (id=11, TCP, dst_port=201)`
- **Service**: `Steam (id=4, TCP, dst_port=445)`
- **Traffic**: `Traffic1 (id=1, TCP, src_port=83, dst_port=850)`
- **Group**: `Group1 (id=1, IP range 130.10.0.0/24, group web server)`
- **Group**: `Group3 (id=3, group web client)`
- **Node**: `NodeWC1 (id=12, IP 40.40.41.1)`

Figure 6.4 shows the enriched representation of the three intents after the translator processed the `for` constructs, including all endpoints, groups, and the corresponding allow/block rules.



```
[{'start': [{'intent': 'one',
             'operations': [{'path': [{'targets': [{'endpoint': {'name': [{'name': '130.10.0.1'}],
                                                                  'raw_name': 'NodeWS1',
                                                                  'type': 'WEBSERVER'}},
                                                    {'group': {'description': 'gruppo '
                                                                              'web '
                                                                              'client',
                                                               'id': 3,
                                                               'ip_range': None,
                                                               'name': 'Group3',
                                                               'serviceG_is': 1,
                                                               'type': 'GROUP'}}]}]},
                            {'operation': [{'rules': {'action': 'allow',
                                                      'matches': [{'matches': [{'service': {'dst_port': '80',
                                                                                            'id': 1,
                                                                                            'name': 'WebService',
                                                                                            'protocol': 'TCP',
                                                                                            'src_port': None,
                                                                                            'type': 'SERVICE'}}]}]}}]},
                            {'operation': [{'rules': {'action': 'block',
                                                      'matches': [{'matches': [{'service': {'dst_port': '43',
                                                                                            'id': 8,
                                                                                            'name': 'WHOIS',
                                                                                            'protocol': 'TCP',
                                                                                            'src_port': None,
                                                                                            'type': 'SERVICE'}}]}]}}]}]}]},
 {'start': [{'intent': 'two',
             'operations': [{'path': [{'targets': [{'group': {'description': 'web '
                                                                              'server '
                                                                              'group',
                                                               'id': 1,
                                                               'ip_range': '130.10.0.0/24',
                                                               'name': 'Group1',
                                                               'serviceG_is': 1,
                                                               'type': 'GROUP'}},
                                                    {'endpoint': {'nameWebServer': '130.10.0.1',
                                                                  'raw_name': 'NodeWC1',
                                                                  'type': 'WEBCLIENT'}}]}]},
                            {'operation': [{'rules': {'action': 'allow',
                                                      'matches': [{'matches': [{'service': {'dst_port': '3074',
                                                                                            'id': 4,
                                                                                            'name': 'XBOXLive',
                                                                                            'protocol': 'TCP',
                                                                                            'src_port': None,
                                                                                            'type': 'SERVICE'}}]}]}}]},
                            {'operation': [{'rules': {'action': 'block',
                                                      'matches': [{'matches': [{'service': {'dst_port': '201',
                                                                                            'id': 9,
                                                                                            'name': 'AppleTalk',
                                                                                            'protocol': 'TCP',
                                                                                            'src_port': None,
                                                                                            'type': 'SERVICE'}}]}]}}]}]}]},
 {'start': [{'intent': 'three',
```

Figure 6.4: Excerpt of the enriched NILE intents.

The first intent illustrates how the `for` construct operates when one of the operands is a single node, while the other is a group composed of multiple nodes. In this case, the single node is `NodeWS1` (IP `130.10.0.1`) and the group is `Group3`, which contains the following web clients explicitly defined in the database:

- `NodeWC1`: IP `40.40.41.1`
- `NodeWC2`: IP `40.40.42.1`

- `NodeWC4`: IP `192.168.1.1`
- `NodeWC5`: IP `192.168.2.1`

Unlike groups defined via an IP range, this group explicitly lists its members in the database. Listing 6.4 shows how the translator recognizes the group and enumerates its member nodes:

```
Recognised PATH with FOR
  Target 1 = {'endpoint': {'type': 'WEBSERVER', 'name': [{'name': '130.10.0.1'}], 'raw_name': 'NodeWS1'}}
  Target 2 = {'group': {'type': 'GROUP', 'id': 3, 'serviceG_is': 1, 'name': 'Group3', 'ip_range': None, 'description': 'group web clients'}}
  IP addresses of nodes for group_id 3: ['40.40.41.1', '40.40.42.1', '88.80.84.1', '192.168.1.1', '192.168.2.1']
```

Listing 6.4: Translator output showing recognition of `Group3` and its member nodes.

As shown in Listing 6.5 he translator systematically expands `Group3`, generating one `nsr` entry for each individual IP in the group. Moreover, since the `for` construct is inherently bidirectional, for every policy the translator produces two complementary rules: one for traffic from the group member to the other endpoint, and another in the opposite direction.

```
    <Property name="ReachabilityProperty" graph="1" src="130.10.0.1" dst="40.40.41.1" lv4proto="TCP" dst_port="80"/>
    <Property name="ReachabilityProperty" graph="1" src="40.40.41.1" dst="130.10.0.1" lv4proto="TCP" dst_port="80"/>
    <Property name="ReachabilityProperty" graph="1" src="130.10.0.1" dst="40.40.42.1" lv4proto="TCP" dst_port="80"/>
    <Property name="ReachabilityProperty" graph="1" src="40.40.42.1" dst="130.10.0.1" lv4proto="TCP" dst_port="80"/>
    <Property name="ReachabilityProperty" graph="1" src="130.10.0.1" dst="192.168.1.1" lv4proto="TCP" dst_port="80"/>
...
```

Listing 6.5: Excerpt of the generated VEREFOO XML configuration for Intent One. Each pair of entries represents bidirectional communication between `NodeWS1` and a node in `Group3`.

The second intent illustrates how the `for` construct operates when one of the operands is a group defined through an IP range. In this case, `Group1` represents all the web servers of the network, identified by the IP range `130.10.0.0/24`.

The Listing 6.6 clearly shows how the translator recognises that `Group1` corresponds to an IP range rather than a set of discrete endpoints.

```
=== Processing intent #2 ===
Recognised PATH with FOR
  Target 1 = {'group': {'type': 'GROUP', 'id': 1, 'serviceG_is':
    1, 'name': 'Group1', 'ip_range': '130.10.0.0/24', 'description'
    : 'web server group'}}
  Target 2 = {'endpoint': {'type': 'WEBCLIENT', 'nameWebServer': '
    130.10.0.1', 'raw_name': 'NodeWC1'}}
  Converted IP CIDR 130.10.0.0/24 -> 130.10.0.-1
```

Listing 6.6: Translator output showing recognition of the IP range and conversion to wildcard notation.

Unlike a group defined as a set of discrete endpoints, the translator recognizes that `Group1` corresponds to a contiguous IP range. Instead of generating a separate `nsr` entry for each individual IP, it uses VEREFOO's `-1` wildcard notation to represent the entire range in a single entry. Moreover, since the `for` construct is inherently bidirectional, the translator produces two complementary `nsr` entries for each policy: one for traffic from the group to the other endpoint, and another for traffic in the opposite direction.

This behavior, including the use of the wildcard notation and the bidirectional `nsr` entries, can be observed in Listing 6.7.

```
<Property name="ReachabilityProperty" graph="1" src="130.10.0.-1"
    dst="40.40.41.1" lv4proto="TCP" dst_port="3074"/>
<Property name="ReachabilityProperty" graph="1" src="40.40.41.1"
    dst="130.10.0.-1" lv4proto="TCP" dst_port="3074"/>
<Property name="IsolationProperty" graph="1" src="130.10.0.-1" dst
    ="40.40.41.1" lv4proto="TCP" dst_port="201"/>
<Property name="IsolationProperty" graph="1" src="40.40.41.1" dst=
    "130.10.0.-1" lv4proto="TCP" dst_port="201"/>
```

Listing 6.7: Generated nsr for Intent Two showing the use of the `-1` wildcard for the IP range.

Finally, the third intent combines two groups, demonstrating the translator's capability to handle the most complex expansion case, where both operands represent multiple entities. As seen in Listing 6.8, the translator systematically generates all possible combinations between the members of `Group1` and `Group3`, producing the full set of bidirectional `nsr` entries for each allowed and blocked element.

```
<Property name="ReachabilityProperty" graph="1" src="130.10.0.-1"
    dst="40.40.41.1" lv4proto="TCP" src_port="83" dst_port="850"/>
<Property name="ReachabilityProperty" graph="1" src="40.40.41.1"
    dst="130.10.0.-1" lv4proto="TCP" src_port="83" dst_port="850"/>
<Property name="ReachabilityProperty" graph="1" src="130.10.0.-1"
    dst="40.40.42.1" lv4proto="TCP" src_port="83" dst_port="850"/>
```

```
<Property name="ReachabilityProperty" graph="1" src="40.40.42.1"
   dst="130.10.0.-1" lv4proto="TCP" src_port="83" dst_port="850"/>
...
```

Listing 6.8: Excerpt of the generated nsr for Intent Three.

As shown in the listings, the translator correctly interprets the `for` construct in all cases, automatically expanding groups and generating the full bidirectional set of `nsr` rules consistent with the semantics of the original NILE intents.

Overall, the translator correctly interprets each `for` intent, enriching it with database information and generating the corresponding bidirectional `nsr` entries, preserving the semantics of the original specification. This use case also demonstrates that multiple intents within the same network can be handled together by the translator and inserted in a single VEREFOO XML input file.

The service graph associated with the intents is reconstructed, and all constructs are properly interpreted, ensuring that the translation captures the intended policies for both endpoints and groups. The resulting XML accurately represents the interaction of multiple intents and the correct enforcement of services and traffic policies. Listing 6.9 shows the complete VEREFOO XML file generated after processing all three intents. This file includes all `nsr` entries for bidirectional enforcement of allowed and blocked services/traffic.

```
<?xml version="1.0" ?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="../../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="1">
      <node name="130.10.0.1" functional_type="WEBSERVER">
      ...
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints/>
    <LinkConstraints/>
  </Constraints>
  <PropertyDefinition>
    <Property name="ReachabilityProperty" graph="1" src="
   130.10.0.1" dst="40.40.41.1" lv4proto="TCP" dst_port="80"/>
    <Property name="ReachabilityProperty" graph="1" src="
   40.40.41.1" dst="130.10.0.1" lv4proto="TCP" dst_port="80"/>
    <Property name="ReachabilityProperty" graph="1" src="
   130.10.0.1" dst="40.40.42.1" lv4proto="TCP" dst_port="80"/>
    ...
  </PropertyDefinition>
  <ParsingString/>
</NFV>
```

Listing 6.9: Complete generated VEREFOO XML file for all intents in Use Case 2.

Overall, this second validation scenario demonstrates the translator's capability to correctly process non-directional intents expressed through the `for` construct. The tests confirmed that both forms of group definition, via explicit membership and via IP range, are properly expanded and represented in the VEREFOO XML output. The translator systematically generates the full set of bidirectional `nsr` entries, ensuring that the semantics of the original NILE intents are preserved across all cases. Additionally, this use case highlights the flexibility of the translation process, showing that the translator can handle multiple intents at once.

These results provide a solid basis for the next validation scenario, which introduces more complex network topologies and mixed use of `from/to` and `for` constructs.

6.4 Use Case 3: Validation with Multiple Intents Across Two Service Graphs

The third validation scenario demonstrates the translator's capability to handle multiple intents simultaneously, combining both `from/to` and `for` constructs, and involving two distinct network graphs. Ten intents were defined in total, with five intents corresponding to each service graph.

```
define intent one:
    for endpoint(NodeWS1), group(Group3)
      allow protocol(DNS) traffic(Traffic3)
      block service(WHOIS) traffic(Traffic4)

define intent two:
    for group(Group1), endpoint(NodeWC1)
      allow service(XBOXLive) traffic(Traffic5)
      block protocol(SMTP) traffic(Traffic6)

define intent three:
    for group(Group1), group(Group3)
      allow  traffic(Traffic1)
      block protocol(SMB)

define intent four:
    from endpoint(NodeWS1) to endpoint(NodeWC4)
```

```
      block protocol(FTP)

define intent five:
    from endpoint(NodeWS3) to endpoint(NodeWC1)
      block protocol(NetBIOS-Session)

define intent six:
    from endpoint(Node2WS1) to endpoint(Node2WC4)
      block protocol(FTP)

define intent seven:
    for group(Group4), endpoint(Node2WC1)
      allow service(XBOXLive) traffic(Traffic5)
      block protocol(SMTP) traffic(Traffic6)

define intent eight:
    for group(Group4), group(Group5)
      allow traffic(Traffic1)
      block protocol(SMB)

define intent nine:
    for endpoint(Node2WS1), group(Group5)
      allow protocol(DNS) traffic(Traffic3)
      block service(WHOIS) traffic(Traffic4)

define intent ten:
    from endpoint(Node2WS3) to endpoint(Node2WC1)
      block protocol(NetBIOS-Session)
```

The first service graph corresponds to the previously described network in Figure 6.1, while the second service graph represents an extended version of the same network with additional nodes and groups, as shown in Figure 6.2.

Each intent is enriched with details from the database, including services, protocols, traffic profiles, endpoints, and group membership, as in the previous use cases.

As can be seen in Listing 6.10, which shows an excerpt of the `nsr` entries generated by the translator, the translator correctly recognized that intents five and six belong to two different service graphs. Accordingly, the `nsr` elements were created with the correct `graph` attribute, reflecting the distinction between the two topologies and preserving the mapping of intents to network graphs.

```
  ...
```

```
<Property name="IsolationProperty" graph="1" src="130.10.0.3" dst=
    "40.40.41.1" lv4proto="TCP" dst_port="139"/>
<Property name="IsolationProperty" graph="2" src="130.10.0.1" dst=
    "192.168.1.1" lv4proto="ANY" dst_port="21"/>
...
```

Listing 6.10: Excerpt of `nsr` elements showing how intents belonging to different graphs are distinguished.

The translator successfully processed all ten intents, generating a single XML input file for VEREFOO that contains `nsr` entries for both service graphs. The file clearly distinguishes which `nsr` elements belong to each service graph, preserving the correct mapping between intents and the underlying network topology. Bidirectional policies, directional policies, and the combination of multiple intent types are correctly represented, demonstrating the translator's ability to manage complex and heterogeneous configurations (see Listing 6.11).

```
<?xml version="1.0" ?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="1">
      <node name="130.10.0.1" functional_type="WEBSERVER">
      ...
    </graph>
    <graph id="2">
      <node name="130.10.0.1" functional_type="WEBSERVER">
      ...
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints/>
    <LinkConstraints/>
  </Constraints>
  <PropertyDefinition>
    <Property name="ReachabilityProperty" graph="1" src="
    130.10.0.1" dst="40.40.41.1" lv4proto="UDP" dst_port="53"/>
    <Property name="ReachabilityProperty" graph="1" src="
    40.40.41.1" dst="130.10.0.1" lv4proto="UDP" dst_port="53"/>
    <Property name="ReachabilityProperty" graph="1" src="
    130.10.0.1" dst="40.40.42.1" lv4proto="UDP" dst_port="53"/>
    ...
    <Property name="IsolationProperty" graph="2" src="130.10.0.1"
    dst="85.80.84.2" lv4proto="UDP" src_port="94" dst_port="78"/>
    <Property name="IsolationProperty" graph="2" src="85.80.84.2"
    dst="130.10.0.1" lv4proto="UDP" src_port="94" dst_port="78"/>
    <Property name="IsolationProperty" graph="2" src="130.10.0.3"
    dst="40.40.41.1" lv4proto="TCP" dst_port="139"/>
  </PropertyDefinition>
  <ParsingString/>
</NFV>
```

Listing 6.11: Complete XML input file generated by the translator for VEREFOO.

This use case confirms that the translation workflow is capable of handling multiple intents within different network graphs, maintaining both semantic correctness and structural consistency in the resulting XML input.

Overall, this validation scenario demonstrates the completeness of the translation process across heterogeneous configurations. The translator not only preserves the logical consistency between intents and network graphs but also ensures interoperability between different intent constructs (`from/to` and `for`) within a unified VEREFOO input. The results confirm the robustness and scalability of the translation mechanism, which can manage multiple, concurrent intent definitions across distinct topologies without loss of semantic fidelity.

6.5 Conclusions

The validation activities presented in this chapter provide a comprehensive assessment of the translator developed to convert NILE intents into VEREFOO XML configurations. Through the three use cases, the translator was systematically tested across a range of scenarios, from simple directional intents to complex multi-intent specifications involving multiple service graphs.

The first use case demonstrated that intents expressed through the `from/to` construct are correctly interpreted, with all directional semantics preserved and accurately reflected in the generated XML. The enrichment process successfully resolved references to services, protocols, and traffic profiles, ensuring that the final configuration was both complete and semantically faithful to the original specification.

The second use case focused on the `for` construct, highlighting the translator's ability to handle bidirectional intents and group expansions. Both types of group definitions, explicit membership and IP range, were correctly processed, generating the full set of corresponding `nsr` entries in the XML output. This scenario also confirmed that multiple intents can be managed simultaneously within a single network, preserving the intended interaction of policies across endpoints and groups.

Finally, the third use case confirmed the translator's robustness in handling complex configurations with multiple intents across distinct service graphs. The translation process maintained clear separation between graphs while correctly integrating different intent constructs and bidirectional rules. The results demonstrate that the translator scales effectively and preserves semantic correctness even in heterogeneous and multi-graph scenarios.

Overall, the validation confirms that the developed translator reliably converts NILE intents into coherent VEREFOO input files, accurately reflecting the semantics of both directional and non-directional constructs. These outcomes establish a solid foundation for the practical deployment of the translator in more extensive and realistic network settings, supporting automated policy enforcement and verification in NFV environments.

Chapter 7

Conclusions and Future Works

This thesis addressed the problem of translating high-level, human-readable network intents expressed in the NILE language into the concrete XML input files required by the VEREFOO framework. To overcome the abstraction gap between intent-based specifications and VEREFOO's input files requirements, a structured translation process was defined and a prototype tool was implemented to automate the conversion from NILE intents to the requirements for the VEREFOO framework.

The comparative analysis of NILE and VEREFOO highlighted several challenges of the translation arising from their different roles. One of these challenges, for example, is that while NILE allows users to describe what the network should achieve without knowing the technical details, VEREFOO focuses on verifying the feasibility and means of achieving those goals, but based on technical requirements. This mismatch in abstraction demanded the introduction of an enrichment phase, supported by a dedicated database, to provide all technical details that are not explicitly defined in high-level intents. The resulting pipeline, composed of parsing, enrichment, NSR generation, and XML construction, proved effective in bridging the two systems.

The decision to adopt a modular architecture facilitated clarity and extensibility, and supporting future evolution of the framework. Validation across three use cases further confirmed the reliability and scalability of the translation process. The tool handled simple and complex intents alike, including scenarios involving multiple service graphs and heterogeneous constructs, consistently producing coherent VEREFOO configurations that faithfully represented the original specifications.

Overall, the outcomes of this work confirm the feasibility of semantic translation as a bridge between intent-based network management and automated verification frameworks. The developed translator constitutes a step toward making network security configuration more accessible, reducing the need for specialized expertise, and moving closer to fully intent-driven security automation.

7.1 Future works

Although the translator provides a functional and validated solution to the interoperability between NILE and VEREFOO, several avenues for improvement and expansion remain open.

7.1.1 Extension of the supported NILE grammar

Currently, the translator handles only a subset of NILE's expressiveness, focusing on constructs directly compatible with VEREFOO. Features such as Quality of Service (QoS) constraints, time-based rules, or add/remove middleboxes operations remain unsupported, primarily due to limitations in the current VEREFOO model. As discussed in the following subsection, one promising direction involves extending the accepted NILE grammar to explicitly include middlebox-related operations (e.g., `add` and `remove`), which could then be delegated to an automated orchestration component capable of managing these actions at runtime. Such an extension would allow intents to describe at the same time not only traffic requirements but also structural modifications to the service graph.

Regarding time-based rules, an intermediate approach could be employed even without native support in VEREFOO. Specifically, time constraints could be handled externally by executing the translation pipeline in advance and scheduling the application of the resulting output only at the times specified by the intent. This mechanism would preserve the semantics of temporal activation while relying entirely on existing components, making it a feasible short-term enhancement before integrating full temporal reasoning directly within the translator or the VEREFOO framework itself.

Future extensions of VEREFOO may enable the native integration of these constructs, thus further enlarging the scope and expressiveness of the translator.

7.1.2 Automated enrichment and dynamic data retrieval

The current enrichment phase relies on a manually populated database that contains technical details about nodes, services, protocols, and topologies. However, the modular design of the enrichment component, combined with its database-centric architecture, opens the door to more advanced future scenarios. In particular in one possible scenario, an external component could be automatically populate the database through intelligent data-collection mechanisms, allowing the user expressing the intent to operate without any knowledge of the underlying network. Automated discovery of topologies, services, and middleboxes could be performed through SDN controllers, orchestration platforms, or even OSINT techniques applied to network analysis. This intelligent module would gather real-time data about the network, dynamically updating the database and ensuring that the system always has the most up-to-date information.

In addition to this, the component could make it possible to integrate the live handling of intent with operations on middleboxes, such as `add` and `remove`, allowing the accepted NILE grammar to be expanded.

These advancements would enable the translator to function in live, continuously evolving network environments. This would significantly reduce the need for manual preparation, streamline operations, and improve the system's ability to adapt to changes on the fly. As a result, the system would become more autonomous and scalable, providing a highly adaptable solution for real-world network management and optimization.

7.1.3 Integration with AI-based intent interpretation

Although the proposed tool effectively translates well-defined NILE intents into the corresponding VEREFOO configuration, it still relies on the assumption that intents are already clear, consistent, and free of conflicts. A possible direction for future development is to introduce a preliminary phase with an AI chatbot capable of guiding the user in the transition from human language to NILE language, as in [12]. In this way, conflicts and problems could be resolved more easily and intuitively, as the AI could interact directly with the user, for example, by asking for clarification or suggesting alternatives when conflicts arise that cannot be resolved automatically. Moreover, the use of an AI interface would make the system more accessible even to less experienced users, thanks to the AI's ability to explain problems at a level appropriate to the user's expertise.

Also in this preliminary phase from human language to the NILE language, the component proposed in the previous section could also be integrated to support the creation of optimal intents. This component could provide critical information from the database to resolve issues such as the presence of two endpoints belonging to different service graphs but included in the same intent, which would lead to errors.

Integrating AI-based intent interpretation represents a promising evolution of the current work. By enabling an interactive dialogue between the user and the system, AI can support the creation of precise, consistent, and conflict-free intents, reducing the cognitive effort required from the operator. This can lay the foundation for a future pipeline capable of handling increasingly complex scenarios with minimal human intervention.

7.2 Final Remarks

The work presented in this thesis demonstrates the practicality and effectiveness of linking the IBN concept intent-based with automated verification security mechanisms. By establishing a structured translation pipeline and validating it in multiple scenarios, this research contributes to the broader vision of intent-driven, verifiable, and automated network security management.

Bibliography

[1] Aris Leivadeas and Matthias Falkner. «A Survey on Intent-Based Networking». In: *IEEE Communications Surveys & Tutorials* 25.1 (2023), pp. 625–655. DOI: `10.1109/COMST.2022.3215919` (cit. on pp. 1, 6–9).

[2] Francesco Pizzato, Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. «An intent-based solution for network isolation in Kubernetes». In: *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. 2024, pp. 381–386. DOI: `10.1109/NetSoft60951.2024.10588939` (cit. on pp. 4, 5, 21).

[3] Yosra Njah, Aris Leivadeas, and Matthias Falkner. «An AI-Driven Intent-Based Network Architecture». In: *IEEE Communications Magazine* 63.4 (2025), pp. 146–153. DOI: `10.1109/MCOM.001.2400143` (cit. on p. 7).

[4] Ahlam Fuad, Azza H. Ahmed, Michael A. Riegler, and Tarik Čičić. «An Intent-based Networks Framework based on Large Language Models». In: *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. 2024, pp. 7–12. DOI: `10.1109/NetSoft60951.2024.10588879` (cit. on p. 7).

[5] Md. Kamrul Hossain and Walid Aljoby. «NetIntent: Leveraging Large Language Models for End-to-End Intent-Based SDN Automation». In: *ArXiv* abs/2507.14398 (2025). URL: `https://api.semanticscholar.org/CorpusID:280270329` (cit. on p. 7).

[6] Yassin Abouelseoud and Minar El-Aasser. «Practical Implementation of Intent-Based Networking Using OpenDaylight and Rasa». In: *2025 International Conference on Machine Intelligence and Smart Innovation (ICMISI)*. 2025, pp. 100–105. DOI: `10.1109/ICMISI65108.2025.11115828` (cit. on p. 7).

[7] Nguyen Tu, Sukhyun Nam, and James Won-Ki Hong. «Intent-Based Network Configuration Using Large Language Models». In: *International Journal of Network Management* 35.1 (2025). e2313 nem.2313, e2313. DOI: `https://doi.org/10.1002/nem.2313`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/nem.2313`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.2313` (cit. on p. 8).

[8] Ankur Chowdhary, Abdulhakim Sabur, Neha Vadnere, and Dijiang Huang. «Intent-Driven Security Policy Management for Software-Defined Systems». In: *IEEE Transactions on Network and Service Management* 19.4 (2022), pp. 5208–5223. DOI: `10.1109/TNSM.2022.3183591` (cit. on p. 9).

[9] Yoshiharu Tsuzaki and Yasuo Okabe. «Reactive configuration updating for Intent-Based Networking». In: *2017 International Conference on Information Networking (ICOIN)*. 2017, pp. 97–102. DOI: `10.1109/ICOIN.2017.7899484` (cit. on p. 9).

[10] Sergio Rivera, Zongming Fei, and James Griffioen. «POLANCO: Enforcing Natural Language Network Policies». In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. 2020, pp. 1–9. DOI: `10.1109/ICCCN49398.2020.9209748` (cit. on pp. 9, 16).

[11] Bingchuan Tian et al. «Safely and automatically updating in-network ACL configurations with intent language». In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. Beijing, China: Association for Computing Machinery, 2019, pp. 214–226. ISBN: 9781450359566. DOI: `10.1145/3341302.3342088`. URL: `https://doi.org/10.1145/3341302.3342088` (cit. on p. 9).

[12] Arthur S. Jacobs, Ricardo J. Pfitscher, Rafael H. Ribeiro, Ronaldo A. Ferreira, Lisandro Z. Granville, Walter Willinger, and Sanjay G. Rao. «Hey, Lumi! Using Natural Language for Intent-Based Network Management». In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 625–639. ISBN: 978-1-939133-23-6. URL: `https://www.usenix.org/conference/atc21/presentation/jacobs` (cit. on pp. 9, 11, 12, 26, 44, 75).

[13] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. «Refining Network Intents for Self-Driving Networks». In: *Proceedings of the Afternoon Workshop on Self-Driving Networks*. SelfDN 2018. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 15–21. ISBN: 9781450359146. DOI: `10.1145/3229584.3229590`. URL: `https://doi.org/10.1145/3229584.3229590` (cit. on pp. 9–11, 25, 26).

[14] Mohammad Riftadi and Fernando Kuipers. «P4I/O: Intent-Based Networking with P4». In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. 2019, pp. 438–443. DOI: `10.1109/NETSOFT.2019.8806662` (cit. on p. 13).

[15] Rafael Hengen Ribeiro, Arthur Selle Jacobs, Luciano Zembruzki, Ricardo Parizotto, Eder John Scheid, Alberto Egon Schaeffer-Filho, Lisandro Zambenedetti Granville, and Burkhard Stiller. «A deterministic approach for extracting network security intents». In: *Computer Networks* 214 (2022), p. 109109. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.`

2022.109109. URL: https://www.sciencedirect.com/science/article/pii/S1389128622002389 (cit. on p. 13).

[16] Mariam Kiran, Eric Pouyoul, Anu Mercian, Brian Tierney, Chin Guok, and Inder Monga. «Enabling intent to configure scientific networks for high performance demands». In: *Future Generation Computer Systems* 79 (2018), pp. 205–214. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.04.020. URL: https://www.sciencedirect.com/science/article/pii/S0167739X1730626X (cit. on pp. 13, 15).

[17] Eder J. Scheid, Cristian C. Machado, Muriel F. Franco, Ricardo L. dos Santos, Ricardo P. Pfitscher, Alberto E. Schaeffer-Filho, and Lisandro Z. Granville. «INSpIRE: Integrated NFV-based Intent Refinement Environment». In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 2017, pp. 186–194. DOI: 10.23919/INM.2017.7987279 (cit. on p. 15).

[18] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. «Automation for Network Security Configuration: State of the Art and Research Trends». In: *ACM Comput. Surv.* 56.3 (Oct. 2023). ISSN: 0360-0300. DOI: 10.1145/3616401. URL: https://doi.org/10.1145/3616401 (cit. on pp. 19–21).

[19] Verizon Business. *2025 Data Breach Investigations Report*. https://www.verizon.com/business/resources/reports/dbir/. Accessed: 2025-09-07. 2025 (cit. on p. 20).

[20] Daniele Bringhenti, Simone Bussa, Riccardo Sisto, and Fulvio Valenza. «Atomizing Firewall Policies for Anomaly Analysis and Resolution». In: *IEEE Transactions on Dependable and Secure Computing* 22.3 (2025), pp. 2308–2325. DOI: 10.1109/TDSC.2024.3495230 (cit. on pp. 21, 22).

[21] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. «A novel abstraction for security configuration in virtual networks». In: *Computer Networks* 228 (2023), p. 109745. DOI: 10.1016/j.comnet.2023.109745. URL: https://www.sciencedirect.com/science/article/pii/S1389128623001901 (cit. on pp. 21, 22).

[22] Daniele Bringhenti and Fulvio Valenza. «GreenShield: Optimizing Firewall Configuration for Sustainable Networks». In: *IEEE Transactions on Network and Service Management* 21.6 (2024), pp. 6909–6923. DOI: 10.1109/TNSM.2024.3452150 (cit. on pp. 21, 23).

[23] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. «Automating VPN Configuration in Computer Networks». In: *IEEE Transactions on Dependable and Secure Computing* 22.1 (2025), pp. 561–578. DOI: 10.1109/TDSC.2024.3409073 (cit. on pp. 21, 23).

[24] Daniele Bringhenti, Simone Bussa, Riccardo Sisto, and Fulvio Valenza. «A Two-Fold Traffic Flow Model for Network Security Management». In: *IEEE Transactions on Network and Service Management* 21.4 (2024), pp. 3740–3758. DOI: `10.1109/TNSM.2024.3407159` (cit. on p. 21).

[25] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Automated optimal firewall orchestration and configuration in virtualized networks». In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 2020, pp. 1–7. DOI: `10.1109/NOMS47738.2020.9110402` (cit. on pp. 21, 22, 42).

[26] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Automated Firewall Configuration in Virtual Networks». In: *IEEE Transactions on Dependable and Secure Computing* 20.2 (2023), pp. 1559–1576. DOI: `10.1109/TDSC.2022.3160293` (cit. on pp. 21, 22, 30, 31, 36).

[27] Netgroup. *VEREFOO: VErified REFinement and Optimized Orchestrator*. Accessed: Sep. 13, 2025. 2025. URL: `https://github.com/netgroup-polito/verefoo` (cit. on pp. 21, 30, 32).

[28] Daniele Bringhenti, Francesco Pizzato, Riccardo Sisto, and Fulvio Valenza. «A Looping Process for Cyberattack Mitigation». In: *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*. 2024, pp. 276–281. DOI: `10.1109/CSR61664.2024.10679501` (cit. on p. 21).

[29] Francesco Pizzato, Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. «Automatic and optimized firewall reconfiguration». In: *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. 2024, pp. 1–9. DOI: `10.1109/NOMS59830.2024.10575212` (cit. on p. 21).

[30] Daniele Bringhenti, Francesco Pizzato, Riccardo Sisto, and Fulvio Valenza. «Autonomous Attack Mitigation Through Firewall Reconfiguration». In: *Int. J. Netw. Manag.* 35.1 (Dec. 2024). DOI: `10.1002/nem.2307`. URL: `https://doi.org/10.1002/nem.2307` (cit. on p. 21).

[31] NetGroup. *NetGroup - Politecnico di Torino*. Accesso: 10 settembre 2025. 2025. URL: `https://netgroup.polito.it/` (cit. on p. 21).

[32] Lumi Chatbot. *Lumi Chatbot*. `https://lumichatbot.github.io/#/`. Accesso il 2 settembre 2025. 2025 (cit. on p. 26).