POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

An abstract model of NSF capabilities for the automated security management in Software Networks

Supervisors:
Prof. Cataldo Basile
Prof. Antonio Lioy

Candidate:
Aurelio Cirella

Academic Year 2021/2022

Abstract

Background Virtualisation mechanisms allow to dynamically start an instance of any kind of software without worrying about the setup and configuration of a dedicated physical machine for each of them. These mechanisms are a huge step forward in terms of service provisioning and network management: an example could be the Network Function Virtualisation approach that proposes network function instances that can be executed as processes on virtual environments hosted on the network node where these functions are needed. However, security issues are still not addressed properly. Security policy enforcement is a sensitive area that can leverage virtualisation tools. In particular, one of the most discussed problems is the development of automated frameworks that help the user to easily configure and implement the desired security level within its network topology. Usually, in order to properly embody the desired security requirements, the user should choose and configure in the proper way the Network Security Functions (NSFs). NSFs are defined as a set of Security Capabilities. Security Capability represents what the NSF can do in terms of security enforcement. Some examples are packet filtering or traffic encryption. In this scenario, the user has to know how NSF from each vendor has to be set up and this can cause errors when switching from one vendor to another.

Objective The thesis aim is to develop an abstract model through which it is possible to formally describe NSFs and how to use them. Since NSFs from different vendors expose different interfaces, a standard is required to reduce complexity when it comes to managing NSFs from several providers. The proposed abstract model offers a common interface with which different NSFs can be operated. Automated tools can leverage this common interface by programmatically querying available functionalities and configuring them according to the user's needs. The outcome is an automated framework that can perform the same reasonings that human security experts would do.

Results The proposed model takes advantage of software development and model definition methods such as Model-Driven Engineering and Information and Data Models. The developed framework offers a formal description of NSFs through the Decorator Design Pattern. This facilitates the definition of further NSF instances and their abstract language. One of the most important abstraction layer functionalities is the introduction of a general way to specify Security Capability values in abstract policy definition, decoupling the NSF low-level syntax from the abstract policy syntax. Also, the model facilitates the provisioning of details that are specific to the selected NSF, such as low-level policy string format. Moreover, it introduces the support to NSF Default Actions when no abstract policy has to be performed and NSF Resolution Strategy when multiple abstract policies are in conflict. In

conclusion, the proposed framework can receive a security policy stated in abstract language and translate it into an equivalent policy stated using low-level language for the selected Network Security Function.

Conclusion The developed solution can cover the above usage scenario. It has been validated in real-world use cases and it has been proven that including new NSF is simple and clear if the proposed workflow is followed. Finally, it has been tested that the proposed solution can produce valid low-level policies and that it can be embedded in the above-mentioned wider refinement framework.

I would like to express my gratitude to my inspiring supervisor Professor Cataldo Basile for the immense patiencewith which he followed and helped me during this thesis work. All of this could not have been possible without its passion and attitude towards the project. I wish to extend my thanks to the Department of Control and Computer Engineering and the committed Professors I have met during these years at the Polytechnic University of Turin.

I would also like to thank my family for supporting me in every decision I have made so far. A special thanks also to the friends and fellow students who shared this study experience with me. It has been a journey of joy, laughter, and sleepless nights.

Finally, I am deeply grateful to Nicla, who motivated me in the most challenging moments and celebrated with me in the most cheerful ones.

Table of Contents

List of Figures

Chapter 1

Introduction

Virtualisation technology has been introduced to solve the problem of rising requirements in terms of computing power. It allows the creation of virtual hardware instances partitioning a single computer's physical resources.

In the past, companies had to use a *one application per server rule* in which when a new application had to be deployed, then a new server had to be bought and configured. This approach was required since each application needed its hardware and software requirements. Also, it was essential to properly isolate different applications to avoid their executions that could cause conflicts in hardware usage or harm each other when malware occurs.

This solution led to many servers to buy and to power up that was mostly idle. Virtualisation proposed a resolution for this kind of issue: partitioning a *single machine* physical resources obtaining *virtual environments* in which each application could run almost in isolation from each other. One of the most important aspects of virtualisation is that it allows the setup of virtual profiles of these virtual environments such as operating systems, virtual hardware, or virtual drivers. In this way, it was possible to deploy several applications on a single server and make companies save a considerable amount of money.

Virtual environments (or *virtual machines*) can easily be set up and deployed, and they can execute any application: from a single process to check network traffic to a whole web server managing hundreds of requests. Moreover, the most crucial aspect is that it is possible to execute those on a single computing machine if it is powerful enough.

With the introduction of virtualisation, computing instances can be spawned at ease based on companies' and users' needs. It is possible to move virtual machines across the network topology, instantiate new of them in strategic locations within the network, or pause them when they are no longer needed. This agility allowed service providers to dynamically set up the network topology instead of buying and configuring a physical computing machine in a physical network topology. As a consequence of the introduction of virtualisation, a new kind of service provider was born: cloud service providers. Cloud providers offer the customer the ability to instantiate virtual machines, set the desired virtual network topology, and run customer applications on these virtual instances. Virtual computing instances are hosted by the cloud provider physical machines whose geographical location or physical network topology is unknown to the customer.

With this kind of dynamic behavior in mind, *Network Function Virtualisation*

and *Software Defined Networking* apply virtualisation advantages in terms of network management.

Network Function Virtualisation (NFV) is a solution adopted by telecommunication providers to provide network services (e.g. traffic analysis, security reports, IP functionalities) through virtual machines. In this way, it is possible to place these services where they are needed without the downside of buying and configuring a physical machine. NFV considers network functionalities as plain software that can be executed on any physical machine within the network topology. NFV functionalities are executed in virtual machines instantiated after the provider needs. Software-Defined Networking (SDN) is a technology for which it is possible to programmatically control the forwarding decision procedures that take action within a network topology during data traffic. This solution is implemented thanks to a centralized controller from which it is possible to spread forwarding information across the network hosts. The advantages of SDN are that it avoids the over-usage of paths for forwarding traffic. Existing forwarding algorithms can find the best path to deliver internet packets. Still, if all the hosts use the same algorithms, they will come up with the same optimal paths to use, resulting in overusing the same paths and underusing the rest of them.

As customers leverage services offered by cloud providers in order to host their web servers or to execute other applications, one of the requirements a customer could have satisfied using virtual machines is *security functionalities*. Customers can ask to spawn virtual machines to enforce security requirements within their network configuration, such as packet filtering, attack mitigation, or detection systems.

Security functionalities or *Network Security Functions* (NSF) represent functions that allow ensuring essential security properties (e.g. confidentiality, authentication, and integrity). Several security providers offer NSFs, and they can execute them through physical or virtual devices. Additionally, each NSF is implemented with different underlying technologies. For this reason, NSFs offered by different vendors usually require a vendor-specific syntax to be configured and instantiated. In this way, security administrators have to learn a different syntax for each NSF from a given vendor, causing errors during configuration phases. NSFs may offer the same *Security Capabilities* but with different naming conventions. These issues make it difficult to have an *automated* enforcement of security requirements.

Interface to Network Security Function (I2NSF) working group proposed solution is taken into consideration for this thesis work. I2NSF aims to determine a formal description of what an NSF can offer regarding security functionalities. This description is done through a data model in which Security Capabilities are described following abstraction, independence, and vendor-independence principles. Thanks to a formal description of NSFs, users can state the desired security requirements using an abstract language, and the I2NSF Security Controller can deploy those requirements using the available NSFs. In this way, the user does not have to learn a new syntax for a given NSF, but it can simply rely on the I2NSF framework that receives security rules in abstract language and translates and deploys them on one of the available NSF.

Despite I2NSF framework, some problems are not faced yet. *Automated security policy enforcement* is a study area that is recently approaching the researcher's attention. The desired scenario is to have an automated tool that can choose, compare and configure NSFs based on user needs. The main advantage is that user security

requirements would be expressed using high-level language that resembles natural language. In order to welcome this kind of framework, some adjustments are needed to the NSF abstract model that I2NSF has introduced to describe NSFs formally. In particular, NSF abstract language has to allow the specification of Security Capability values using a generic syntax independent from NSF low-level language. Additionally, a way to compare available NSF and their Security Capabilities is also required for the refinement framework. Finally, the abstraction model and the tool that translates from abstract policy to low-level ones have to be exposed as a service.

In this context, the designed framework faces the problem of improving the above-mentioned abstract language for defining NSFs and their security policies. Additionally, the solution has to receive abstract language policies correctly and translate them using NSF-specific low-level syntax. This abstract language was, at first, thought to ease security administrator work when it comes to setting up several NSFs provided by different vendors. In addition to the usage from a user point of view, the intended framework can be used in the final phase of the work of an automated security enforcement tool. The thesis work solves the problem of decoupling abstract policy syntax from the NSF actual low-level syntax. This solution eases the functioning of the broader refinement tool. Additionally, the thesis work covers the problem of enabling NSF functionalities left behind by the previous thesis works, such as Resolution Strategy or Default Action implementation. Finally, a service for comparing NSFs and their Security Capabilities has been developed, as well as a service for making the proposed tool exploitable by external frameworks.

The thesis work has accomplished to solve the above-mentioned usage scenario. In particular, compared to previous thesis work, it has been introduced an approach with which it is possible to state NSF details related to how policy strings have to be constructed. It is now possible to specify which string to place at the beginning or at the end of low-level security policies; these details have to be provided by the framework developer while before this was final user duty. Thanks to the exact mechanism, further NSF information can be added, such as which Security Capabilities have to be placed as default in each abstract policy, which *policy-level* attribute the tool has to take into consideration to perform the low-level policy translation correctly. Additionally, it is possible to specify which Security Capability provided values must be replaced in the low-level policy for specific use cases.

The usage of generic operators to specify Security Capability values is another significant improvement the thesis work introduces. Operators are intended as the format used to specify which value to use for a specific Security Capability. Some examples are union operator to provide a set of values or range operator to provide a range of values. It can happen that some NSFs do not support all the designed operators, but in order to obtain the most generic abstract language, the framework proposes a mechanism to allow any value operators within abstract language policy. When an operator that the NSF does not support is used within the abstract policy, the tool can adequately adapt the policy toward the operators supported by the NSF.

Resolution Strategy is another addition to the current thesis work. It makes it possible to state how NSF should deal with conflicting security rules. It is now possible to specify which Resolution Strategy (e.g. First Matching Rule, Last Matching Rule) an NSF supports and which external data must be provided in the abstract language policy to implement that Resolution Strategy properly. Currently, First

Matching Rule is supported for IpTables NSF; it allows specifying the priority that has to be coupled with each security rule provided within the abstract language policy.

Default Action is about the actions that the NSF should apply when security rules conditions are not met. Hence they do not have to be executed. Default Actions are modelled as an additional abstract rule within the abstract policy definition; hence, it can list any Security Capabilities available for the selected NSF.

The thesis work also proposes the definition of generic NSFs. It allows defining an NSF composed of selected Security Capabilities (e.g. packet filter functionalities) and using the correspondent abstract language without the need to specify toward which actual NSF it has to be translated. In this way, generic policies that cover required security protection could be defined without worrying about which NSF to use. Then, based on the available NSFs, it is possible to choose toward which translate the generic policies defined at the beginning.

NSF Catalogue as a service is one of the most meaningful outcome: a BaseX web server has been configured to perform queries on XML files containing practical details about NSF composition. In particular, it is possible to query the web server to discover which NSFs have common Security Capabilities, which NSF has a particular set of Security Capabilities, or which NSF can implement a specific security rule. Each of these queries is useful for the excellent functioning of the automated security enforcement tool.

To make the thesis framework exploitable by broader refinement tools, a web server has been developed. The web server can be used through a GUI or using HTTP requests. It offers the thesis work as a service; each framework component can be executed by providing custom input files or already available ones. This service is the central aspect that allows linking the proposed work with other external automated components.

This thesis work leverages methods already approved by the computer engineering area in terms of software development and model definition, such as Model-Driven Engineering and Information and Data Models. In particular, the exposed thesis work is developed around the definition of UML Information and Capability Models to describe the entities of NSF and Security Capability. Thanks to these abstraction techniques, the framework functioning is entirely general; it does not have hard-coded details that could make the tool locked to a specific set of NSFs. The framework is easily extendable for future usage when additional NSF introduction is needed.

Chapter 2 is about the technologies and software engineering methodologies that are used in the exposed thesis discussion. Chapter 3 illustrates already used methods in security function literature; these methods lay the foundation for the thesis work development. Chapter 4 presents the problems for which the thesis proposes a solution and the main ideas that have driven the development of the proposed framework. Chapter 5 shows more details of the underlying structure of the thesis work, the technologies and the solutions that have been chosen to implement it. Chapter 6 displays an actual use case scenario through which it is possible to validate the framework usage against each problem that the framework itself addresses. Chapter 7 proposed some possible future development of the thesis work, particularly how this can be used in a broader context of policy refinement framework. Finally, User and Developer Manuals are presented in Appendix A and Appendix B.

Chapter 2

Background

The technical tools and languages referred to in the following chapters are described here. Furthermore, there is a theoretical description of software engineering fundamentals. Markup languages and XML databases are discussed, too. Finally, a description of the Flask Web Application framework is proposed.

2.1 UML

Unified Modeling Language (UML) is a tool used for modeling systems statically and dynamically. It is a standard introduced by *Object Management Group* (OMG), and it can represent system entities and system functions. UML provides a graphical representation of a system through the usage of *diagrams*; a system view composed of elements of different types.

UML diagrams that describe system structure are *class diagrams*, *package diagrams*, or *component diagrams*. UML diagrams that describe system behavior are *use case diagrams*, *sequence diagrams*, and *activity diagrams* [1].

2.1.1 Class diagram

In this thesis work, class diagrams will be mainly used. The main element of a class diagram is a *class*. A class (Figure 2.1) is a descriptor of objects with similar properties, names, attributes, and functions that identify this.

Objects represented by a class are said *class instances*, each instance has the same attributes declared in the class. The attribute name will be the same across the objects of the same class, while attribute value can be different among different objects. Specific symbols are defined to represent class-to-class relationships. *Asso-*

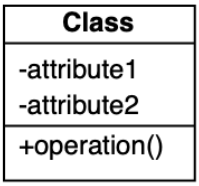

Figure 2.1: Class example

ciations are used to state a *simple* relationship among classes. Class associations

represent links among class instances, in order to not lose instance-wise link information, *multiplicity* is used (Figure 2.2). Multiplicity is placed upon each association end; it is used to indicate how many objects of a given class can take part in the relationship. Multiplicity is stated using n, *, m..n, m..*, 0..1.

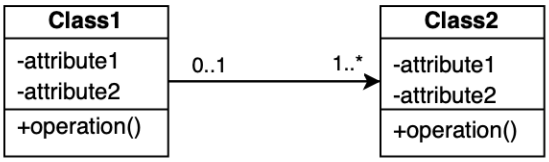

Figure 2.2: Simple association

Associations can be directed in a mono-directional or bidirectional way. It is also possible to state class role within the relationship upon association ends.

Aggregation is a special relationship among classes: it states that a class can be an attribute of another class. In Figure 2.3, Class 2 *is-part-of* Class 1, the lifecycle of these classes is independent from each other.

Figure 2.3: Aggregation

Composition states that a class *is composed of* one or more classes (Figure 2.4). In this kind of relationship, the classes lifecycle is strictly bounded: if one class stops to exist, also the composite class does. It is possible to use *Association Class*

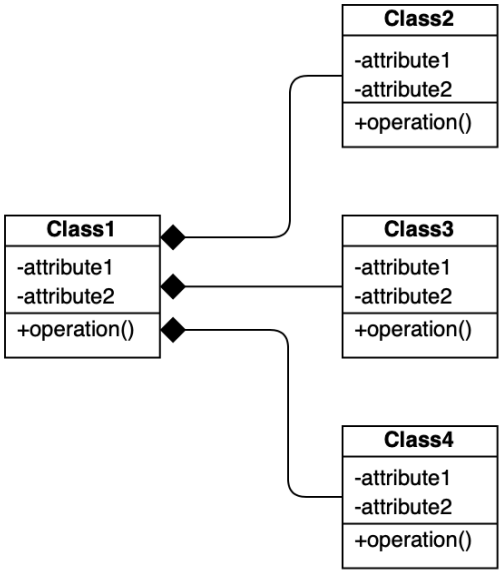

Figure 2.4: Composition

to attach attributes to associations (Figure 2.5).

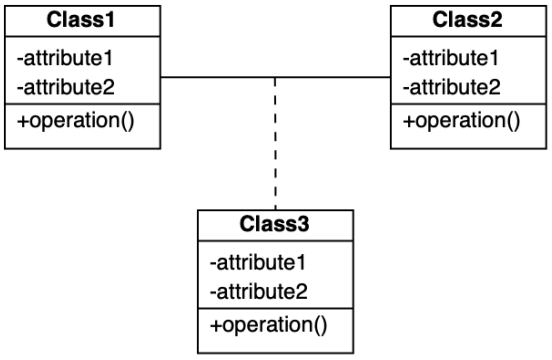

Figure 2.5: Association class

2.2 Design patterns

In software development, it is more than frequent to face the same problem on several occasions. *Design patterns* aim to describe an optimal solution to face recurring and frequent problems with a particular focus on code reusability and maintenance. The following description characterizes a design pattern:

- *Name* to easily identify it.
- *Problem* it solves.
- *Solution* to the problem, it describes, in an abstract way, which element to use and how to link them.
- *Consequences* due to the solution proposed, it can happen that some patterns provide advantages but also disadvantages both in terms of flexibility, portability, etc.

Design patterns are a description of objects and classes that communicate with each other and that are used to solve a general problem.

Patterns can categorized based on what they do. *Purpose* classification includes: *creational patterns* (object creation), *structural patterns* (object composition), and *behavioral patterns* (object intercommunication and responsibility). Then, another categorization can be introduced based on *scope*; it depends on whether the pattern can be applied for classes or objects.

The former classification regards relationships that are statically defined at *compile time*, the latter classification regards relationships that can evolve at *runtime* [2].

2.2.1 Reuse mechanisms: Inheritance versus Composition

Class inheritance is used to implement a new class starting from a parent one; descendant classes can override or extend parent class interfaces. This technique is usually referred to as *white-box reuse*: parent class implementation is visible to the descendant classes.

Class composition, on the other hand, allows the implementation of new functionalities through composing objects from different classes. This technique is referred to as *black-box reuse*: component objects details are not visible.

Inheritance is the most straightforward approach: it can be defined at compile time and allows changing the behavior of the ancestor class. The main disadvantages of inheritance are that: it can only be defined at compile-time, and changes in the parent class inevitably are reflected into descendant class implementation, causing descendant classes to not be suitable anymore for a given problem.

Object composition can be defined at runtime: objects link to others referencing them. Object to object communication is possible only if they both respect each other interfaces. The main advantage is that, differently from inheritance, an object can be replaced by another one as long as it exposes the same interface, providing the same services. In opposition, the main composition disadvantage is that it resorts to a runtime binding that can be more or less efficient based on implementation specifics. Object composition supports objects encapsulation and separation of tasks, avoiding the usage of a deep inheritance tree.

A much more specialized type of Composition is *Delegation* in which a request is served by two objects of different classes: a *receiver* and a *delegate*. The former receives the request and forwards it to the latter that has to handle it: the receiving object has a reference to the delegate object. As opposed to inheritance, in which descendant classes operations refer to parent class ones (that are fixed), in delegation is possible to dynamically change the referenced delegate object and vary how the incoming request will be handled.

Aggregation versus acquaintance

Aggregation is a relationship for which one object *is responsible for* another one; the latter is part of the former. Both objects have the same lifetime. Acquaintance (i.e. association) is a relationship for which an object *knows of* another one. Aggregation relationship is considered stronger than acquaintance.

2.2.2 Pattern catalog

As previously said, patterns can be grouped based on purpose or scope. In this section, the *purpose* classification will be discussed.

Creational patterns

Creational patterns are about how an object is instantiated, and its main objective is to make the object creation independent of the surrounding system. The core of these patterns is not on the class implementation but on the object composition needed to obtain objects that perform complex actions. Going over the basic knowledge that the system has of the instantiate objects (i.e. their interfaces), creational patterns aim to describe additionally: what gets created, when it is created, who creates it, and how it is created.

- *Factory method*: creation methods defined as interfaces in superclasses, then subclasses can implement those methods changing the type of object created.
- *Abstract factory*: the creation of varieties of objects without correspondent concrete class specification.

- *Builder*: step by step creation of an object in its different representations, using the same construction code.
- *Prototype*: copy of an existing object without making the resulting code depending on the copied classes.
- *Singleton*: make sure a class has only one instance while providing a single point of access to this instance.

Structural patterns

Structural patterns are about classes and objects composition in order to obtain complex structures and new functionalities.

- *Adapter*: collaboration among objects with incompatible interfaces.
- *Bridge*: split a large set of classes into two subsets in order to be developed independently.
- *Composite*: object composition in a tree structure fashion that can be used as a whole.
- *Decorator*: attach new functionalities to objects placing these into a wrapper object that describe the functionalities.
- *Facade*: simple interface to access a complex set of classes.
- *Flyweight*: share of object resources to consume less computing resources.
- *Proxy*: usage of a substitute for the original object, it can perform access control or other action before the original object is reached.

Behavioral patterns

Behavioral patterns are about algorithms and objects' responsibility: in addition to objects and classes, these patterns also cover the communication part. Behavior distribution is done using inheritance or composition.

- *Chain of responsibility*: requests sent over a chain of objects, each of them decides if it has to process the request by itself or if it has to forward it to the next object.
- *Command*: store request details in an object in order to pass the request as a parameter, queue, or delay it.
- *Composite*: object composition in a tree structure fashion that can be used as a whole.
- *Iterator*: traverse elements of a collection without knowing the actual implementation.
- *Mediator*: object intercommunication is done by means of a mediator object to avoid object-to-object dependencies.

- *Observer*: subscription mechanism that allows an object to be signaled when another object experiences an event.
- *State*: object behavior changes based on its internal state; it is like an object has changed its original class.
- Strategy: group a set of algorithms in a class to make their instances interchangeable.
- *Template method*: algorithm skeleton defined in a superclass, subclass override the details of the algorithm.
- *Visitor*: the algorithm is decoupled by the class on which it has to operate, and it is moved in an outside class. The algorithm object receives as a parameter the objects on which the algorithm has to operate.

2.3 Model-Driven Engineering

Since computer science early days, one of the most important practice is to *abstract* complex concepts and to present them through a *higher level view* that simplifies it. Some examples could be operative systems or assembly programming languages that hide the actual technology complexity.

However, these types of abstractions were only limited to technology concepts and not about decoupling the design of software solutions from the implementation details. Some *computer-assisted software development tools* was introduced to follow the above direction. These tools were able to translate a software solution expressed using graphical representations or state machines into actual code in low-level programming languages. Even if these tools were not appreciated and famous, they lead software engineers to use *diagrams* to describe software solutions.

With new generation programming languages, developers have to produce reliable software without forgetting about system integration, evolving APIs, or system deployment. Consequently, developers are often not too careful about system-wide correctness and performance.

In order to properly approach system-dependent complexity, *Model-Driven Engineering* (MDE) could be exploited. MDE is led by two main principles: usage of domain-specific modeling languages and transformation engines.

Domain-specific modeling languages allow software engineers to design a solution using models in which the domain of interest imposes entities and relationships. In this way, it is possible to state what the solution should look like instead of how it should be implemented.

Transformation engines are tools that parse the model expressed in a domain-specific modeling language and produce various deliverables based on the previous: source code, XML descriptions, or new models. This transformation enforces the concept of "*correct-by-construction*" opposed to "*construct-by-correction*" typical of traditional software engineering approaches.

The main advantage of using MDE technologies is to have a model in which elements are bound to the domain of interest: this facilitates the communication between software engineers and product managers thanks to a *standardization* of the terminology used [3].

Model Driver Architecture (MDA) is a software approach proposed by OMG that follows MDE principles. MDA detaches application logic from implementation technologies. This approach allows system designers to system functionalities through standard languages such as UML models or *Business Process Modelling Notation* (BPMN). In this way, system description exists independently of system implementation or the technology used. MDA allows designing a system without worrying about low-level details, resulting in fewer misunderstandings among people specialized in different areas.

These properties make it possible to implement the same system across different underlying platforms having the system model as guideline [4].

2.4 Information Model and Data Model

Information Model is the representation of system entities and correspondent attributes, relationships, and actions that can be executed on them. Entities can be abstract or real objects. Information Models are used to describe a closed environment (or system) and the entities that contribute to its functioning. Describing the system without worrying about the underlying software implementation is the critical point of using the Information Model. Information Models can be represented using *natural* language or *specific* languages. Examples of the latters are Entity-Relationship diagrams, EXPRESS language, or UML.

Data Model, instead, is used to provide system concepts more in detail compared to the description offered by the Information Model. In particular, the Data Model introduces low-level details on how system entities must be stored and implemented. A Data Model allows designers to determine the structure of data explicitly. In Data Models, it is possible to specify entity attributes type or relationships cardinality.

Since a Data Model represents implementation and technical details, several Data Models could be derived given an Information Model.

2.5 Markup languages and data representation

In order to represent system components and correspondent relationships, specific file standards will be used as XML, XSD, and XMI.

2.5.1 XML

Extensible Markup Language (XML) is a markup language that is used to represent data in a human and machine-readable way. XML has been introduced by *World Wide Web Consortium* (W3C) as a simple and usable language to store data structures across the Internet. The XML file format is mainly used to make different systems exchange data even if these systems use an inner data format different than the other.

XML file is a string of Unicode characters that can be *markup* or *content*. *Markup strings* start with a "<" and ends with a ">" while not markup strings are *content strings*. A markup string is also a tag, this can come in the following forms: *start-tag*, *end-tag*, and *empty-element tag*. Start and end tags identify elements, strings between tags are the element's *content*. It is possible to have an *attribute* within a

start-tag; this is a name-value pair representing a sort of element property. All XML documents have to start with an *XML declaration*.

XML documents have to be *well-formed*, it has to respect a set of syntax rules as using only Unicode characters, correct nesting of start and end tag couples, start and end tag names have to be the same, all elements have to be nested in a single root element following a tree structure. XML documents have to be also *valid*: elements within the document have to be defined following the rules, and the grammar of elements declared in a *Document Type Definition* (DTD). DTDs define which elements can or cannot be used in an XML document.

XML documents are parsed by XML processors or parsers that can be able to check for document well-form, and document validity with respect to a DTD [5].

2.5.2 XSD

XML Schema Definition (XSD) is a successor of Document Type Definition. It is more powerful and complete: it provides a more detailed description of elements that are allowed within an XML document, and it supports *various data types* and *user-defined types*. As with other XML schema languages, it describes rules that an XML document has to respect to be considered valid. An XSD file defines the structure of the XML file, i.e. which elements are in which order, how many times, with which attributes, or how they are nested.

Without an XSD, an XML file is a relatively free set of elements and attributes. A schema is a collection of components: *attribute* and *element declarations* and *data type definition*. XSD introduces a new important aspect: schema components are grouped by *namespaces* to avoid conflicts due to using the same name for more than one attribute or element. To validate an XML document against an XML Schema is required to provide the XML Schema as an input parameter to the *validator* or to set the XML Schema reference within the document to validate itself. Schema components are:

- *Element declarations*, they define name, type, namespace and the possibility to have children or attributes of an element.
- *Attribute declaration*, they define name, type and namespace of attributes. Attributes can have default or fixed values.
- *Types*: simple or complex.
  - *Simple types* define which textual values can be used in an element or an attribute. Simple types are nineteen primitive data types (e.g. boolean, string, decimal, float). New data types can be obtained by restriction, concatenation, or union of these ten primitive data types.
  - *Complex types* allow stating which is the allowed configuration of an element: content, attributes, and children. Complex types example is element-only content in which no text is allowed, simple content in which text is allowed but children are not. Complex types can be derived from other complex types.
- *Groups*: macros of elements or attributes that can be reused in several types definitions.

- *Attribute use* states if an attribute is mandatory or not in a complex type.
- *Element particle* states the minimum and the maximum number of times the element can be used as a complex type content.

XML documents can be validated using XML Schema, but XSD is also used to produce the XML structure as a data model starting from elements, attributes, relationships, and data types. The information about an XML structure is called *Post Schema Validation Infoset* and is used in order to handle the XML document as an *Object-oriented programming* object. XSD can also be used as a generator for XML file documentation by means of *annotations* [6].

2.5.3 UML and XML: XMI

In this thesis work, Unified Modeling Language (UML) will be used to describe and design the system implementing a solution about the topic of interest; UML allows illustrating system details using diagrams representing entities and their relationships. UML language is characterized by a *Meta-Object Facility* (MOF), a meta-model language that is used to describe other modeling languages. In order to exchange models (i.e. structured data) among MOF-based languages, XML could be used.

XML Metadata Interexchange (XMI) delineates a standard way to save any MOF-based metamodel into an XML file. An XML document storing UML models is then called XMI. XMI is though, as XML, to be a format to exchange structure of data and, additionally, to handle supplementary information (e.g. constraints on values, data structures) needed for a complete and meaningful data exchange [7].

2.6 XML Databases

The tool proposed in this thesis work will make huge use of XML files to provide the discussed services (e.g. Capability description, NSF abstract language). In order to properly store and, if needed, compare the content of those files *XML Databases* will be discussed. A possible use case of XML Databases could be to compare between different NSF capabilities and check if one has the capability of the other and vice versa [8].

XML Database is a *data storage management system* that handles data in XML format. It offers all the traditional database properties: a query language, integrity constraints, or resiliency. XML files allow storing data with a simple syntax so that they are easy to parse. XML is said to represent *semistructured* data: it is possible to represent data with loose structure (all entries have the same attribute except some that have some exceptional ones) or data with no structure at all. Finally, XML files support nested elements and hybrid (data and textual) information [9]. There are two main classes of XML Databases:

- *XML-enabled databases* that map XML file content to a traditional database by splitting XML data into relational attributes, entries and tables. This approach is suitable for structured documents. A DBMS acts as middleware and performs the translation from XML file structure to relational database and

vice versa. XML attributes can be deduced using a DTD file or from the XML structure itself.

- *Native XML databases* that use a logical model to store the XML files, then the same model is used to perform queries. In order to do so, custom optimized data structures are used. Opposed to relational databases that consider a tuple as a fundamental unit, Native XML Databases consider the XML document itself the fundamental model unit. In fact, some systems have introduced an XML data type to handle XML documents. When an XML document is retrieved, it is parsed and manipulated by the database system itself.

2.6.1 Differences between XML Data and Relational Data

A relational model is based on a mathematical relationship based on one or more unordered tuples. Each tuple can have one or more attributes. While in XML model, elements order within the document can be relevant. An additional attribute could be used in a relational model to represent that ordering. XML model supports nested relationships while the relational model does not. The relational model does not allow the same attribute in a single tuple more than once in the XML model; it is possible to have the same child element appearing more than once in the same entity. The relational model only supports primitive data types, while the XML model also supports user-defined data types.

2.6.2 Mapping of XML Documents to Relational Data

There are several types of mapping XML documents to relational models:

- *Table-based*: the XML Document has to have the same structure as a relational system; *database, table, row, and column* elements must be used within the XML document. This approach is the most simple: there is a one-to-one mapping between the XML structure and the relational database one.

- *Schema-Oblivious*: mapping is performed without the support of DTD or XSD correspondent documents. These methods could be used for irregular XML in which elements do not share the same structure. In order to obtain a most general structure, a solution could be to transform a document into a three structure. XML elements will be considered as tree nodes; they will be stored as relational tuples with the following attributes: *id, parent, type, tag name, value, and order*. Finally, using SQL queries will be possible to retrieve data from this tree-like table. An alternative could be to create more tables in order to reduce query's computational complexity.

- *Schema-Aware*: this method creates a database schema starting from an existing XSD or DTD document related to the XML file of interest. The mapping between tables and attribute with XML elements will be simplified by the DTD/XSD that already describes the XML document structure.

2.6.3 XPath

XPath is a language that allows accessing all XML document elements. XPath considers an XML document as a tree structure in a way that XML elements, tags,

or text are treated as tree nodes. XML navigation is performed traversing a set of subsequent nodes until the target content is reached.

`doc()` command is used to retrieve the root node of the current document, "/" is used to navigate to the immediately lower level from the current node. If needed, it is possible to skip more than one level using "//". The following command will return all the wine elements from an XML document called "`winecellar.xml`" that has `winecellar` as the main node:

```
doc"(winecellar."xml)/winecellar/wine
```

To retrieve only the second element within the main node `winecellar`:

```
doc"(winecellar."xml)/winecellar/wine[2]
```

It is worth noting that XPath index starts from 1. If it is needed to retrieve only wine elements which have price attribute above a certain threshold, the following command should be used:

```
doc"(winecellar."xml)/winecellar/wine[price > 20]/name
```

2.6.4 XML-enabled and Native XML databases

In XML-enabled databases, it is not possible to query for a specific XML document, nor is it said that it is possible to reconstruct the XML document starting from its shredded form as it is stored in the underlying relational database. For this reason, exporting existing relational data as XML documents is the primary usage of XML-enabled databases. Native XML databases that store XML documents as text have better performance when it comes to retrieve XML documents or XML documents fragments thanks to indexing techniques. While in XML-enabled databases, a large number of joins is required to reconstruct the original XML document [10].

2.6.5 Querying languages

XML-enabled databases support both *SQL/XML* and *XQuery* languages while Native XML databases only support XQuery.

SQL/XML

SQL/XML is an SQL language extension that allow to create XML documents from relational table tuples based on input queries. SQL/XML defines an XML data type and a set of scalar and aggregate functions to handle this data type. For example, `XMLElement` keyword allows accessing XML object/fields within table tuple.

```
SELECT XMLElement"(sparkling "wine, PRODNAME)
FROM PRODUCT
WHERE PRODTYPE"="sparkling
```

With the previous query will return the name of XML elements from PRODUCT table with PRODTYPE tag equals to "sparkling". In particular, the result will be enclosed in further XML tags named "sparkling_wine":

```
<sparkling_wine>Meerdael, Methode Traditionnelle
Chardonnay, 2014 </sparkling_wine>
<sparkling_wine>Jacques Selosse, Brut Initial, 2012
```

```
</sparkling_wine>
<sparkling_wine>Billecart-Salmon, Brut Réserve, 2014
</sparkling_wine>
...
```

Additionally, adding further attributes taken from the database to the output XML elements is possible.

XQuery

Both XML database models allow access to XML data through the usage of *XQuery* language. In particular, XML-enabled databases have an intermediate layer that performs the translation from input queries stated in XQuery to actual low-level queries in SQL/XML language.

XQuery language allows performing a structural search within an XML document. Structural search is different from full-text search; the latter will search for a set of words within the XML document while the former will be able to perform the search taking into account each XML element attributes within the document. For example, if an XML is storing information about books published, it will be possible to search for author name attribute instead of search any text content (that could not be written by the author of interest) which contain author name. With the support of *XPath expressions*, XQuery makes it possible to iterate XML document elements, comparing them and performing SQL-like operations.

XQuery paradigm is based on *FLWOR*: For, Let, Where, Order By, and Return:

```
FOR $variable IN expression
LET $variable:=expression
WHERE filtercriterion
ORDER BY sortcriterion
RETURN expression
```

A possible example could be:

```
FOR $wine IN doc"(winecellar."xml)/winecellar/wine
ORDER BY $wine/year ASCENDING
RETURN $wine
```

This query will return `wine` elements within `winecellar.xml` XML document ordering `year` attribute in ascending fashion. Additionally, `WHERE` clause can be used to refine query output.

In the following example the query will output a `cheap_wine` XML element containing wine name and price of wines whose price and currency satisfy the condition within the `WHERE` clause:

```
FOR $wine IN doc"(winecellar."xml)/winecellar/wine
WHERE $wine/price < 20 AND $wine/price/@currency"="EURO
RETURN <cheap_wine> {$wine/name, $wine/price}</cheap_wine>
```

2.7 BaseX

BaseX is the XML Database used to implement some of the key services of the thesis work. As claimed on the correspondent documentation: "*BaseX is a light-weight,*

high-performance and scalable XML Database and an XQuery 3.1 Processor". It allows storing and processing textual data such as XML, HTML, XSD, and others. BaseX can be used to create databases, code, and execute queries. BaseX provides a GUI-based application and a command-line client. BaseX provides additional services as a *Database Server* enabling a client/server approach, a Web Application that acts as an HTTP server, and Database Administration (DBA) that allows managing BaseX databases through a browser-based approach [11].

2.7.1 Web Application: REST Service

The services of the thesis work mentioned above are about handling and processing XML and XSD documents. Those services could be exposed to the final user through a web application that had to act as a proxy toward the XML Database. The web application itself could have introduced undesired bugs; for this reason, it was chosen to use the BaseX Web Application service that allows exposing simple APIs through HTTP protocol. In order to run the BaseX Web Application service, `basexhhtp` executable has to be launched. This will make BaseX Web Application listen on `localhost` port `8984`. Launching this executable will have a folder in which the Web Server will search when it comes to execute local queries.

2.7.2 REST API

Database resources can be processed in BaseX through REST APIs. `GET`, `PUT`, `DELETE` and `POST HTTP` methods are available. REST services are available at

```
http://localhost:8984/rest/
```

It will be needed to provide default user credentials using the HTML authentication page that will prompt up or through Basic Authentication or Digest Authentication (based on server setup). Through HTTP requests as the following, it is possible to access `factbook.xml` document from `factbook` database

```
http://localhost:8984/rest/factbook/factbook.xml
```

It is possible to send the following HTTP requests:

- `GET`: it is used to request database resources by means of XQuery. It can be accompanied by the following operations:
  - `query`: execute an XQuery instruction using the database given in the URL as query context.
  - `command`: execute an atomic database command.
  - `run`: execute an XQuery instruction from a `xq` file stored on the folder mentioned above.
- `POST`: it is used to send data to the database by means of instructions stated in XML fragment
- `PUT`: it is used to create new databases or to add and update existing ones. Databases are created using XML documents provided in the request body. When it comes to adding a new document resource, it will be added as a new

resource if not already stored; otherwise, it will replace the existing one. Non-XML resources can be stored as binary data or converted to XML thanks to the available parsers.

- `DELETE`: it is used to delete databases or resources.

2.8 Flask Web Service framework

Flask is a Python *microframework* that has the objective to provide a minimal web server keeping the core extensible but straightforward. Flask does not include all the functionalities of a basic web server, such as database selection, but it allows developers to introduce these using third-party *extensions*. Flask project belongs to Pallets Projects and is based on others of them:

- *Werkzeug*: a Python web application library that provides various utilities for web server communication. It is used for the creation of request objects composed of headers, query args, form data, or files. It also manages the creation of response objects. Werkzeug offers a routing system to link URLs to endpoints, allowing to state variables into the URL itself.

- *Jinja*: a Python template engine to generate HTML dynamically. It allows defining an HTML file composed of a static and a special syntax part. The latter allows describing how the dynamic content will be inserted into the final HTML file. It allows sandbox execution and automatic escaping for security-sensitive applications, too.

- *MarkupSafe*: a Python string library that implements a string object in which it is possible to use escape characters. MarkupSafe can handle those characters and properly use them in HTML contexts. In this way, it is possible to prevent injection attacks.

- *ItsDangerous*: a Python data serialization library that secures passing data to untrusted parties. Data securing is done by cryptographically signing data. In this way, when data is received after it has been sent, it is possible to check if data has been tampered with or not.

The main advantage of using Flask is that it allows web developers to quickly deploy a basic web server using intuitive and not cluttered code. Flask's micro-framework approach allows the developer to decide by himself when and how to implement each area of their web application. This approach lets the developer free to use whatever modules, files, or directories to use.

It is worth noting that available Flask extensions are managed and supervised by Flask developers themselves.

Chapter 3

Related works

Network Operators provide security services with huge expenses regarding the required technical skills and costs. *Network Function Virtualisation* and *Software-defined Networks* try to ease those drawbacks bringing more dynamism and scalability to a network paradigm that, otherwise, would be challenging to configure and to evolve based on customer needs.

3.1 Network Function Virtualisation

The telecommunication industry has always been dependent on physical devices to provide every network function. Additionally, network physical components have to be sorted in a defined order to make network functions work correctly. High quality and stringent standards, coupled with the previous requirements, led to long production cycles and poor service agility.

In this scenario, *Telecommunication Service Providers* had to frenetically purchase and configure the physical product in order to satisfy customer needs for services needed only for a short period. Also, device configuration has to be considered in terms of technical skills and the required time to configure the device. These aspects led to increased expenses for Telecommunication Service Providers that customer fees cannot back up.

Network Function Virtualisation (NFV) has been proposed to introduce more dynamic services, reduce product cycles and improve service agility. NFV aims to *decouple* physical network devices from functions executed on them. Network functions are, in this way, implemented as basic software functions that can be executed on any hardware device (e.g. data centers, network nodes). This philosophy allows instantiating as many network function instances on whatever physical device is available at the moment, leading to *high service agility* without the need to buy and configure new physical devices. Software and hardware development can be performed in parallel, independently. This solution also makes it possible to assign different tasks to different hardware and software components [12].

NFV technologies bring the following advantages:

- *Reduced cost* for equipment and power consumption in IT scenarios.
- *Minimized Time to Market* for network devices since it is only required that they are general-purpose enough to run any provided software.

- *Single device usage* for several applications and tenants in order to share resources across services.
- Delivery of *custom service* based on user geographical location.
- *Technology openness* due to ease of development of network software solutions.

In order to reach such benefits, it is needed to handle the following technical issues:

- Development of *efficient* virtualised network services that are *portable* across different hardware and hypervisors.
- Reach *co-existence* among already present hardware-based software services and virtualised solutions.
- *Resistance* to possible security attacks and hardware or software failures while correctly managing virtual services.
- *Integration* of virtual appliances from different vendors.

NFV covers a wide range of use cases such as traffic analysis, security reporting, IP functions implementation (*NAT*, *firewalls*), substitution to customer specialized network hardware, virtualisation of network services in mobile networks to improve energy consumption and hardware usage.

The spread of NFV undoubtedly leads to changes in the telecommunication industry. Traditional network equipment was not thought to execute virtual network services. For this reason, a redesign of these devices has to be performed. Standardization across devices could also lead to advantages to vendors: they could offer a virtualised version of their products that can be executed on competitors' machines if they respect the industry standards.

NFV follows the main principles of *Cloud Computing* solutions that leverage the concepts of virtualisation and improvement of hardware resource usage. The increasing presence of data centers across the world has undoubtedly put the proper attention to the needs of abstracting the usage of several software services without the need to buy dedicated hardware.

NFV is not dependent on SDN, but these techniques can be used together based on what is needed to achieve in terms of flexibility and resource usage [13].

3.2 Software-defined Networking

Software-defined Networking (SDN) is a technology that allows the dynamic and programmatic set up of network configuration using centralizing network intelligence. This is done decoupling *data plane* (moving packets from one interface to another) from *control plane* (fill routing tables, decide routing path). The Control plane is then directly programmable, and the network devices are unaware of application and network functions.

Traditional network architecture is *static* when it comes to monitoring and controlling data flow. A set of algorithms and rules implemented in network devices defines the routing path and destination device. This approach overuses the same

path if packets are forwarded to the same destination device. Only some devices offer the ability to set up different priorities based on packet types, but they have to be configured by skilled administrators, and still, there is a performance loss in terms of speed, scalability, and reliability [12]. Conventional infrastructure also requires skilled engineers and network administrators to set up and maintain multi-vendor networks. This requirement is due to different network devices with different software interfaces.

SDN overcomes these downsides since the data plane and the control plane are treated separately: SDN operations are dictated by the *logical* network topology, and it does not depend on the underlying physical devices. SDN takes advantage of a *centralized controller* that spreads the forwarding strategies for each node within the network topology.

Due to the increasing demands of various flavors of *Cloud Computer services*, SDN has to ensure some critical requirements in terms of support for additional servers or network components and terms of network security. In particular, the main requirements are:

- *Scalability*: SDN has to be able to handle the increase of customer service requests.
- *Reliability*: SDN has to alert in real-time customers and administrators when data is not sent successfully.
- *High availability*: defining availability in terms of service uptime, SDN has to guarantee this parameter to be as high as possible.
- *Elasticity*: SDN has to adapt to network sudden changes rapidly.
- *Security*: SDN has to protect the main security properties of data and, additionally, it has to take care of hardware security too.
- *Performance*: SDN has to provide as services as possible concerning the amount of available physical resources and execution time spent.
- *Resilience*: SDN has to provide a coherent and *correct* service even if there is a failure of any type.
- *Dependability*: SDN has to avoid faults and maintain correct behavior even if some faults happen.

3.3 Interface to Network Security Functions

Interface to Network Security Functions (I2NSF) is a working group whose goal is to facilitate the implementation of security functions within SDN and NFV scenarios while being independent of underlying technologies and vendor's solutions.

More and more challenges are arising for small and medium companies when it comes to managing secure infrastructure, complying with regulatory requirements, and controlling cost. For this reason, it is always more common for them to rely on hosted security services. Hosted services raise companies from the need to have a specialized IT department and train employees responsible for network security

issues. Emerging companies are asking for this kind of service; also, larger ones do it.

The increasing demand for hosted security services has made *Network Security Functions* (NSF) being developed in a variety of conditions and offered by several service providers.

I2NSF has tried to design a framework in which a *Security Controller* will be able to receive *security policies* from the service provider client and to provide those policies to the available NSFs and monitor them [14]. It is appropriate to distinguish security policies from security policy rules. *Security policy rule* is a single statement that expresses a single security functionality. *Security policy* is a set of security policy rules. The Security Controller can manage security policies thanks to:

- *Capability Layer*: it describes the rules with which a controller can interact and control an NSF. These rules have to be implemented by the NSF.
- *Service Layer*: it describes how a client can provide security policies to the security controller. The controller can implement those policies only if the NSFs support them.

3.3.1 Network Security Functions

Network Security Function (NSF) is a term introduced by *Interface to Network Security Functions* (I2NSF) working group. NSFs are defined as functions that allow the implementation of some of the most important security properties, among which there are: integrity, confidentiality, and asset availability. Security providers offer nSFs; these functions are supported by physical or virtual devices and implemented through technologies and devices from different vendors. Additionally, NSFs allow monitoring network traffic and intervention if malicious activities are detected.

Obviously, NSFs from different vendors provide different features and interfaces. Also, NSFs can be deployed in several places within a network and can serve in many ways. Security providers can use several NSFs from different vendors to guarantee a wide range of security properties.

Some NSFs roles can be:

- *External intrusion and attack protection*: firewall, Intrusion Detection System, Intrusion Prevention System.
- *Security functions in a Demilitarized Zone*: NAT, proxies, application filtering (plus the previous functions).
- *Centralized or Distributed Security Functions*: security functions can be deployed in a centralized way when it is needed that it is more easily manageable or in a distributed way when the system size is a crucial point in terms of security. In both cases, it is preferred that NSFs have the same interfaces.
- *Internal Security Analysis and Reporting*: logs, forensic analysis.
- *Internal Data and Content Protection*: encryption, authorization, key management.

- *Security Gateways and VPN Concentrators*: IPSec gateways, VPN concentrators to bridge secure VPNs.

Given the above roles that an NSF can have, the I2NSF objective is to *standardize* the interface, which allows the configuration of those NSFs even if these come from *different vendors* or they use *different underlying technologies*. Doing so, a network admin can use standard interfaces to control which rules are enforced by which NSF and how each NSF has to threat packets.

3.3.2 Problems addressed by I2NSF

The primary assumption is that security functions are not hosted on customer physical facilities. Service providers provide security functions that can be external or internal to the customer company.

It is possible that service providers rely on different vendors or technologies to offer security functions to their customers. The main problem is that different NSFs expose different interfaces to allow experts to configure and manage them. Even if some NSFs offered an *automated way* to be configured, that method would always be specific for that single NSF, and there will not be a universal way to handle all kinds of NSFs automatically.

NSFs from different vendors are used by the service provider to expand the offer to their customer. Even if different NSFs represent the same kind of security functions, it can happen that they do not support the same set of functionalities. For this reason, it is needed that the *management system* (or security controller) of the service provider has to be able to retrieve the available security functions offered by their NSFs. In terms of security functions offered by NSFs, I2NSF wants to develop a *standard method* thanks to which vendors can dynamically update the capabilities of security functions offered by the NSFs they sell. Also, I2NSF aims to introduce standards when it comes to NSFs having to rely on external *metadata sources* like signature profiles, blacklists.

On the customer side, the I2NSF solution has to deal with customers who would like to state which communications should be preserved during critical events, report attacks, and manage network connectivity. Additionally, customers usually require to integrate remote hosted security functions with on-premise local ones that already existed.

Since a customer may rely on multiple service providers, they will express security policies and requirements to them, and the service providers will have to *translate* those requirements based on which NSFs they are using. How customer commonly express those requirements are not standard. This could cause: not matching interfaces used by the customer and required by the selected NSF, no easy and immediate way to update security policies, difficulty to update scripts that produce NSF-specific low-level configuration when the service provider introduces an NSF from a new vendor.

Customers also desire to check which policies are currently enforced, how much bandwidth is covered by those policies and collect other *statistics* in order to perform better *security analysis* and *risks estimations*. I2NSF wants to introduce a simple method through which customers can collect this essential information [15].

3.3.3 Use cases proposed by I2NSF

Through the use of standard interfaces, service providers would be able to set up a *Security Controller* that act as intermediates between a *Client* that want to upload or update a specific security policy and one or more *NSFs* that will have to enforce them (Figure 3.1).

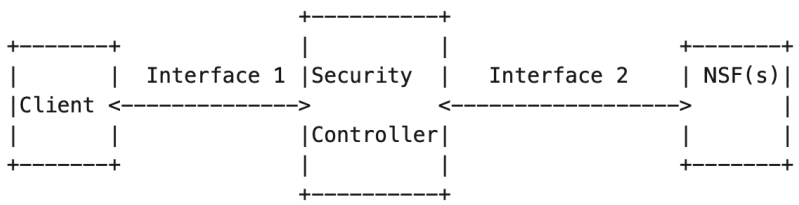

Figure 3.1: Entities Interaction

In a *basic scenario*, Interface 1 has the task to receive security configurations from a client and translate them to commands that the NSFs can execute. The Security Controller returns NSFs reports. Interface 2 has the task of interacting with the NSFs based on the input security configurations received by the client. Security Controller can also provide to the user information about multiple NSFs and hence offer an infrastructure-wide status description. In this way, the Security Controller can check NSF integrity, isolation, and provenance. In order to provide the last-mentioned information, Security Controller should double-check with third-party added data. For this reason, Interface 1 will be used not only by users but also by a trusted third party.

In *access networks*, users require specific security requirements hosted in the Network Service Provider infrastructure. Different users represent different scenarios: *residential* users will ask for parental control or threat management, *enterprise* users will ask for flow security policies, *service providers* will ask for protection of their network against threats, *mobile* users will ask quality of service, parental control or threat management. It can happen that some users do not care about which NSFs are used to support their security requirements, while others, like enterprise customers, would like to manage directly the NSF that has to implement their requests.

In *cloud data centers*, security functions can dynamically be added or removed based on user needs or to respect *Service Level Agreements* (SLA) between the cloud provider and the user. In these cases, service providers tend to allocate virtual hardware to ensure the execution of such security functions, which is transparent to the final user. Some challenges for I2NSF Security Controller in data centers are: client requested firewalls will for sure not be hosted by physical devices, but by virtual ones, client traffic is based on virtual source/destination IP Address and ports instead of the correspondent 5-tuple of the physical device that implements security functions, VPN configurations (e.g. keys management) cannot be updated directly by the client.

DDoS, Malware, and Botnet scenario consists of undesired traffic traversing client network devices. DDoS attacks are increasingly challenging to identify and prevent; malware attacks are becoming frequent, especially in information-heavy services (e.g. IoT, VoIP, VoLTE). I2NSF desired to develop a framework that could allow clients to implement security policy configuration searching for specific malware [15].

3.3.4 I2NSF Framework

I2NSF *reference model* allows the user to configure security functions in its network easily. The main goal is to provide a way with which it is possible to interact with NSFs from different vendors through a set of standard interfaces. The reference model describes the *Network Operator Management System* and how it interacts whit the I2NSF User and the available NSFs. NSF developer point of view is not taken in consideration (Figure 3.2).

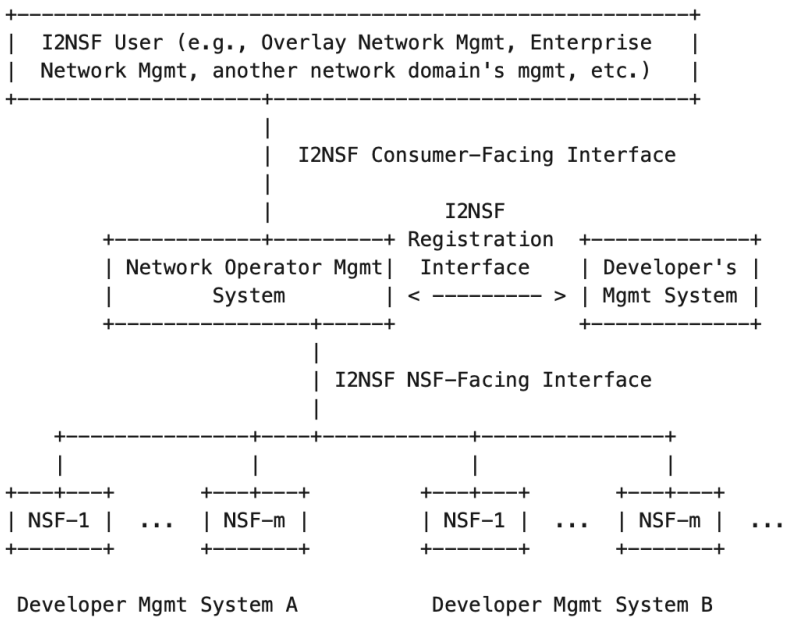

Figure 3.2: Reference Model

I2NSF Interfaces are agnostic of network topology, NSF vendor-specific implementation, NSF control plane, NSF data plane, and NSF provider itself. These interfaces require at least mutual authentication and authorization.

I2NSF Customers interact with the Management System through the *Consumer-Facing Interface*. *NSF-Facing Interface* is used by the Management System to interact with the available NSFs. It is not said that each NSF security function has to be used as a whole; instead, Management System can treat each security function offered by each NSF as a single building block to implement customer's security policies. Finally, the *Registration Interface* is required to NSF vendors to add new NSF to the Management System and to state which security functions the new NSF offers.

Allowing external vendors to add their own NSFs is a crucial concept in terms of flexibility but also introduces *security concerns* to be addressed. Malicious users could access legitimate user security policies, change predefined privileges to these policies, install malicious elements to control the NSF or the whole provider network. Malicious providers could modify NSF behavior. I2NSF highly recommends the basic means of security when it comes to access specific resources, such as authentication, authorization, account, and audit.

To avoid NSF being always the same, I2NSF has thought each NSF as an *evolving virtual device* that can have more security functionalities based on upcoming security issues [16].

3.3.5 Flow security policy structure

In order to standardize different kind of security functions, flow-based NSFs use *Policy Rules*. Policy Rules only use imperative paradigm, in this context. *Event-Condition-Action* (ECA) policy rule is composed of:

- *Event* clause states the event that triggers the evaluation of the Condition clause.
- *Condition* clause determines if the set of Actions can be executed.
- *Action* clause states the type of operations that has to be operated on the incoming packet.

Each of the above is a Boolean clause that will evaluate in TRUE or FALSE. In order to ease policy definition for the user, the *Customer-Facing Flow Security Policy* Structure should be abstract with respect to the low-level language used by the NSFs. These policies could, in fact, express user expectations or guidelines regarding specific traffic flow. A single policy flow could need more than one NSF to be implemented. It is not needed that a user policy rule has to be similar to the NSF low-level language; in this way, the user can construct policies without worrying to know the exact NSFs syntax [16].

3.3.6 I2NSF Capability Information and Data Model

Since NSFs provided by different vendors offer different *Security Capabilities*, it is possible to combine multiple NSFs to satisfy most of the security requirements that a user could express. NSF combination is done without worrying if NSFs are provided through a physical or virtual one. Security Capabilities are the description of the security functions that an NSF supports.

In this context, every NSF will be described through the set of Security Capabilities that it offers. The most crucial point is that Security Capabilities, coupled to a specific NSF, have to be described in a vendor-independent way.

I2NSF describes these Security Capabilities using a Data Model, and an Information Model [17]. Security Capabilities can be grouped in: *event* capabilities, *condition* capabilities, *action* capabilities, *resolution strategy* capabilities, and *default action* capabilities (Figure 3.3).

Through a *Capability Information Model* (CapIM) it is possible to formalize NSF security functionalities. NSFs are precisely specified in what they can do to enforce security policies. In the I2NSF framework, the CapIM is also used to avoid ambiguities when describing security functions offered by an NSF whose definition and functioning can create confusion (e.g. the difference between stateful and stateless packet filter).

A CapIM also defines how an NSF can advertise its functionalities; those can also be a subset of the overall available functionalities, based on client privileges to access them. CapIM design principles are:

- Security capabilities have to be *independent*.
- Security capabilities description has to be *vendor-independent*.

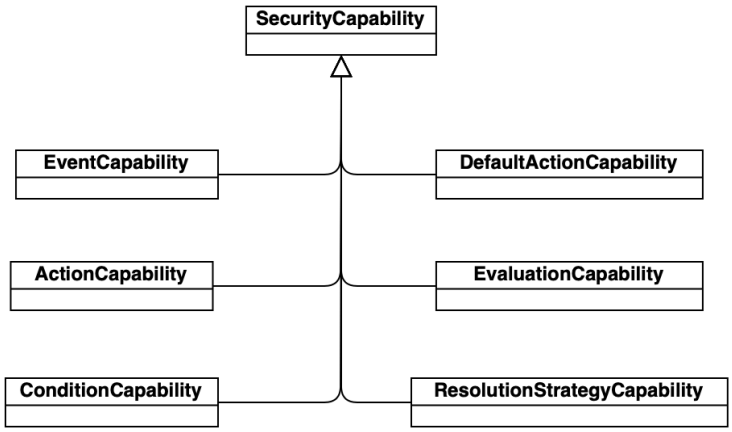

Figure 3.3: I2NSF Capability Data Model

- NSFs have to be able to *advertise* their security capabilities.
- Possibility to *monitor* and *instantiate* security capabilities.
- Security capabilities have to be discovered, negotiated, and updated *automatically.*

In conclusion, the set of supported capabilities by the NSF states the *security levels* it provides. After receiving the required security policy, the security controller can decide which NSF has the needed functionalities to implement application or client requests. This choice will be made only by comparing Security Capabilities without considering the low-level implementation or the correspondent vendor.

Vendors can update their NSFs after the discovery of security threats; this means that the I2NSF framework should offer a way to add, update or delete Security Capabilities [17].

The Capability Model is based on the Event-Condition-Action policy model, in which:

- *Event*: change within the system to be managed. It is used to determine if the condition clause has to be evaluated or not.
- *Condition*: a set of attributes to compare with given values to decide if the policy rule set of actions has to be executed or not.
- *Action*: action to apply in order to control or monitor NSF behavior in terms of packet handling.

Policy rules are three Boolean clauses: Event, Condition, and Action. Each of them can be composed of several Boolean terms that can assume TRUE or FALSE values combined by logic conjunctions. Policy rules have the following semantic:

```
IF <event-clause> is TRUE
    IF <condition-clause> is TRUE
        THEN execute <action-clause>
    END-IF
END-IF
```

Other than the three above clauses, policy rules can include metadata as rule priority when conflicting actions have to be applied simultaneously. A conflict can happen when two Event clauses evaluate TRUE, two Condition clauses evaluate TRUE, or two Action clauses evaluate TRUE.

In order to evaluate which policy rule has to be applied when conflicts happen, *Resolution Strategies* are taken into consideration. Instead, when traffic does not trigger any Condition clause of any policy rule, that traffic should be treated performing a *Default Action*.

Finally, it could be needed to specify how Boolean clauses must be evaluated. For this reason, I2NSF has introduced two additional capabilities for an NSF:

- *Resolution strategies* to state how to resolve conflicts between actions on the same traffic.
- *Default action* to state a default behavior when no rule matches for the current traffic.
- *Evaluation criteria* to define if the Condition Clause is TRUE after the evaluation of the single Condition Clauses.

I2NSF Policy Rules

I2NSF Policy Rules work as follow: if the incoming Event clause is true and the correspondent Condition clause is true, then the Action clause is performed if and only if it is not influenced by attached metadata. Condition clauses in this context are expressed using *conjunctive* or *disjunctive* forms, specified by the *Evaluation criteria*. In the conjunctive clause, components contain OR and/or NOT operators, and they are logically ANDed, while in the disjunctive clause, components contain AND and/or NOT operators and are logically ORed. Condition clauses can be exact-match (can use regex-match), range-based, and custom-match (based on client-provided patterns). It could be possible that several Action clauses on the same object, triggered by the same Event clause and Condition clause, have to be performed, the order of execution has to be defined through *Resolution strategies*. When no policy rule Event clause is satisfied, a Default action should be configured. External data in the form of attached metadata can also influence policy rule Action clause execution.

Decorator pattern in I2NSF model

The model proposed by I2NSF supposes the existence of an external Information Model that abstracts the concepts of Policy rules, Events, Conditions, and Actions. In this way, I2NSF Data Model classes can be seen as specializations of the classes defined in the external Information Model. It is also assumed that the external

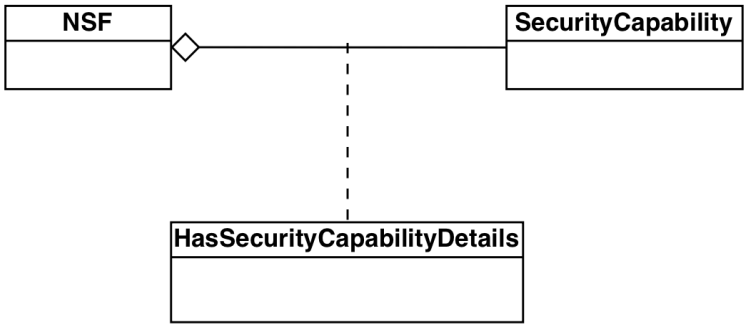

Figure 3.4: I2NSF Capability Information Model

Information Model can append metadata to the class that it describes. I2NSF Capabilities are represented as subclasses of the external Information Model classes. New Capabilities could be added in two ways: adding a correspondent parent class either in the Capability Data Model or in the external Information Model; adding a new class that has to wrap the new Capability class (*decorator pattern*). Capability classes represent NSF functionalities. Capabilities can be used as a component for an event, condition, or action. I2NSF Capability Information Model uses the Decorator Pattern in which one or more decorator objects can extend an object (Figure 3.4). The decorator could provide additional features to the base object; this is done by extending the base object interface. NSF is the extended base object in this scenario, and Security Capability will be the decorator object. As an aggregation of Security Capabilities, NSF states that an NSF owns/provides that Security Capability. An association class is needed to provide more details about NSF and Security Capabilities relationship: the HasSecurityCapabilityDetail defines which Security Capabilities of the specific NSF are accessible and how.

Chapter 4

System design

The proposed framework has been designed following the previously discussed engineering good practices such as MDE and design patterns. Framework requirements, area of application, and design are discussed in this chapter. Also, framework entities and correspondent interactions are explained.

4.1 Problem statement

NSFs can be difficult to configure and dispatch for network administrators since NSFs can be developed by different vendors, deployed on different virtual environments, and use different low-level languages. Starting from the principles drawn by I2NSF, this thesis work focuses on implementing a framework in which it is possible to describe NSFs through Security Capabilities in an abstract way.

The proposed framework has the purpose of offering an *abstraction layer* through which network administrators can use and set up NSFs without worrying about NSF underlying technologies and languages. This layer is possible thanks to an *Abstract Language* that allows describing NSFs' Security Capabilities without taking into account the vendor from which they are from or the language that is required to configure them. The framework is based on Capability Data and Information Model that will be explored later. These models specify how to generate an abstract language through which it is possible to interact with the available NSFs. Finally, using this abstract language, it is possible to state security policies that will be translated into low-level language based on the selected NSF.

It is what noting the difference between Security Policy and Security Policy Rule. *Security Policy* is a set of Security Policy Rules. While *Security Policy Rule* is a single statement expressing a single security functionality.

4.2 Use cases

An abstract description of what an NSF can do could lead to huge steps forward regarding automatic security policy enforcement. Human network administrators can understand differences or similarities between NSF security functions, but this task is still difficult to perform for an automated process. Modeling an abstract model of Security Capabilities could allow an automated process to perform security policy enforcement based on high-level external requirements. In this way, some examples

of task that could be performed automatically are: evaluate which NSF devices are equivalent in terms of Security Capabilities or choose the NSF that offer the most appropriate Security Capabilities based on predefined security requirements.

In *cybersecurity market* it is often hard to compare Security Capabilities offered by different vendors. Security Capabilities comparison difficulty happens because vendors overuse buzzwords to deceive users and convince them to buy their new product even if they do not have any improvement. The proposed framework aims to describe in a precise way which security functionalities are offered by NSFs.

Vendor lock-in is another scenario in which the proposed framework could be helpful. Often, companies tend to stick with the same vendor since they have already set up the required security policy using the specific language that the vendors offer for their NSFs. It could happen that companies would like to change products and rely on NSFs provided by different vendors. Nowadays, this would mean redefining from the foundation their set of security policies: deciding which NSF offers the desired Security Capabilities, expressing the security policies with the low-level language of the newly chosen NSF, checking for bugs induced by the usage of the new low-level language. Thanks to a formal description of the Security Capabilities an NSF can provide, it is possible to query which NSF offers which Security Capabilities to check which NSF can reproduce a specific security policy. Additionally, through an abstract language, it is not needed to worry about the expression of the policy in the new low-level language since the proposed framework can translate the policy from the abstract language to the low-level one of the selected NSF.

The proposed framework focuses on developing an abstract language through which it is possible to interact with the available NSFs. So it regards the *translation* from policy expressed in abstract language to policy expressed in low-level language. It could be thought to use this framework as a component of a *wider framework* that can receive from the user a security policy expressed in natural language (higher level of abstraction)[12]. Automated refinements can be performed on the policy expressed in natural language, obtaining the same policy in abstract language. At this point, the proposed framework could easily translate the policy using low-level language.

4.3 System requirements and design principles

The proposed framework has been developed trying to respect the following requirements:

- NSFs have to be characterized by a *formal description* of their Security Capabilities.
- *Available NSFs* and their *Security Capabilities* can be leveraged with a generic syntax that does not depend on the NSF vendor or low-level implementation.
- *NSF querying system* has to expose information about available NSFs and which Security Capabilities they offer.
- *Abstract language* has to be generic for all the NSFs. It has to depend on the NSF low-level syntax or implementation as little as possible. The NSF abstract language generation has to be based on the Security Capabilities owned by the NSF itself.

- NSF *low-level* policy translation has to depend on the correspondent abstract language policy.
- *Policy translation* from abstract language to low-level language has to be always feasible: abstract language should always be correctly managed by the translation tool. There will be no restriction or exception on the expressiveness of the language based on the target NSF.
- Security Capability values can be expressed using a set of predefined *operators*. The objective is to allow the user to specify Security Capability values using a syntax that resembles natural language. For Security Capability values whose type is an integer, it is possible to use: exact-match, range, and union operators. For string type, it is possible to use: exact-match and union. While for IP address type, it is possible to use: exact-match, union, range, rangeMask(e.g. `0.0.0.0/0.0.0.0`), and rangeCIDR(e.g. `0.0.0.0/24`). When the abstract language policy uses operators that are not supported by the NSF low-level syntax, the translation tool has to correctly perform an expansion of the policy rule towards supported operators.
- Abstract language generation tool and translation tool have to retrieve translation information from the proposed *Capability Information Model* and *Capability Data Model*.
- Abstract language should be able to support the definition *Resolution Strategy* and *Default Action* for the target NSF.
- The proposed framework features a *basic web service* that allows the upload of the required files and the subsequent generation of the desired abstract language and the translation from abstract to low-level language policies.
- It has to be possible to state an abstract security policy without specifying to which NSF it belongs. This results in the definition of *generic NSFs* that can include a set of Security Capabilities belonging to specific areas of interest.

4.4 System outline

The proposed framework will implement some new and critical functionalities described in the previous sections. It is crucial to notice that while introducing new ways to express Security Capability details, the proposed framework implements *backward compatibility* toward the way used to express Security Capability in the previous thesis works.

In detail, this framework comes in handy when a security network administrator or a service provider user wishes to introduce new security functionalities in a particular network. It could be possible that this person does not know the low-level language of a specific NSF or which NSF supports which Security Capabilities. The proposed framework allows external actors to query a database based on an NSF Catalogue to discover information as which NSF supports a set of Security Capabilities, which NSF can implement a given policy rule, or if two NSFs are equivalent. Thanks to the implementation of the abstract language, users could express policies using it without worrying about the underlying NSF technology. The framework's

translator will be responsible for producing that low-level policy, starting from the same policies expressed in abstract language.

NSF formal description is possible thanks to *Capability Information Model* (CapIM) and *Capability Data Model* (CapDM).

CapIM is used to describe an NSF main components that should carry the needed information to generate the abstract correspondent language and translate policies that use the abstract language into policies expressed in the selected NSF low-level language. In this model, NSFs are coupled to the correspondent Security Capabilities using the *decorator pattern*.

The CapDM, on the other hand, is used to describe Security Capabilities formally. In fact, in the CapDM, Security Capabilities are arranged and split based on their primary area of interest. Other than composing an NSF, Security Capabilities will be used as basic blocks of the policy rules expressed using the abstract language; they can be used as Event clause, Condition clause, Action clause, Resolution Strategy clause, Default Action, and Evaluation Criterion clause.

In both cases, the proposed framework is based on CapIM and CapDM representation designed by Avallone in his thesis work. Then the needed changes have been implemented to guarantee the implementation of the before-mentioned system requirements.

The mentioned models have been designed using Modelio modeling environment. This application has been chosen since it easily allows exporting UML models using XMI format. This feature is used when it comes to exporting the CapIM and CapDM in order to process them through the framework components.

The tools that compose the framework have been developed in Java, mainly because Java optimized management of XML, XMI, and XSD files. Also, Java external custom libraries have been used, such as Xerces for parsing and validating XML files, Genere and Automaton for regular expression management, and IPAddress for IP Addresses manipulation.

4.5 System architecture and workflow

The main tools that compose the proposed framework are:

- *XMI to XSD Converter*: it is responsible for generating an XSD file that describes how the NSF Security Capabilities can be expressed in the NSF Catalogue.
- *Abstract language generator*: it is responsible for generating the specific NSF abstract language in the form of an XSD file that specifies how it is possible to state NSF policies using the abstract language.
- *Policy translator*: it is responsible for translating abstract language based policy rules into the same policy rules expressed in NSF-specific low-level language.

The above-introduced files are in XMI, XML, or XSD format. XML (as well as XMI) format has been chosen because XML files can be validated through XSD ones. As described in Section 2.5, XSD languages allow stating what elements can be used in the corresponding XML file and in what order those elements can be used.

It is possible to state which element attributes are allowed, too. Finally, XML is well-known in the Computer Science field, and it has the appreciated characteristic to be human-readable and highly expressive.

The proposed framework introduces a new way to express Security Capability attributes using abstract language. In order to abstract even more the definition of Security Capabilities in policy rules expressed in abstract language, the first idea has been to introduce some general operators that would have been the same for each Condition Capability. This differentiates from the previous implementation since each Security Capability attribute had its naming convention. The newly introduced operators are:

- For *integer-typed* attributes: exact-match, union and range;
- For *string-typed* attributes: exact-match and union;
- For *IP Address-typed* attributes: exact-match, union, range, rangeMask, and rangeCIDR.

For Action Capability, on the other hand, one operator has been introduced explicitly for IKE algorithm agreement Security Capabilities: the *proposal* operator. This operator is similar to the union one for Condition Capabilities; it allows stating a list of proposed IKE protocol algorithms to stick with the IKE mechanism itself. The above operators will be discussed more in detail in the following chapters. The idea to introduce general operators

This thesis work also adds *Resolution Strategy* support. In particular, it has been introduced the possibility to add some Resolution Strategies to each supported NSF. Available Resolution Strategies are: First matching rule, Least matching rule, Deny Take Precedence, Allow Take Precedence, and some of their variation. Nonetheless, the actual framework implementation supports only the First matching rule. In order to implement it, abstract policy rules can now be coupled with *priority* details.

It is also implemented one of the most helpful tasks that has been described in the previous chapters: a BaseX-based XML database that can be queried to compare NSF Security Capabilities, check if a given NSF can implement a policy rule, or discover which NSF has a user-provided set of Security Capabilities.

Default Strategy support is also introduced with the proposed framework: it is possible to state which Action Security Capability has to be performed when none of the stated policy rules Event or Condition Clauses matches the incoming traffic.

Finally, a *web service* based on the Flask Python framework has been implemented in order to allow the usage of the proposed framework through a simple web interface. This web service allows the user to update the needed files for the tool's correct behavior and perform the needed steps to obtain abstract language or translate user-defined NSF policies.

Framework low level design is presented below in Figure 4.1, the presented components are:

- *Capability Information Model* is defined through Modelio modeler software, which describes how Security Capabilities can be coupled with the available NSF. In this Information Model, the decorator and the adapter patterns have been used, and they will be explained in the next chapters. CapIM defines the entities that characterize the NSF Catalogue.

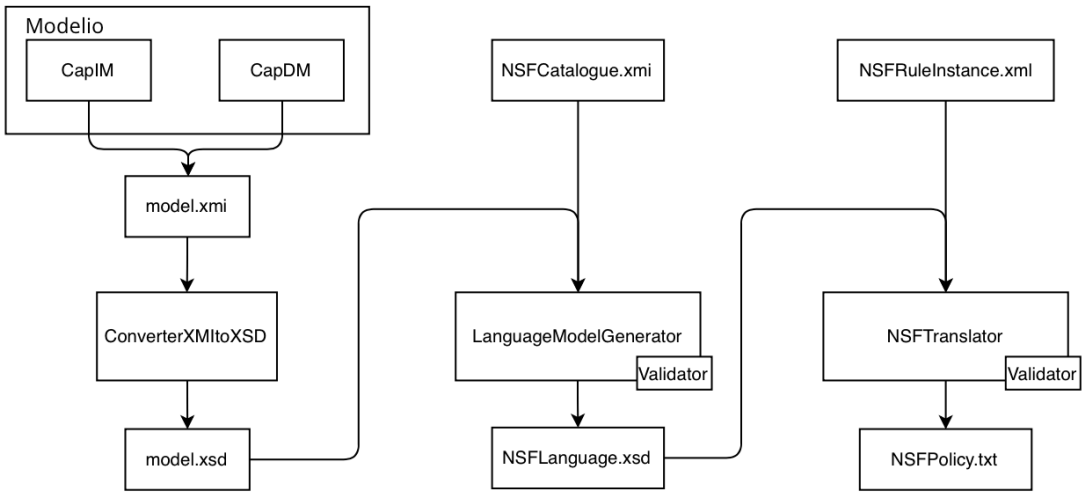

Figure 4.1: System Design

- *Capability Data Model* is defined through Modelio modeler software; it describes Security Capabilities in terms of required attributes and arranges them based on their functional area. Some characterization is Event, Condition, Action, Resolution Strategy, Default Action, and Evaluation Criterion clauses.
- *model.xmi* is obtained thanks Modelio exporting functionality, it stores both CapIM and CapDM element using XMI format. This file represents the starting point for the functioning of the proposed framework.
- *Converter XMI to XSD* is a generic tool that, starting from XMI files, creates an XSD based on the XML elements stored in the input. This tool is needed to create the XSD file in which it is stated how and which elements can be declared in the NSFCatalogue.xml file.
- *model.xsd* is the output of the previous tool in XSD format. It defines the constraints that the NSFCatalogue.xml elements have to respect. In particular, this file contains a formal description of how the Security Capabilities have to be defined and which attributes of which type can have each of them.
- *NSFCatalogue.xml* is the XML file that stores the available NSF and which Security Capabilities are coupled to them. It also contains essential XML elements such as abstract language generation details and low-level translation details for each Security Capability. The framework developer produces this file.
- *Language Model Generator* is a tool that can produce the NSF abstract language starting from model.xsd and NSFCatalogue.xml files.
- *NSFLanguage.xsd* is the output of the previous tool. It is an XSD file that states how policies can be expressed following the abstract language of a specific NSF.
- *NSFRuleInstance.xml* is an XML file storing the NSF policy rules expressed using NSF abstract language.

- *NSF Translator* is the tool able to translate NSF policy expressed using abstract language to policy expressed in NSF low-level language.
- *NSFPolicy.txt* is an XML file storing the translated policy rules.
- *Validator* is a tool that is used to validate XML files against the correspondent XSD files. It is used every by Language Model Generator to check if the NSFCatalogue.xml respects the constraints stated by the model.xsd file. NSF Translator also uses it to check if the input abstract language policies respect the costrains of the abstract language defined in NSFLanguage.xsd file.

Chapter 5

System implementation

The proposed framework, as described above, is heavily dependent on CapIM and CapDM developed through Modelio modeling software. Those Security Capability models are then processed through three Java tools: XMI to XSD Converter, Abstract language generator, and the NSF translator. These tools can be executed as jar files or thanks to the simple web service interface proposed. BaseX queries can be performed using a Docker image that runs a BaseX server instance that can be queried through `curl` command.

5.1 Capability Information Model

Information models provide a general description of the entities within a system and their functionalities. It does not cover details about technological implementation and working environments.

Taking inspiration from I2NSF Capability Information Model, the proposed framework has been developed on a new, extended CapIM. It specifies at a high level of abstraction how NSFs can be formally described and how Security Capabilities can be coupled. In CapIM, the decorator pattern is used for the relationship between NSFs and Security Capabilities. It results in the possibility of dynamically assigning Security Capabilities to NSFs and efficiently providing further translation details through the appropriate entities. The entities included in the CapIM are shown in Figure 5.1 are described in the following:

- *NSF*: class to describe an NSF, it is considered a container of decorations.
- *SecurityCapability*: class extended from NSF, it represents the security functionalities NSFs can have.
- *HasSecurityCapabilityDetails*: class representing further details about how a Security Capability belongs to the correspondent NSF.
- *NSFCatalogue*: class representing the set of available NSFs and the correspondent Security Capabilities. NSFCatalogue is a container class composed of: NSF class, CapabilityTranslationDetails class, LanguageGeneratorDetails class, and ResolutionStrategyDetails class.

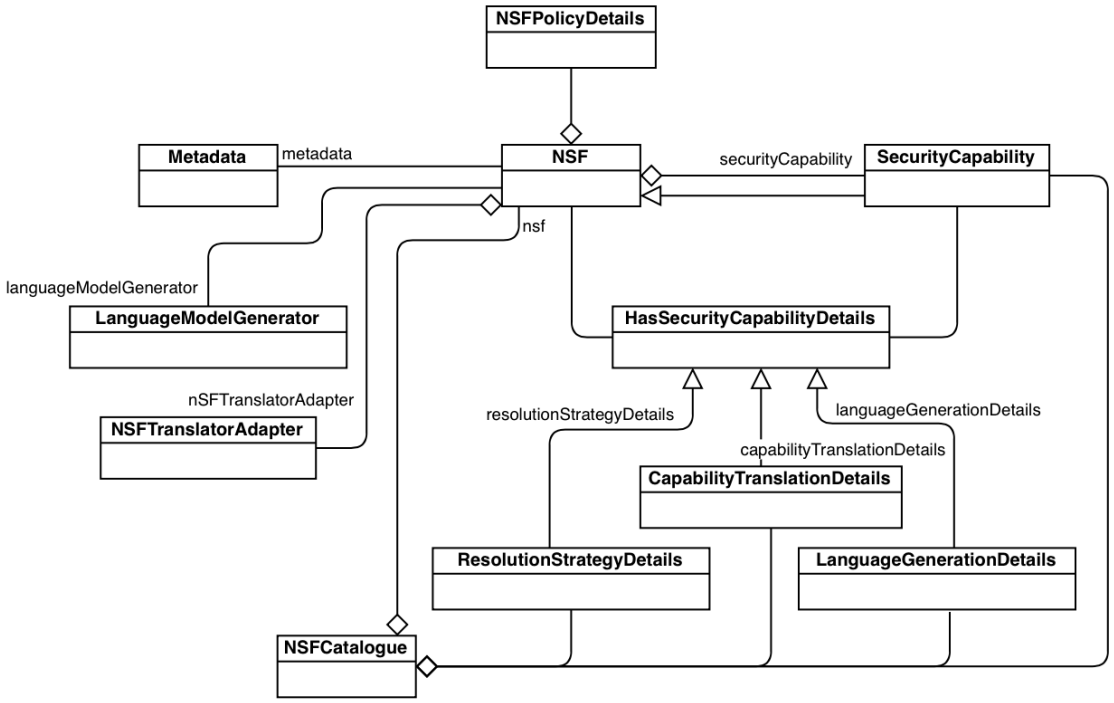

Figure 5.1: Capability Information Model

- *LanguageGeneratorDetails*: class extended from HasSecurityCapabilityDetails, it states the required NSF details in order to generate the correspondent abstract language.
- *CapabilityTranslationDetails*: class extended from HasSecurityCapabilityDetails, it states how abstract language policies can be translated into low level language ones.
- *ResolutionStrategyDetails*: class extended from HasSecurityCapabilityDetails, it states which Resolution Strategy a given NSF supports and which external data is required to be specified for each policy rule.
- *NSFPolicyDetails*: class representing additional information about how NSF low level policy rules have to be translated.
- *LanguageModelGenerator*: singleton class that perform the generation of the NSF abstract language, this is done using LanguageGeneratorDetails.
- *Metadata*: class representing possible data that can be NSF related. This is used by LanguageModelGenerator.
- *NSFTranslatorAdapter*: singleton class that perform the translation of policy rules from abstract language to NSF low level language, this is done using CapabilityTranslationDetails class.

Further details about some of the critical entities are proposed in the following sections. The following relationships characterize the previous classes:

- *NSF-SecurityCapability*: these relationships allow the usage of the decorator pattern; this allows decorating NSF with new functionalities using additional SecurityCapability classes.
  - *Aggregation*: NSF entity is formally described as an aggregation of its Security Capabilities.
  - *Generalization*: SecurityCapability is a descendant class of NSF one. Thus it inherits its attributes (and methods).
- *NSF-HasSecurityCapabilityDetails* and *SecurityCapability-HasSecurityCapabilityDetails*: these relationships allow to provide additional information about how NSF provide its Security Capabilities. It has been redesigned compared to the CapIM provided by I2NSF (Figure 3.4) in which this was an Association Relationship Class. This has been done to give more expressive power to HasSecurityCapabilityDetails class.
- *NSF-NSFPolicyDetails*: aggregation that provide further details about general data of how policies for the NSF have to be translated.
- *NSF-NSFTranslatorAdapter*: relationship that links NSF to the tool that is in charge to translate low level policies of the correspondent NSF.
- *NSF-LanguageModelGenerator*: relationship that links NSF to the tool that is in charge to generate the abstract language of the correspondent NSF.
- *NSF-Metadata*: relationship that allows stating additional metadata related to the NSF. This relationship is mainly used by the LanguageModelGenerator tool.
- *NSF-NSFCatalogue*: aggregation relationship that allows defining an NSFCatalogue in which it is possible to provide a list of supported NSF classes.
- *HasSecurityCapabilityDetails* generalization relationships allow extending the set of details about how NSF and Security Capabilities are linked together.
  - *CapabilityTranslationDetails* generalization allows specifying how abstract policies have to be translated into low level ones.
  - *LanguageGenerationDetails* generalization allows specifying how each Security Capability can be expressed using abstract language.
  - *ResolutionStrategyDetails* generalization allows specifying which external data a given Resolution Strategy needs.
- *NSFCatalogue* aggregation relationships allow to state the NSFCatalogue as a container of the following entities:
  - *NSF*
  - *SecurityCapability*
  - *CapabilityTranslationDetails*
  - *LanguageGenerationDetails*
  - *ResolutionStrategyDetails*

5.2 Capability Data Model

Data models, instead, are used to propose a more specific description of system entities and their relationships. They are meant for developers that have to know which design or technologies to use for the final solution.

Following I2NSF CapDM and Avallone proposed work, the proposed framework provides a formal description of Security Capabilities using a Capability Data Model. In the CapDM shown in Figure 5.2, Security Capabilities are arranged using the following 6-tuple:

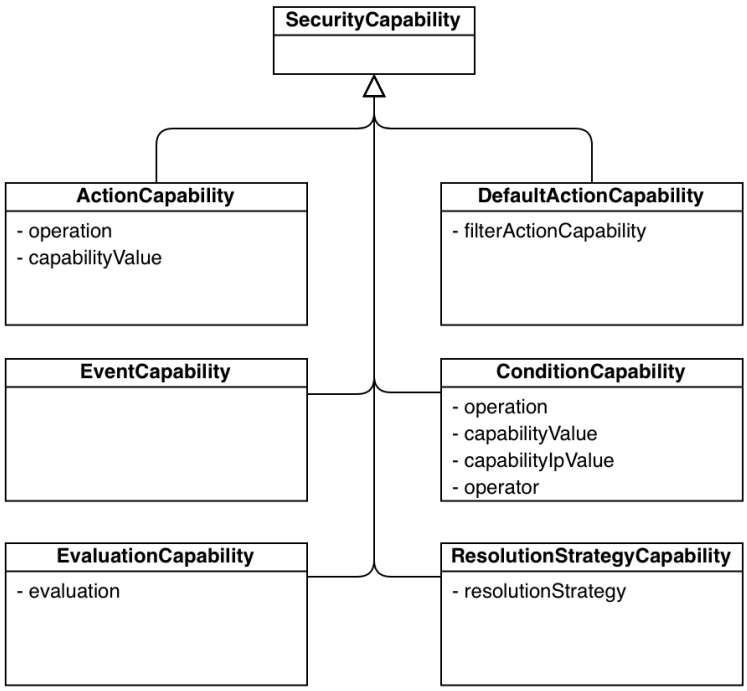

Figure 5.2: Capability Data Model

- *Event Capability*: class that represents the capability to detect whether a specific event is occurring or not.
- *Condition Capability*: class that represents the capability to check if a given condition is satisfied. Its attributes are thought to specify which of the supported Condition operator has to be used, the value of the Condition Capability or if the Condition Capability has to be negated.
- *Action Capability*: class that represents the capability to perform a specific security action. Its attributes are thought to specify which of the supported Action operator has to be used, the value of the Action Capability.
- *Resolution Strategy Capability*: class that represents the capability to enforce a resolution strategy when more than one Event and Condition evaluate as `TRUE`. Its attributes are thought to specify which method of resolution strategy the NSF follows. Following I2NSF proposal, allowed resolution strategy

methods are: First Matching Rule (FMR), Last Matching Rule (LMR), Prioritized Matching Rule (PMR), Prioritized Matching Rule with Errors (PMRE), Prioritized Matching Rule with No Errors (PMRN), *Deny Take Precedence* (DTP), and *Allow Take Precedence* (ATP). Among these, *FMR* is the resolution strategy currently supported.

- *Default Action Capability*: class that represents which Action has to be performed when none of the provided policy rules Events or Conditions do not evaluate as `TRUE`. Its attributes are thought to specify which default action has to be performed; only *filtering actions* have been implemented since it has been developed with more attention to IPSec scenarios.

- *Evaluation Criterion Capability*: class that represents how to evaluate Boolean clauses in Condition Capabilities. Its attributes are thought to specify which of the supported evaluation method has to be used. Allowed logical operators but not implemented yet are Negation Normal Form (NNF), Disjunctive Normal Form (DNF), Conjunctive Normal Form (CNF), and Algebraic Normal form (ANF).

Default Action Capability and *Resolution Strategy Capability* functionalities have been introduced in the current thesis work since they were not supported before.

Default Action Capability allows specifying only *filtering actions* since it has been developed with more attention to IPSec scenarios. Additionally, it is possible to specify Default Action Capability only for IpTables NSFs since its default behavior can be defined as an additional policy rule. When Default Action Capability is used in an abstract policy definition for IpTables, that will be translated as an additional policy rule.

Resolution Strategy Capability also supports only IpTables related resolution strategy methods: First Matching Rule. The remaining strategies are allowed, and this thesis work already has configured the framework so that the remaining ones can be implemented as well.

Finally, the proposed CapDM discerns Security Capabilities based on other hierarchies.

5.3 XMI to XSD generator

The XSD generator is the first tool on the XMI file obtained from the Modelio modeling tool. The XMI exported from Modelio contains both the CapIM and CapDM class definitions but without the correspondent diagrams. Compared to the previous thesis work, this tool has not been modified since its functioning is still compliant with the new additions to the framework.

The functioning of the XMI to XSD generator is based on an iteration of the input XMI nodes. For each of them, based on the kind of each node determines the production of an XSD node in the output files. As said, the XSD file format allows describing how another XML file should be defined. In particular, the XSD file produced from this tool will specify the format of the NSF Catalogue file nodes. Also, the produced XSD is used as a knowledge container for the other two tools in the framework.

The XMI from Modelio lists all the UML classes with their attributes and their literal enumerations. A correspondent XSD node will be generated in the output file for each of these items. The XSD output embodies a strict restriction on how NSF Catalogue elements should be defined in naming, attributes, and inheritance tree.

As an example of how the XMI to XSD Generator works, the following listings can be observed. With the proper omissions, Listing 5.1 displays how `NSFCatalogue` class is described from Modelio export in XMI file format. This XMI node can not be used as a structure description for the same node in other XMI files.

```
 <packagedElement xmi:type="uml:Class"
    xmi:id="_TzXbmpcgEeyZN708OqQtlQ" name="NSFCatalogue">
    <ownedAttribute xmi:id="_TzXbm5cgEeyZN708OqQtlQ"
       name="nSF"/>
    <ownedAttribute xmi:id="_TzXbnZcgEeyZN708OqQtlQ"
       name="securityCapability"/>
    <ownedAttribute xmi:id="_TzXbn5cgEeyZN708OqQtlQ"
       name="languageGenerationDetails"/>
    <ownedAttribute xmi:id="_TzXbopcgEeyZN708OqQtlQ"
       name="capabilityTranslationDetails"/>
    <ownedAttribute xmi:id="_TzXbpZcgEeyZN708OqQtlQ"
       name="resolutionStrategyDetails"/>
</packagedElement>
```

Listing 5.1: XMI node representing NSFCatalogue class

Hence, it is needed an XSD representation as shown in the Listing 5.2, in which it is properly described which structure and aspect the `NSFCatalogue` class should have in additional XMI files.

```
<xs:complexType name="NSFCatalogue">
    <xs:choice maxOccurs="unbounded" minOccurs="0">
        <xs:element maxOccurs="unbounded" name="nSF"
           type="NSF"/>
        <xs:element maxOccurs="unbounded"
           name="securityCapability" type="SecurityCapability"/>
        <xs:element maxOccurs="unbounded" minOccurs="0"
           name="languageGenerationDetails"
           type="LanguageGenerationDetails"/>
        <xs:element maxOccurs="unbounded" minOccurs="0"
           name="capabilityTranslationDetails"
           type="CapabilityTranslationDetails"/>
        <xs:element maxOccurs="1" minOccurs="0"
           name="resolutionStrategyDetails"
           type="ResolutionStrategyDetails"/>
    </xs:choice>
</xs:complexType>
```

Listing 5.2: XSD node describing NSFCatalogue class

5.4 Abstract language generator

Abstract language generator tool is responsible for producing NSF abstract language. After that, CapIM and CapDM XSD files are produced by the XMI to XSD Converter after the XMI export from Modelio; this tool takes this XSD file and the NSF as input Catalogue. Within the NSF Catalogue, it is specified which Security Capabilities the NSF provides, and LanguageGenerationDetails classes are listed there too.

Abstract language generator produces an XSD file containing NSF language; this file sets the constraints about which Security Capabilities can be used in the abstract language policies of the NSF and which values can be used for them.

5.4.1 LanguageGenerationDetails class

LanguageGenerationDetails class is a support entity that allows storing all the needed information to generate the NSF abstract language. This class is a subclass of HasSecurityCapabilityDetail; it is possible to state to which Security Capability it is linked.

In particular, LanguageGenerationDetails attributes are thought to specify, through a Model-Driven approach, which values are allowed and which not for each of the Security Capabilities coupled to the target NSF. Additionally, it is possible to define new custom types using enumeration entities.

LanguageGenerationDetails class and classes that uses as attributes are shown in Figure 5.3. Compared to Avallone's thesis work, these classes have not been redesigned.

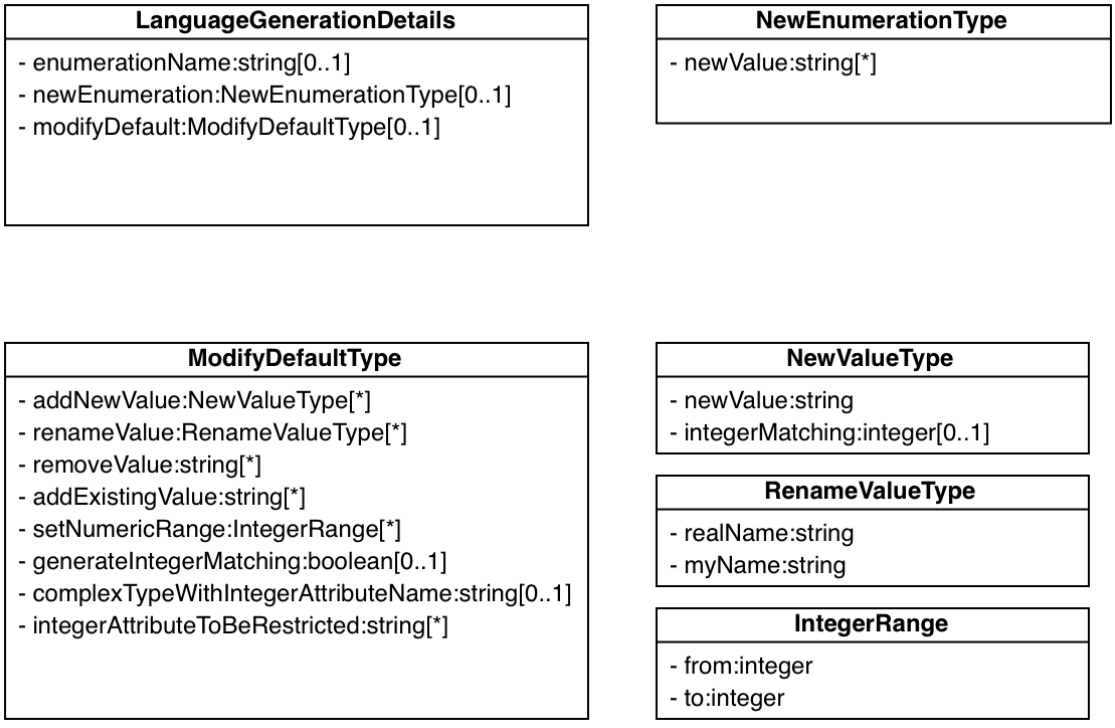

Figure 5.3: LanguageGenerationDetails

The main goal of LanguageGenerationDetails class is to make the user define a new *custom enumeration* type starting from a base enumeration (also referred as *default enumeration*)already present in the model. For example, the new enumeration can take attributes from the previous enumeration, remove or rename them. Also, it is possible to add brand new attributes to the custom enumeration. LanguageGenerationDetails is composed of the following entities:

- `enumerationName`: string attribute that allows specifying the name of the custom enumeration.
- `newEnumeration`: `NewEnumerationType` attribute which is composed by:
  - `newValue` string attribute, that allows expressing the string values that have to build the custom enumeration.
- `modifyDefault`: `ModifyDefaultType` attribute that allows changing values or set constraints on already existing Security Capability attributes. This attribute type is composed by:
  - `addNewValue`: `NewValueType` attribute composed of two further ones; a `newValue` string representing new literal to be added to the custom enumeration and a `integerMatching` optional integer which is linked as literal string substitution (e.g. this is used to couple Protocol Type name to correspondent integer).
  - `renameValue`: `RenameValueType` attribute composed of two further ones; a `realName` string representing the default enumeration attribute to rename and a `myName` string representing the new desired name for that attribute.
  - `removeValue`: string attribute that allows stating which attribute to remove from the default enumeration.
  - `addExistingValue`: string attribute that allows stating which attribute from the default enumeration has to be addedd in the custom one.
  - `setNumericRange`: `IntegerRange` attribute composed of two further ones; a `from` and a `to` integer representing the allowed integer range for the Security Capability inner value.
  - `generateIntegerMatching`: boolean attribute used to state if the language generator has to produce the correspondence betwee the string value of the enumeration and the correspondent integer.
  - `complexTypeWithIntegerAttributeName`: string attribute that allows specifying the default `complexType` in the CapDM from which the customization has to take place.
  - `integerAttributeToBeRestricted`: string attribute that allows stating on which default complexType attributes have to be restricted.

5.5 NSF low-level language translator

NSF translator tool is responsible for receiving the abstract policies for a given NSF and producing the low-level correspondent policies. The translation is made possible

by CapabilityTranslationDetails class and other ones described in the following. These classes are responsible for storing all the required information to make the tool correctly translate the policies based on the target NSF low-level language. Additionally, this tool checks if the provided abstract policy rules are compliant with the NSF abstract language generated by the Abstract language generator.

This is the framework component that had the significant changes compared to Avallone's thesis work. It has been introduced the NSFPolicyDetails class that allows specifying additional data about *how a policy rule for a given NSF has to be translated*. It is possible to specify the prefix (e.i. command tool name) or suffix of each translated policy rule. Also, it can be specified if specific keywords have to be postponed at the end of all the policy rules translated (e.g. `COMMIT` for IpTables policies). Finally, it can be specified a *Default Security Capability*; for a given NSF, the default Security Capability will always be added to the abstract language policy rules, even when that Security Capability has not been added to the policy rule itself.

Operators (e.g. exact-match, union, or range) have been introduced in order to provide values more efficiently and in the same way to all the Condition Capabilities. The same has been done for Action Capabilities. The main point about operators is that they can be used for any Action or Condition Capability. If the specific Capability does not support it, then the tool will be able to adapt the provided operator in the abstract language policy to the operator that the Security Capability supports. This is possible thanks to dynamically called expansion functions that rely on CapabilityTranslationDetails class related to a given NSF.

Resolution Strategies Capability has been introduced; the proposed framework implements the possibility to specify also Resolution Strategies Capability to a given NSF. In particular, it is possible to state any of the Resolution Strategies listed in Section 5.2 hence the only supported one is First Matching Rule (FMR). Thanks to FMR, it is possible to state a priority attribute for each abstract language policy rule then the tool will sort the output low-level policy rules in the order stated by the priority integer. In this way, it is possible to state NSF priorities that support FMR Resolution Strategy.

Default Action Capability functionality has been implemented too; it is possible to state a Default Action Capability node at the end of the abstract policy. That Action Capability can store any Security Capability offered by the target NSF.

Generic NSFs is another aspect introduced by the thesis work. Those NSFs are characterized by the fact that they are composed of Security Capabilities belonging only to specific security areas (e.g. filtering capabilities, IPSec capabilities). The abstract language obtained is not linked to any real-world NSF when using these generic NSFs. Then, when it comes to translating an eventual abstract policy of the generic NSF, it is possible to specify toward which real-world NSF that policy has to be translated. The advantage of this functionality is that automated tools or real users can define security policy and, during abstract policy definition, not worry about which NSF should be used for it.

5.5.1 NSFPolicyDetails class

NSFPolicyDetails class is an entity that stores all the information about how an NSF low-level policy has to be translated. This class is used as the inner attribute

of the NSF class, stored in the NSF Catalogue file. When the NSF Translator tool is instantiated, it retrieves from the NSF entity the correspondent NSFPolicyDetails data and, during the tool processing, these details are used to compound the final low-level policy rules properly.

NSFPolicyDetails class and classes that uses as attributes are shown in Figure 5.4.

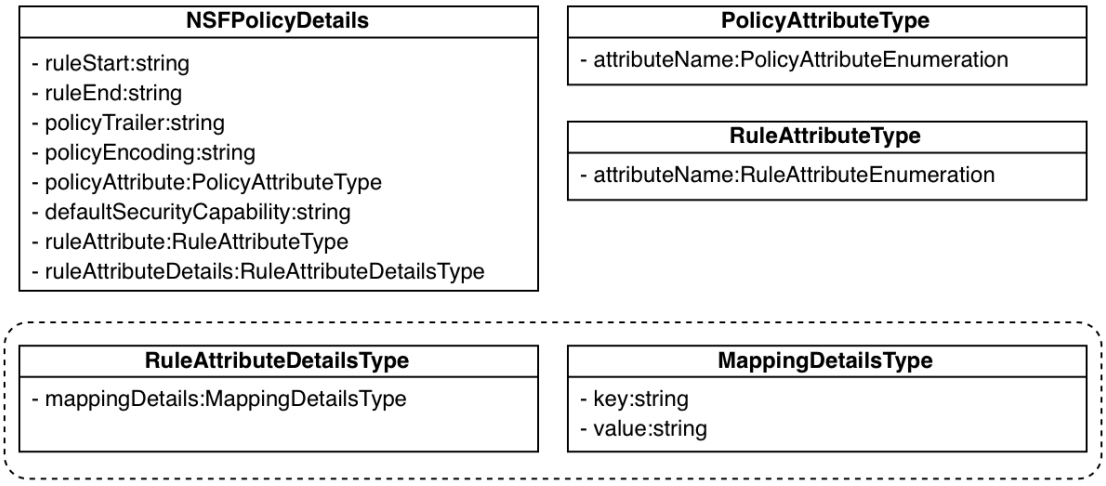

Figure 5.4: NSFPolicyDetails

NSFPolicyDetails class makes the final user avoid providing command-line information such as the command name that has to be placed at the beginning of or what has to be placed at the end of each policy rule translated. Additionally, it allows the user to specify a Default Security Capability that must always be added to a policy rule even if the user does not state it.

NSFPolicyDetails is made up of the following entities:

- `ruleStart`: string attribute that stores a string that the NSFTranslator will place at the beginning of every policy rule. It substantially replaces the `+s` input parameter of the previous version of the tool; in this way, information such as which is the command name of each NSF is stored by the model, and it has not to be provided by the final user through an input parameter.

- `ruleEnd`: string attribute that stores a string that the NSFTranslator will place at the end of every policy rule. It replaces the `+e` input parameter of the previous version of the tool; in this way, such information is stored by the model, and it has not to be provided by the final user through an input parameter.

- `policyTrailer`: string attribute that stores a string that the NSFTranslator will place at the end of every policy. It replaces the `+cre` input parameter of the previous version of the tool; in this way, such information is stored by the model, and it has not to be provided by the final user through an input parameter. This attribute could be used to store `COMMIT` value to be added at the end of the IpTables policy.

- `policyEncoding`: string attribute that stores the character encoding supported by the correspondent NSF. Based on that, the NSFTranslator tool will encode the output policy using that specified encoding.
- `policyAttribute`: `PolicyAttributeType` attribute that allows stating which attribute it is required in the policy element of the abstract policy file provided by the user. This attribute is composed of:
  - `attributeName`: string attribute that can have values from `PolicyAttributeEnumeration`: string enumeration containing the *allowed* additional attributes that can be stated in the policy element of the abstract policy file.
- `defaultSecurityCapability`: string attribute that state a Security Capability that will be always put by the NSFTranslator in each policy rule even if that specific Security Capability was not stated by the user.
- `ruleAttribute`: `RuleAttributeType` attribute that allows stating which attribute it is required in the rule element of the abstract policy file provided by the user. This attribute is composed of:
  - `attributeName`: string attribute that can have values from `RuleAttributeEnumeration`: string enumeration containing the *allowed* additional attributes that can be stated in the policy element of the abstract policy file.
- `ruleAttributeDetails`: `RuleAttributeDetailsType` attribute that allows stating optional information about how rule attributes have to be treated. In particular, it can happen that rule attribute value has to be replaced by NSF-specific values. In order to store this kind of detail, this attribute is composed of `mappingDetails` attribute. This is `MappingDetailsType` attribute that allows storing a list of mappings between which value has to be replaced and which is the string that has to be used, instead. `MappingDetailsType` is composed of `key` string attribute representing the value that has to be replaced and `value` string attribute that has to be used as substitution.

5.5.2 CapabilityTranslationDetails class

CapabilityTranslationDetails class is a support entity that allows storing all the needed information to translate the NSF low-level language policy rules. This class is a subclass of HasSecurityCapabilityDetail; it is possible to state to which Security Capability it is linked.

CapabilityTranslationDetails class allows defining NSF-specific ways of how a Security Capability used in an abstract policy rule has to be translated; in this way, the same Security Capability will be translated differently based on the target NSF toward which has to be translated.

CapabilityTranslationDetails class attribute values substantially specify the low-level policy rule semantic and syntax. Its attributes are shown in Figure 5.5, and those attributes govern the structure of the translated policy rule for each Security Capability.

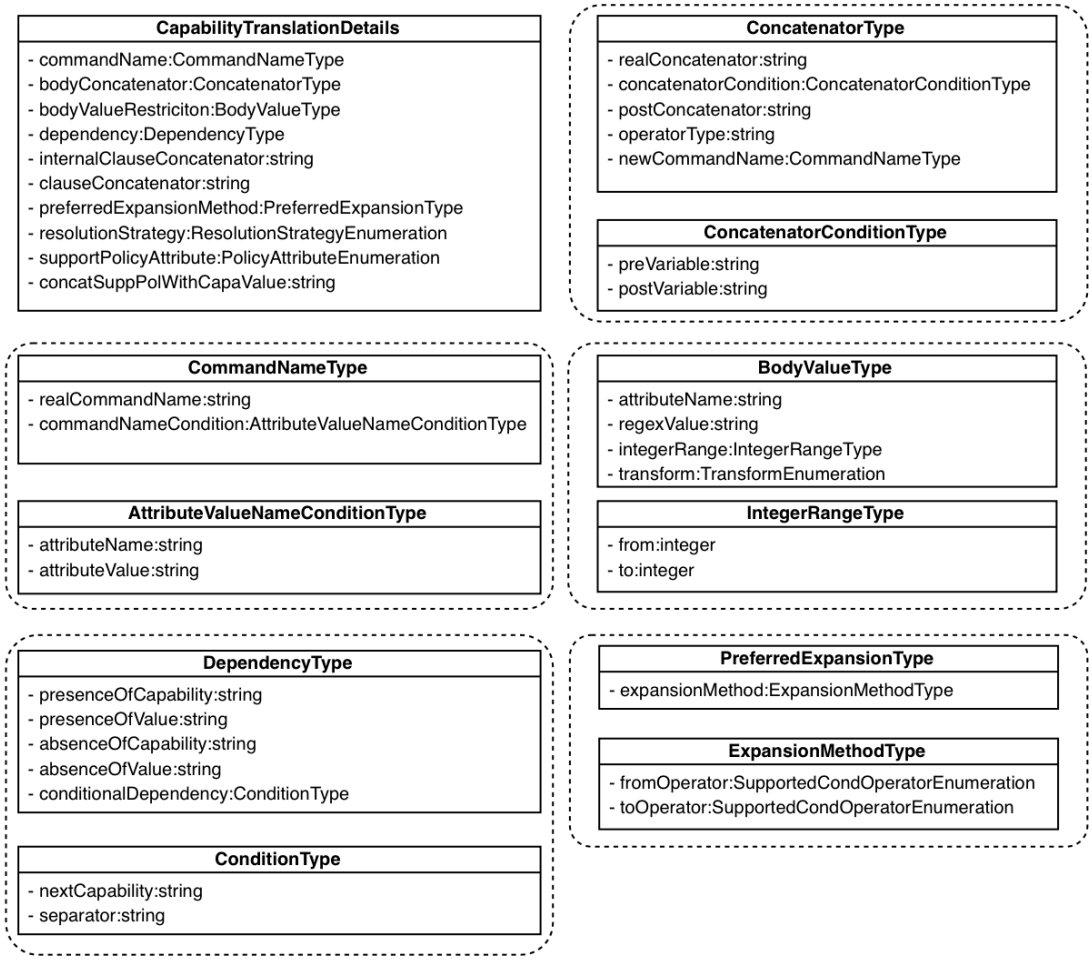

Figure 5.5: CapabilityTranslationDetails

CapabilityTranslationDetails class inner attributes `nSF` and `securityCapability` are needed in order to specify to which NSF and which Security Capability the class refers to. CapabilityTranslationDetails class is composed of the following entities:

- `commandName`: `CommandNameType` attribute used to specify which is the command string that corresponds to the related Security Capability for the correspondent NSF. This is stated thanks to:
  - `realCommandName`: string attribute that stores the command string correspondent to the related Security Capability.
  - `commandNameCondition`: `AttributeValueNameConditionType` attribute that allows setting conditions that has to be verified in order to use the before mentioned command string. The current attribute is composed of *attributeName* a string attribute that state which attributes within the policy rule has to be considered, and `attributeValue` string attribute that states which is the value that the above attribute has to have in order to consider the condition satisfied. This functionality is used to make the user use negated commands, for example.

- `bodyConcatenator`: `ConcatenatorType` attribute that stores the needed information to properly arrange the core parts that compose a low level rule. `bodyConcatenator` states how the NSF Translator has to parse input values for each Security Capability within the policy. This attribute is the one that has been more remodeled compared to Avallone's proposed framework; it allows handling operators for Condition and Action Security Capability. It is composed of the following attributes:
  - `realConcatenator`: string attribute that has to be interleaved between Security Capability provided values when there are more than one of them.
  - `concatenatorCondition`: `ConcatenatorConditionType` attribute that is used to determine if the realConcatenator string has to be used. It is composed of `preVariable` string attribute that contains the string that has to be found before the actual Security Capability value and `postVariable` string attribute that contains the string that has to be found after the actual Security Capability value.
  - `postConcatenator`: string attribute that has to be postponed between the last Security Capability provided value.
  - `operatorType`: string attribute that is used to couple the bodyConcatenator to one of the introduced operators. Different operators require to concatenate Security Capability values differently. Supported operators are exact match, union, range for integer-based or string based values following I2NSF principles described in Section 3.3.6. Additionally, other operators are rangeMask and rangeCIDR for IP Address values.
  - `newCommandName`: `CommandNameType` attribute already discussed above. It is worth noting that when is required to use an operator for a Security Capability value, it could be needed to override the original `commandName` stated in the `CapabilityTranslationDetails`.
- `bodyValueRestriciton`: `BodyValueType` attribute that stores several kind of restrictions that will be applied on the Security Capability values. It is possible to handle different types of values, hence this attribute is composed of:
  - `attributeName`: string attribute stating which attribute name has to be restricted.
  - `regexValue`: string attribute storing the regular expression that the provided value has to match.
  - `integerRange`: `IntegerRangeType` attribute that states `from` which value `to` which value the integer Security Capability value has to range.
  - `transform`: string `TransformEnumeration` that can store special transformations that could be applied on the provided Security Capability value.
- `dependency`: `DependencyType` attribute that stores contour requirements that the Abstract Language Rule has to respect. It is composed of:

  - `presenceOfCapability`: string attribute stating which Security Capability is mandatory within the Abstract Language Rule in respect to the current one.
  - `presenceOfValue`: string attribute stating which value is mandatory within the Abstract Language Rule.
  - `absenceOfCapability`: string attribute stating which Security Capability is denied within the Abstract Language Rule in respect to the current one.
  - `absenceOfValue`: string attribute stating which value is denied within the Abstract Language Rule.
  - `conditionalDependency`: `ConditionType` attribute stating which Security Capability has to immediately follow the current one. It is composed of `nextCapability` string and `separator` string that has to be interleaved between the current Security Capability and the one that comes after it.

- `internalClauseConcatenator`: string attribute that has to be interleaved between `realCommandName` and Security Capability value.

- `clauseConcatenator`: string attribute that has to be interleaved between the translation of the current Security Capability and the translation of the next one.

- `preferredExpansionMethod`: `PreferredExpansionType` attribute that is checked when the Abstract Language Policy uses an operator that is not supported for a Security Capability value. The standard behavior is to expand the current Security Capability value operator toward the immediately lower one. Using this attribute is possible to prevent that standard behavior and to state explicitly toward which operator the expansion has to be performed. This attribute is composed of:

  - `expansionMethod`: `ExpansionMethodType` attribute containing `fromOperator` string that represents the operator that is not supported and `toOperator` string that represents the operator toward which perform the expansion.

- `resolutionStrategy`: string attribute that can have values from `ResolutionStrategyEnumeration`. It is used only in a special bodyConcatenator that states which Resolution Strategy the referred NSF uses. At the beginning of the translation, NSF Translator will pick up this information and the low level policy will be influenced by it.

- `supportPolicyAttribute`: string attribute that can have values from `PolicyAttributeEnumeration`. It is used to state if the current NSF relies on attributes stated in the `Policy` node or the Abstract Language Rule file.

- `concatSuppPolWithCapaValue`: string attribute used as a boolean value to state if the NSF translator has to concatenate the value from supportPolicyAttribute and the value provided in the Abstract Language Rule.

5.5.3 ResolutionStrategyDetails class

ResolutionStrategyDetails class is a support entity that NSF Translator uses to discover if a specific Resolution Strategy requires some external data to be provided in the Abstract Language Rule. In this way, the NSF Translator can look up for that external data during Abstract Language Policy translation.

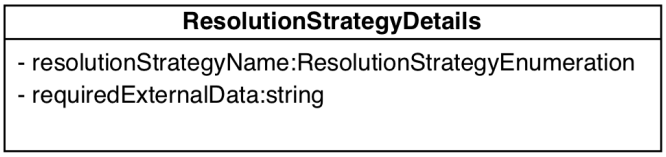

Figure 5.6: ResolutionStrategyDetails

This class is shown in Figure 5.6 and it is composed of the following attributes:

- `resolutionStrategyName`: string attribute that can have values from `ResolutionStrategyEnumeration`, it states for which Resolution Strategy the class is expressing additional information.
- `requiredExternalData`: string attribute that states which kind of external data the correspondent Resolution Strategy requires in order to be appropriately managed.

An example of the usage of this class is to specify that *First Matching Rule* resolution strategy, the default for IpTables NSF, requires external data of type priority. Based on that external data, NSF Translator will be able to properly arrange the low-level rules.

5.6 System extendibility verification

A general workflow of how the framework has to be used is displayed in Figure 5.7. It is worth noting that the proposed steps are always the same for every new NSF the developer needs to add to the proposed framework. The proposed figure verifies that the proposed framework can be flawlessly enriched with further NSF instances.

In particular, XMI, XML, and textual files have to be edited through a text editor of choice. At the same time, the proposed framework is provided through Java JAR files and simple web service APIs.

Starting from the CapIM and CapDM already extracted from Modelio and converted in XSD format, the workflow consists of the following steps:

1. *Create a new NSF in NSFCatalogue*: add a new `nSF` element within NSFCatalogue file.
2. *Attach Security Capabilities to the newly created NSF*: add `securityCapability` elements and `nsfPolicyDetails` element within the `nSF` element. In this way, it is stated that the NSF provides the coupled Security Capability and that it owns the provided policy details.

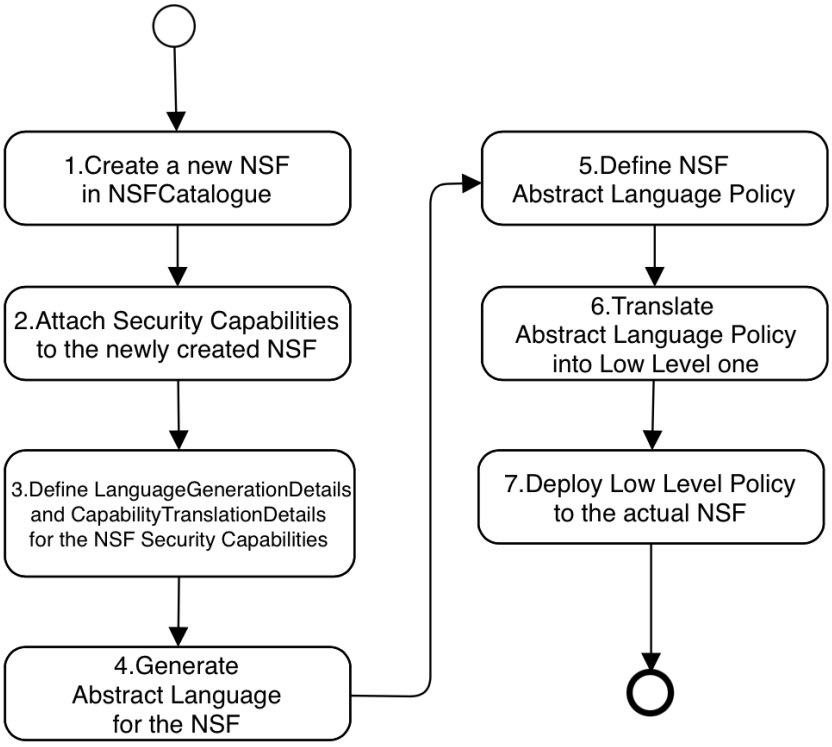

Figure 5.7: Framework workflow

3. *Define LanguageGenerationDetails and CapabilityTranslationDetails for the NSF Security Capabilities*: if it is needed to change existing Security Capability details, add a `LanguageGenerationDetails` for that Security Capability. Then, for each Security Capability, add `CapabilityTranslationDetails` in which the NSF Translator can find details about how each Security Capability has to be translated into low-level language.

4. *Generate Abstract Language for the NSF*: execute Language Generator tool.

5. *Define NSF Abstract Language Policy*: create an Abstract Language Policy containing the desired security functionalities for the selected NSF.

6. *Translate Abstract Language Policy into Low Level one*: execute NSF Translator tool providing Abstract Language Policy file, the output will be low-level policy file for the selected NSF.

7. *Deploy Low-Level Policy to the actual NSF*: leverage the previously generated low-level policy file containing data that can be directly provided to the actual NSF instance.

Low-level implementation of how the workflow can be executed is described in the following. Further details about technical implementation can be found in Appendix B.

5.6.1 Java implementation

Previous sections delineate the existence of three main components: ConverterXMItoXSD, LanguageModelGenerator, and NSFTranslator. Each of these is represented by a Java class plus some auxiliary Java classes for specific operations. The details about what each of these components resolves have already been described.

The workflow in Figure 5.7 does not include the generation of XSD file containing CapIM and CapDM. This is done by executing the ConverterXMItoXSD component. Step 1, Step 2, Step 3, and Step 5 are about properly editing NSFCatalogue and RuleInstance files. While Step 4 and Step 6 are about executing, respectively, LanguageModelGenerator and NSFTranslator components.

A possible usage scenario of framework components is to execute Java files using Java Runtime Environment and provide input files through file path specification. This is possible thanks to the fact that framework components Java classes are provided through JAR files. More details are shown in Appendix A.

5.6.2 Server implementation

Additionally, it is possible to leverage the proposed framework functionalities using a web server provided through Flask Web Service framework running in a Docker container. The web server emulates the same directory structure as the one required when executing the components JAR files. The web server serves four web pages:

- *Home*: this web page provides a view of the server working directories, allows the user to delete the content of the directory structure, and to navigate across the web pages dedicated to each framework component.
- *Upload Capability Data Model*: this web page allows the user to upload the CapIM and CapDM XMI file obtained from Modelio Export XMI functionality. Then it is possible to instantiate the generation of the security capability model file into XSD file format.
- *Upload NSF Catalogue*: this web page allows the user to upload the NSF Catalogue XMI file. Then the user can specify for which NSF it desires to generate the corresponding abstract language and to instantiate the abstract language's actual generation.
- *Upload Rule Instance*: this web page allows the user to upload a user-defined Rule Instance for the desired NSF. Thanks to it, it is possible to instantiate the translation of the abstract language policy into a low-level language one. The web server will assume that the NSF for which the Rule Instance is configured is stated within the file XML elements.

The web server is able to handle the majority of possible errors during the framework components execution and possibly misspelled input strings by the user.

5.7 NSF Catalogue as a service

Finally, a BaseX-based XML Database is provided through BaseX REST API Web Server functionality. The BaseX web server does not come with a GUI, but it has to be used through `curl` or Python `requests` package.

The web server does not regard the framework components' execution. Instead, it is used to provide the user the functionality to check in advance the available NSFs, which are the NSFs that have Security Capabilities in common or which NSF can implement which Rule Instances. The NSF Catalogue database satisfies the goal of offering the user the ability to compare different NSFs and check NSF inner details regarding which Security Capabilities their offer.

As described in Section 2.7, BaseX framework provides a simple web server that is able to serve REST HTTP requests. Through HTTP Requests, it is possible to ask the web server to run BaseX `.xq` query-format files.

BaseX web server is set up in a Docker container which stores all the needed queries and support files to allow the user to run one of the following queries:

- *Two NSFs comparison*: query that receives two NSF names and checks if their Security Capability Sets are: contained each other, equivalent, or their intersection is empty.
- *One NSF Comparison*: query that receives an NSF name and returns the list of NSFs that provide the same Security Capabilities provided by the input NSF.
- *Find NSF based on rules*: query that receives the path of a Rule Instance file and returns the NSF that can implement the policies stated within the above file.
- *Find NSF based on Security Capabilities*: query that receives a set of Security Capabilities and returns which NSF provides those items.
- *Find all the Security Capabilities*: query that returns the list of all the available Security Capabilities arranged by type.

NSF Catalogue database is a support tool that an external refinement framework could leverage when performing high-level queries about NSF details. In particular, this web server could be queried when high-level policy requirements have to be implemented through some provider security services. After the security requirement refinement, the refinement framework could have information about which Security Capabilities are required to enforce them. It could be possible to query the NSF Catalogue web server to discover which NSF provides the required Security Capabilities, which NSFs are equivalent in terms of Security Capability and other NSF details as stated in the proposed queries above. After this phase, the refinement tool could decide to instantiate the selected NSF, define the correspondent Rule Instance using NSF Abstract Language, and then execute the NSF Translator tool.

Chapter 6

System validation

In order to properly validate the proposed system, the framework's overall functioning is displayed in the following sections. Additionally, the proposed tool has been validated in the context of a wider refinement tool, and it has proven to work correctly in terms of service exposure and desired low-level policy translation.

Some of the problems that are faced are syntax customization of the output policy strings, general way to provide Security Capability values, usage of Resolution Strategy or Default Action functionalities, the definition of specific rule types, usage of generic NSFs, and exposing all the proposed functionalities as a web server.

The following sections provide an example of usage of the main features introduced with the proposed thesis work. The newly introduced functionalities are shown with attention to which is the correspondent output for each of them. *IpTables* and *XFRM* NSFs are taken into consideration in order to show off the proposed framework functioning properly.

6.1 NSF-specific policy details

In order to ease the framework usage from the point of view of the final user, `NSFPolicyDetails` class is used to store additional information that before had to be provided by the user itself. Introducing some important automation, NSFPolicyDetails structure for IpTables is shown in Listing 6.1.

```
<nsfPolicyDetails>
  <ruleStart>iptables</ruleStart>
  <ruleEnd/>
  <policyTrailer/>
  <policyEncoding/>
  <policyAttribute>
    <attributeName>targetRuleSet</attributeName>
  </policyAttribute>
  <defaultSecurityCapability>
      appendRuleActionCapability
  </defaultSecurityCapability>
</nsfPolicyDetails>
```

Listing 6.1: NSFPolicyDetails for IpTables

This element has to be instantiated within IpTables `nSF` element, in NSF Catalogue XML file. It stores the start and the end string that have to be used for low-level rule translation; in this way, the user does not have to provide these external and immutable data each time NSF Translator is executed. In the IpTables case, only `ruleStart` is used, while `ruleEnd` is empty, meaning the translator will assume the newline character to be used at the end of each rule. Additionally, it is also possible to state `policyTrailer` that is a string that is placed at the end of the policy, after all the translated rules. A use case could be COMMIT keyword when IpTables table-based low-level rules are produced. `policyEncoding` attribute allows specifying which string encoding the actual NSF requires. `policyAttribute` for IpTables is `targetRuleSet`, this means that NSF Translator will search for an attribute named like that within the `policy` element in Abstract Language file. This functionality is possible thanks to `supportPolicyAttribute` attribute in `bodyConcatenator` node as shown in Listing 6.2, this attribute expresses the name of the policy node attribute that the referenced Security Capability requires. `defaultSecurityCapability` for IpTables is *appendRuleActionCapability* since all its policy rules requires to append the rule in one of the IpTables available chains. In this way this Security Capability is added at the top of each rule, even if the user has not to provided it.

```
<capabilityTranslationDetails>
  <nSF ref="IpTables"/>
  <securityCapability ref="AppendRuleActionCapability"/>
  <commandName>
    <realCommandName>-A</realCommandName>
  </commandName>
  <supportPolicyAttribute>targetRuleSet</supportPolicyAttribute>
  ...
</capabilityTranslationDetails>
```

Listing 6.2: BodyConcatenator with supportPolicyAttribute

For the sake of clearness, in Listing 6.3 is shown how `policy` node attributes are stated.

```
<policy nsfName="IpTables" targetRuleSet="INPUT" ... .../>
```

Listing 6.3: Policy attribute for IpTables

Given the above details about IpTables, one of the output low level rules is shown in Listing 6.4: `iptables` string is concatenated at the beginning of the low level rule, *AppendRuleActionCapability* (`-A`) is added by the NSF Translator automatically, and `INPUT` keyword is retrieved by `targetRuleSet` policy attribute.

```
iptables -A INPUT -p TCP -s 203.0.113.0/24 -j DROP
```

Listing 6.4: Low level rule for IpTables

As an adjunct of `nsfPolicyDetails` element, it is possible to state if an NSF requires a specific Resolution Strategy through `capabilityTranslationDetails`

element. IpTables uses *First Matching Rule* strategy and it is possible to express this as shown in Listing 6.5.

```
<capabilityTranslationDetails>
  <nSF ref="IpTables"/>
  <securityCapability ref="ResolutionStrategyCapabilitySpec"/>
  <resolutionStrategy> FMR </resolutionStrategy>
</capabilityTranslationDetails>
```

Listing 6.5: Resolution Strategy specification for IpTables

In order to use the above `capabilityTranslationDetails`, it is needed to state if the First Matching Rule strategy requires some external data. In this case it does and it requires *priority* specification for each Abstract Language rule. Listing 6.6 demonstrates how to specify the above through `resolutionStrategyDetails` element.

```
<resolutionStrategyDetails>
  <resolutionStrategyName> FMR </resolutionStrategyName>
  <requiredExternalData>priority</requiredExternalData>
</resolutionStrategyDetails>
```

Listing 6.6: Resolution Strategy details for IpTables

While in Abstract Language rule, priority is stated as shown in Listing 6.7. In this way output low level rules will be sorted based on the priority value provided by the user.

```
<policy nsfName="IpTables">
    <rule>
        <externalData type="priority">2</externalData>
        ...
    </rule>
    ...
</policy>
```

Listing 6.7: Resolution Strategy priority for IpTables

6.1.1 IpSec rule type case

For the sake of clearness, XFRM nsfPolicyDetails is proposed in Listing 6.8. It can be observed that similar to the IpTables case, XFRM low-level rules will be translated placing `ip xfrm` at the beginning of the string, that `ipSecAction` attribute is required in policy node, and `ipSecRuleTypeActionCapability` is the default Security Capability to be automatically added for each abstract policy. The most important aspect for XFRM is the fact that it also requires `ruleType` attribute to be stated within the rule node and the fact that this attribute values are constrained to `SecurityAssociation` and `SecurityPolicy` strings. In order to prepare the thesis work for future enhancement, IPSec functionalities have to be split based on if they

regard the definition of Security Association (SA) or Security Policy (SP). For this reason, the ruleType attribute describes XFRM rules and makes the NSF Translator understand if the rule regards SA or SP. As a consequence, the NSFPolicyDetails `ruleAttributeDetails` node allows the NSF Translator to know how to handle SA or SP. In fact, this node stores the strings to be replaced when SA or SP rule type is defined. For XFRM low level rules, the strings `state` or `policy` have to be used properly.

```
<nsfPolicyDetails>
  <ruleStart>ip xfrm</ruleStart>
  <policyAttribute>
    <attributeName>ipSecAction</attributeName>
  </policyAttribute>
  <ruleAttribute>
    <attributeName>ruleType</attributeName>
  </ruleAttribute>
  <ruleAttributeDetails ref="ruleType">
    <mappingDetails>
      <key>SecurityAssociation</key>
      <value>state</value>
    </mappingDetails>
    <mappingDetails>
      <key>SecurityPolicy</key>
      <value>policy</value>
    </mappingDetails>
  </ruleAttributeDetails>
  <defaultSecurityCapability>
      ipSecRuleTypeActionCapability
  </defaultSecurityCapability>
</nsfPolicyDetails>
```

Listing 6.8: NSFPolicyDetails for XFRM

6.2 Condition and Action Security Capabilities operators

Operator usage is one of the essential functionalities introduced. The main objective has been to provide to the user a *further abstraction* in terms of how Security Capability values can be provided. In fact, the proposed framework generates an Abstract Language in which all the operators can always be used independently, even if the NSF or the Security Capability supports the operator. This is possible through an *expansion* mechanism that works as follows: when Abstract Language rules use not available operators, the NSF Translator will take care of it, replacing the used operator with the one supported by the NSF. The introduced operators are described in Table 6.1.

Operator	Value type	Security Capability
exactMatch	Integer, string, IPAddress	Condition, Action
union	Integer, string, IPAddress	Condition
range	Integer, IPAddress	Condition
rangeMask	IPAddress	Condition
rangeCIDR	IPAddress	Condition
proposal	String	Action

Table 6.1: Available Security Capability operators

In the following, more details about each operator:

- *exactMatch* operator is used when only one value is provided for Condition or Action Security Capability. It supports integer, string, and IPAddress types.
- *union* operator is used when a set of values is provided for Condition Security Capability. It supports integer, string, and IPAddress types.
- *range* operator is used when a start and end values are provided for Condition Security Capability. It supports integer and IPAddress types.
- *rangeMask* operator is used to provide as Condition Security Capability value an IPAddress in the format of x.x.x.x/x.x.x.x where x is a decimal value between 0 and 255.
- *rangeCIDR* operator is used to provide as Condition Security Capability value an IPAddress in the format of x.x.x.x/y where x is a decimal value between 0 and 255 and y is a decimal value between 0 and 32.
- *proposal* operator is used to provide as Action Security Capability value a set of strings representing the desired encryption algorithms for IPSec operations.

In Section 6.2.1 it is shown an example of how integer-based operators function. In particular, it properly describes how the expansion mechanism works. For the sake of readability, IP-based operators are omitted.

6.2.1 IpTables SourcePortConditionCapability

SourcePortConditionCapability for IpTables requires the user to provide integer values representing TCP ports. This Security Capability supports exactMatch, union, and range operators as can be seen in Listing 6.9 and Listing 6.10. exactMatch bodyConcatenator is omitted.

Union operator uses "`,`" to concatenate the single value and/or the range value that the user provides in order to represent TCP ports.

```
<bodyConcatenator>
  <operatorType>union</operatorType>
  <realConcatenator>,</realConcatenator>
  <concatenatorCondition>
      <preVariable>elementRange</preVariable>
      <postVariable>elementValue</postVariable>
    </concatenatorCondition>
    <newCommandName>
      <realCommandName>-m multiport --sports</realCommandName>
    </newCommandName>
  ...
  ...
</bodyConcatenator>
```

Listing 6.9: Union bodyConcatenator for SourcePortConditionCapability

Range operator uses ":" to concatenate the start and the end range values that the user provide in order to represent TCP ports.

```
<bodyConcatenator>
  <operatorType>range</operatorType>
  <realConcatenator>:</realConcatenator>
  <concatenatorCondition>
    <preVariable>start</preVariable>
    <postVariable>end</postVariable>
  </concatenatorCondition>
  <newCommandName>
    <realCommandName>-m multiport --sports</realCommandName>
  </newCommandName>
    ...
    ...
</bodyConcatenator>
```

Listing 6.10: Range bodyConcatenator for SourcePortConditionCapability

In both cases, it is specified that the Security Capability command name has to be translated according to the specific operator.

SourcePortConditionCapability values can be provided using one of the formats displayed in the following listings. By default, SourcePortConditionCapability supports the mentioned operators. Nonetheless, the expansion functioning is demonstrated, too.

When the operator used within the Abstract Language Rule is not supported, the NSF Translator will still allow the user to use it. Then no error will happen during the translation. In fact, the NSF Translator will *expand* the used operator toward one of the Security Capability available operators.

Exact match operator

For exactMatch operator, no kind of expansion is needed since the framework has made all the available Security Capabilities to support it. Listing 6.11 and List-

ing 6.12 show how exactMatch operator is translated.

```
<sourcePortConditionCapability operator="exactMatch">
  <capabilityValue>
    <exactMatch>
        80
    </exactMatch>
  </capabilityValue>
</sourcePortConditionCapability>
```

Listing 6.11: Abstract language rule using exactMatch

```
--sport 80
```

Listing 6.12: Low level rule using exactMatch

Union operator

Listing 6.13 shows how union operator is used within the abstract language rule. Listing 6.14 represents the case in which union operator is supported and Listing 6.15 describes the case in which union is not supported and then the expansion toward exactMatch has to be performed.

```
<sourcePortConditionCapability operator="union">
  <capabilityValue>
    <union>
      <elementValue>10</elementValue>
      <elementValue>20</elementValue>
      <elementValue>30</elementValue>
    </union>
  </capabilityValue>
</sourcePortConditionCapability>
```

Listing 6.13: Abstract language rule using union

```
-m multiport --sport 10,20,30
```

Listing 6.14: Low level rule using union

```
--sport 10
--sport 20
--sport 30
```

Listing 6.15: Low level rule expansion from union to exactMatch

Range operator

Listing 6.16 shows how range operator is used within the abstract language rule and Listing 6.17 represents the case in which range operator is supported. Listing 6.18 describes the case in which range is not supported but union operator is, hence the expansion toward union has to be performed and Listing 6.19 describes the case in which range nor union are supported and then the expansion toward exactMatch has to be performed.

```
<sourcePortConditionCapability operator="range">
  <capabilityValue>
    <range>
      <start>30</start>
      <end>35</end>
    </range>
  </capabilityValue>
</sourcePortConditionCapability>
```

Listing 6.16: Abstract language rule using range

```
-m multiport --sport 30:35
```

Listing 6.17: Low level rule using range

```
-m multiport --sport 30,31,32,33,34,35
```

Listing 6.18: Low level rule expansion from range to union

```
--sport 30
--sport 31
...
--sport 35
```

Listing 6.19: Low level rule expansion from range to exactMatch

6.3 Default Action Capability

Support for Default Action is another primary functionality introduced. This kind of Security Capability is important since it represents what the NSF must perform when no previous rule conditions are satisfied. Default Action Capability is handled by the proposed tool as an additional rule, but with a different name: `defaultActionCapabilitySpec`. At the moment, IpTables and ethereumWebAppAuthz NSFs can handle Default Action functionality. As shown in Listing 6.20, this Security Capability is handled by means of `capabilityTranslationDetails` class.

```
<capabilityTranslationDetails>
  <nSF ref="IpTables" />
  <securityCapability ref="DefaultActionCapabilitySpec" />
  <commandName>
    <realCommandName>-P</realCommandName>
  </commandName>
  <supportPolicyAttribute>targetRuleSet</supportPolicyAttribute>
</capabilityTranslationDetails>
```

Listing 6.20: IpTables Capability Translation Details for Default Action

Thanks to `capabilityTranslationDetails`, it is possible to understand which NSF the Default Action refers to and how that NSF handles the Default Action within the low-level language syntax. In the case of IpTables, this Security Capability is translated using the `-P` command that is specifically for setting a policy strategy for a given chain, resulting in a Default Action to be applied. In Listing 6.20, an example of `supportPolicyAttribute` is displayed again; meaning that the NSF Translator will require a `tergetRuleSet` attribute in policy node in order to properly produce low level language for the Default Action Capability.

The low-level language output regarding this Security Capability is shown in Listing 6.21.

```
iptables -P INPUT -j ACCEPT
```

Listing 6.21: IpTables low level policy for Default Action

6.4 Defining a generic NSF

One of the problems security administrators faces is deciding which NSF to use to enforce a specific range of security policies, leading to decision time that could be used to fine-tune the security policy itself. For this reason, the proposed framework introduces the concept of *generic* Network Security Functions. Generic NSFs are thought to allow the user to set up specific security policies using the required Security Capabilities without selecting which NSF to use in advance.

In this way, the security policy is defined first, and the NSF that has to implement it is fixed second. At the moment, a *generic packet filter NSF* is provided by the thesis tool. It includes all the Security Capabilities related to layer 3 filtering as shown in Listing 6.22.

When a user or an external refinement tool would like to define a packet filter policy, generic packet filter NSF could be used. The abstract policy definition will not require to specify the specific real-world NSF but the generic packet filter instead. Even if no real NSF is specified, it is still possible to use Security Capabilities and provide their correspondent values as shown in Listing 6.23.

```
<nSF id="genericPacketFilter">
  <securityCapability ref="IpProtocolTypeConditionCapability"/>
  <securityCapability ref="IpSourceAddressConditionCapability"/>
  <securityCapability
     ref="IpDestinationAddressConditionCapability"/>
  <securityCapability ref="SourcePortConditionCapability"/>
  <securityCapability ref="DestinationPortConditionCapability"/>
  <securityCapability ref="AcceptActionCapability"/>
  <securityCapability ref="RejectActionCapability"/>
  <securityCapability ref="DefaultActionCapabilitySpec"/>
</nSF>
```

Listing 6.22: Generic packet filter NSF definition

Subsequently, only when the abstract policy has to be translated into low level ones, the real NSF to use has to be specified to the NSF Translator. Optionally, the NSF Catalogue as a service can be exploited to understand which NSF supports the Security Capabilities owned by the generic packet filter itself. More details about how to use the NSF Catalogue as a service are shown in Appendix A.5.

```
<policy nsfName="genericPacketFilter" targetRuleSet="INPUT">
    <rule id="0">
      <ipProtocolTypeConditionCapability operator="exactMatch">
        <capabilityValue>
          <exactMatch>tcp</exactMatch>
        </capabilityValue>
      </ipProtocolTypeConditionCapability>
      <ipSourceAddressConditionCapability operator="exactMatch">
        <capabilityIpValue>
          <exactMatch>192.168.1.0</exactMatch>
        </capabilityIpValue>
      </ipSourceAddressConditionCapability>
      <ipDestinationAddressConditionCapability
         operator="exactMatch">
        <capabilityIpValue>
          <exactMatch>192.168.1.2</exactMatch>
        </capabilityIpValue>
      </ipDestinationAddressConditionCapability>
      <destinationPortConditionCapability operator="exactMatch">
        <capabilityValue>
          <exactMatch>80</exactMatch>
        </capabilityValue>
      </destinationPortConditionCapability>
      <rejectActionCapability/>
    </rule>
</policy>
```

Listing 6.23: Abstract policy for generic packet filter NSF

As a validating procedure, it is possible to consider the case in which the above abstract policy must be translated toward IpTables NSF. In order to do so, the

NSF Translator tool requires an additional input parameter `destinationNSF` that states toward which NSF the generic NSF abstract policy has to be translated. Here, it is possible to appreciate the importance of `defaultSecurityCapability` in Listing 6.1. IpTables low-level rule requires to specify the chain to which the rule has to be appended, but generic packet filter NSF does not have this kind of Security Capability since it does not make sense for a generic packet filter. The NSF Translator can understand that IpTables requires the Security Capability `appendRuleActionCapability` even if this Security Capability is not listed in the abstract policy. Moreover, the chain value to use for this Security Capability is retrieved by `targetRuleSet` attribute within the policy node.

6.5 NSF Catalogue as a service: NSF details discovery

The proposed framework is thought to be used by external automated tools. The first information that the framework has to provide to the outside is an accurate description of the NSFs it supports. This is done using a BaseX web server that stores the NSF Catalogue file and performs predefined XML Database queries. These predefined queries are:

- `two_nsf_comparison.xq`: it receives two NSF names and checks if their Security Capability Sets are: contained each other, equivalent, or their intersection is empty.
- `one_nsf_comparison.xq`: it receives an NSF name and returns the list of NSFs that provide the same Security Capabilities provided by the input NSF.
- `rule_nsf_search.xq`: it receives the path of a RuleInstance file and returns the NSF that can implement the policies stated within the above file.
- `capa_set_search.xq`: it receives a set of Security Capabilities and returns which NSF provides those items.
- `overall_search.xq`: it returns the list of all the available Security Capabilities arranged by type.

A Docker container hosts the Base X server and it is possible to perform one of these queries using `curl` or Python `requests` module. As an example the curl command in Listing 6.24 can be executed to compare the Security Capabilities provided by *IpTables* and *XFRM*.

```
curl -i
   "http://localhost:8984/rest?run=two_nsf_comparison.xq&nsf_a=
   IpTables&nsf_b=XFRM" -u admin:admin
```

Listing 6.24: BaseX NSF comparison query request

The BaseX server would respond as shown in Listing 6.25.

```
Common SecurityCapability =
["ConnmarkConditionCapability",
   "DeviceDestinationConditionCapability",
   "IcmpConditionCapability", "Icmp6ConditionCapability",
   "MobilityHeaderConditionCapability",
   "PolicyDirConditionCapability",
   "PolicyReqidConditionCapability",
   "PolicySpiConditionCapability", "AcceptActionCapability",
   "RejectActionCapability", "SourcePortConditionCapability",
   "DestinationPortConditionCapability",
   "IpProtocolTypeConditionCapability",
   "IpSourceAddressConditionCapability",
   "IpDestinationAddressConditionCapability"]
```

Listing 6.25: BaseX server query response

The external refinement tool would request to know which NSF provides certain Security Capabilities, in Listing 6.26 is shown the curl command to do so.

```
curl -i
   "http://localhost:8984/rest?run=capa_set_search.xq&capa_set=
   SourcePortConditionCapability,
   IpDestinationAddressConditionCapability" -u admin:admin
```

Listing 6.26: BaseX NSF search query request

The BaseX server would respond as shown in Listing 6.27.

```
NSF genericPacketFilter supports SourcePortConditionCapability,
   IpDestinationAddressConditionCapability.
NSF XFRM supports SourcePortConditionCapability,
   IpDestinationAddressConditionCapability
NSF StrongSwan supports SourcePortConditionCapability,
   IpDestinationAddressConditionCapability.
NSF IpTables supports SourcePortConditionCapability,
   IpDestinationAddressConditionCapability.
```

Listing 6.27: BaseX server query response

6.6 Using the framework as web server

Since the thesis framework is developed in Java, it can be obviously executed as JAR files. Hence, using JAR files is not the best and most dynamic provisioning method for eventual external entities. For this reason, a Flask-based web server is proposed to interact with the thesis work seamlessly.

In particular, the web server is developed inside a Docker container, and it can execute JAR files underneath thanks to the Python back end. The web server offers a GUI as shown in Figure 6.1.

Figure 6.1: Web server home page

Through the web server it is possible to navigate across Converter, Language Generator, or NSF Translator pages. On each page, as shown in Figure 6.2, it is possible to load the required files in order for the tools to work correctly or, if already existent, the tool execution can be directly requested.

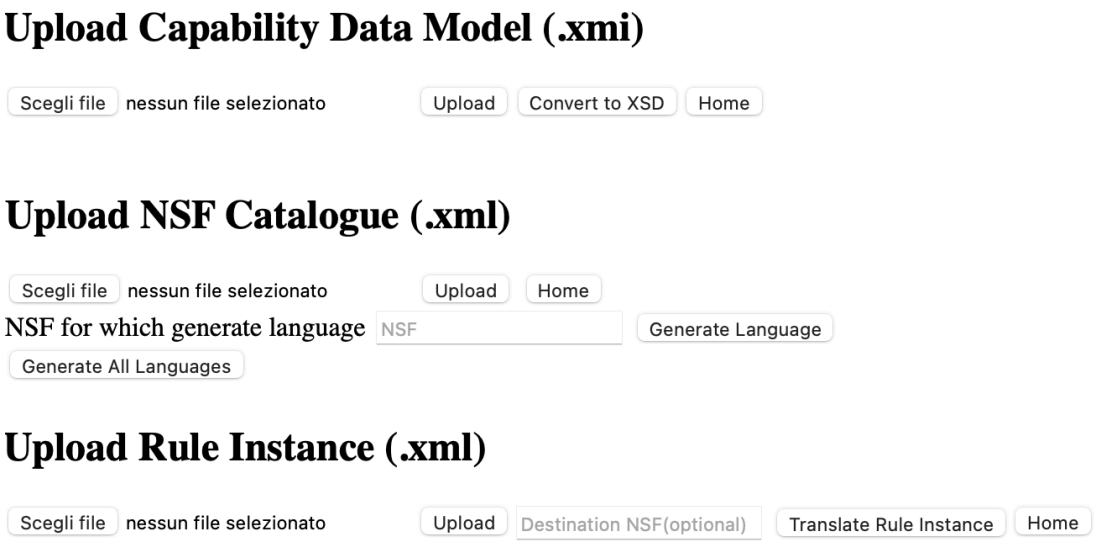

Figure 6.2: Web server tools pages

It has been verified that it is possible to use the framework components if the proper files are uploaded and the correct execution sequence is respected. The produced low-level policies satisfy the requirements. In general, it can be said that the web server acts as a proxy layer toward the low-level execution of JAR files. Hence, the tool's functioning is the same as described up to this point.

The GUI is well-suited for a human user. If an automated tool would require to use it, it is possible to do so using Python `requests` module as shown in Listing 6.28.

```
import requests

files = {'file':open('path/to/model.xmi','rb')}
r = requests.post('http://127.0.0.1:8080/converter',
   files=files, data={'upload':'Upload'})

if('File uploaded correctly' in r.text):
    r = requests.post('http://127.0.0.1:8080/converter',
       data={'generate':'Generate'})
```

Listing 6.28: Web server usage through Python request

In the proposed example, requests for the Converter web page are shown: one for uploading the XMI file containing the CapIM and CapDM and one for requesting the web server to execute the XMI to XSD converter. Python requests have been proven to work for possible usage in combination with an external automated framework.

Chapter 7

Conclusions

The proposed thesis work has been inspired by the I2NSF working group and the thesis of previous students. The main objective has been to develop further an abstraction layer that allows a general and well-defined description of the Security Capabilities that an NSF can offer. These referenced work laid the groundwork to obtain that model, and the current thesis work has expanded its functionality.

Using Model-Driven Engineering and UML Modelling features, it has been possible to add new functionalities to the abstraction model. NSF can be defined with the addition of NSF-related details that are not required anymore to be provided by the final user of the proposed tool. Thanks to an additional model generalization, Security Capability attribute values can be provided using standard and always the same syntax, resulting in a more straightforward way to use the abstract language for each available NSF. New capabilities have been added with respect to the previous student's thesis work, such as the support for Resolution Strategy that allows specifying in which order the security rules have to be translated. Default Action support is now available with the ability to define which set of Security Capability Actions have to be performed when no security rule conditions are satisfied. NSF comparison methods have been developed, making the knowledge process about NSF details straightforward and widely available, employing a dedicated web service. A new kind of generic NSF has been introduced, too. This makes it possible to use an NSF related to a specific interest area without the worry to select during the security rule design an actual NSF to implement it. Additionally, it has been shown how straightforward it is to add a new NSF with the correspondent Security Capabilities thanks to the formal way the proposed tool deals with NSF implementation details. Finally, the same tools are now available as a web service, overcoming the limitations since specific configuration knowledge was needed to deploy these tools.

The results obtained from the thesis development have satisfied all the proposed requirements. The tools are now more usable, and more or less expert users can exploit their functioning. During the thesis work, several problems occurred, such as the need to introduce new UML Classes and UML types for modeling the supported operators or the Default Action and Resolution Strategy functionalities, the decision to maintain backward compatibility with the previous thesis work functionalities, and the requirement to modify existing UML classes in order to support capabilities like NSF-related details or new attributes to be used in abstract security policies and rules.

The proposed thesis work has concentrated on the main functionalities for IpT-

ables NSF and, in a limited way, XFRM NSF. This is because of the linear way with which IpTables allows defining security-related aspects such as Security Capability values employing range or unions operators or to specify Default Action to apply when no security rule conditions are satisfied. Even if there are functionalities (i.e. operators) that can be easily applied to all the Security Capabilities of all the available NSF; future works can focus on adapting all the proposed functionalities to better reflect the syntax-related aspects of each specific NSF.

The reason for defining an abstraction layer to describe NSFs and their Security Capability has been widely discussed during this thesis dissertation: a way to conveniently describe security requirements and policies without worrying about the specific low-level syntax of the device that implements them. The development of this thesis solution is not only for non-expert users that would like to configure their network environments following the main security fundamentals but also for automated tools that have human-like ability to decide which security capabilities are needed for specific network topology. The proposed framework has been proved to work correctly coupled with an external security refinement engine. This can be observed in Mattia Bencivenga's thesis work [18] in which an automated refinement tool has been developed. Mattia's framework's main job is to receive high-level security requirements and select which Security Capabilities are needed and on which node those have to be deployed based on the available network topology. This automated refinement tool then leverages the NSF comparison web service to discover which NSF offers which Security Capability. Finally, it generates the required security policies for each selected node using the proposed abstract language developed in the current thesis work. In this way, it can be observed how the proposed thesis work can be embedded in a wider scenario in which the end-user is a non-expert one. The end-user expresses its security necessities using a simple file format coupled with a GUI-based approach, both proposed by Mattia's work. And then, the automated refinement tool with the proposed abstraction layer performs the needed reasoning to output a low-level security policy that can be deployed on the desired network topology. It is worth noticing that the framework has been successfully deployed in the context of the *FISHY* research project[1] funded by the European Union's Horizon 2020 research programme.

In conclusion, this thesis work has allowed me to dive deeply into software modeling and system security design. These months have been beneficial for discovering a computer science area in which I was not an expert. I would like to express how exciting it is to see my own work be used in a broad application context that can ease a person's everyday life, even in a specific area.

[1] https://fishy-project.eu

Appendix A

User manual

This chapter displays how to use each proposed tool as JAR files. The same tools can also be used employing a web server as described in Section 6.6.

A.1 XMI to XSD generator

This tool converts an XMI file to an XSD one based on the features that Modelio uses to describe XMI classes. This tool is used in the thesis architecture to facilitate the creation of an XSD file containing the constraints about how Security Capability and other classes have to be specified during the framework usage. The following syntax can be used to run the tool from the command line:

```
java -jar newConverter.jar input_path [output_path]
```

Each parameter role is:

- `input_path`: path of the XMI file to be converted.
- `output_path`: path of the output file that the tool will generate. The output file is produced in the current working directory with a standard file name if this argument is not specified.

The command's execution may or may not succeed. If the execution is successful and `output_path` is present, an XSD file `capability data model.xsd` is generated in the position specified by the provided path. `output_path` parameter is also used to fill the values of `xmlns` and `targetNamespace` XSD file attributes. The successful outcome is confirmed by a screen message.

If the execution is unsuccessful, the tool displays a response that includes information about the kind of problem.

A.2 Abstract language generator

This tool generates an XSD file that represents the selected NSF abstract language. The tool functioning is based on an XML file describing the NSF Catalogue: it contains all the available NSFs and the Security Capabilities that can be assigned to

them. This tool has to be used every time a new NSF is created or NSF's Security Capabilities are added, removed, or modified.

The abstract language generator can work with files that state metadata. Those files have to be stored in a folder named `metadata` located in the tool working directory. The following syntax can be used to run the tool from the command line:

```
java -jar newLanguage.jar xsd_model_path nsfCatalogue_path
   nsfName [output_path]
```

Each parameter role is:

- `xsd_model_path`: path of the XSD file converted by the previous tool, this file contains the description of how Security Capability classes have to be defined in the NSF Catalogue file.
- `nsfCatalogue_path`: path of the NSF Catalogue, this file contains the list of available NSF, the Security Capability they offer and other system classes such as Capability Translation Details or Language Generation Details.
- `nsfName`: name of the NSF for which the abstract language has to be generated.
- `output_path`: path of the the output file that the tool will generate. The output file is produced in the current working directory with a standard file name if this argument is not specified.

The command's execution may or may not succeed. If the execution is successful, an XSD file with the name `language_nsfName.xsd` is created, this file includes the NSF's language, and it is stored in the position specified by the provided `output_path`. The successful outcome is confirmed by a screen message. The XML file stated by `nsfCatalogue_path` is used to collect information from any `languageGenerationDetails` classes that allows specializing the NSF abstract language during the language generation itself.

This tool may also generate Metadata files. This occurs when a new enumeration is created, or an old one is updated using `languageGenerationDetails` class. A new Metadata file with the name of the referenced NSF is created for this purpose. These files could store the coupling between the numerical and alphabetic values of specific Security Capability values. Any metadata files are stored in the tool working directory as the abstract language XSD file.

If the execution is unsuccessful, the tool displays a response that includes information about the kind of problem.

A.3 NSF low-level language translator

Given a security policy stated using the NSF abstract language, this tool can translate the before-mentioned policy into a low-level one using the actual NSF syntax. The output file is generated based on the Security Capabilities used in the abstract security policy. The following syntax can be used to run the tool from the command line:

```
java -jar newTranslator.jar xsd_language_path nsfCatalogue_path
   nsfRule_path [output_path] [parameter5] [parameter6]
   [parameter7] [parameter8] [parameter9] [parameter10]
```

Each parameter role is:

- `xsd_language_path`: path of the XSD file containing the abstract language description for the selected NSF (e.g. the XSD file obtained from the previous tool).
- `nsfCatalogue_path`: path of the XMI file containing the NSF Catalogue, this file contains the list of available NSF, the Security Capability they offer, and other system classes such as Capability Translation Details or Language Generation Details.
- `nsfRule_path`: path of the XML file containing the security policy stated using NSF abstract language.
- `output_path`: path of the the output file that the tool will generate. The output file is produced in the current working directory with a standard file name if this argument is not specified.
- `parameter5`: optional parameter that allows the user to provide a static alphanumeric string that will be added to at the beginning of each output rule; this parameter must begin with the sequence of characters "+s". Its functioning can be replaced by NSF Policy Details implementation.
- `parameter6`: optional parameter that allows the user to provide a static alphanumeric string that will be added to at the end of each output rule; this parameter must begin with the sequence of characters "+e". Its functioning can be replaced by NSF Policy Details implementation.
- `parameter7`: optional parameter to force the generation of valid rules even if some policy rules do not respect specific dependency criteria, it is expressed as "-f".
- `parameter8`: optional parameter that allows the user to specify a static alphanumeric string to be placed just after the first string of each rule, this parameter must begin with the sequence of characters "+crs".
- `parameter9`: optional parameter that allows the user to specify a static alphanumeric string to be placed just before the last string of each rule, this parameter must begin with the sequence of characters "+cre".
- `parameter10`: optional parameter that allows the user to state which NSF to choose when the abstract input policy regards a generic type of NSF (e.g. genericPacketFilterNSF).

The first three parameters must be given in the given order, while the optional parameters can be entered in any order.

The command's execution may or may not succeed. If the execution is successful, a textual file named `policy_nsfName.txt` containing the translated low-level policy

for the given NSF is produced. The low-level policies are obtained from the abstract language ones. If specified, the file is produced at the location specified by `output_path`. Otherwise, the file is saved in the tool working directory.

If the execution is unsuccessful, the tool displays a response that includes information about the kind of problem.

A.4 Framework web server

The proposed web server provides access to various tools oriented to Abstract Language generation and Low-Level Policy Translation. From the web server home page, it is possible to clear folders reserved for each tool. Also, it is possible to navigate across the offered tools.

The web server provides access to the following web pages:

- *Converter*: page that makes the user upload a Security Capability Model in XMI format that will be converted in XSD format.
- *Language Generator*: page that makes the user upload an NSFCatalogue storing the available NSFs and the owned Security Capabilities. It allows the creation of the Abstract Language for a selected NSF.
- *Translator*: page that makes the user upload a Rule Instance expressed using NSF Abstract Language, and that translates the Rule Instance items into Low-Level Policies, based on NSF low-level language.

In order to run the proposed framework, Docker is required. The following steps are needed to set up the web server:

- Clone the Git repository:

```
git clone git@github.com:torsec/security-capability-model.git
```

- Move into project folder:

```
cd security-capability-model
```

- Move into server folder:

```
cd enforcer
```

- Build the Docker image from `Dockerfile`:

```
docker build -t enforcer .
```

- Run the Docker container containing the proposed enforcer:

```
docker run -p 8080:5000 enforcer
```

At this point, the web server is listening on `http://localhost:8080`.

It is possible to use the Web Server directly from its GUI using the Security Capability Model files stored in `src/` directory.

Other than the GUI-based usage, three Python test scripts are available under `/enforcerTests` directory. They perform the required Python `requests` that embody the steps with which it is possible to interact with the web server GUI. In order to execute the Python scripts, the above Docker container has to be running. From `enforcerTests/` folder:

- Create the Python venv:

```
python3 -m venv ./venv
```

- Activate the Python venv:

```
source venv/bin/activate
```

- Install Python requirements:

```
pip install -r requirements.txt
```

- Upload the CapIM and CapDM from Modelio in XMI format and convert it in XSD format:

```
python testConverter.py
```

- Upload the most recent NSF Catalogue and generate abstract language for `nsfName`. If `nsfName` is omitted, the enforcer will generate the Abstract Language for all the available NSFs.

```
python testLanguage.py [nsfName]
```

- Upload a desired Rule Instance providing its path and to translate it to low level policies. For generic NSFs it is possible to set the `destinationNSF` toward which the translation has to be performed.

```
python testTranslator.py rule/instance/path [destinationNSF]
```

A.5 NSF Catalogue web server

In order to run the proposed NSF Catalogue Database web server, Docker is required. The following steps are needed to set up the web server:

- Clone the Git repository:

```
git clone git@github.com:torsec/NSF-Catalogue.git
```

- Move into project folder:

```
cd NSF-Catalogue
```

- Build the Docker image from `Dockerfile`:

```
docker build -t register_and_planner .
```

- Run the Docker container containing the proposed NSF Catalogue Database:

```
docker run -p 8984:8984 register_and_planner
```

At this point, the NSF Catalogue web server is listening on `http://localhost:8984`.

Examples of curl commands to send requests to the web server, can be found in Section 6.5.

Appendix B

Developer manual

Code details and tool implementation are described in this chapter. System logical flow and software decisions will be taken care of, from adding new Security Capabilities to the CapDM to improving software implementation. Tool's components are developed using Java programming languages. Java for XML Processing APIs has been used as well.

B.1 Update the Capability Data Model

As already described, CapIM and CapDM are the main UML models designed to describe Security Capability and NSFs properly. Those UML models are defined using Modelio modeling software.

CapDM, in particular, is related to the characterization of Security Capabilities and their area of application. As already done by previous thesis works, Security Capability classes inherit from the super-class Security Capability. Then they are arranged based on the 6-tuple defined in Figure 5.2. Each group defined by the 6-tuple classification can be split into sub-groups that describe the area of interest where the Security Capability operates.

Using UML itself (Figure B.1) to describe the relationship among CapDM elements, it is possible to consider that *CapDM* itself is composed by *Security Capabilities*. Security Capability class acts as super-class for each of the six *tuples*: Action, Condition, Event, Resolution Strategy, Evaluation, and Default Action. Each defined tuple acts as a super-class for the *area of interest* for which Security Capabilities can be sorted. Finally, the actual Security Capability (named *Leaf Capability*) can be considered as inherited from the area of interest. Security Capability class can refer to *custom data types* that can be defined in the correspondent class diagram.

As a consequence of the relationships described above, it is possible to add a new Security Capability following the proposed steps:

- Identify which *tuple* the Security Capability belongs to.
- Identify the class diagram used for that tuple.
- Identify in which *area of interest* the Security Capability could be most suited.
- Identify the class diagram used for that area of interest.

- Create a new class for the Security Capability in the identified area of interest class diagram.
- Define any details about the attributes needed to manipulate the new Security Capability class.

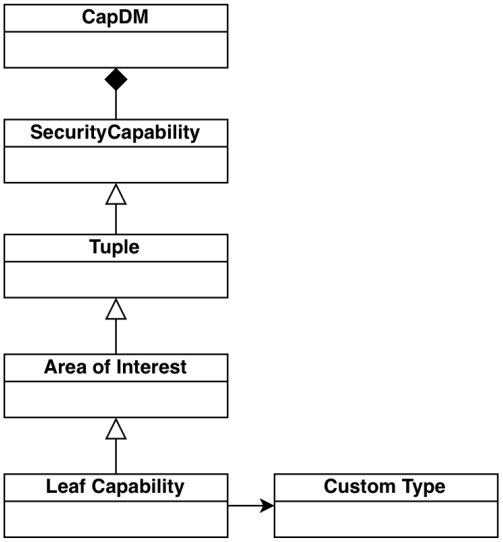

Figure B.1: CapDM UML design

If custom data types are required for the newly added Leaf Capability, the following steps can be performed:

- Identify how to compose the new type.
- Create a new class in the type class diagram.
- Define the new class with a generic name.
- Define the details of the new attributes needed for the Leaf Capability.
- Assign the type of the class just defined to attribute type of the Leaf Capability just created.

B.2 XMI to XSD generator

The XMI to XSD generator tool has the job to convert into an XSD file the XMI classes exported from Modelio. In this way, it is possible to specify how Security Capability classes should be defined in the NSF Catalogue file. The XMI to XSD generator workflow is proposed in Figure B.2.

The tool behaves based on the following steps:

- Read the input XMI file.
- Create an XSD complexType based on if the XMI class represents a UML Class or a UML Enumeration.

- For each UML Class attribute: find the type, set Min and Max occurrences values, and set Default Values. Then, add it to the previously created complexType for UML Class.
- For each UML Enumeration literal: add it to the previously created complexType for UML Enumeration.
- Generate the output XSD file.

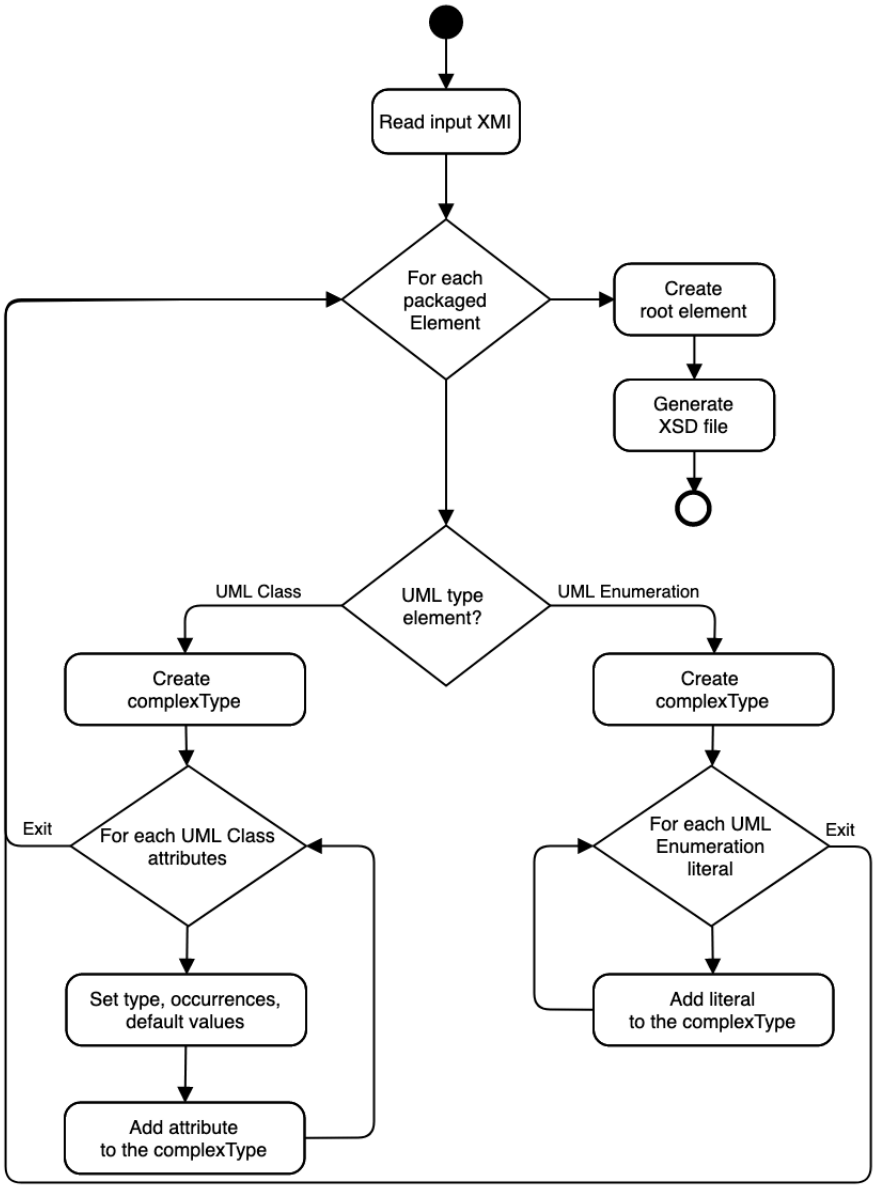

Figure B.2: XMI to XSD generator workflow

B.2.1 Tool architecture

The proposed tool architecture is composed of two main classes, represented in Figure B.3, and a support class for the creation of objects to be included in the document.

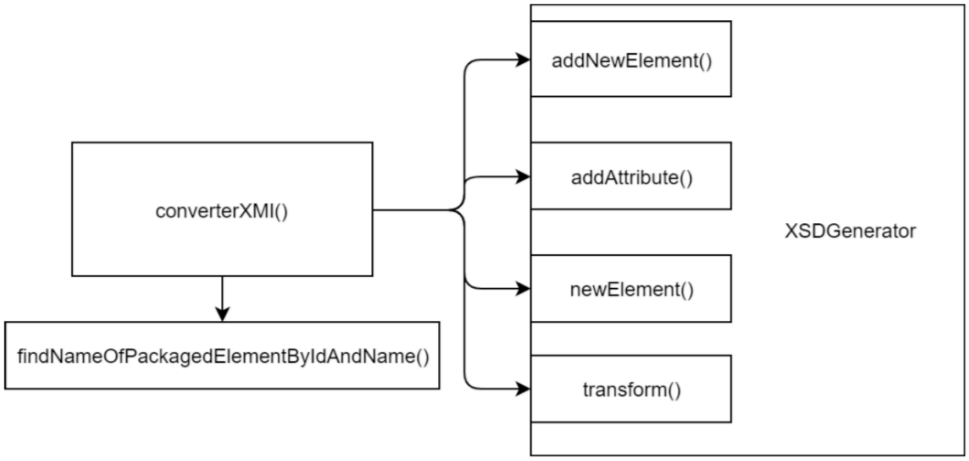

Figure B.3: XMI to XSD generator architecture

`Converter` class operates on the input file and recognizes the XMI elements that have to be used to generate elements in the output XSD file. This class offers the following methods:

- `public Converter()`:
  constructor method that allows instantiating an object corresponding to the Converter class. It takes care of reading the input file and recognizing the XMI elements that have to be used to generate the output XSD file.

- `private String findNameOfPackagedElementByIdandType(NodeList nl, String id, String class)`:
  this method's job is to find the name of an XMI element starting from the element ID assigned by the Modelio Export Tool. The XMI element is searched among the list of XMI nodes passed as a parameter. If the search is not successful, "null" is returned.

`XSDgenerator` class has the responsibility to generate XSD elements for the output XSD file. Each XSD element is appended to the root of the output file. Then it writes the output file itself. This class defines the following instance variables:

- `private final static String NS PREFIX = "xs:"`:
  variable that stores all the XSD elements with which prefix will be generated.

- `private Document doc`:
  variable that stores the output XSD document to which new XSD elements are appended.

- `private Element schemaRoot`:
  variable that stores the root XSD element of the output XSD file.

- `private NameTypeElementMaker elMaker`:
  variable that stores XML Javax support tool in charge of generating new XSD elements based on input XMI ones.

`XSDgenerator` class defines the following methods:

- `public XSDgenerator()`:
  constructor method that allows instantiating an object corresponding to the XSDgenerator class responsible for creating XSD elements for the output file.
- `public boolean transform(String outputName)`:
  method that is responsible of actually writing the output XSD file, setting up the proper file name and XSD root node properties.
- `public Element newElement(String name)`:
  method that generates a new XSD element, the `name` parameter is used to give the name to the element itself.
- `public void addAttribute(Element element, String nameAttr, String attrValue)`:
  method that is responsible for adding to the XSD element `element` an attribute named `nameAttr` with the value `attrValue`.
- `public void addNewElement(Element e)`:
  method that appends the XSD element `e` to the root of the output XSD file to be generated.

`NameTypeElementMaker` support class for the XSDgenerator provides the required methods for managing XSD elements. This class defines the following instance variables:

- `private String nsPrefix`:
  variable that stores the XSD prefix that has to be used for the generated XSD elements.
- `private final static String NS PREFIX = "xs:"`:
  variable that stores all the XSD elements with which prefix will be generated.
- `private Document doc`:
  variable that stores the output XSD document to which new XSD elements are appended.

`NameTypeElementMaker` class defines the following methods:

- `public NameTypeElementMaker(String nsPrefix, Document doc)`:
  constructor method that allows instantiating an object corresponding to the `NameTypeElementMaker` class while providing the preferred XSD prefix and the target output document.
- `public Element createElement(String elementName)`:
  method responsible for creating the XSD element, appending it to the proper XSD document with the proper XSD prefix.
- `public void setAttribute(Element element, String nameAttr, String attrValue)`:
  method that set to XSD `element` the attribute `nameAttr` with the value `attrValue`.

B.3 Abstract language generator

This tool is used to generate a given NSF Abstract language. In particular, it creates an output XSD file that describes how Security Capabilities have to be used within abstract language policies. All of this is done based on the NSF Catalogue XMI file. Thanks to the languageGenerationDetails class, it is possible to specify custom data types if a given Security Capability needs it. NSF Catalogue also stores the information about which Security Capability a given NSF owns. Hence, every time a Security Capability is added, removed, or updated, the Abstract language generator tool must be performed.

This tool can also take advantage of a *Metadata* file in which it is possible to provide further details about how to configure a custom data type. This file must be stored in the same folder of the tool executable. More in detail, the metadata file can store the coupling between string values and their correspondent integer ones. This kind of mapping can be a handful when managing TCP ports or IP protocol types since both properties can be stated by a string or by an integer. The coupling is expressed in the following format: each row states a coupling, first comes the integer value, then there is a space, and then the correspondent string value. The abstract language generator workflow is proposed in Figure B.4.

The tool behaves based on the following steps:

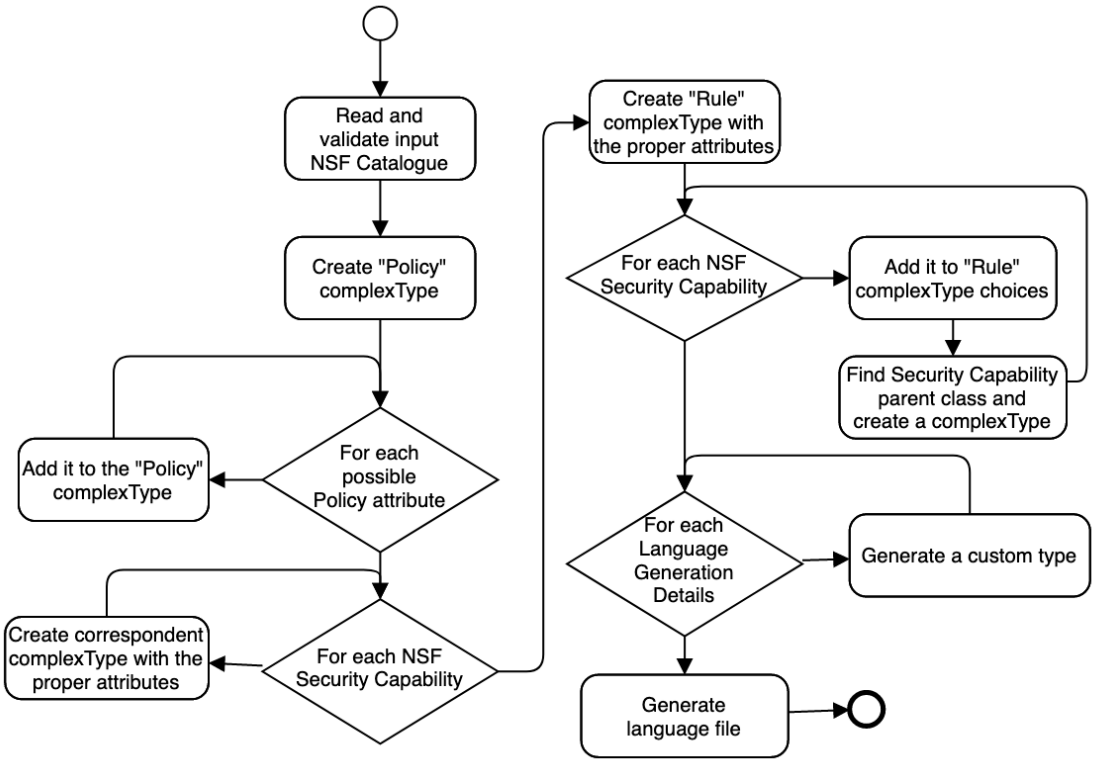

Figure B.4: Abstract Language Generator workflow

- Read and validate the input NSF Catalogue XMI file.
- Create a complexType `Policy` that describes an XMI node that will contain abstract Rule nodes.

- For each possible Policy attributes: add it to the Policy complexType. Policy attributes will be used to store further abstract policy details.
- For each NSF Security Capability: create a complexType with the proper attributes. In this way, XSD specifies how to compose Security Capability XMI nodes in the abstract language policy.
- Create a complexType `Rule` that represents abstract security requirements.
- For each NSF Security Capability: add it as `choice` of the Rule complexType. In this way, it is said that Rule XMI elements can have Security Capability XMI elements as inner children. Then find the Security Capability parent class and generate a complexType for it. The last step is needed for making the XSD engine to understand the type of each Security Capability complexType.
- For each languageGenerationDetails class: generate the custom type described in the languageGenerationDetails itself.
- Generate the output abstract language XSD file.

B.3.1 Tool architecture

The architecture of the abstract language generator can be seen as a solution to two main problems: the generation of the abstract language based on the Security Capabilities owned by the target NSF and the generation of the custom types expressed by languageGenerationDetails classes. As a consequence, two main architectures allow solving these aspects. These are shown in Figure B.5 and Figure B.6. For both parts, generateLanguage() method in `LanguageModelGenerator` class acts as a starting point from which all the required methods are executed. Additionally, `XSDGenerator` and `NameTypeElementMaker` support classes are used in order to generate the output XSD file with the desired structure.

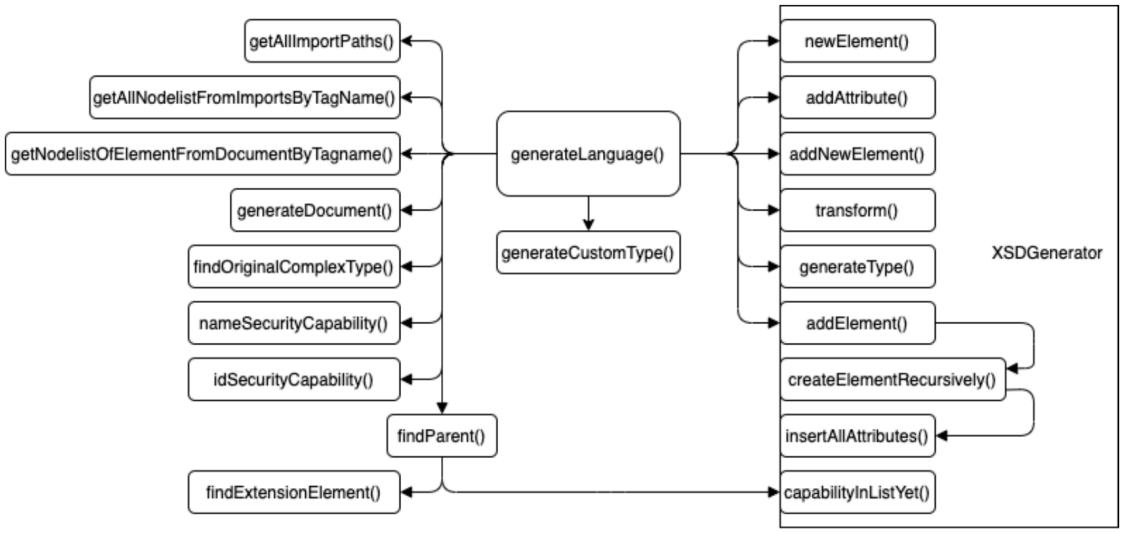

Figure B.5: Abstract Language Generator architecture first part

`LanguageModelGenerator` class methods related to reading the input files and creating the abstract language are displayed in Figure B.5. In particular, the following variables are defined:

- `private String xsd`:
  variable that stores the Security Capability Model XSD file path.
- `private String xml`:
  variable that stores the path of the NSF Catalogue XMI file containing the NSF Security Capability list and the related `languageGenerationDetails` and `capabilityTranslationDetails` classes.
- `private String outputName`:
  variable that stores the folder path where the output XSD file has to be stored. It can also store the desired output file name.
- `private List<String> imports`:
  variable that stores the list of Security Capability Model XSD files. In this way, it is possible to extend the tool's usage to a list of Security Capability Models.
- `private List<NodeList> complexTypeNodeLists`:
  variable that stores the list of the complexTypes retrieved from the `imports` XSD files.
- `private List<NodeList> simpleTypeNodeLists`:
  variable that stores the list of the simpleTypes retrieved from the `imports` XSD files.
- `private XSDgenerator gen`:
  variable that stores the instance object of `XSDGenerator` class, explained later.
- `private List<String> metadataPath`:
  variable that stores the list of paths of support metadata file.

`LanguageModelGenerator` class defines the following methods:

- `public LanguageModelGenerator(String xsd, String xml, String outputName)`:
  constructor method that allows instantiating an object corresponding to the `LanguageModelGenerator` class.
- `public boolean transform(String outputName)`:
  method that is responsible of actually writing the output XSD file, setting up the proper file name and XSD root node properties.
- `public boolean generateLanguage()`:
  method that generates the abstract language. It starts from creating the `Policy` and `Rule` XSD elements which are expressed in a way that the Policy XMI node can contain Rule inner children. Each `rule` complexType is composed of Security Capability XSD nodes, expressing that Rule XMI node can store several Security Capabilities XMI children.

- `private List<String> getAllImportPaths()`:
  method that retrieves a list of paths stated by "xs:import" attribute in the Security Capability Model XSD files. This method is needed if the Security Capability Model is split into multiple XSD files.

- `private List<NodeList> getAllNodelistFromImportsByTagName(List<String> imports, String tag)`:
  method that retrieve from the `imports` files the XSD or XMI elements with the name `tag` and collects them in a List of NodeLists items.

- `private static NodeList getNodelistOfElementFromDocumentByTagname(Document d, String tagname)`:
  method that retrieve from a single Document `d` the XSD or XMI elements with the name `tag` and collects them in a NodeList.

- `private static Document generateDocument(String path)`:
  method that generates a Document type based on the input path.

- `private static Element findOriginalComplexType(NodeList complextype, String capa)`:
  method that retrieves a complexType element from a NodeList, based on its name provided in `capa` parameter.

- `private Element findParent(Element complextype)`:
  method that recursively search the NodeList of complexTypes and retrieve the parent complexType of the `complexType` provided element. At each iteration, the current element is added to the elements necessary to use for the language, any elements already present are not further inserted.

- `private Element findExtensionElement(Element e)`:
  method that search `extension` attribute of the provided Element `e`.

- `private String nameSecurityCapability(Element item)`:
  method that extract the name of the provided Element `item`.

- `private String idSecurityCapability(Element item)`:
  method that extract the id of the provided Element `item`.

`LanguageModelGenerator` class methods related the generation of custom types based on `languageGenerationDetails` classes are displayed in Figure B.6. In particular, the following methods are defined:

- `private boolean generateCustomType(Element element, NodeList n)`:
  method that allows generating a custom type for the selected NSF. This method is executed if `languageGenerationDetails` classes are recognized in NSF Catalogue input file.

- `private boolean generateCustomizedEnumeration(String enumerationName, Element modifyDefaultEnumeration, NodeList modeledNL)`:
  method that is responsible generating a custom enumeration type. It receives the name of the enumeration to modify and the NodeList from which retrieve the original enumeration. The `modifyDefaultEnumeration` is the

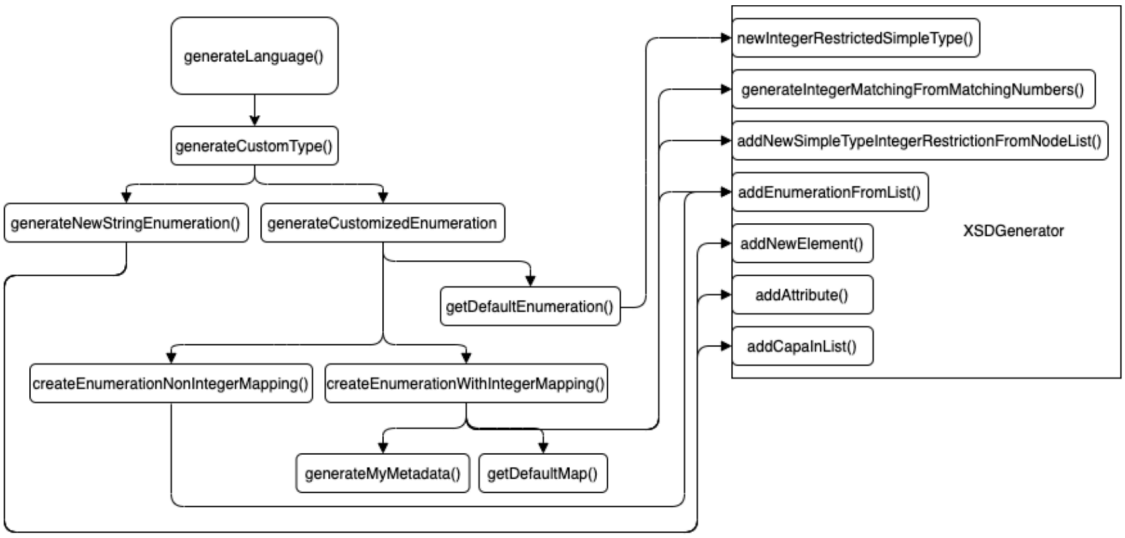

Figure B.6: Abstract Language Generator architecture second part

`languageGenerationDetails` class which stores how to modify it. Also in this method metadata files are taken in consideration, if used.

- `private boolean createEnumerationWithIntegerMapping(...)`:
  method that generates an enumeration containing a mapping between an integer value and a string.

- `private void generateMyMetadata(Map<String, Integer> nameValueMap, String metadata)`:
  method that generates a metadata file containing the integer-string couplings used during the abstract language generation.

- `private Map<String, Integer> getDefaultMap(String metadata)`:
  method that extract a Map object starting from an input specifically formatted `metadata` file.

- `private boolean createEnumerationWithIntegerMapping(...)`:
  method that generates an enumeration containing a mapping between an integer value and a string.

- `private boolean createEnumerationNonIntegerMapping(...)`:
  method that generates an enumeration that does not contain a mapping between an integer value and a string.

- `private Element getDefaultEnumeration(String name, NodeList modeledNL)`:
  method that retrieve an enumeration from the XSD simpleTypes given its name.

- `private boolean generateNewStringEnumeration(String name, NodeList valueNL)`:
  method that generates a simple enumeration named `name` with the literals stored in `valueNL` NodeList item.

`XSDgenerator` support class is responsible for managing the generation of XSD output elements. This class defines the following variables:

- `private final static String NS PREFIX = "xs:"`:
  variable that stores the XSD prefix of the elements generated.
- `private Document doc`:
  variable that stores the document instance representing the output XSD file.
- `private Element schemaRoot`:
  variable that stores XSD root element of the output file.
- `private NameTypeElementMaker elMaker`:
  variable that stores the instance responsible of generate new XSD elements.
- `private TreeSet<String> capability`:
  variable that stores the list of Security Capability names encountered during abstract language generation.
- `private List<String> type`:
  variable that stores the list of the types encountered during abstract language generation.

`XSDgenerator` class defines the following methods:

- `public XSDgenerator()`:
  constructor method that allows instantiating an object corresponding to the `XSDgenerator` class.
- `public boolean transform(String outputName)`:
  method that is responsible of actually writing the output XSD file, setting up the proper file name and XSD root node properties.
- `public Element newElement(String name)`:
  method that generates a new XSD element, the `name` parameter is used to give the name to the element itself.
- `public void addAttribute(Element element, String nameAttr, String attrValue)`:
  method that is responsible for adding to the XSD element `element` an attribute named `nameAttr` with the value `attrValue`.
- `public void addNewElement(Element e)`:
  method that appends the XSD element `e` to the root of the output XSD file to be generated.
- `public void addElement(Element e)`:
  method that is responsible for appending an XSD element `e` to the root of the output XSD file.
- `private Element createElementRecursively(Element e)`:
  method that generates a copy of the provided XSD element taking care of iterating recursively the element in order to retrieve all the element children nodes and attributes.

- `private void insertAllAttributes(Element fromElement, Element toElement)`:
  method that is responsible for adding attributes of `fromElement` XSD node to the `toElement` XSD node.

- `public void addCapaInList(String s)`:
  method that appends the encountered Security Capability name in a list of string. This strings are used in order to make sure to add to the abstract language all the Security Capability referenced by the NSF Security Capabilities.

- `public boolean capabilityInListYet(String s)`:
  method that checks if a given Security Capability has been already added to the previous list of strings.

- `public void generateType(NodeList typeComplex)`:
  method that iterates over all the complex and simple types referenced during language generation and generates these types in the output language XSD files.

- `public void addEnumerationFromList(String name, List<String> valueList)`:
  method that creates an enumeration whose name is provided through `name` attribute and its literals are provided through `valueList` list of strings.

- `public void addNewSimpleTypeIntegerRestriction-FromNodeList(List<String> capabilityAndAttributesToBeChanged, NodeList setNumericRangeNodeList)`:
  method that generates a new element based on the provided parameters. `capabilityAndAttributesToBeChanged` store the target Security Capability name and the Security Capability attributes that have to be modified. `set-NumericRangeNodeList` stores the kind of integer restriction to be specified for each provided attribute.

- `private void changeElementTypeInExistingComplexType(List <String> capabilityAndAttributesTo BeChanged, String newType-Name)`:
  method that changes the type of a target Security Capability. Security Capability name is stored as first item of `capabilityAndAttributesToBeChanged`, following items are the attributes to update. The type to which the attributes have to be changed is provided through `newTypeName`.

- `public void generateIntegerMatchingFromMatchingNumbers(List <String> capabilityAndAttributesToBeChanged, Map<String, Integer> nameValueMapSortedByIntegerValue)`:
  method that generates an integer restriction for the attributes of the Security Capability whose name is stored as first string of `capabilityAndAttributesToBeChanged`.

- `public void newIntegerRestrictedSimpleType(NodeList setNumericRangeNodeList, List<String> capabilityAndAttributesTo-BeChanged)`:

method that generates a new integer restriction using the values contained in the `capabilityAndAttributesToBeChanged`. The first item is the Security Capability name while the following ones are the attributes for which the restriction is needed.

`NameTypeElementMaker` class is a support class for generating elements in a specific document. XSDgenerator class methods leverage this class. The details of this class are already discussed in Appendix B.2.1.

B.4 NSF low-level language translator

The NSF Translator is the one that has been mainly re-designed in order to make the proposed framework to provide the functionalities designed and proposed in Chapter 4. It is worth noting that, to not lose all the good introduction from the previous thesis work, the new framework is backward compatible with the previous syntax with which it was possible to state Security Capability values within the abstract rules.

This tool receives as input the target NSF Rule Instance file that stores the security rules expressed using the previously defined NSF abstract message. The NSF Translator workflow is shown in Figure B.7. The tool behaves based on the following steps:

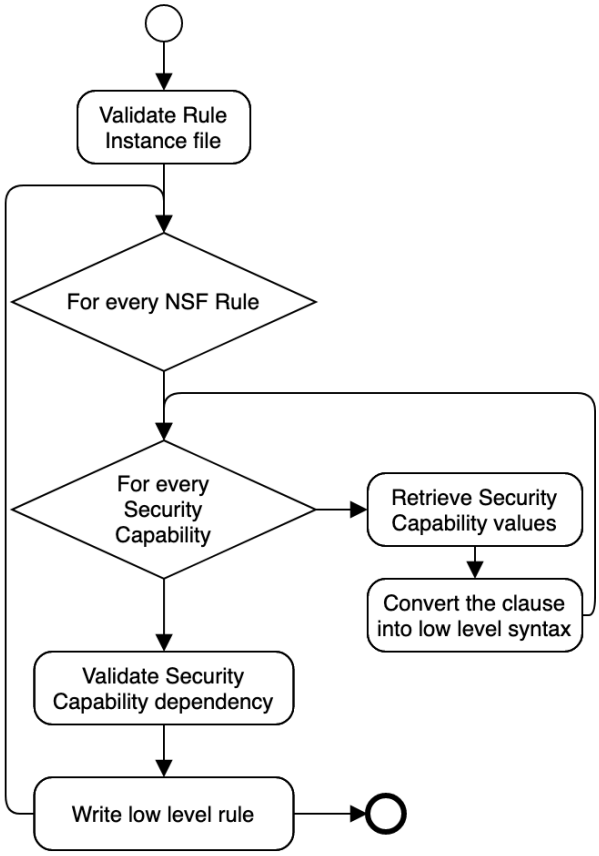

Figure B.7: NSF Translator workflow

- Validate the input Rule Instance XML file against the constraints defined by the NSF Abstract Language XSD file.
- For every NSF rule in the Rule Instance file:
  - Iterate over the Security Capability within the rule.
  - Retrieve the Security Capability value.
  - Convert the Security Capability respecting the NSF low-level syntax using the capabilityTranslationDetail for that Security Capability stored in the NSF Catalogue.
  - Check if the Security Capability dependencies expressed by the NSF Catalogue XMI file are respected.
  - Write the low-level rule.

B.4.1 Tool architecture

The NSF Translator architecture is composed of the NSF Translator class itself, whose architecture is shown in Figure B.8 and Figure B.9. Also, NSFPolicyDetails, BodyConcatenator, and CommandName support classes have been defined.

`NSF Translator` class defines the following variables:

- `private String xsdLanguage`:
  variable that stores the path of the abstract language XSD file.
- `private String xmlRule`:
  variable that stores the rule instance XML file path that stores the security rules expressed in abstract language.
- `private String xmlCatalogue`:
  variable that stores the NSF Catalogue XMI file path that stores the NSF Security Capability details.
- `private String outputName`:
  variable that stores the path to use for the output file storing the translated security rules using NSF low-level syntax.
- `private String temporaryRule`:
  variable that stores the translated low-level rule as the Security Capability values are collected and translated using NSF low-level syntax. This variable is also used to verify the correctness of the Security Capability dependencies.
- `private String temporaryCapabilityAndAttributes`:
  variable that stores the current analysed Security Capability name and the sequence of the attribute and correspondent values expressed for it. These values are retrieved from the Security Capability node in each security rule. For example: “securityCapabilityName attribute1 value1 attribute2 value2”.
- `private String temporaryCapability`:
  variable that stores the current analysed Security Capability name.

- `private List<String> temporaryListCapabilityOfRule`:
  variable that stores the list of Security Capability names encountered during the translation of the current rule.

- `private NodeList translationNodes`:
  variable that stores the NodeList of capabilityTranslationDetails element for each NSF Security Capability.

- `private String nsfName`:
  variable that stores the name of the NSF for which the translation is performing. External parameters can specify this variable when a Generic NSF Rule Instance has to be translated.

- `private String nextCapabilityTemp`:
  variable that stores the name of the subsequent Security Capability. This is used to verify if the Security Capabilities are provided in a proper order, stated in capabilityTranslationDetails dependencies for a given Security Capability.

- `private NodeList nodes`:
  variable that stores a NodeList containing all the Rule nodes stored in the input Rule Instance XML file.

- `private Element myCapabilityTranslation`:
  variable that stores a capabilityTranslationDetails class related to the current Security Capability analysed.

The first set of methods shown in Figure B.8 and defined by `NSF Translator` class are:

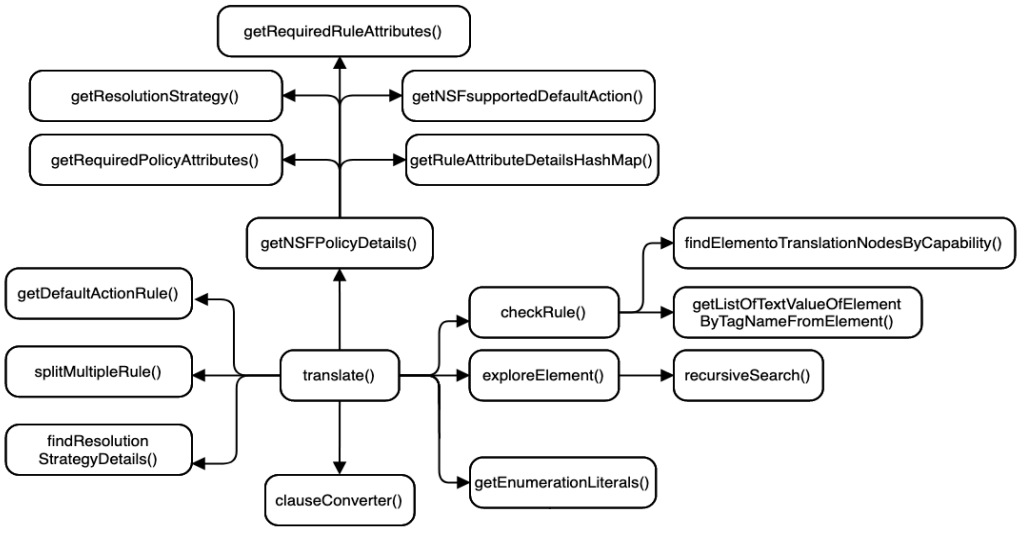

Figure B.8: NSF Translator architecture, first part

- `public void translate(String xsd, String xmlCatalogue, String xmlRule, String outputName, String startString, String endString, String forced, boolean scr, boolean ecr, String destinationNSF)`:
  method that instantiates the execution of the `NSF Translator` operations. This method uses `capabilityTranslationDetails` class to understand how to translate each security policy that is found in the Rule Instance XML file. After each rule translation, it checks the correctness of the rule based on Security Capabilities dependencies. This method takes care of the newly introduced: NSF-related details, Default Action Capabilities, Resolution Strategy approaches, and operator support.

- `private NSFPolicyDetail getNSFPolicyDetail(NodeList nsfList)`:
  method that retrieve NSF-related details stored in the NSF XML item stored in the NSF Catalogue XML file. The retrieved data is returned in a `NSFPolicyDetail` class.

- `private List<String> getRequiredPolicyAttributes(Element e)`:
  method that retrieves eventual Policy Attributes that the NSF requires for proper low-level rules translation. It returns a list of string in which those attributes are described.

- `private List<String> getRequiredRuleAttributes(Element e)`:
  method that retrieves eventual rule Attributes that the NSF requires for proper low-level rules translation. It returns a list of string in which those attributes are described.

- `private List<String> getNSFsupportedDefaultAction(Element e)`:
  (unused) method for previous implementation of Default Action Capability.

- `private HashMap<String, HashMap> getRuleAttributeDetailsHashMap`:
  method that retrieves a set of string couples for each possible Rule attribute. This functionality is required to manage different rule types for XFRM NSF. In particular, it is used as a dictionary of strings that have to be replaced when encountered during the translation process.

- `private boolean checkRule()`:
  method that controls if the dependencies of the Security Capabilities in the current rule are satisfied.

- `private void exploreElement(Element e)`:
  method that retrieves, from the abstract security rule, the Security Capability node name, children element name and correspondent inner values.

- `private List<String> getEnumerationLiterals(String enumerationName)`:
  method that retrieves the enumeration literals given its `enumerationName`.

- `private Element findElementoTranslationNodesByCapability()`:
  method that retrieves the literal string values of the available enumeration classes.

- `private List<String> getListOfTextValueOfElementByTagNameFromElement(Element e, String tagName)`:
  method that, given an XML element `e`, returns the inner string stored in the children node named `tagName`.

- `private void recursivesearch(Element e)`:
  method that recursively search the XML element `e` in order to retrieve the XML element inner children name and attributes.

- `private String getDefaultActionRule(NodeList defaultActionList, String destinationNSF)`:
  method that receives the XML Default Action element through `defaultActionList` and iterates over the inner Security Capability nodes. Finally, it returns a low-level rule that can be used to specify the NSF default action strategy.

- `private String splitMultipleRule(String rule)`:
  support method for the *expansion* process defined for the operator management. This method checks if the rule has to be expanded toward an exactMatch operator, it controls the existence of a *break-rule* character and split the input string `rule` every time it is encountered.

- `private String getResolutionStrategy()`:
  method that retrieve `resolutionStrategyDetails` class for the target NSF.

- `private HashMap<String, String> findResolutionStrategyDetails(NodeList resolutionStrategyNodeList)`:
  method that retrieves from NSF Catalogue file, the content of the `resolutionStrategyDetails` class. This stores which external information a given Resolution Strategy requires (e.g. `priority` external data for First Matching Rule).

The second set of methods shown in Figure B.9 and defined by `NSF Translator` class are:

- `private String clauseConverter()`:
  method that is responsible of producing a low-level syntax for expressing the target Security Capability starting from the Security Capability name and values in abstract language format.

- `private String getPre()`:
  method that retrieves from `capabilityTranslationDetails` the low-level command name related to the target Security Capability.

- `private String getMid()`:
  method that retrieves from `capabilityTranslationDetails` the string to place between the low-level Security Capability command name and the actual low-level desired options for the Security Capability itself.

- `private String getBody()`:
  method that retrieves from `capabilityTranslationDetails` the needed string in order to specify the options and the correspondent values from the abstract language security rule for the target Security Capability.

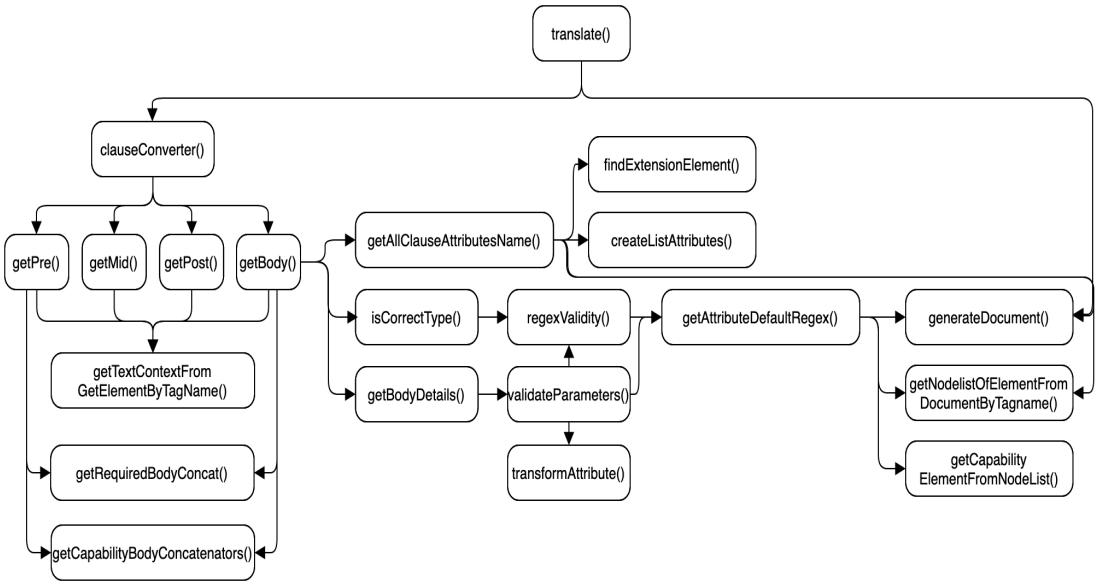

Figure B.9: NSF Translator architecture, second part

- `private String getPost()`:
  method that retrieves from `capabilityTranslationDetails` the string to place at the end of the translated Security Capability.

- `private String getTextContextFromGetElementByTagName(Element e, String s)`:
  method that search within the XML element `e` and returns the inner text of the child element named `s`.

- `private List<BodyConcatenator> getCapabilityBodyConcatenators()`:
  method that, given a `capabilityTranslationDetails` of the target Security Capability, returns a list of `BodyConcatenator` object storing information about the inner Body Concatenator XML elements. Body Concatenators stores information about how to handle multiple Security Capability attribute values. There is one of this for each Security Capability supported operators, too.

- `private BodyConcatenator getRequiredBodyConcat(List<BodyConcatenator> bodyConcatenators, String requestedConditionOperator)`:
  method that, given the required operator within the abstract language security rule, returns the correspondent BodyConcatenator class.

- `private List<String> getAllClauseAttributesName()`:
  method that recursively searches the XSD element representing the target Security Capability to retrieve the allowed Security Capability abstract language attributes name. These attributes names are used to understand which string is an attribute in the user provided abstract language security rule. It leverages createListAttributes method.

- `private void createListAttributes(NodeList elementNL, List<String> ls, Element e)`:
  method that recursively creates a list of the XSD element `e` inner attribute names. If an inner attributes is a `complexType` then the same function is called until the current attributes type is a `simpleType`.

- `private Integer isCorrectType(String parameterName, String capabilityParameter)`:
  method that, given the target Security Capability, controls if the `capabilityParameter` value provided for `parameterName` attribute is compliant with the constraints required by the Security Capability itself. In this way it is possible to understand if the encountered string in the abstract language Security Capability element is a attribute value or one of the supported operators.

- private String getBodyDetails(String bodyDetails, String[] s, List<String> clauseAttributesName): method that iterates over the abstract language strings from the target Security Capability XML element. Suppose a string is the same of one of the possible attribute names for the target Security Capability (retrieved by getAllClauseAttributesName()). In that case, it validates the provided attribute values and appends it to the support string for the body of the current Security Capability.

- `private Element findExtensionElement(Element e)`:
  method that search the `xs:extension` child node within the XSD element `e`.

- `private boolean regexValidity(String value, String regex)`:
  method that evaluates if the provided `value` respects the provided regular expression `regex`.

- `private String validateParameter(String parameterName, String capabilityParameter)`:
  method that, given the target Security Capability, controls if the `capabilityParameter` value provided for `parameterName` attribute is compliant with the constraints required by the Security Capability itself.

- `private String transformAttribute(String value, String transform)`:
  method that perform the transformation `transform` on the specified attribute `value`.

- `private String getAttributeDefaultRegex(String attributeName)`:
  method that, given the target Security Capability `capabilityTranslationDetails`, returns any regular expression for the Security Capability attribute `attributeName`.

- `private Element getCapabilityElementFromNodeList(String capa, NodeList nl)`:
  method that, given the Nodelist `nl` of available XSD elements, returns the Security Capability XSD element whose name is `capa`.

- `private static Document generateDocument(String path)`:
  method that generates a `Document` object based on the path specified by `path`.

`NSFPolicyDetails` is a support class used to store the NSF-related details expressed by the NSFPolicyDetails XML elements that can be found in each NSF XML element within the NSF Catalogue file. This class defines the following variables:

- `String ruleStart`:
  variable that stores the string to place at the beginning of each translated security rule.
- `String ruleEnd`:
  variable that stores the string to place at the end of each translated security rule.
- `String policyTrailer`:
  variable that stores the string to place at the end of the translated security policy, meaning at the end of all the translated security rules.
- `String policyEncoding`:
  variable that stores which string encoding the target NSF requires for the input security rule commands.
- `String resolutionStrategyInfo`:
  variable that stores which Resolution Strategy the target NSF requires to use during the low-level security rule translation.
- `String defaultSecurityCapability`:
  variable that stores a Security Capability name that has to be inserted for each NSF abstract security rule even if the user does not state it. It represents a *default* Security Capability for the NSF.
- `List<String> requiredPolicyAttributes`:
  variable that stores a list of attribute names that can be stated in the abstract language Policy XML node.
- `List<String> requiredRuleAttributes`:
  variable that stores a list of attribute names that can be stated in the abstract language Rule XML node.
- `HashMap<String, HashMap<String,String>> ruleAttributeDetails`:
  variable that stores a list of aliases for the values that Rule Attributes can have. In particular, for each Rule Attribute, there is a list of coupling of the target string to replace with the string to use as replacement. This variable is needed in order to provide Rule Type definition for XFRM NSF.

`BodyConcatenator` is a support class used to store BodyConcatenator XML class details. Those classes are stored in `capabilityTranslationDetails` for each Security Capability, within the NSF Catalogue XML file. BodyConcatenator class defines the following variables:

- `String realConcatenator`:
  variable that stores the string to place between the `preVariable` and `postVariable` strings found within the abstract language Security Capability element.

- `String preVariable`:
  variable that stores a string which identify an attribute value for the target Security Capability abstract language definition. The identified attribute acts as the first part to compose the Security Capability attribute.

- `String postVariable`:
  variable that stores a string which identify an attribute value for the target Security Capability abstract language definition. The identified attribute is the second part of the Security Capability attribute.

- `String postConcatenator`:
  variable that stores the string to place after `postVariable` strings found within the abstract language Security Capability element.

- `String operatorType`:
  variable that stores the operator type to which the BodyConcatenator refers to.

- `String supportPolicyAttribute`:
  variable that stores the name of the Policy attribute needed to manage the target Security Capability.

- `String concatSuppPolWithCapaValue`:
  variable that stores a Boolean information about if the Support Policy attribute value has to be concatenated with the actual value provided by the abstract language Security Capability XML element or not.

- `ArrayList<CommandName> commandNames`:
  variable that stores a list of `CommandName` jnstances for the target BodyConcatenator class.

`CommandName` is a support class used to store CommandName XML class details. Those classes are stored in `BodyConcatenator` for each Security Capability, within the NSF Catalogue XML file. CommandName class defines the following variables:

- `String commandName`:
  variable that stores the low-level command name to be used for the target BodyConcatenator and hence the target Security Capability.

- `HashMap<String,String> commandNameCondition`:
  variable that stores a list of conditions in which the key is the attribute to observe and the value is the required attribute value. If those conditions are satisfied, then the related `commandName` string has to be used as the low-level command name.

Bibliography

[1] Addison Wesley. *UML Distilled 3rd Edition*. 2003.

[2] Erich Gamma, Richard Helm, and Ralph Johnsons. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1984.

[3] Douglas C. Schmidt. *Model-Driven Engineering*. URL: https://web.archive.org/web/20060909034327/http://www.cs.wustl.edu/%5C%7Eschmidt/PDF/GEI.pdf.

[4] Object Management Group. *Model Driven Architecture*. URL: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[5] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2013. URL: https://www.w3.org/TR/xml/.

[6] World Wide Web Consortium. *W3C XML Schema Definition Language (XSD) 1.1*. 2012. URL: https://www.w3.org/TR/xmlschema11-1//.

[7] Object Management Group. *XML Metadata Interchange (XMI) Specification*. 2015. URL: https://www.omg.org/spec/XMI/2.5.1/PDF.

[8] *A Comparative Analysis of XML Documents, XML Enabled Databases and Native XML Databases*. URL: https://arxiv.org/pdf/1707.08259.pdf.

[9] EBart Baesens, Seppe vanden Broucke, and Wilfried Lemahieu. *Principles of Database Management: The Practical Guide to Storing, Managing and Analyzing Big and Small Data*. 2018.

[10] Massimo Franceschet. *Native XML Databases*. URL: https://users.dimi.uniud.it/~massimo.franceschet/caffe-xml/xschema/xschema-native.html.

[11] *BaseX Documentation*. URL: https://docs.basex.org/wiki/Main_Page.

[12] Cataldo Basile et al. *Towards the Dynamic Provision of Virtualized Security Services*. URL: https://iris.polito.it/retrieve/handle/11583/2621480/426198.

[13] European Telecommunications Standards Institute. *Network Functions Virtualisation*. URL: https://docbox.etsi.org/isg/nfv/open/Publications_pdf/White%5C%20Papers/NFV_White_Paper1_2012.pdf.

[14] *Interface to Network Security Functions*. URL: https://datatracker.ietf.org/wg/i2nsf/about/.

[15] Interface to Network Security Functions. *Problem Statement and Use Cases*. URL: https://datatracker.ietf.org/doc/rfc8192/.

[16] Interface to Network Security Functions. *Framework for Interface to Network Security Functions*. URL: `https://datatracker.ietf.org/doc/rfc8329/`.

[17] Interface to Network Security Functions. *I2NSF Capability YANG Data Model*. URL: `https://datatracker.ietf.org/doc/draft-ietf-i2nsf-capability-data-model/`.

[18] Mattia Bencivenga. *Towards the automatic refinement of high-level security policies in computer networks*.