Politecnico di Torino

Master's Degree Course in Computer Engineering

Master's Degree Thesis

A formal model of security controls implementing the IPsec and IKE protocols

Supervisors:
Prof. Cataldo Basile
Prof. Antonio Lioy

Candidate:
Andrea Avallone

Academic year 2020/2021
Torino

To my mum and my dad

"My parents gave me the greatest gift anyone could give to another person: they believed in me."

Table of Contents

List of Figures

Chapter 1

Introduction

Virtualization is a technology that allows you to create services by exploiting resources traditionally tied to hardware. It allows you to exploit all the capabilities of a physical machine by distributing the functionality among multiple users or environments.

The concept of virtualization is believed to have its origins in the late 1960s, when IBM invested a lot of time and effort in developing robust resource sharing solutions over time. Sharing of time resources refers to the shared use of computer resources among a large group of users, with the aim of increasing the efficiency of both users and the expensive computer resources they share. This model represented a major breakthrough in information technology: the cost of providing computing capability has dropped considerably and it has become possible for organizations, and even people, to use a computer without actually owning one. Similar reasons drive virtualization by industry standards: the capability in a single server is so large that it is nearly impossible for most workloads to use it effectively. The best way to optimize the use of resources and at the same time simplify data center management is through virtualization. Today, data centers use virtualization techniques to accomplish the abstraction of physical hardware, create large aggregates of logical resources consisting of CPU, memory, disks, file storage, applications, networks, and offer these resources to users or customers in the form of Agile, scalable and consolidated virtual machines. Although technology and use cases have evolved, the main meaning of virtualization remains the same: allowing a computing environment to run multiple independent systems at the same time.

Technologies that enabled virtualization, such as *hypervisors*, were developed in order to provide multiple users with simultaneous access to computers. However, initially, the problem of having multiple users on a single computer was addressed by choosing solutions other than virtualization. Among these, temporal sharing, which isolated users within operating systems and which inadvertently led to the birth of

other operating systems, such as UNIX, up to Linux. All this, while virtualization remained an underutilized niche technology. In the 1990s, most companies used physical servers and IT stacks associated with a single vendor, which did not allow existing applications to run on other vendors' hardware. Companies began updating their IT environments with cheaper commodity servers, operating systems, and applications from various vendors. However, they were tied to underused physical hardware, as each server could only perform one task. At this point virtualization began to spread. It was the natural solution to two problems: it allowed companies to get partitions on their servers and run existing applications on multiple types and versions of operating systems. The use of servers has been made more efficient, in some cases abandoned, thus reducing the costs associated with purchase, configuration, cooling and maintenance. The broad applicability of virtualization helped reduce vendor lockdown and laid the foundation for cloud computing. Today it is used by many companies, and it is often necessary to have specific software for virtualization management that allows you to monitor the environment.

Advances in cloud computing and Network Functions Virtualization technologies have made it possible to provide network services through virtual service functions running on cloud servers. It is also possible to outsource these virtual service functions to third party solution providers. This cloud-based service delivery model offers numerous benefits such as cost savings and flexible and efficient use of resources. This service model is particularly useful for providing users with network security services. For example, in the scenario of a Distributed Denial of Service (DDoS) attack, it is possible to quickly and flexibly respond to the intense traffic of attacks by dynamically increasing the number of DDoS mitigation instances. In addition, this cloud-based security service model facilitates the implementation of various security features developed by multiple vendors. This is suited to meet the growing needs of corporate network systems to integrate these security functions to create more secure systems. The Network Security Functions developed by different vendors have different interfaces for their configuration and management because there is no industry standard of interfaces for *NSF*. This heterogeneity introduces complexity to managing the NSFs of multiple suppliers, resulting in increased management costs. Therefore, standardization is essential to successfully implement NSF offered by various vendors. Recently, some standardization activities have started a development process such as the *Internet Engineering Task Force Interface to Network Security Functions* (IETF I2NSF) working group to meet these needs.

The IETF I2NSF working group develops a series of information models and standard data models that are the key to building the standard interfaces of the

I2NSF architecture. On the basis of the models defined by I2NSF, this thesis has as its starting point the realization of a generic model of the security capabilities of a network. The model is created using the UML graphic representation and some design patterns to optimize the expressive features of the model. The XML schema representation for validating NSF configured according to arbitrary capacities is obtained from the model. In fact, the primary objective of the thesis is to find a method to represent, in a generic way, the features that are offered by the security services. Having available the generic representation of the capabilities of the devices, various essential activities can be carried out both in the field of research and in the field of network security administration. For example, different security devices can be compared in a formal and precise manner to verify if they have characteristics in common for, for example, being able to apply the same security policies. To verify the expressiveness of the model, tests will be carried out with packet authentication and encryption devices through the use of network security protocols such as IPSec.

The definition of a generic data model is similar to the definition of a natural language. For example, a generic data model can define relationship types as a "classification relationship", being a binary relationship between a single element and an element type (a class) and a "part-whole relationship", being a relationship binary between two elements, one with the role of the part, the other with the role of the whole, regardless of the type of elements that are related. Given an extensible list of classes, this allows you to classify each individual item and specify relationships for each individual object. By standardizing an extensible list of relationship types, a generic data model allows the expression of an unlimited number of data types and will approach the capabilities of natural languages. Conventional data models, on the other hand, have a fixed and limited domain scope, since the instance of this model only allows expressions of predefined data types in the model. Generic data models are developed as an approach to address some shortcomings of conventional data models. For example, different modelers usually produce different conventional data models of the same domain. This can lead to difficulties in bringing together the models created by different people and is an obstacle to the exchange and integration of data. This difference is attributable to different levels of abstraction in the models and differences in the types of data that can be instantiated. Modelers have to communicate and agree on some elements that need to be rendered more concretely, in order to make the differences less significant.

Making a model generic, even in a specialist application area, can present difficulties. The essential requirements of a generic description must be identified first and an appropriate framework must be established to provide the flexibility needed

to allow a more specific set of needs to satisfy that generic representation. It may be necessary to represent a system at different levels of detail in the different phases of a design project and this must also be possible with the generic approach. This means that submodels may be required, which represent specific parts of the entire physical system, at different levels of complexity, ranging from purely functional forms in the initial phase to highly detailed and fully validated models in the subsequent phases of the project. Ideally, the structures for the different levels of the model will be directly related and the models at different resolutions will form an integrated group. The relationship between the different levels of each sub-model within the generic structure must be fully understood by users. The most important benefit of the generic approach is probably a faster and cheaper development process for new models than the traditional approach, which involves developing a new model specific to each new design task on a one-time basis. Other advantages are likely due to the development and application of a generic model that requires a more systematic and rigorous approach to model validation problems, together with better documentation. Having a generic model allows you to reason at a very abstract level such that it makes its functions and relationships visible for each element. The generic model provides information on the behavior of the system.

Based on these standard information and data models, the I2NSF architecture is able to provide an efficient and flexible security service environment driven by security policies. The *Software-Defined Networking* (SDN) paradigm enables dynamic and flexible changes in network behavior at code level by controlling and managing the configurations of network resources, such as switches and routers. This capability makes it possible to apply some packet filtering rules to switches by checking their packet forwarding rules. In particular, the switches apply simple packet filtering rules that can be translated into their packet forwarding rules, while the NSFs apply security rules relating to the security capabilities available between the NSFs. Therefore, if switches can make decisions about some received packets based on their packet forwarding rules programmed by a switch controller, we can avoid unnecessary latency for packets taken by an NSF for a time-consuming inspection task. Also, since all packets don't necessarily go through an NSF, we can reduce the possibility of congestion in an NSF.

I2NSF defines two types of interfaces at two different levels: a service level and a capability level. The level of service specifies how a customer's security policies can be expressed to a security officer. The capability level specifies how to control and monitor flow-based security functions at the functional implementation level. The policies on the service level interface do not care which NSFs are used to enforce the policies. There may be multiple NSFs to enforce a service level policy. The

policies on the capability level interface are specific to NSF. To express flow-based security policies, the *event-condition-action* (ECA) paradigm is used both on the service level interface and on the capability level interface.

From analysis in the literature, confirmed during this thesis, it appears that the languages made available by the suppliers of devices to express security policies are heterogeneous, so there may be incompatibilities or misunderstandings in the use of security control devices produced by different suppliers; therefore, to express security policies to devices from different manufacturers, it is necessary to know the details of the languages of each manufacturer of security features. For this reason, during this thesis I worked on finding a method to express security policies with a generic representation based on the security capabilities of the device that was also independent of the technology and any characteristics determined by the manufacturer. As a result, the problem arose of finding a system for translating into the device-specific language.

Model transformations are used to automate design activities by allowing you to move from an abstract, technology-independent model to a concrete, platform-dependent one. This concept is expressed by the model driven transformation paradigm. The work of this thesis develops a security policy conversion tool for NSF from a generic model to a specific model. This allows a user not to have to know the "specific language" with which each single rule will be expressed, since he will be able to provide the rules in a "generic language" to a translator, which will take care of translating the rules into the "specific language" of the NSF that will apply them.

This thesis, following the principles set out above, leads to the following. An NSF makes certain security capabilities available. The Capability Information Model (CapIM) defines an NSF as an aggregate of security capabilities. So to make the concept general, a generic NSF can be considered as a container of the security capabilities that have been assigned to it. This creates a relationship between an NSF and a security capability. In order to manage and manipulate every potential that makes up a security feature, an abstract representation of the security capabilities was created in a data model (CapDM), using the UML graphic modeling language. In this model, a security capability is represented as a class that makes attributes available. By defining the type of data that a capability can handle, you can specify any restrictions or determine the values it can represent, for example by using regular expressions or enumerations. Each security capability has been grouped into subsets of competence, and inherits characteristics according to the category in which it is located. The most abstract element of security capabilities is a class called "SecurityCapability" that allows you to define attributes or

features common to all the security capabilities that inherit from it.

Outlined the method for defining the security capabilities of a device or a security function, a system was designed for the generation of abstract languages for security controls starting from an instance of the CapDM that describes the device itself. This language allows you to express security policies using a generic syntax, taking advantage of the semantics defined by the security capabilities of the CapDM or any details specified during the allocation of capacities to the NSF. For this purpose and for the development of the generic language, the XML language was used. Security policies expressed in the general language of the NSF cannot be applied by the devices because they are not expressed in their specific language, consequently a method of translation from the generic language to the concrete language of the NSF has been devised. This system uses a translation mechanism based on the definition of syntax and semantics, specific to an NSF, at the time of assigning each security capability. In this way the tool created during the thesis work allows the generation of a configuration file containing the rules, in the concrete language of the NSF, of the policy expressed in the generic language of the same.

Chapter 2 presents the background relating to the topics addressed for the design and development of the report, with insights into the most used features. Chapter 3 introduces the technologies and related works that form the basis for developing the model design work. Chapter 4 will define in more detail the problem to be addressed, the technologies and criteria used will be exposed and the architecture designed for the solution will be made known. Chapter 5 shows the information model of the capabilities and the model of the data produced. Chapter 6 analyzes a concrete case of authentication and encryption of a package applied to the developed architecture and verifies that the proposed representation allows to cover the functionality of the concrete case. Chapter 7 illustrates the conclusions derived from this thesis work. Furthermore, the user and programmer manuals for each tool developed as part of the thesis will be specified in the appendices.

Chapter 2

Basic concepts

This chapter describes the methods from which the thesis takes its cue and on which certain choices are based instead of others.

2.1 Graphic representations

Graphical models are a class of statistical models which combine the rigour of a probabilistic approach with the intuitive representation of relationships given by graphs [1].

2.1.1 UML

The *Unified Modelling Language* (UML) is a universal language of modeling standardized by *Object Management Group* (OMG) and *International Organization for Standardization* (ISO).

The UML is a standard visual modeling language intended to be used for:

- modeling business and similar processes;
- analysis, design, and implementation of software-based systems.

UML is a common language for business analysts, software architects and developers used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems. UML can be applied to diverse application domains. It can be used with all major object and component software development methods and for various implementation platforms.

UML is a standard modeling language, not a software development process.[1]

[1] https://www.uml-diagrams.org/

The UML language is characterised by diagrams. A UML diagram is a partial graphical representation (view) of a model of a system under design, implementation, or already in existence. UML diagram contains graphical elements (symbols) that represent elements in the UML model of the designed system.

The kind of the diagram is defined by the primary graphical symbols shown on the diagram. For example, a diagram where the primary symbols in the contents area are classes is class diagram. A diagram which shows use cases and actors is use case diagram. A sequence diagram shows sequence of message exchanges between lifelines. UML specification does not preclude mixing of different kinds of diagrams.

UML specification defines two major kinds of UML diagram:

- Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

- Behavior diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.[2]

2.1.2 Class diagram

The class diagram is part of the group of *Structure diagram*. It is the most used in the development of this thesis.

This type of diagram describes the structure of a system showing the classes and the relationships between them. A class models a set of entities, called instances of the class, that share the same characteristics, limitations, and semantics (the same attributes, associations, and operations). The class is a type of classifier whose purpose is to give a classification of objects and to indicate the characteristics that describe their structure and behavior. A class is distinguished from other classes by a name and a set of properties and operations. Properties represent the structural characteristics of a class and can be represented by attributes. An attribute describes a property with a line of text, indicating the name and other optional information such as visibility, type, multiplicity and default value. The definition of an attribute is local to the class in which it is defined, in this way another class can have an attribute with the same name but with a different definition.

Two class can have different types of relationship:

[2]https://www.uml-diagrams.org/uml-25-diagrams.html

- association: an association is a relationship between two or more classifiers and also indicates a connection between their instances.
- aggregation: it is a particular type of association that expresses the concept "is part of". This type of relationship occurs when a whole is related to its parts.
- composition: it is a particular case of aggregation in which the parts exist as a function of the "whole". In other words, the part can be included at most in a whole and only the whole object can create and destroy its parts. If a component is destroyed, normally all components are destroyed at the same time.
- generalization: it is a relationship that connects a more generic element to a more specific one, it is used to describe the inheritance relationship between the various classes of a project
- association class: it is an element that possesses characteristics of both an association and a class. It can be seen as an association with class properties or as a class with association properties.
- class ≪Enumeration≫: it is a particular type of class, it works as a container of "enumeration literal", well-defined and prefixed string elements for the creation of the enumeration.

2.2 Data representation

2.2.1 XML

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.[3]

XML is a W3C approved standard that provides a generic syntax used to mark tagged documents, readable by both humans and machine. XML is a general

[3]https://www.w3.org/TR/xml/

purpose specification for defining markup languages, therefore a language based on a syntactic mechanism that allows you to define and control the meaning of the elements contained in a text document. It is extensible because it allows you to create new markup elements. This language is commonly used when sharing data. The main purpose of XML is to simplify the sharing and distribution of information between various independent systems.

XML has a number of advantages:

- redundancy: XML markup is very detailed. This allows the computer to detect common errors such as incorrect nesting;
- self-descriptive: the readability of XML and the presence of element names and attributes in XML means that people who look at an XML document can often gain an advantage in understanding the format and facilitate the identification of any errors;
- network effect and XML promise: any XML document can be read and processed by any XML tool.

The information contained in XML files is easily usable by programming languages. There are APIs implemented specifically for handling this kind of data. For example, for the JAVA language there is Java Architecture for XML Binding (JAXB), which provides a set of APIs to simplify access and creation of documents in XML format.

2.2.2 XML Schema Definition

The structure of an XML document can vary a lot and therefore there are many different ways of expressing the same information. This may not be a problem for people, but it may prevent proper communication between different computer systems. Therefore it is a good idea to create a set of rules for the document that defines which elements and attributes are allowed. This way both communicating parties know what to anticipate and can prepare their application logic accordingly.

Document Type Definition (DTD) is the original modeling language for describing the tree structure of XML documents. Using DTDs, XML is able to create a template for document markup so that the positioning of elements and their attributes can be checked and validated. Since 2009 more suitable methods have been introduced to create schema definitions.

The best known is the *XML Schema Definition* (XSD) language whose purpose is to define the nature of the schema and their parts, provide an inventory of XML markup constructs with which to represent the schema and define the application of the schema to XML documents.

The purpose of an XSDschema is to define and describe a class of XML documents using schema components to constrain and document the meaning, use and relationships of their constituent parts: data types, elements, their contents, attributes and their values.

An XMLschema specifies the general structure of an XML document and the constraints for the entities it contains. The following components of an XML document are the main elements described by the schema [2]:

- element: each element used in the XML document is defined by a declaration that includes the name, namespace and type of the element. The element namespace does not have to be explicitly specified but can be derived from its parent element. The element is of a simple or complex type;
- attribute: each attribute used within an XML element is defined by an attribute declaration. The attribute has a target (derived) namespace and is always of the simple type. Also, you can declare the fixed or default value;
- simple type: instances of this type (also called data types) are single values, that is, in strings or general numbers. With the use of restrictions it is possible to specify their format or possible values;
- complex type: a complex type describes the content of an element, that is, which child elements are allowed, in what order and quantity. It also specifies the attributes of the element. Complex types can be limited or extensive. Using the XML Schema metalanguage we can characterize these elements, attributes and types of an XML document.

2.3 Information model vs Data model

Information models (IM) are used to model conceptually managed objects, regardless of any specific protocol used to transport the data. The degree of specificity (or detail) of the abstractions defined in the IM depends on the modeling needs of its designers. In order to make the overall design as clear as possible, an IM should hide all protocol and implementation details. Another important characteristic of an IM is that it defines relationships between managed objects.

IMs are primarily useful for designers to describe the managed environment, for operators to understand the modeled objects, and for implementors as a guide to the functionality that must be described and coded in the DMs. The terms "conceptual models" and "abstract models", which are often used in the literature, relate to IMs. IMs can be implemented in different ways and mapped on different protocols. They are protocol neutral.

An important characteristic of IMs is that they can (and generally should) specify relationships between objects. Organizations may use the contents of an IM to delimit the functionality that can be included in a *Data model* (DM).

IMs can be defined using a formal language or a semi-formal structured language. One of the possibilities to formally specify IMs is to use class diagrams of the Unified Modeling Language (UML). An important advantage of UML class diagrams is that they represent objects and the relationships between them in a standard graphical way. Because of this graphical representation, designers and operators may find it easier to understand the underlying management model[3].

Compared to IMs, DMs define managed objects at a lower level of abstraction. They include implementation- and protocol-specific details. Data models are often represented in formal data definition languages specific to the management protocol used. Most of the standardized management models so far are DM and to express the relationships between objects techniques such as UML and ER diagrams still give the best results, because they are the easiest diagrams to understand.

Because conceptual models can be implemented in several ways, it is possible to derive multiple data models from a single information model (Figure 2.1). Although information models and data models serve different purposes, it is not always easy to decide which details belong to an information model and which details belong to a data model. Similarly, it is sometimes difficult to determine whether an abstraction belongs to an information model or a data model.

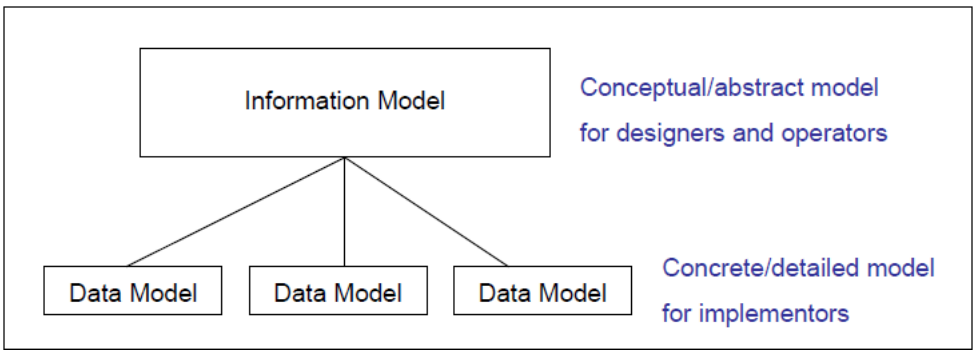

Figure 2.1: Relationship between an Information Model and a Data Model

2.4 Reusing code

Code reuse is the practice of using existing code for a new function or software. But in order to reuse code, that code needs to be high-quality. And that means it should be safe, secure, and reliable.

2.4.1 Design Patterns

Designing object-oriented software is hard, and designing reusable object oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it.

Design patterns make it easier to reuse successful designs and architectures. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"[4].

23 types of design patterns were defined in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by a group of four software designers (the "Gang of Four") in 1994 [5], thanks to which they began to study methods and software designs by reusing previously tested architectural solutions. In particular, the patterns were cataloged using a very precise formalism. In fact, each pattern is presented by the name of the pattern, the problem to which it can be applied, the solution (not in a particular case), the consequences of applying the pattern. Gamma defined the term Design pattern as "a general reusable solution to the problem commonly encountered in software design" [5]. A design model is a description for how to solve a problem that can be used in many different situations. Design patterns improve software documentation, accelerate the development process, enable large-scale reuse of software architectures, enable specialist knowledge, create design trade-offs, and can help restructure systems.

With the design patterns it is possible to find help for the structuring of the project, avoiding the possible creation of risky solutions with not very consistent architectures or that do not respect the paradigms of object-oriented programming.

The categories defined by the "Gang of Four" into which the design patterns can be divided are:

- creational patterns;
- structural patterns;
- behavioral patterns.

2.4.2 Structural patterns - Decorator pattern

One of structural patterns is the Decorator pattern. In this section I will explain why it is the most used for the realization of the models in this thesis.

Sometimes we want to add responsibilities to individual objects and not to an entire class. One way to add responsibility is with inheritance. Inheriting properties causes the characteristics of the parent to be added to all instances of the subclasses (Figure 2.2). However, this is not flexible as the choice of additional properties is done statically. A client cannot control how and when to apply new functionality to the new component but can only rely on the creation of existing classes. A more flexible approach is to enclose the component, to which you want to add responsibilities, in another object that actually adds the desired feature. The object it contains is called a decorator. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component client. The decorator forwards the requests to the component and can perform additional actions before or after forwarding. Transparency allows decorators to be nested recursively, thus allowing for an unlimited number of additional responsibilities.

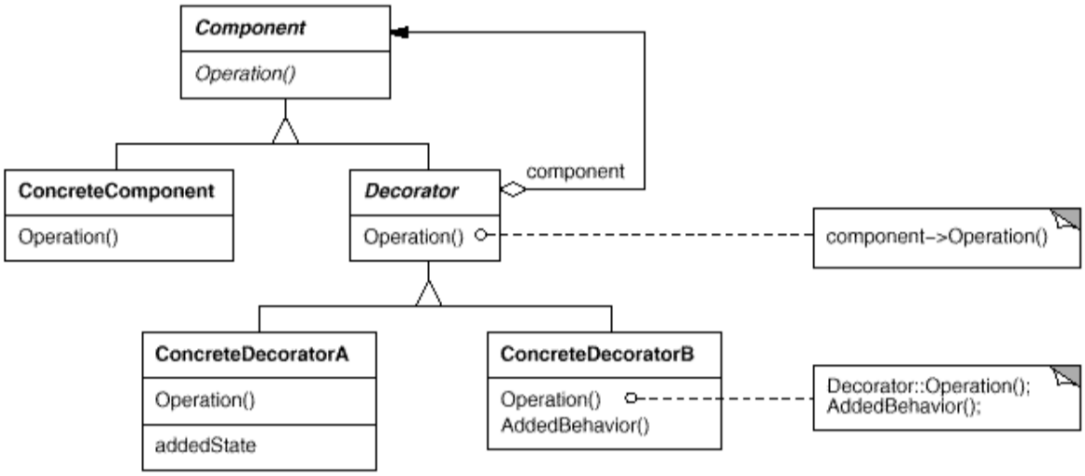

Figure 2.2: Decorator pattern object diagram

The main elements of Pattern decorator are:

- component: interface for objects to which responsibilities can be added dynamically;
- concreteComponent: object to which responsibilities can be added;
- decorator: maintains a reference to the Component object and defines an interface that conforms to the component interface;

- concreteDecorator: classes that actually extend the functionality of the Component to which they are attached.

2.5 Model driven development

Model-Driven Development (MDD) is an emerging paradigm which solves a number of problems associated with the composition and integration of large-scale systems while also leveraging the advances in software development, such as component-based middleware. The focus of MDD is to elevate software development to a higher level of abstraction than that provided by third generation programming languages. The MDD approach relies on the use of models to represent the system elements of the domain and their relationships [6].

Models are widely used in software development, but in current practice they are mainly used for communication between stakeholders, analyzing the problem, and documenting the system, while detailed design is code-centric. *Model-Driven Engineering* (MDE) is the term used for development processes that are model-centric as opposed to code-centric. In MDE models are the prime artifacts and developing high-quality systems depends on developing high-quality models and performing transformations that preserve quality or even improve it [7].

Model Driven Architecture (MDA) is an approach to software design, development and implementation spearheaded by the OMG. MDA provides guidelines for structuring software specifications that are expressed as models. MDA separates business and application logic from underlying platform technology. Platform-independent models of an application or integrated system's business functionality and behavior, built using UML can be realized through the MDA on virtually any platform, open or proprietary [8]. The MDA has three main objectives: interoperable, reusable, portable software components. The fundamental element in MDA is the model as the term Model Driven Architecture already underlines. Within the MDA, a model refers to a representation of a part of the function, structure and / or behavior of a system. To guide modeling, MDA provides two approaches to a developer:

- model refinement: concept that provides the means to add more detailed system information to an existing model;
- model transformation: model transformation will create new models based on existing ones.

Chapter 3

State of the art

In this section some techniques and some projects relevant for the development of the thesis are introduced and explained.

3.1 Network Security Functions

In the context of this thesis it deals with *Network Security Functions* (NSF). The term NSF is defined by I2NSF in the work *Interface to Network Security Functions* (I2NSF) as: "Software that provides a series of security related services".

A NSF is a function used to ensure integrity, confidentiality, or availability of network communications, to detect unwanted network activity, or to block or at least mitigate the effects of unwanted activity. NSFs are provided and consumed in increasingly diverse environments. Users could consume network security services enforced by NSFs hosted by one or more providers, which may be their own enterprise, service providers, or a combination of both. Similarly, service providers may offer their customers network security services that are enforced by multiple security products, functions from different vendors, or open source technologies. NSFs may be provided by physical and/or virtualized infrastructure. Without standard interfaces to control and monitor the behavior of NSFs, it has become virtually impossible for providers of security services to automate service offerings that utilize different security functions from multiple vendors [9].

Security features are also defined as functions responsible for the specific handling of received packets. A network security function can act on various layers of a protocol stack (for example, at the network layer or other OSI layers).

More sophisticated examples of NSF service can be:

- firewall;
- Intrusion Detection System (IDS) / Intrusion Prevention System (IPS);

- Deep Packet Inspection (DPI);
- Application Visibility and Control (AVC);
- network virus and malware scanning;
- sandbox;
- Data Loss Prevention (DLP);
- Distributed Denial of Service (DDoS) mitigation;
- proxy TLS;

3.2 Interface to Network Security Functions

NSFs are being provided and used in increasingly diverse environments. Users can use NSF-enforced network security services provided by one or more providers, which can be their company, service providers, or a combination of both. Likewise, service providers may offer their customers network security services that are applied by multiple products, functions from different vendors, or open source technologies. NSFs can be provided by physical and / or virtualized infrastructures. Without standard interfaces to control and monitor NSF behavior, it has become virtually impossible for security service providers to automate service offerings that use different security features from multiple providers.

Not only business customers, but also residential and mobile customers are becoming increasingly aware of the need for network security, only to find that security services are difficult to operate and become expensive in the case of reasonably sophisticated ones. This general trend has caused numerous security operators and vendors to start leveraging cloud-based models to deliver security solutions. In particular, methods related to *Virtualization of Network Functions* (NFV) are intended to facilitate the elastic deployment of software images that provide network services and require the management of various resources by customers, which may not possess or physically host such networking features.

The goal of I2NSF is to define a set of software interfaces and data models for controlling and monitoring aspects of physical and virtual NSFs, enabling clients to specify rulesets [9].

The definition of such interfaces has several advantages. Operators could provide more flexible and customized security services for specific users and this would provide more efficient and secure protection for each user.

I2NSF will specify interfaces at two functional levels for the control and monitoring of network security functions [9]:

- The I2NSF Capability Layer: specifies how to control and monitor NSFs at a functional implementation level. The term "Functional Implementation" is used to emphasize that the rules (for control and monitor) of NSFs have to be implementable by most NSFs. I2NSF will standardize a set of interfaces by which a security controller can invoke, operate, and monitor NSFs[9];
- The I2NSF Service Layer: defines how clients' security policies may be expressed to a security controller. The controller implements its policies according to the various capabilities provided by the I2NSF Capability Layer. The I2NSF Service Layer also allows the client to monitor the client specific policies[9].

A client may leverage the I2NSF Service Layer interface to express security policies to a security controller, which in turn interacts with one or more NSFs through the I2NSF Capability Layer interface. Alternatively, a client may interact with one or more NSFs directly through the I2NSF Capability Layer interface [9].

3.2.1 Use Cases I2NSF

The growing challenges and complexities of maintaining a secure infrastructure, complying with regulatory requirements and controlling costs are leading companies to use the network security features hosted by service providers. The hosted security service is particularly attractive to small and medium-sized businesses that suffer from a lack of security experts to continually monitor networks, acquire new skills, and propose immediate mitigation measures to ever-increasing series of security attacks.

The demand for hosted (or cloud-based) security services is growing. Small and medium-sized enterprises (SMBs) are increasingly adopting cloud-based security services to replace local security tools, while large enterprises are implementing a mix of traditional and cloud-based security services [10].

To meet the demand, more and more service providers are providing hosted security solutions to provide cost-effective security services to corporate customers. Hosted security services are primarily aimed at enterprises (especially small and medium sized ones) but could also be provided to any type of mass market customer. As a result, network security functions (NSF) are provided and used in a wide variety of environments. Users using NSF can use network security services hosted by one or more providers, which can be their company, service providers, or a combination of both.

There are many types of NSF. NSFs from different vendors may have different functionalities and interfaces. NSFs can be deployed in multiple locations in a given

network and can have different roles. Here are some examples of security functions and locations or contexts in which they are often implemented:

- protection from intrusions and external attacks: examples of this function are firewall / ACL, IPS, IDS;
- demilitarized zone (DMZ) security features: examples of this feature are firewall / ACL, IDS / IPS, one or all AAA services, NAT, forwarding proxy and application filtering;
- Centralized or Distributed Security Functions: Security functions could be implemented centrally to facilitate network management and design or in a distributed manner for scaling needs. Regardless of how a security function is deployed, it is preferable to have the same interface for deploying security policies; otherwise, the security administration task is more complex and requires knowledge of the firewall and network design;
- internal security analysis and reports: examples of this function are security logs, event correlation and forensic analysis;
- protection of data and internal content: examples of this function are encryption, authorization and management of public / private keys for internal databases.

Given the diversity of security functions, the contexts in which these functions can be implemented, and the constant evolution of these functions, standardizing all aspects of security functions is a challenge and probably not feasible. Fortunately, it is not necessary to standardize all aspects. For example, from an I2NSF perspective, there is no need to standardize how each firewall's filter is created or applied. Some filter functions from a specific vendor may be unique to the vendor's product, so there is no need to standardize these capabilities.

Standard interfaces for monitoring and controlling the behavior of NSFs are essential building blocks for security service providers and enterprises to automate the use of different NSFs from multiple vendors by their security management entities [11].

Below are some use cases identified by I2NSF's work:

- Basic Framework: sers request security services through specific clients and the appropriate NSP network entity will invoke the (virtual) NSFs according to the user service request. This network entity is denoted as the security controller. The interaction between the entities discussed above is shown in Figure 3.1.

  Interface 1 is used for receiving security requirements from a client and translating them into commands that NSFs can understand and execute. The

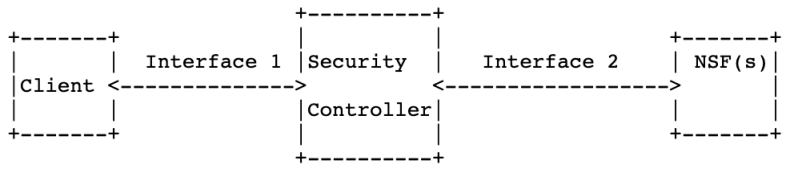

Figure 3.1: Interaction between Entities

security controller also passes back NSF security reports (e.g., statistics) to the client that the security controller has gathered from NSFs. Interface 2 is used for interacting with NSFs according to commands (e.g., enact/revoke a security policy or distribute a policy) and collecting status information about NSFs [11].

- Access Networks: This scenario describes use cases for users (residential user, enterprise user, mobile user and management system) that request and manage security services hosted in the NSP infrastructure. Given that NSP customers are essentially users of their access networks, the scenario is essentially associated with their characteristics as well as with the use of vNSFs [11].

- Cloud Data Center: In a data center, network security mechanisms such as firewalls may need to be dynamically added or removed for a number of reasons. These changes may be explicitly requested by the user or triggered by a pre-agreed-upon demand level in the Service Level Agreement (SLA) between the user and the provider of the service. This capability expansion could result in adding new instances of firewalls on existing machines or provisioning a completely new firewall instance in a different machine [11].

- Preventing DDoS, Malware, and Botnet Attacks: On the Internet, where everything is connected, preventing unwanted traffic that may cause a DoS attack or a DDoS attack has become a challenge. Similarly, a network could be exposed to malware attacks and become an attack vector that may jeopardize the operation of other networks, by means of remote commands for example. In order for organizations to better secure their networks against these kind of attacks, the I2NSF framework should provide a client- side interface that is use case independent and technology agnostic. Technology agnostic is defined to be generic, technology independent, and able to support multiple protocols and data models. Similarly, botnet attacks could be easily prevented by provisioning security policies using the I2NSF client-side interface that prevents access to botnet command and control servers [11].

3.2.2 The I2NSF framework

The I2NSF framework allows the use of heterogeneous NSFs developed by various security solution providers in the NFV (Network Functions Virtualization) environment using the capabilities of such NSFs via I2NSF interfaces.

I2NSF use cases require standard interfaces for users of an I2NSF system to inform the system which functions should be applied to which traffic (or traffic patterns). The I2NSF system realizes it as a set of security rules for monitoring and controlling the behavior of different traffic. It also provides standard interfaces for users to monitor flow-based security functions hosted and managed by different administrative domains.

Figure 3.2 shows a reference model (including major functional components and interfaces) for an I2NSF system. This figure is drawn from the point of view of the Network Operator Management System; hence, this view does not assume any particular management architecture for either the NSFs or how the NSFs are managed (on the developer's side) [12].

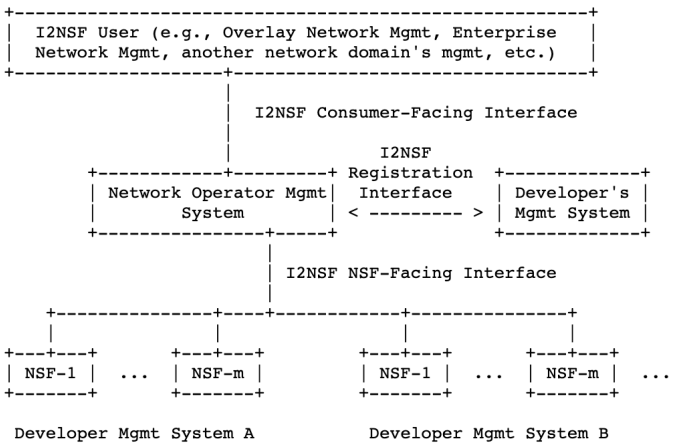

Figure 3.2: I2NSF Reference Model

When defining I2NSF Interfaces, this framework adheres to the following principles:

- It is agnostic of network topology and NSF location in the network;
- It is agnostic of provider of the NSF;
- It is agnostic of any vendor-specific operational, administrative, and management implementation, hosting environment, and form factor (physical or virtual);
- It is agnostic to NSF control-plane implementation;
- It is agnostic to NSF data-plane implementation.

In general, all I2NSF Interfaces should require at least mutual authentication and authorization for their use [12]. In the Figure 3.2 there are three interfaces:

- The I2NSF Consumer-Facing Interface: is used to enable different users of a given I2NSF system to define, manage, and monitor security policies for specific flows within an administrative domain. The location and implementation of I2NSF policies are irrelevant to the consumer of I2NSF policies.

- The I2NSF NSF-Facing Interface: is used to specify and monitor flow-based security policies enforced by one or more NSFs. Note that the I2NSF Management System does not need to use all features of a given NSF, nor does it need to use all available NSFs. Hence, this abstraction enables NSF features to be treated as building blocks by an NSF system; thus, developers are free to use the security functions defined by NSFs independent of vendor and technology.

- The I2NSF Registration Interface: NSFs provided by different vendors may have different capabilities. In order to automate the process of utilizing multiple types of security functions provided by different vendors, it is necessary to have a dedicated interface for vendors to define the capabilities of (i.e., register) their NSFs. This interface is called the I2NSF Registration Interface.

3.2.3 I2NSF Flow Security Policy Structure

Even though security functions come in a variety of form factors and have different features, provisioning to flow-based NSFs can be standardized by using policy rules.

In this version of I2NSF, policy rules are limited to imperative paradigms. I2NSF is using an Event-Condition-Action (ECA) policy, where:

- An Event clause is used to trigger the evaluation of the Condition clause of the I2NSF Policy Rule.

- A Condition clause is used to determine whether or not the set of Actions in the I2NSF Policy Rule can be executed or not.

- An Action clause defines the type of operations that may be performed on this packet or flow.

Each of the above three clauses are defined to be Boolean clauses. This means that each is a logical statement that evaluates to either TRUE or FALSE.

3.3 Information Model of NSFs Capabilities

NSFs produced by multiple security vendors provide various security capabilities to customers. Multiple NSFs can be combined together to provide security services over the given network traffic, regardless of whether the NSFs are implemented as physical or virtual functions.

Security Capabilities describe the functions that Network Security Functions (NSFs) are available to provide for security policy enforcement purposes. Security Capabilities are independent of the actual security control mechanisms that will implement them.

Every NSF should be described with the set of capabilities it offers. Security Capabilities enable security functionality to be described in a vendor-neutral manner. That is, it is not needed to refer to a specific product or technology when designing the network; rather, the functions characterized by their capabilities are considered. Security Capabilities are a market enabler, providing a way to define customized security protection by unambiguously describing the security features offered by a given NSF.

A *Capability Information Model* (CapIM) is a formalization of the functionality that an NSF advertises. This enables the precise specification of what an NSF can do in terms of security policy enforcement, so that computer-based tasks can unambiguously refer to, use, configure, and manage NSFs. Capabilities MUST be defined in a vendor- and technology-independent manner.

The CapIM is intended to clarify these ambiguities by providing a formal description of NSF functionality. The set of functions that are advertised MAY be restricted according to the privileges of the user or application that is viewing those functions. I2NSF Capabilities enable unambiguous specification of the security capabilities available in a (virtualized) networking environment, and their automatic processing by means of computer-based techniques.

This includes enabling the security controller to properly identify and manage NSFs, and allow NSFs to properly declare their functionality, so that they can be used in the correct way [1] [13].

Some basic design principles for security capabilities and the systems that manage them are:

- Independence;
- Abstraction;

[1] https://datatracker.ietf.org/doc/html/draft-ietf-i2nsf-capability-05

- Advertisement;
- Execution;
- Automation;
- Scalability.

Based on the above principles, I2NSF defines a capability model that enables an NSF to register (and hence advertise) its set of capabilities that other I2NSF Components can use. These capabilities MAY have their access control restricted by policy; this is out of scope for this document. The set of capabilities provided by a given set of NSFs unambiguously define the security offered by the set of NSFs used. The security controller can compare the requirements of users and applications to the set of capabilities that are currently available in order to choose which capabilities of which NSFs are needed to meet those requirements.

Furthermore, when an unknown threat (e.g., zero-day exploits and unknown malware) is reported by an NSF, new capabilities may be created, and/or existing capabilities may be updated. This results in enhancing the existing NSFs (and/or creating new NSFs) to address the new threats. New capabilities may be sent to and stored in a centralized repository, or stored separately in a vendor's local repository. In either case, a standard interface facilitates the update process[1].

The "Event-Condition-Action" (ECA) policy model [12] is used as the basis for the design of the capability model. The following three terms define the structure and behavior of an I2NSF imperative policy rule:

- Event: is defined as any important occurrence in time of a change in the system being managed, and/or in the environment of the system being managed.
- Condition: is defined as a set of attributes, features, and/or values that are to be compared with a set of known attributes, features, and/or values in order to determine whether or not the set of Actions in that (imperative) I2NSF Policy Rule can be executed or not.
- Action: An action is used to control and monitor aspects of flow- based NSFs when the event and condition clauses are satisfied. NSFs provide security functions by executing various Actions.

An I2NSF Policy Rule is made up of three Boolean clauses: an Event clause, a Condition clause, and an Action clause. This structure is also called an ECA (Event-Condition-Action) Policy Rule. A Boolean clause is a logical statement that evaluates to either TRUE or FALSE. It may be made up of one or more terms; if more than one term is present, then each term in the Boolean clause is combined using logical connectives (i.e., AND, OR, and NOT).

An I2NSF ECA Policy Rule has the following semantics:

```
IF <event-clause> is TRUE
  IF <condition-clause> is TRUE
    THEN execute <action-clause> [constrained by metadata]
  END-IF
END-IF
```

Since there can be many types of NSF that have many different types of I2NSFSecurityCapabilities, the definition of a *SecurityCapability* must be done using the context of an NSF. This is realized by an association class in UML. *HasSecurityCapabilityDetail* is an association class Figure 3.3.

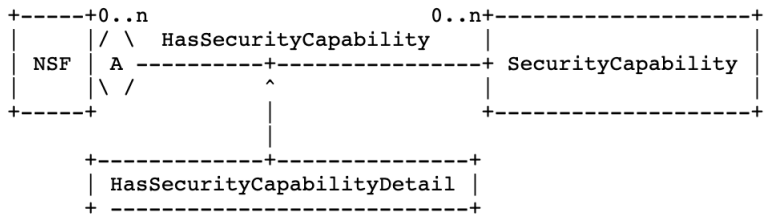

Figure 3.3: Defining SecurityCapabilities of an NSF [13]

This enables the HasSecurityCapabilityDetail association class to be the target of a Policy Rule. That is, the HasSecurityCapabilityDetail class has attributes and methods that define which I2NSFSecurityCapabilities of this NSF are visible and can be used [13].

Chapter 4

Solution design

In this chapter I will deal specifically with the definition of the problem to which I want to give a solution in this thesis, I will show some use cases on which I relied to devise the solution and for the development of the tools. In addition, the concept of abstract language for security controls will be introduced and the design of the solution will be presented.

4.1 Problem definition

The goal of I2NSF is to define a set of software interfaces and data models for the control and monitoring of aspects of NSF implemented in a physical and virtual way, allowing customers to specify sets of rules from one or more management entities. From this concept my thesis focuses on the problem of representing abstract data models of network security features. The problem that arises the interest of my thesis concerns the management of data models that represent network security features. Data models based on *Capability Information Model* (CapIM) are expressive enough to describe real NSFs. In fact, starting from the information model created in the basic framework defined by I2NSF in Figure 3.3, the thesis proposes the definition of the data model and an expansion of the expressive capability of this model in order to cope with the problem of the generation of abstract NSF languages and the subsequent translation of the policies. There is a need to define a data model that represents, in a generic way, the security capabilities for network security services, sufficiently expressive to be able to represent the potential of the devices that offer security functions, and to be able to compare these devices to verify whether they can express the same security policies. The CapDM includes all the functionalities that can be used to assign capability to an NSF following the decorator pattern paradigm.

Now the problem of generating abstract languages for security controls arises to allow the definition of abstract policies for NSF with network services. Obviously, each NSF has its own language for applying a security policy. The language must therefore be generic as it can be used by each NSF but specific based on the security capabilities assigned to the NSF concerned. Since the maintenance of abstract languages associated with each NSF is difficult, cumbersome and time-consuming, the problem arose of having to define an automatic procedure for deriving the abstract configuration language of an NSF starting from the description of its network security capabilities. We encountered the problem of having to associate an automatic transformation mechanism based on the instances of the CapDM. This automatic transformation requires support for the automatic refinement of abstract configurations written in the NSF-specific language model into concrete configuration settings applicable by the NSF. Consequently, criteria or rules are needed to define the semantics and syntax that allow the translation of the policies expressed in the generic language of the NSF into the low-level language relating to the NSF itself.

4.2 Generation of abstract languages

Our goal is to generate a new high level language that allows us to express concrete security concepts with a common syntax for another NSF.

This project is based on the work of N. Noceti in his thesis and improve the ability to have more defined NSF to use the same language.

An abstract security control language should satisfy the following requirements:

- Abstraction: the language must contain abstract, vendor-independent security configurations or product-specific representation and storage. The reason for this requirement is that the configuration semantics are independent of the actual representation. That is, the same configuration settings can be represented and applied in different security controls;
- Diversity: must support description of configurations for a variety of security functions. The configuration metamodel must also support the configuration of these security capabilities, which follow different policies and concepts and are applied to different types of security controls;
- Flexibility and extensibility: it must be flexible and extensible enough to support the introduction of new security controls;
- Continuity: must ensure continuity of the policy chain, allowing translation of security control settings. This is useful for keeping track of which policy is actually applied in the policy.

A generic security control language is defined to abstract configuration languages with a vendor- and control-system-independent format organized by capabilities. For example, one vendor's firewall may implement packet filtering functionality in its capability set, while another vendor may implement packet inspection capability in its firewall as well. Unfortunately, defining this abstraction is not trivial because each security check has a specific syntax. Therefore the mapping can be unmanageable in a generic syntax and the generic language must be organized according to security features. A security feature is a basic functionality offered by a security check. Therefore, a generic language for security checks is organized according to a general model that defines the high-level concepts (policies, rules, conditions, actions, etc.) and a set of submodels to acquire the specific concepts of semantics such as attributes, types of conditions, methods, etc.. On the other hand, a specific configuration language of the control depends on the formats and functions available in the actual security control: each security function defines its own configuration language according to its security. The generation of abstract languages allows to standardize the method by which network security policies can be expressed.

In this proposed solution, to define the generic format to express the abstract language of NSF, the use of XML language was chosen, as there are numerous advantages.

The first advantage is that the XML language is designed for the definition of markup languages that can be defined as *XMLSchema* (XSD), this allows the generation of languages according to the capabilities that have been attributed to a specific NSF. Using the open source *modelio* framework for the graphical representation of the Capability Information Model and the Capability Data Model, the XML language is immediately compatible since in the *modelio* framework there is a feature that allows you to export the model in the adaptable XMI format through the use of a tool created specifically for translation to the XSD format. A fundamental property for the generation of abstract languages for security checks is the possibility of dynamically defining the semantic characteristics of the language itself, using the XML language allows you to define, using special tools, a fixed structure dependent on the semantic characteristics of the language itself. Furthermore, the XML language is widely known in the IT field.

4.3 The main goal

One of the use cases showed in the I2NSF contest (page.27) is the aim of this thesis: there is a need to insert a security policy into the network. We do not know

the specific low-level language with which the various network security services can express this policy, but we are able to recognize which are the necessary security capabilities and assign them the required values. The thesis proposes a solution that allows the possibility of using a generic language to express the necessary policy and therefore to have a tool that translates it into the specific language of the chosen NSF or of the set of NSFs needed to apply this policy.

A new file will be created in which all the security capabilities and the NSF are declared. Each NSF will have an instance of the linked security capability, in this way we know the requirements of the concerned NSF. This is an advantage as if you find that the performance of a given NSF is not sufficient to enforce the required standards, it is necessary with an instance of a better performing NSF. Using the new NSF, the policy is translated into the configuration with the specific language of the chosen NSF.

4.4 Design of the solution

In this paragraph the planning and the design of the solution to the previously exposed problems are presented. In addition, the architecture that was conceived and produced will be exhibited.

This discussion is based on the work carried out by N. Noceti in his thesis [14], from which ideas and foundations are taken to carry out a work dedicated to a different protocol. In the implementation of the latter, a different path had to be chosen so that his work and the new one were compatible. This has led to substantial changes in the design and development of the tools.

The main reason is given by the fact that with these changes the reuse of the code and the addition of new NSFs is simpler and more immediate. However, this denotes a significant increase in the complexity of the work carried out.

For the realization of an information model of security capabilities sufficiently expressive to describe real NSFs, the *Capability Information Model* (CapIM) is used (detailed in the Section 5.1). In order to describe generic NSFs, the thesis work takes up the abstract NSF vision to which security capabilities are assigned according to the desired security functionality. To achieve this, the decorator pattern paradigm has been applied, which allows you to assign characteristics in a dynamic way.

In order to describe the characteristics to be assigned to an NSF, the thesis work refers to the representation of the security functions of the network services already used in Noceti's thesis. Depiction generates a security capabilities data model or the *Capability Data Model* (CapDM), which will be detailed in Section 5.2.

The purpose of the CapDM is to represent in a generic way every single part that makes up a security feature, called security capability, by specifying its implementation details in a generic way. The CapDM used groups the security capabilities into sets defined according to the characteristics of the capabilities themselves.

The hierarchy used for the representation of security capabilities starts from the division into the 6-tuple defined by I2NSF (Section 3.2), consisting of:

- event;
- condition;
- action;
- default action;
- resolution strategy;
- evaluation.

The attributes for each security capability were also represented in the same model in order to define a unique and generic representation, thus allowing the creation of a common generic language. Since these attributes are specified as optional, they have been reused for my specific case.

For the graphic representation of CapIM and CapDM, multiple graphic environments were taken into consideration for the creation of the class diagram, the final choice was to use the extensible modeling environment and open source modelio. This environment allows the management of diagrams using the UML language in a simple and intuitive way. Modelio is a standalone application based on Eclipse RCP.

To address the problem of the representation of generic policies, the thesis proposes a tool for the creation of the generic language of the NSF. The generic language of NSF is the set of generic representations of its security capabilities, within which there is the semantics useful for expressing any abstract policies. The proposed NSF language is implemented using the specific security capabilities that have been assigned to it.

To better express and make the language of the NSF generic, it is necessary to use a metalanguage that allows the definition of further languages. The choice fell on XML as it was already used for Noceti's work and proved to be the best choice, because it allows the definition of markup languages, that is a language based on a syntactic mechanism that allows you to define and control the meaning of elements contained in a document. It also allows the generation of a language that is both human-readable and machine-readable, independent of the surrounding technologies and with a high capability for expression. Consequently, one of the

proposed tools allows the generation of a file containing the abstract language of NSF in XSD format. This allows the generation of grammar-based policies without context.

The tool proposed for the generation of the language is based on the representations of the CapIM and the CapDM, in fact it manages the generation of the language according to the description of the assigned security capabilities and any specific constraints defined on the capabilities, using a model driven approach.

Finally, the problem of translating the policies from the NSF generic language to the specific low-level language of the same was addressed. Since in Noceti's work the problem of having multiple NSFs and therefore multiple different translations for the same security capability was not addressed, the translation tool has been changed and improved.

The proposed tools are made using the JAVA language. The advantages of choosing the JAVA programming language are also related to the high compatibility in the management of files in XML format, in fact there are APIs that allow optimized management of the latter, such as JAXB and JAXP.

A further reason that led to the choice of these technologies is that the export functionality of the UML model in XMI format is integrated in the model development environment. The first modified tool is used for the conversion from XMI language to XML language developed by Noceti. The changes made were necessary due to the fact that the CapIM was not considered as an input but only the CapDM. This resulted in instances with NSF or the *HasSecurityCapabilityDetails* not being translated.

Furthermore, a functional tool was used that allows the validation of an XML file with respect to a reference schema.

The design of the solution and the connection between the elements are shown in the Figure 4.1.

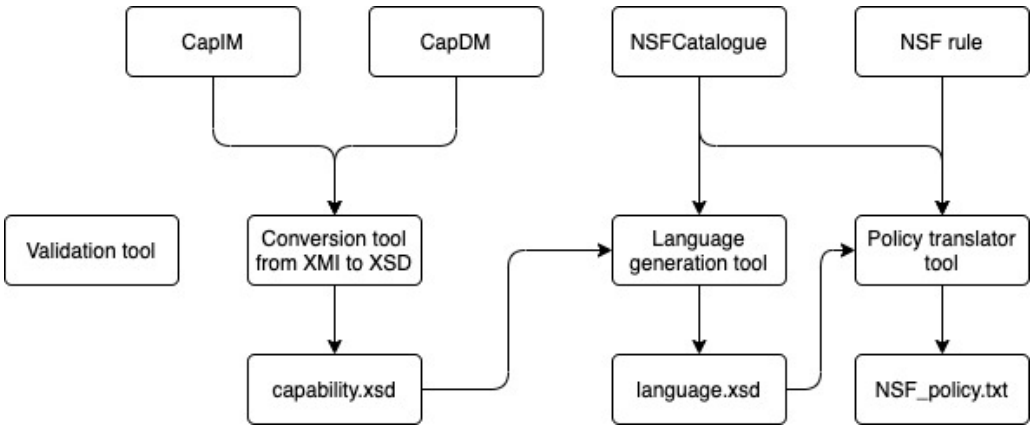

Figure 4.1: Solution Design

Every entity represent:

- CapIM: the Capability Information Model is the reference model for the development of the solution, it represents the main entities that are involved in the study of the thesis. This model is created with the aid of the chosen model tool for modeling the classes in the relevant class diagram (Section 5.1);
- CapDM: the Capability Data Model is the reference model where the security capabilities are hierarchically represented, created with the aid of the model tool chosen for modeling the classes in the related class diagram (Section 5.2);
- Conversion tool from XMI to XSD: in order to use and validate a set of security capabilities to be assigned to an NSF, a list of valid security capabilities must be obtained. This tool allows the generation of a set of valid capabilities in the XSD format on the basis of the capacities modeled in the CapDM with references to the CapIM. In input we use the XMI file complete with name *definitivo.xmi* as export from modelio;
- capability.xsd: the output of the previous tool in XSD format;
- NSFCatalogue: in the proposed solution this element represents the set of all NSF instanced with the reference security capabilities;
- Language generation tool: this tool allows the generation of the abstract NSF language, with NSFCatalogue and capability.xsd as input;
- language.xsd: entity containing the semantic characteristics that allow the generation of security policies. It is the output of the previous tool in XSD format;
- NSF rule: file in txt format where are listed security capabilities in generic language to make a new policy;
- Policy translator tool: this tool allows the translation of a policy generated with the language of the NSF into the low-level language of the NSF;
- NSF_policy.txt: policy in the corresponding NSF language;
- Validation tool: tool that allows the validation of an XML file with respect to its XMLSchema;

The realization of each tool has been designed and adapted for use from the command line, from this you get the advantage of being able to automate its use. This kind of character has not been changed.

Chapter 5

Security Capabilities Model

In this chapter we will show the modeling of the security capabilities and the structure of the solution related to the information model and the data model.

5.1 Information Model

An information model allows to represent and model managed objects on a conceptual level, independent of any circumstance and surrounding technology. Information models focus on representing the abstract entities and functionalities of an environment; they highlight the relationships between the objects involved and the dynamics related to the entities. They can have different degrees of detail of the abstractions defined according to the needs of the designers and hide the implementation details and the realization of the represented entities.

I2NSF defines in the basic framework the information model for an NSF (Figure 3.3), where the following objects are defined:

- NSF: entity representing any NSF;
- SecurityCapability: entity representing any security capability;
- HasSecurityCapabilityDetail: Entity representing the details of the relationship between a SecurityCapability and its NSF.

The following relationships are represented in this information model:

- HasSecurityCapability aggregation between NSF and SecurityCapability: relationship indicating the belonging of zero or more security capabilities to zero or more NSF;
- HasSecurityCapabilityDetail association class: relationship used to represent the HasSecurityCapabilityDetail entity belonging to a SecurityCapability relationship to an NSF.

Given the problems presented in the previous chapter, a solution was therefore devised, based on the CapIM defined by I2NSF, the thesis proposes the CapIM represented in Figure 5.1.

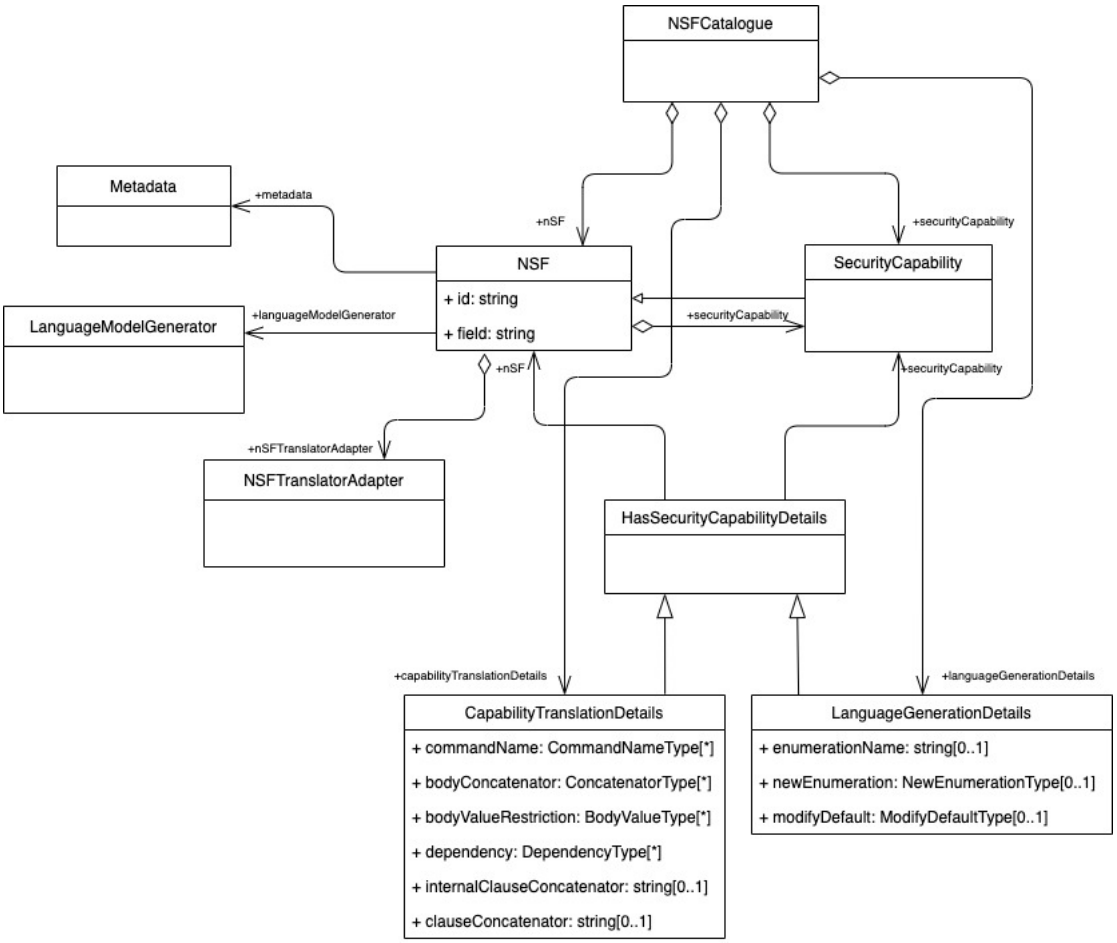

Figure 5.1: CapIM

The following objects are defined:

- NSF: entity representing any NSF;
- SecurityCapability: entity representing any security capability;
- HasSecurityCapabilityDetail: Entity representing the details of the relationship between a SecurityCapability and its NSF;
- LanguageModelGenerator: entity representing the language generation of the instantiated NSF;
- NSFTranslatorAdapter: entity representing the translation of the policy into the low-level language of the instantiated NSF;
- Metadata: entity that represents any NSF metadata, for example the list of protocol names and their associated number, known by the NSF;

- CapabilityTranslationDetails: entity that inherits from HasSecurityCapabilityDetail, and represents, specific details given by the relationship of a security capability to an NSF regarding the creation of the NSF language;
- LanguageGeneratorDetails: entity that inherits from HasSecurityCapabilityDetail, and represents, specific details given by the relationship of a security capability to an NSF, concerning the translation of policies from the generic language of the NSF to the low-level language of the same.

The following relationships are represented in the proposed information model:

- relationship +metadata: relationship that allows the NSF to possess metadata, with indefinite cardinality to allow the relationship of multiple metadata corresponding to multiple NSFs;
- relationship +languageModelGenerator: generic relationship to the tool that allows the generation of an abstract language of the NSF;
- relationship +nSFTranslatorAdapter: generic relationship to the tool that allows the transformation of policies expressed in the generic language of the NSF by producing a configuration file adapted to the characteristics of the NSF;
- aggregation +securityCapability between NSF and SecurityCapability: relationship that indicates the belonging of a SecurityCapability instance to an NSF instance;
- aggregation +securityCapability between NSFCatalogue and SecurityCapability: essential relationship to have a list of all the security capabilities in a catalogue;
- aggregation +nSF between NSFCatalogue and NSF: essential relationship to have a list of all the NSF instantiated in a catalogue;
- aggregation +capabilityTranslationDetails between NSFCatalogue and CapabilityTranslationDetails: essential relationship to have a list of translations to be carried out for each security capability in a catalogue;
- aggregation +languageGenerationDetails between NSFCatalogue and LanguageGenerationDetails: essential relationship to have a list of translation constraints to be carried out for a specific security capability in a catalogue;
- association +securityCapability between HasSecurityCapabilityDetails and SecurityCapability: relationship that allows you to have a link between a security capability and its details, so as to create an essential reference with the specific instance;

- association +nSF between HasSecurityCapabilityDetails and NSF: relationship that allows you to have a link between an NSF and its details, so as to create an essential reference with the specific instance;
- generalization between NSF and SecurityCapability: relationship that allows, together with aggregation +securityCapability, to apply the decorator pattern and to have a inheritance from NSF to SecurityCapability;
- generalization between LanguageGenerationDetails and HasSecurityCapabilityDetails: relation that allows LanguageGenerationDetails to be a child of HasSecurityCapabilityDetails;
- generalization between capabilityTranslationDetails and HasSecurityCapabilityDetails: relation that allows capabilityTranslationDetails to be a child of HasSecurityCapabilityDetails;

The information model presented in the thesis focuses on the management of NSF, in fact it represents the NSF as its main element, how it is related to security capabilities and how it interacts with the assignment of policies. The model allows to describe an NSF and to assign security features dynamically through decorator patterns and associations with securityCapabilityDetails. This way when a new NSF is instantiated it will be easy to instantiate a securityCapabilityDetail and make a reference to the NSF and securityCapability name. From here we can define the characteristics necessary for the management of the NSF language generation and for the adaptation to the low-level language of the NSF instance. As part of the thesis, the security capabilities are described in the proposed Capability Data Model.

5.2 Data Model

Compared to Information Models, Data Models define managed objects at a lower level of abstraction. Although information models and data models have different purposes, it is not always easy to decide which details belong to the former and which belong to the latter. Similarly, it is sometimes difficult to determine whether an abstraction belongs to an information model or a data model. They are intended for implementers and include specific details relating to the implementation, such as rules that explain how to map managed objects on lower-level constructs or define the structure of the various entities. Since information models can be implemented in different ways, it is possible to derive multiple data models from a single information model.

The data model proposed in the thesis is based on the work of *N. Noceti* in his thesis [14]. His work uses the subdivision of the SecurityCapabilities into a 6-tuple as defined by I2NSF:

- event: ability to recognize an event, the occurrence of a fact;
- condition: ability to assess whether the conditions are met;
- action: ability to carry out a command;
- default action: action used by default;
- resolution strategy: methodology according to which the rules are evaluated with respect to the occurrence of an event;
- evaluation criterion: methodology according to which the parts of the rule are correct.

The difference with respect to the definition of I2NSF, which can be seen in Figure 5.2, is the presence of an aggregation relationship between the class corresponding to the default action (DefaultAction-Capability class) and the class relating to actions (ActionCapability class). This relationship allows you to define the ability to perform an action only once and to be able to use the same definition for the predefined action. So the set of predefined actions can be equal to the set of actions or a subset of it.

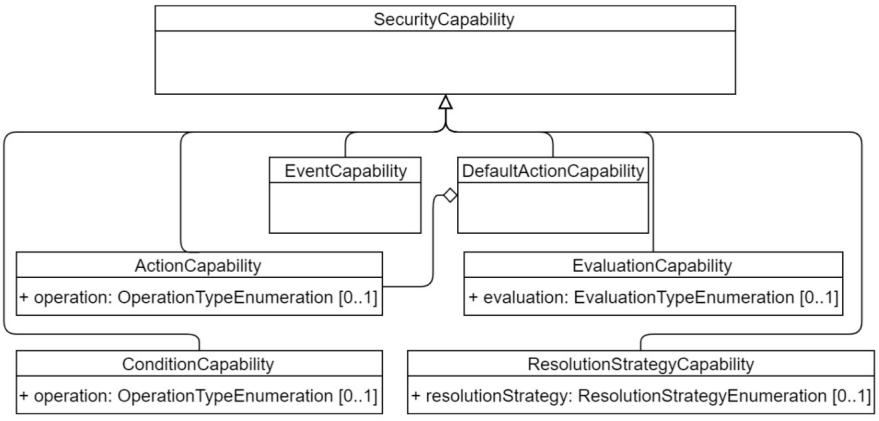

Figure 5.2: 6-Tupla

The proposed CapDM broadly develops the condition and action classes. Starting from the 6-tuple and using UML inheritance, further subclass hierarchies have been defined that allow to differentiate the scope of the security capability being considered.

5.3 LanguageGenerationDetails Class

To address the problem of the generation of an abstract language of the NSF, the thesis proposes a tool that exploits the mechanism of Model-driven transformation. This transformation makes it possible to pass from the generic representation of security capabilities, defined in the CapDM, to the specific definition of the abstract language of the NSF, to which certain security capabilities have been assigned with the possibility, if necessary, to apply details for customization. In fact, the *LanguageGenerationDetails* class is proposed as a subclass of the association class defined by I2NSF *HasSecurityCapabilityDetail*. This allows to make explicit the capability to which it belongs and the context NSF.

The structure governing the generation of the language of NSF proposed by the thesis is represented in Figure 5.3. Depending on the values specified in the "LanguageGenerationDetails" class during the assignment of security capabilities, the language generation tool interprets these values and generates the language accordingly. This allows you to customize some aspects of the NSF semantics and allows you to define non-standard semantics when assigning security capabilities to the NSF.

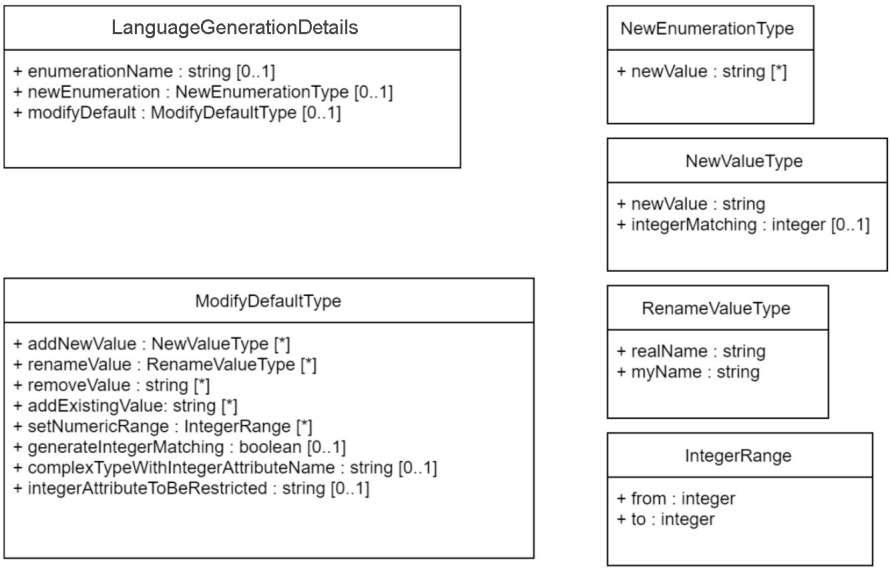

Figure 5.3: LanguageGenerationDetails

Specifically, the goal of the "LanguageGenerationDetails" class is to allow the customization of the values that can be assigned to certain parameters, such as integer parameters or values belonging to enumerations.

The proposed class "LanguageGenerationDetails" is composed of the following structure:

- enumerationName: any name of the enumeration to be modified;
- newEnumeration: any parameter that allows the generation of its own enumeration; is composed by:
  - newValue: value that must be assigned to the new enumeration, has cardinality [0..n] in order to generate a list of new values;
- modifyDefault: parameter that allows you to modify the default values of an attribute, also allows the restriction of integer values; is composed by:
  - addNewValue: parameter that allows you to enter a new value to the default enumeration. This element is further composed of two parameters, one of which is optional, to allow, if necessary, the matching between string value and numeric value;
  - renameValue: parameter that allows you to rename an existing value in the default enumeration, this element is further composed of two parameters to allow the new nomenclature with respect to the default nomenclature;
  - removeValue: parameter that allows you to remove existing values in the default enumeration and to consider all the others valid. This parameter cannot be used together with "addExistingValue";
  - addExistingValue: by specifying this element, you declare that you want the past value in your enumeration that is already present in the default enumeration. This parameter cannot be used together with "removeValue";
  - setNumericRange: parameter that allows you to indicate the range of integer values that can be accepted in the instantiation of the capability when defining a rule of a policy;
  - generateIntegerMatching: parameter that, if true, generates the correspondence between the string value of the enumeration and the integer value of the corresponding parameter;
  - complexTypeWithIntegerAttributeName: parameter that allows you to indicate the name of the complex type declared in the CapDM to which the change will be imposed;
  - integerAttrobuteToBeRestricted: parameter that allows you to indicate any list of parameters belonging to "complexTypeWithIntegerAttributeName" on which to apply the restriction.

The main difference with the work of N. Noceti [14] (beyond the modification of the class name) is that this "LanguageGenerationDetails" class is instantiated

in the NSF catalogue with reference to the NSF and the SecurityCapability we are referring to. In this way it will not be an attribute of the SecurityCapability class and can be independent from other instances.

5.4 CapabilityTranslationDetails Class

To address the problem of translating NSF's generic language policy into its low-level language, Noceti's thesis [14] proposes a tool that interprets a syntax expressed by the *CapabilityTranslationDetails* class at the time of assigning security capabilities to the NSF.

The mechanism is to instantiate each time a languageGen class that has a reference to the chosen NSF and SecurityCapability. In this way, in our catalogue, we can have multiple translations of the same SecurityCapability but which differ from the NSF we are referring to as obviously each has a different translation.

This work implies that every time we ask the tool for the translation, it is necessary to indicate the NSF as it knows the abstract language of the same, so as to generate the policy translated into the specific language.

The CapabilityTranslationDetails class is proposed as a subclass of the association class defined by I2NSF *HasSecurityCapabilityDetail*. This allows to make explicit the security capability to which it belongs and the context NSF. The structure governing the translation into the specific language of NSF proposed by the thesis is represented in Figure 5.4. Depending on the values specified in the CapabilityTranslationDetails class during its instantiation, the language generation tool interprets these values and translates the policy accordingly. Some parameters of the CapabilityTranslationDetails class can influence the structure of the translation format of each security capability, this can affect the method of use of the file produced by the translation.

Specifically, the goal of this class is to allow the customization of the semantics and syntax concerning the low-level language of the NSF. This class is based on the definition of the CapDM and allows you to customize the characteristics used for the particular security capability to which it refers, in the context of the NSF.

The proposed class CapabilityTranslationDetails is composed of the following structure, the names of the attributes to which reference will be made below are those defined in the CapDM:

- commandName: parameter that allows you to specify the name of the command in the low-level language of the NSF. Since there can be several commands related to a security capability, this parameter is composed of:

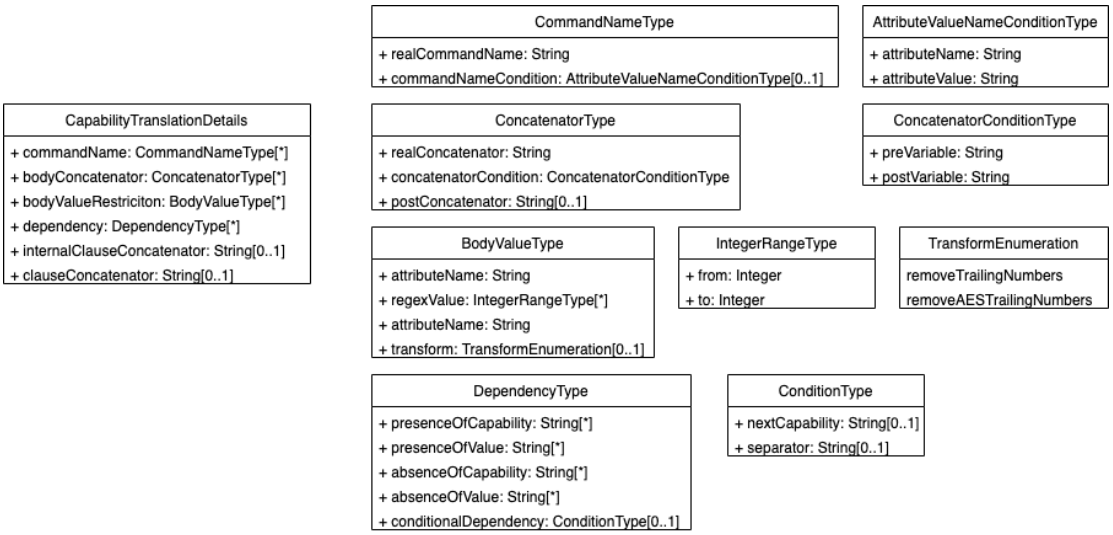

Figure 5.4: CapabilityTranslationDetails

  - realCommandName: parameter that contains the actual name to be used when the "commandNameCondition" is respected;
  - commandNameCondition: parameter that allows you to form conditions based on the value of specified attributes. This parameter consists of:
    * attributeName: name of the attribute to consider to evaluate whether to use the "realCommandName";
    * attributeValue: attribute value that must be compared to evaluate whether to use the "realCommandName".

- bodyConcatenator: parameter that allows to specify the separator of the values that can be attributed to the relative command. Since there can be several separators relating to the expressible attributes, this parameter is composed of:
  - realConcatenator: value of the separator to be used if the "concatenatorCondition" is verified;
  - concatenatorCondition: parameter that allows you to form conditions based on the value of specified attributes. This parameter consists of:
    * preVariable: name of the attribute to be considered whose value will be found before the relative concatenator;
    * postVariable: name of the attribute to be considered whose value will be found after the relative concatenator;

- bodyValueRestriction: parameter that allows you to restrict the translation range of parameters. Since there can be more types of restrictions, this parameter is composed of:

  - attributeName: name of the attribute to which you want to give translation restrictions;
  - regexValue: parameter that allows the declaration of a regex to restrict the range of translatable values;
  - integerRange: parameter that allows the restriction of integer values that can be translated. This parameter consists of:
    * from: starting value of the range;
    * to: end of range value;
  - transform: parameter that allows to make a transformation on the attribute specified. For now we can have two types of transformation:
    * removeTrailingNumbers;
    * removeAESTrailingNumbers.

- dependency: parameter that allows you to express dependencies of the security capability to which it is assigned with respect to other security capabilities Since there can be multiple types of dependencies, this parameter is composed of:
  - presenceOfCapability: imposes the presence of a certain security capability;
  - absenceOfCapability: imposes the absence of a certain security capability;
  - presenceOfValue: forces the presence of a certain value;
  - absenceOfValue: forces the absence of a certain value;
  - conditionalDependency: indicates whether there should be a dependency based on the condition specified in the class:
    * nextCapability: indicates that the presence of an immediately succeeding capability is required to verify this condition;
    * separator: if the condition is true, insert this separator between the two capabilities.

- internalClauseConcatenator: parameter that allows the definition of a value that separates the name of the command and the value attributed to it;
- clauseConcatenator: parameter that allows the definition of a value that separates the translated construct of the security capability to which it is assigned from the next.

Chapter 6

Analysis and validation on a concrete case: IPsec

In this chapter the workflow proposed by the thesis for the management of NSF in the field of network security will be explained. Subsequently, a concrete case is analyzed, IPsec, concretely showing the application of the workflow.

6.1 Workflow

The reference workflow, shown in Figure 6.1, is based on the information model and the data model described in the previous chapter.

The phases described by the workflow develop as follows:

1. *Instantiate a new NSF*: decision to create a new NSF;
2. *Instantiate new Security Capabilities*: decision to create a new Security Capability;
3. *Create references between the new NSF and the Security Capability*: insert a "re" field that defines the name of the security capability in the new language. Insert this new security capability in the reference NSF in order to create a link between them;
4. *Definition of CapabilityTranslationDetails and LanguageGenerationDetails for associated security capabilities*: create an instance of these two classes for each security capability by inserting a "ref" field to indicate the security capability and the NSF we are referring to;
5. *Language generation*: use of the tool proposed in the thesis that allows the generation of the language of the NSF;

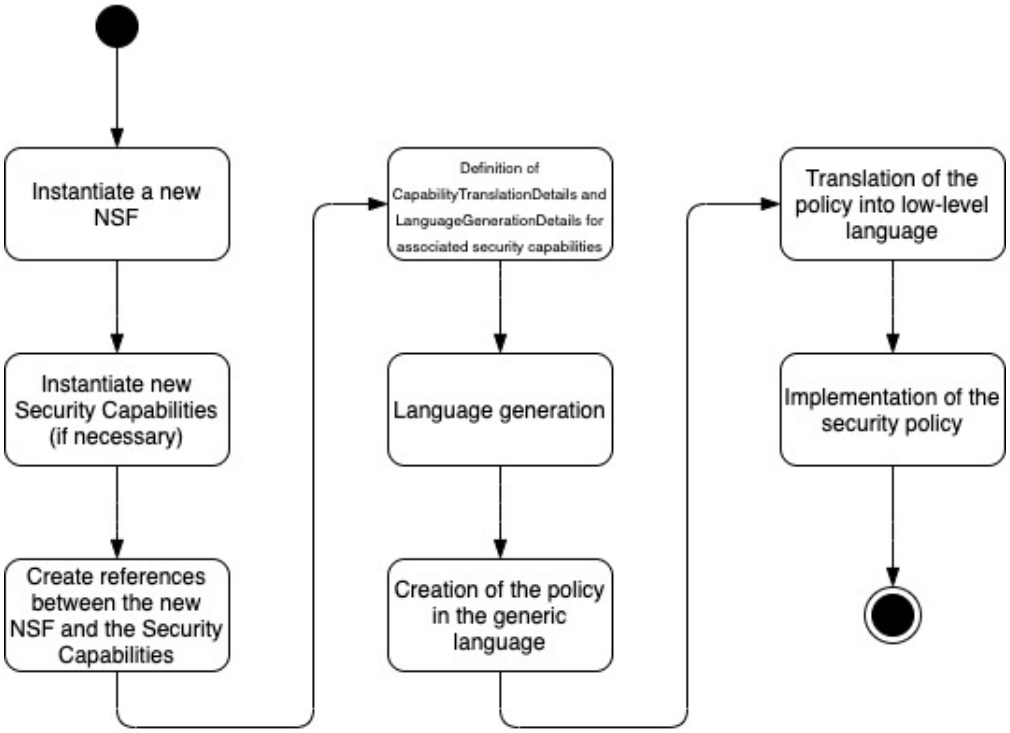

Figure 6.1: Workflow

6. *Creation of the policy in the generic language*: using the generic language of the NSF, the policy can be defined using this language;

7. *Translation of the policy into low-level language*: use of the proposed tool as part of the thesis that allows the translation of the policy expressed in the generic language of the NSF into the policy expressed in the low-level language and in the format necessary for the NSF;

8. *Implementation of the security policy*: use of the output file of the previous phase to directly take advantage of the policy translated into the NSF low-level language.

6.2 IPsec analysis

This section analyzes the previously described workflow with a concrete case of NSF network security features: authentication and encryption the packets of data.

This type of secure communication between two hosts on an IP network is guaranteed by IPsec.

IPsec (Internet Protocol Security) is a suite of protocols that provides security to Internet communications at the IP layer. The most common current use

of IPsec is to provide a Virtual Private Network (VPN), either between two locations (gateway-to-gateway) or between a remote user and an enterprise network (host-to-gateway); it can also provide end-to-end, or host-to-host, security[15]. The components required to provide security services at the IP layer are:

- SA (Security Association): a one-way (inbound or outbound) agreement between two communicating peers that specifies the IPsec protections to be provided to their communications. This includes the specific security protections, cryptographic algorithms, and secret keys to be applied, as well as the specific types of traffic to be protected[15];
- SPI (Security Parameters Index): a value that, together with the destination address and security protocol (AH or ESP), uniquely identifies a single SA[15];
- SAD (Security Association Database): each peer's SA repository. The RFC describes how this database functions (SA lookup, etc.) and the types of information it must contain to facilitate SA processing; it does not dictate the format or layout of the database. SAs can be established in either transport mode or tunnel mode[15];
- SPD (Security Policy Database): an ordered database that expresses the security protections to be afforded to different types and classes of traffic. The three general classes of traffic are traffic to be discarded, traffic that is allowed without IPsec protection, and traffic that requires IPsec protection[15].

IPsec protections are provided by two special headers: the Encapsulating Security Payload (ESP) Header and the Authentication Header (AH). In IPv4, these headers take the form of protocol headers; in IPv6, they are classified as extension headers[15].

The Authentication Header (AH) provides integrity protection; it also provides data-origin authentication, access control, and, optionally, replay protection. A transport mode AH SA, used to protect peer-to-peer communications, protects upper-layer data, as well as those portions of the IP header that do not vary unpredictably during packet delivery. A tunnel mode AH SA can be used to protect gateway-to-gateway or host-to-gateway traffic; it can optionally be used for host-to-host traffic. This class of AH SA protects the inner (original) header and upper-layer data, as well as those portions of the outer (tunnel) header that do not vary unpredictably during packet delivery. Because portions of the IP header are not included in the AH calculations, AH processing is more complex than ESP processing. AH also does not work in the presence of *Network Address Translation* (NAT) [15].

The IP Encapsulating Security Payload (ESP) provides confidentiality (encryption) and/or integrity protection; it also provides data-origin authentication, access control, and, optionally, replay and/or traffic analysis protection. A transport mode ESP SA protects the upper-layer data, but not the IP header. A tunnel mode ESP SA protects the upper-layer data and the inner header, but not the outer header [15].

IPsec can be implemented by the use of a framework for transforming packets *XFRM* and by *Strongswan*. The most important feature that differentiates them:

- XFRM: manual key exchange;
- Strongswan: automatic key exchange by IKE.

These two frameworks are supported equally and both are fully compatible. For simplicity, only the cases of XFRM will be treated within this thesis. For Strongswan the approach does not change but we may have only some slightly different capabilities due to the presence of an automatic key exchange by IKE. The examples below can also be used for Strongswan but of course we will have a different final output with a different language.

This paragraph reports in detail the most significant examples of the development of the concrete case analyzed without fully reporting all the steps, details and configuration files really necessary for the implementation carried out, in order not to burden the reading of the report.

The assumption on which the following phases are based is to have generated the file containing the CapIM, the CapDM and the "CapabilityTranslationDetails" and "LanguageGenerationDetails" classes in XMLSchema format using the appropriate tool with the command:

```
java -jar newConverter.jar definitivo.xmi
```

6.2.1 Study of capabilities

In the first phase, the functionalities offered by IPsec were studied in detail. Since IPsec has some specific characteristics, a section dedicated to key exchange (automatic and manual) and to packet exchange has been added to the model that represents the behavior of the protocol in a generic way.

Some specific security capabilities mapped into the CapDM after studying IPsec are shown in Figure 6.2.

These new security capabilities are all transformations, which is why they have been categorized as children of the abstract "TransformPacketDecorator" class. The latter is the decorator of the generic class "TransformPacketCapability" which

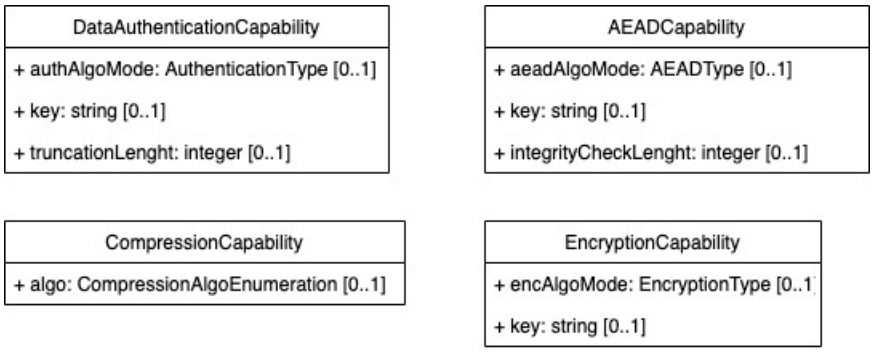

Figure 6.2: IPsec specific classes

is identified as "ActionCapability", a generic subclass of the 6-tuple in Figure 5.2. They have also been made generic and any NSF can use them if they are assigned to them.

6.2.2 Instantiate new Security Capabilities

After studying the security capabilities offered by the IPsec protocol, you can proceed to drafting the list of them in a generic format compatible with the translation of the CapDM. Since the use of XML language has been defined in the architecture as a generic support language, Figure 6.3 shows the assignment of some internal security capabilities to the NSF configuration XML file. Each security capability once instantiated must have a new specific name to refer to the 6-tuple.

```
<securityCapability id="IpSourceAddressConditionCapability"
    xsi:type="p:IpSourceAddressCapability"/>
<securityCapability
    id="IpDestinationAddressConditionCapability"
    xsi:type="p:IpDestinationAddressCapability"/>
<securityCapability id="DataAuthenticationActionCapability"
    xsi:type="p:DataAuthenticationCapability"/>
<securityCapability id="EncryptionActionCapability"
    xsi:type="p:EncryptionCapability"/>
<securityCapability id="AEADActionCapability"
    xsi:type="p:AEADCapability"/>
<securityCapability id="CompressionActionCapability"
    xsi:type="p:CompressionCapability"/>
<securityCapability id="ManualOperationActionCapability"
    xsi:type="p:ManualOperationCapability"/>
```

Figure 6.3: Security Capabilities instantiation example

The security capabilities chosen to show the example are:

- IpSourceAddressCapability: this generic security capability of the condition type allows you to identify the source IP address of a packet;
- IpDestinationAddressCapability: this generic security capability of the condition type allows you to identify the destination ip address of a packet;
- DataAuthenticationCapability: this IPsec specific security capability, but made generic, of the action type, allows you to declare which authentication algorithm I want to use for the specific packet;
- EncryptionCapability: this IPsec specific security capability, but made generic, of the action type, allows you to declare which encryption algorithm I want to use for the specific packet;
- AEADCapability: this IPsec specific security capability, but made generic, of the action type, allows you to declare which encryption with authentication algorithm and mode I want to use for the specific packet;
- CompressionCapability: this IPsec specific security capability, but made generic, of the action type, allows you to declare with which algorithm you want to perform compression on the specific packet;
- ManualOperationCapability: this specific IPsec security capability, but made generic, of the action type, allows us to declare how we want to act on the Security Association Database;

6.2.3 Instantiate a new NSF and create references

Now it is necessary instantiate a new NSF and create the references between the security capabilities within the catalogue and the NSF itself. Precisely for this reason each security capability has had a new name, useful to create the reference with the NSF. As shown in Figure 6.4, instantiating a new NSF is done by declaring the name of the same and then declaring the references to the security capabilities through the name created.

Subsequently, the configuration file of the security capabilities to be assigned to an NSF can be validated using the specific validation tool. Since the transcription of the validation file for the verification of the security capabilities has been done by hand, the validation avoids transcription errors or errors due to non-existent security capabilities in the model. In this phase the security capabilities are used in an abstract way and only according to the name, in fact it is not necessary to define or specify the attributes of each security capability since they are defined in the CapDM.

```
<nSF id="XFRM">
    <securityCapability
        ref="IpSourceAddressConditionCapability"/>
        <securityCapability
            ref="IpDestinationAddressConditionCapability"/>
        <securityCapability
            ref="DataAuthenticationActionCapability"/>
        <securityCapability ref="EncryptionActionCapability"/>
        <securityCapability ref="AEADActionCapability"/>
        <securityCapability ref="CompressionActionCapability"/>
        <securityCapability
            ref="ManualOperationActionCapability"/>
</nSF>
```

Figure 6.4: NSF instantiation example

6.2.4 Definition of CapabilityTranslationDetails and LanguageGenerationDetails for associated security capabilities

After studying the security capabilities offered by the IPsec protocol, we learned its syntax and semantics, which allowed us to instantiate the CapabilityTranslationDetails and LanguageGenerationDetails classes necessary for the tools for language generation and policy translation. These classes allow you to specify details about the security capability in the specific NSF system.

These two classes are instantiated separately to the NSF and the security capability. The connection with the latter is made as they inherit the association towards them from the parent class "HasSecurityCapabilityDetail". In this way, in the instance of one of the two classes, it is necessary to specify the reference both to the NSF and to the security capability. The LanguageGenerationDetails class allows you to customize the generation of the NSF language relative to the single security capability. The CapabilityTranslationDetails class, on the other hand, allows you to specify the details of translation and dependencies of the single security capability within the NSF. The definition of the details of the CapabilityTranslationDetails class affects the semantics and the actual structure of the translation of the security capabilities. Therefore, for the definition of the characteristics of an CapabilityTranslationDetails, the purpose and use that will be made of the translation of the policy must also be taken into account.

The following examples refer to some of the previously listed security capabilities.

In the context of IPsec, the command needed to work on SAD is expressed as *"state mode"*. To allow this translation the Adapter class will be expressed as shown in Figure 6.5. The main element of this Figure is the definition of the "realCommandName" field, where the specific semantics of the NSF are expressed to declare the security capability to which the CapabilityTranslationDetails class is assigned. In addition to the declaration of the command, the dependence on the fact that the "policy" command (used to operate on the Security Policy Database) must be absent has been expressed. The names used to express dependencies are those declared by the CapDM for security capabilities.

```
<capabilityTranslationDetails>
    <nSF ref="XFRM"/>
    <securityCapability ref="ManualOperationActionCapability"/>
        <commandName>
                <realCommandName>state</realCommandName>
        </commandName>
        <dependency>
                <absenceOfValue>policy</absenceOfValue>
        </dependency>
</capabilityTranslationDetails>
```

Figure 6.5: ManualOperationActionCapability

To analyze a more complex case of CapabilityTranslationDetails, reference can be made to the "DataAuthenticationActionCapability" security capability.

In the case of IPsec, this command is used to declare the algorithm, mode and key to authenticate the packet. Furthermore, this command can also be used to perform truncation by means of its command variant. In this case, the truncation length must also be specified. This condition is expressed as *"auth — auth-trunc mode (algoHash) key [truncationLenght]"*. In order to map this type of translation, the CapabilityTranslationDetails is created in Figure 6.6. From this definition we can see the presence of two "realCommandName", which represent the translations of the security capability in authentication and authentication with truncation. In this way the tool is able to use both translations and will use the right one according to the value of the "operation" parameter. Furthermore, a further difference with respect to Figure 6.5 concerns the definition of the separator, in fact the command allows you to define the algorithm between the round brackets after defining the mode and define the key after defining the algorithm by means of a space. In this way the separators between attributes can be expressed. To allow the identification of which attributes are to be separated, special constructs have been specified, in

which the values of the parameters to refer to are entered. These parameters are expressed with the nomenclature defined in the CapDM.

```
<capabilityTranslationDetails> <nSF ref="XFRM"/> <securityCapability
    ref="DataAuthenticationActionCapability"/>
        <commandName> <realCommandName>auth</realCommandName>
            </commandName>
        <commandName> <realCommandName>auth-trunc</realCommandName>
                <commandNameCondition>
                        <attributeName>operation</attributeName>
                        <attributeValue>NOT_EQUAL_TO</attributeValue>
                </commandNameCondition> </commandName>
        <bodyConcatenator> <realConcatenator>(</realConcatenator>
                <concatenatorCondition>
                        <preVariable>mode</preVariable>
                        <postVariable>algoHash</postVariable>
                </concatenatorCondition>
                <postConcatenator>)</postConcatenator>
                </bodyConcatenator>
        <bodyConcatenator> <realConcatenator> </realConcatenator>
                <concatenatorCondition>
                        <preVariable>algoHash</preVariable>
                        <postVariable>key</postVariable>
                </concatenatorCondition> </bodyConcatenator>
        <bodyConcatenator> <realConcatenator> </realConcatenator>
                <concatenatorCondition>
                        <preVariable>key</preVariable>
                        <postVariable>truncationLenght</postVariable>
                </concatenatorCondition> </bodyConcatenator>
        <dependency>
                <presenceOfCapability>
                IpProtocolTypeConditionCapability
                </presenceOfCapability>
                <presenceOfValue>ah</presenceOfValue>
        </dependency>
        <dependency>
                <presenceOfCapability>
                IpProtocolTypeConditionCapability
                </presenceOfCapability>
                <presenceOfValue>esp</presenceOfValue>
        </dependency>
</capabilityTranslationDetails>
```

Figure 6.6: DataAuthenticationActionCapability

The complete file is validated using the appropriate tool against the XSD file containing the CapDM. This file will be used for the generation of the NSF language.

6.2.5 Language generation

In the language generation phase, the tool proposed in the thesis project is used, created by Noceti [14] but modified specifically for the purpose. In order to use the language generation tool, the catalogue file generated (or modified) in the previous step and the file containing the representation in XMLSchema format of the CapDM are required. For example, in the case considered, the following command can be used:

```
java -jar newLanguage.jar capability_data_model.xsd
    NSFCatalogue.xml XFRM
```

where the passed parameters correspond to:

- capability_data_model.xsd: represents the path to the file that contains the complete list of security capabilities mapped in the CapDM and which contains the definition of the "CapabilityTranslationDetails" and "LanguageGenerationDetails" classes;
- NSFCatalogue.xml: represents the path to the file created in the previous paragraphs.
- XFRM: represents the name of the NSF for which we want the language

This command generates the specific language of the NSF with the standard name "language_XFRM.xsd". The language is considered specific because it is generated mainly by modeling the security capabilities in the CapDM for the NSF concerned. It is considered the specific language of the NSF because it is characterized by the choice of the security capabilities it knows and by any further customization. In this way only security capabilities instantiated for the specific NSF are inside the new language. Furthermore, the language generation tool eventually generates text files containing metadata regarding the associations of numerical values with respect to alphabetic values of the enumerations that require such matching.

6.2.6 Creation of the policy

In the policy creation phase, the main information to keep in mind is the language of the NSF to which you want to assign the policy. The Figure 6.7 shows a valid rule of a possible policy in the example NSF language. Considering the case of IPsec, the proposed rule is divided into the following parts:

- nsfName: you have to specify the name of the NSF we are referring to. In this way, during the translation phase, the tool will search among the security

```
<policy nsfName="XFRM"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="language_XFRM.xsd">

    <rule>
        <ruleDescription></ruleDescription>
        <manualOperationActionCapability>
            <operationType>add</operationType>
        </manualOperationActionCapability>
        <ipSourceAddressConditionCapability>
            <ipAddress>
                <address>192.168.1.1</address>
            </ipAddress>
        </ipSourceAddressConditionCapability>
        <ipDestinationAddressConditionCapability>
            <ipAddress>
                <address>192.168.1.2</address>
            </ipAddress>
        </ipDestinationAddressConditionCapability>
        <ipProtocolTypeConditionCapability>
            <protocolType>
                <protocolTypeName>esp</protocolTypeName>
            </protocolType>
        </ipProtocolTypeConditionCapability>
        <policySpiConditionCapability>
            <spi>0x100</spi>
        </policySpiConditionCapability>
        <encryptionActionCapability>
            <encAlgoMode>
                <algoEnc>aes128</algoEnc>
            </encAlgoMode>
            <key>0x00112233445566778899AABBCCDDEEFF</key>
        </encryptionActionCapability>
    </rule>
</policy>
```

Figure 6.7: Instance of rule in the language of the NSF.

capabilities connected to the reference NSF as each security capability can have different translations based on the NSF;

- rule: parameter indicating the start of the rule. This is inserted for each rule since in the file we can write more rules for the same NSF;
- ruleDescription: parameter that is used to enter a description for the following rule. At the moment no function has been implemented on it, it has been

included for future developments;

- manualOperationActionCapability: with this rule we indicate that we are referring to the indicated security capability. The "operationType" parameter indicates what type of operation we want to perform on the Security Association Database (SAD). When we want to insert something between the parameter, a list of possibilities will come out and in this case we choose the "add" operation;

- ipSourceAddressConditionCapability: using this security capability you can define on which source IP address you want to provide a condition, the address referred to can be defined in the "address" attribute and possibly its mask in the "mask" attribute. In this case, the IP address "192.168.1.1" was defined without a mask;

- ipDestinationAddressConditionCapability: using this security capability you can define on which destination IP address you want to provide a condition, the address referred to can be defined in the "address" attribute and possibly its mask in the "mask" attribute. In this case, the IP address "192.168.1.2" was defined without a mask;

- ipProtocolTypeConditionCapability: using this security capability you can define on which protocol you want to provide a condition, the protocol to which reference is made is defined in the "protocolTypeName" attribute because it is expressed with a valid name with respect to the language. In this case we want to carry out a transformation of the package to guarantee security through encryption and authentication. To do this we choose the "esp" protocol from the list of choices that is granted to us by the language itself;

- policySpiConditionCapability: with this parameter we insert the Security Parameter Index (SPI). The latter is an identifier used in the IPsec protocol to identify IPsec Security Associations. In this case, being manually established as XFRM foresees, the identification number is entered in the "spi" field;

- encryptionActionCapability: using this security capability we indicate which algorithm and which mode in the "encAlgoMode" field we want to use to ensure security. While the key with which the packet is encrypted is entered in the "key" field. In this case the "aes128" algorithm is chosen without specifying the mode. The key, on the other hand, is expressed in hexadecimal.

6.2.7 Translation of the policy

The translation phase of the policy is strictly dependent on how you want to use the output file of the policy translation tool. In addition to the CapabilityTranslationDetails class, the translation tool was designed to allow output types of different formats, the details of using the tool will be analyzed in more detail in section A.3.

Since there may be different formats available through which an NSF can receive the policies or rules to be implemented, it is necessary to study the various formats also in order to decide which is more convenient to use. Depending on this, different types of CapabilityTranslationDetails classes can be defined and the translation tool can be used, in specific ways, to generate different translation structures.

In the example of IPsec that we are dealing with, we have analyzed the case in which we want to be able to have more rules inside the output file so that we can then use them on the command line Figure 6.8.

```
ip xfrm state add src IPSrc dst IPDst proto esp spi 0x1000 enc
    aes 0x00112233445566778899AABBCCDDEEFF
```

Figure 6.8: NSF language rule instance for command line configurations.

This format allows you to generate a file that contains a single autonomous rule for each line. This file can be used to verify the possibility of entering each rule directly from the command line. The keyword “ip xfrm” was passed to the translation tool as one of the specific optional parameters made available by the tool itself, used in the command line to start the rule. This parameter allows you to generate an output containing this keyword as the starting element for each translated rule.

To obtain this type of format using the information in the example, the translation tool was used with the following command:

```
java -jar newTranslator.jar language.xsd NSFCatalogue.xml
    XFRM_RuleInstance.xml "+sip xfrm "
```

where the passed parameters correspond to:

- newTranslator.jar: translation tool created specifically to translate from the generic language into the specific language of the security protocol;
- language.xsd: represents the path to the file that contains the NSF language;

- NSFCatalogue.xml: represents the path to the file created in 6.2.2, 6.2.3 and 6.2.4;
- XFRM_RuleInstance.xml: represents the path to the file created in 6.2.6;
- "+sip␣xfrm␣": it allows you to insert the keyword "ip␣xfrm␣" at the beginning of each rule, the quotation marks have been put specifically to insert the space character as well.

This command generates a text file with the standard name "policy_ipxfrm.txt" with the rule translated as in Figure 6.8.

6.2.8 Implementation of the security policy

In order to verify the actual compatibility with IPsec, two virtual machines with Ubuntu operating system have been allocated. After that, an attempt was made to exchange packets between them. Once the basic exchange has been carried out successfully, we tried to use the IPsec protocol in the XFRM manual, giving security to the exchanged packet, using the rules created in the previous paragraphs.

Chapter 7

Conclusion

The study examined the information model defined by the Interface To Network Security Functions working group and defined a data model by analyzing certain security capabilities for network services. Together with this, the initial information model has been extended to allow the management of security policies for network services, making this model compatible with a policy-based management approach, that is an approach of network administration through policy-based management.

The study carried out has expanded a method already developed by the previous graduate student to represent, in a generic way, the functions offered by the security services. Using this method for the generic representation of the capabilities of the devices, it was possible to carry out several essential activities both in the field of research and in the field of network security administration. During the study, the problem was faced that each element that applies security features uses its own language, therefore an abstract language for security checks was used, which allows to express security policies with a common syntax.

The language definition criterion is based on the security capabilities assigned to the NSF specification, in this way an abstract and generic language is obtained, but specific for each NSF present in a specific Catalogue, containing all the NSFs with the relative security capabilities instantiated. This language allows you to specify security policies with a generic syntax, but this would not have solved the problem of being able to use that specific security protocol through this abstract language. Because of this, it was necessary to translate the policy into the concrete language of the security feature's actuator. For this reason, a valid methodology had to be used to avoid having to generate a specific translator for each low-level language.

To allow the generation of an abstract language, a new tool (*newLanguage.jar*) has been developed, improving the one previously used, which, starting from a catalogue, could provide us with the desired language for the NSF specification.

Subsequently, a new tool (*newTranslator.jar*) was created that allowed translation into the specific language.

Starting from the environment specified by I2NSF, using highly known techniques and languages such as UML and XML and using specific patterns, it was possible to obtain a generic, easily understandable and widely usable representation of security capabilities, adding these specifications to the pre-existing one.

Taking into consideration the data of the previous graduate student, it was decided to change the approach to adequately express the parameters of the association classes regarding the languageGenerationDetails and capabilityTranslationDetails classes. Through these two classes, the proposed tools adapt the generation of the abstract language of the NSF and the consequent translation into the specific low-level language of the NSF. The main difficulty is encountered only during the assignment of the aforementioned classes when they must be referred to a security capability of the NSF. Once the configuration, present in the catalogue, has been defined, the proposed tools adapt to the inputs, thanks to a Model-driven approach. Another advantage brought by the definition of the abstract language dependent on security capabilities is the portability of the policies defined in this language. In fact, the translation of policy into the concrete language of the device depends only on the characteristics of the capabilityTranslationDetails class: for two NSFs to which the same policy is assigned, the abstract language allows the exchange of the policy itself, since it is defined in the same way in the generic language of both NSFs but from which a different translation is obtained.

Using the proposed CapDM, a description of the security capabilities that is broad enough to cover the IPsec protocol work cases was obtained. The solution used can theoretically be applied to any NSF that can map its security capabilities to those expressed in the CapDM. Furthermore, the use of the Model-driven approach offers the possibility to add, modify, remove security capabilities from the CapDM allowing the direct applicability of the latter to characterize the NSF instances, without having to change the code of the tools but only the model. In the event that a new security capability is added to the CapDM, in order to use it, it is sufficient to assign it to the NSF and define the classes that allow the translation and generation of the language related to the new capability, in this way the proposed tools work without need to change the code.

During this thesis path, two NSFs were added to the CapDM proposed by another graduate student previously. The tools have been modified and made more efficient in order to have a generic situation with multiple NSFs present in a catalogue. After that, a new way of retrieving translation and language details was developed so that NSF and security capability could be referenced in XML,

without the possibility of making mistakes. Furthermore, the possibility of having a tool that generates a specific language for the NSF passed as a parameter in the request has been implemented (*newLanguage.jar*).

From the current situation of the tools, we can think that future developments can be concentrated on adding new NSFs starting from the definition of new security capabilities or referring to existing ones. Or you can place the proposed work in a context of centralized management of network controls. A central controller can be allowed to manage the insertion of new security capabilities in the CapDM to allow an insertion method or a standardized modification method. This controller can handle the possible known NSFs and the instances currently active in the network it controls. The controller can also manage the tools proposed to allow centralized control of their use to verify any permissions of use and the correct expression of the commands to be used. We can think of introducing an automatic choice of NSF based on the level of security or the purposes requested by the customer.

Appendix A

User Manual

This section shows how a user can use the proposed tools. The only fundamental prerequisite for using these tools is to have JAVA installed with a minimum version 1.8, and possibly the concrete NSFs to which the generated configurations can be assigned. Being executable in JAR format it is not necessary to install further programs. The tools are used via the command line using the syntax:

```
java -jar jar_name
```

Otherwise JAR files can be included in JAVA projects in order to use their functionality.

A.1 Transformation tool from XMI to XSD

This tool allows the transformation of an XMI file into an XMLSchema file following the characteristics with which Modelio transforms a class diagram into an XMI file. In the architecture of the thesis scope, this tool is used to allow the generation of an XMLSchema file containing all the security capabilities and related details, represented in the class diagram of the security capabilities data model (CapDM) and the information model (CapIM) containing the aggregation between NSF and security capabilities.

A.1.1 Command line use

To invoke the tool using the command line, the following syntax is used:

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

Where the parameters have the following function:

- input_path: path of the XMI file to convert;
- output_path: path where to generate the output file. This parameter is optional, if it is not expressed, the output file is created in the current directory of the executable with the file name determined in a standard way.

The execution of the command can be successful or not.

When the execution is successful, an XMLSchema file is generated in the position represented by "output_path" if present with name *capability_data_model.xsd* and the result is confirmed with a screen printout. During the generation of the output file, the "xmlns" and "targetNamespace" fields of the file are set using the values passed by "output_path", consequently to refer to this XMLSchema it is necessary to use the "targetNamespace" and the path indicated.

When the execution is unsuccessful, the command generates a response containing details on the relative type of error.

A.1.2 Use as a JAVA library

This implementation did not serve the purpose of the project but is easy to apply for those who will need it in the future.

A.2 NSF language generation tool

This tool allows the generation of an XMLSchema file containing the language of all NSFs. The output file is generated using an XML format file that is valid against the XMLSchema file containing all the security capabilities that can be assigned to its instantiated NSFs. In the thesis architecture, this tool is used every time a new NSF is added or security capabilities have been assigned, removed or added to an existing NSF. This tool can make use of files with metadata capabilities. The folder containing such metadata must be called "metadata" and must be in the same folder that contains the executable.

A.2.1 Command line use

To invoke the tool using the command line, the following syntax is used:

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

Where the parameters have the following function:

- xsd_path: path to the reference XSD file, containing the complete list of existing security capabilities (for example the file generated by the previous command (*capability_data_model.xsd*));
- nsfCatalogue_path: path of the XML catalogue file containing all the instances of the desired security capabilities with the NSF instances and the related instances of *languageGenerationDetails* and *capabilityTranslationDetails*;
- nsfName: name of the NSF;
- output_path: path where to generate the output file. This parameter is optional, if it is not expressed, the output file is created in the current directory of the executable with the file name determined in a standard way.

The execution of the command can be successful or not.

When the execution is successful, an XMLSchema file is generated with name *language_nsfName.xsd* containing the language of the NSF, in the position represented by the relative parameter and the result is confirmed with a screen printout. During the generation of the file, the XML file passed as parameter2 is used to get the information of any languageGenerationDetails classes defined to specialize the NSF language. This tool may eventually generate some metadata files as well. This happens when a new enumeration is created in class languageGenerationDetails or an existing one is modified. For this purpose, a new metadata file is generated with the name of the reference NSF. Within these files you can find information about the relationship between the numerical value and the alphabetic value of certain enumerations. Any metadata files will be in the same location as the output XMLSchema file.

When the execution is unsuccessful, the command generates a response containing details on the relative type of error in output, on the command line.

A.2.2 Use as a JAVA library

To use the tool as a JAVA library in a project, you can instantiate an object of the "LanguageModelGenerator" class and call the "generateLanguage" function.

The definition of the "LanguageModelGenerator" function are as follows:

```
public␣LanguageModelGenerator(String␣xsd,␣String␣xml,␣String␣
    nsfName,␣String␣outputName)
```

It allows to instantiate an object of type "LanguageModelGenerator" with parameters:

- xsd: path to the XMLSchema file containing the complete list of security capabilities;
- xml: path of the XML catalogue file containing all the instances of the desired security capabilities with the NSF instances and the related instances of *languageGenerationDetails* and *capabilityTranslationDetails*;
- nsfName: name of the NSF for which we want to create the new language;
- outputName: path and name of the output file.

The definition of the "generateLanguage" function are as follows:

```
public boolean generateLanguage()
```

implements the actual conversion of the files with which the instance was generated with a boolean return value indicating the success or failure of the execution.

A.3 NSF low-level language translation tool

This tool allows the generation of a text file containing the policy expressed in the NSF low-level language. The output file is generated according to the security capabilities present in the security policy expressed in the generic language of the NSF. In the thesis architecture, this tool is used every time you want to translate a policy for a specific NSF.

A.3.1 Command line use

To invoke the tool using the command line, the following syntax is used:

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

Where the parameters have the following function:

- xsd_path: path to the reference XSD file, containing the complete list of existing security capabilities (for example the file generated by the previous command (*language_nsfName.xsd*));
- nsfCatalogue_path: path of the XML catalogue file containing all the instances of the desired security capabilities with the NSF instances and the related instances of *languageGenerationDetails* and *capabilityTranslationDetails*;

- nsfRule_path: path of the XML file containing the instance of the policy to translate;
- output_path: path where to generate the output file. This parameter is optional, if it is not expressed, the output file is created in the current directory of the executable with the file name determined in a standard way.
- parameter5: optional parameter that allows you to add a static alphanumeric part to the output at the beginning of each rule, this parameter must begin with the sequence of characters "+s";
- parameter6: optional parameter that allows you to add a static alphanumeric part to the output at the end of each rule, this parameter must begin with the sequence of characters "+e";
- parameter7: optional parameter to force the generation of valid rules even if some policy rules do not respect certain dependency criteria, it is expressed as "-f";
- parameter8: optional parameter that allows you to wrap after the initial parameter of each rule (ie parameter5), this parameter must begin with the sequence of characters "+crs";
- parameter9: optional parameter that allows you to wrap before the final parameter of each rule (ie parameter6), this parameter must begin with the sequence of characters "+cre".

The order in which the parameters are entered is only important for the first three, the optional parameters can be entered in any order.

When the execution is successful, a text file is generated with the name "policy_nameofNSF" containing the translation into the low-level language of the NSF of the policy expressed in the generic language of the NSF. The file is generated in the location as expressed by parameter4. The format of the content of the output file is strongly influenced by the presence or absence of optional parameters passed to the command.

When the execution is unsuccessful, the command generates a response containing details on the relative type of error.

A.3.2 Use as a JAVA library

To use the tool as a JAVA library in a project, you can instantiate an object of the "NSFTranslatorAdapter" class and call the "translate" function.

The definition of the "translate" function are as follows:

```
public void translate(String xsd, String xmlAdapter, String
    xmlPolicy, String outputName, String startString, String
    endString, String forced, boolean scr, boolean ecr)
```

Where the parameters have the following function:

- xsd: path to the reference XSD file, containing the complete list of existing security capabilities;
- xmlAdapter: path of the XML catalogue file containing all the instances of the desired security capabilities with the NSF instances and the related instances of *languageGenerationDetails* and *capabilityTranslationDetails*;
- xmlPolicy: path of the XML file containing the instance of the policy to translate;
- outputName: path where to generate the output file. This parameter is optional, if it is not expressed, the output file is created in the current directory of the executable with the file name determined in a standard way.
- startString: optional parameter that allows you to add a static alphanumeric part to the output at the beginning of each rule;
- endString: optional parameter that allows you to add a static alphanumeric part to the output at the end of each rule;
- forced: optional parameter to force the generation of valid rules even if some policy rules do not respect certain dependency criteria;
- scr: optional parameter that allows you to wrap after the initial parameter of each rule (ie startString);
- ecr: optional parameter that allows you to wrap before the final parameter of each rule (ie endString).

A.4 Validation tool

This tool allows the validation of an XML file against an XMLSchema file both passed as parameters to the tool. In the architecture of the thesis, this tool is used whenever an XML file necessary for one of the proposed tools needs to be instantiated.

A.4.1 Command line use

To invoke the tool using the command line, the following syntax is used:

```
java -jar validate.jar xsd_path xml_path
```

Where the parameters have the following function:

- xsd_path: path to the referenced XML Schema file;
- xml_path: path of the XML file to validate.

The execution of the command can be successful or not.

When the execution is successful, the string "true" is returned. Otherwise, in case of failure, the command generates a response containing details on the relative type of error in output, on the command line.

A.4.2 Use as a JAVA library

To use the tool as a JAVA library in a project, you can instantiate an object of the "Validation" class and call the "validate" function.

The definition of the "validate" function are as follows:

```
public boolean validate(String xsd, String xml)
```

Where the parameters have the following function:

- xsd: path to the reference XSD file;
- xml: path of the XML file to validate;

Return value: Boolean value that indicates the success or failure of the function execution.

Appendix B

Developer Manual

This section contains the specifications of the developer's manual relating to the capability model and to each tool, in particular the architectures of the tools and the various functions that make up the code of each tool will be explained in detail. The common feature of the tools is that they are all written in JAVA. The APIs offered by Java API for XML Processing (JAXP) were used to manage XML files. The added libraries belong to the "Apache Xerces Project" project, downloadable directly from the project site[1], the specific package used is "xerces-2_6_2" and the libraries present in the projects are:

- resolver.jar;
- serializer.jar;
- xercesImpl.jar;
- xml-apis.jar.

B.1 CapDM management using Modelio

The proposed project for the modification of the CapDM is developed in a set of diagrams of the classes related to each other. Starting from the definition of the more generic element, namely the "SecurityCapability" class, the first hierarchy was generated, that is the division into the tasks defined by the 6-tuple as in Figure 5.2. Using the generalization relationship, further subdivision classes were generated, up to "leaf" elements, which determine a concrete security capability. We started from the CapDM generated by Noceti [14] in his thesis project, in order to improve

[1] http://xerces.apache.org/mirrors.cgi

it and make it compatible with the additions that have been made within this thesis and that can be made in the future.

The relationships between the multiple class diagrams are represented by Figure B.1, which also shows the fact that the Capability Information Model (CapIM, Figure 5.1) uses the security capabilities defined in the Capability Data Model (CapDM). To define the specific security capability, we start from the base in the definition of the 6-tuple. Once the generic scope of the security capability is found, it is assigned to that particular group. After that, you can think about creating a more specific (abstract) subgroup before creating the actual capability. This is defined as "Leaf Capability" as it is the last in the chain starting from the CapDM. This ability can use the types that are defined in the class diagram specifically defined for types.

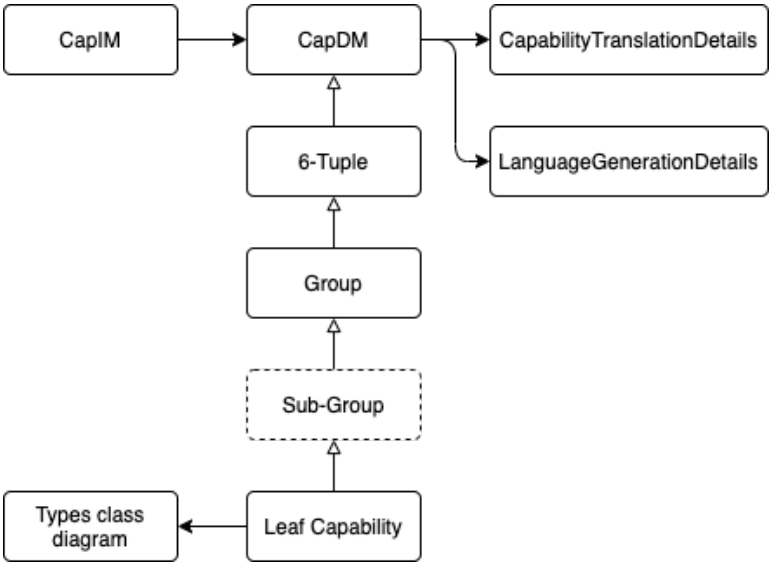

Figure B.1: Relationship between class diagrams

Add a new capability

To add a new capability you can proceed with the following steps:

- identify which group this capability belongs to;
- identify the class diagram used for that capability group;
- create a new class in the class diagram of the identified group;
- define the new class with generic name;
- define any details about the attributes needed to manipulate the new capability.

If the attributes defined for the new class are not of generic type, such as string or integer type, you can define a new generic type by proceeding with the following steps:

- identify how to compose the new type;
- create a new class in the type class diagram;
- define the new class with generic name;
- define the details of the new attributes needed to manipulate the new capability;
- assign the type of the class just defined to the parameter of the new capability created previously.

In this way, standard types can be defined that each new security capability can use.

B.2 XMI to XSD transformation tool

This tool allows the transformation of an XMI file into an XML-Schema file following the characteristics with which Modelio transforms a class diagram into an XMI file. In the thesis architecture, this tool is used to allow the generation of an XML-Schema file containing all the security capabilities and related details, represented in the class diagram of the security capability data model (CapDM). The tool previously developed by Noceti [14] has been modified and improved to allow the transformation not only of the CapDM but also of the CapIM. The tool develops the output following the workflow in Figure B.2.

The main steps in the operation of this tool are as follows:

- reading the input file;
- cycle for each element. The reference point of the tool is a tag called "packagedElement" which contains all the classes generated in that folder, whose type determines the type of element being analyzed;
- when a correct element is recognized, an element of type complexType is generated to be inserted in the output file;
- each generic element can be made up of further elements, so each element is explored to find the attributes useful for generating the complex type necessary for the output file;

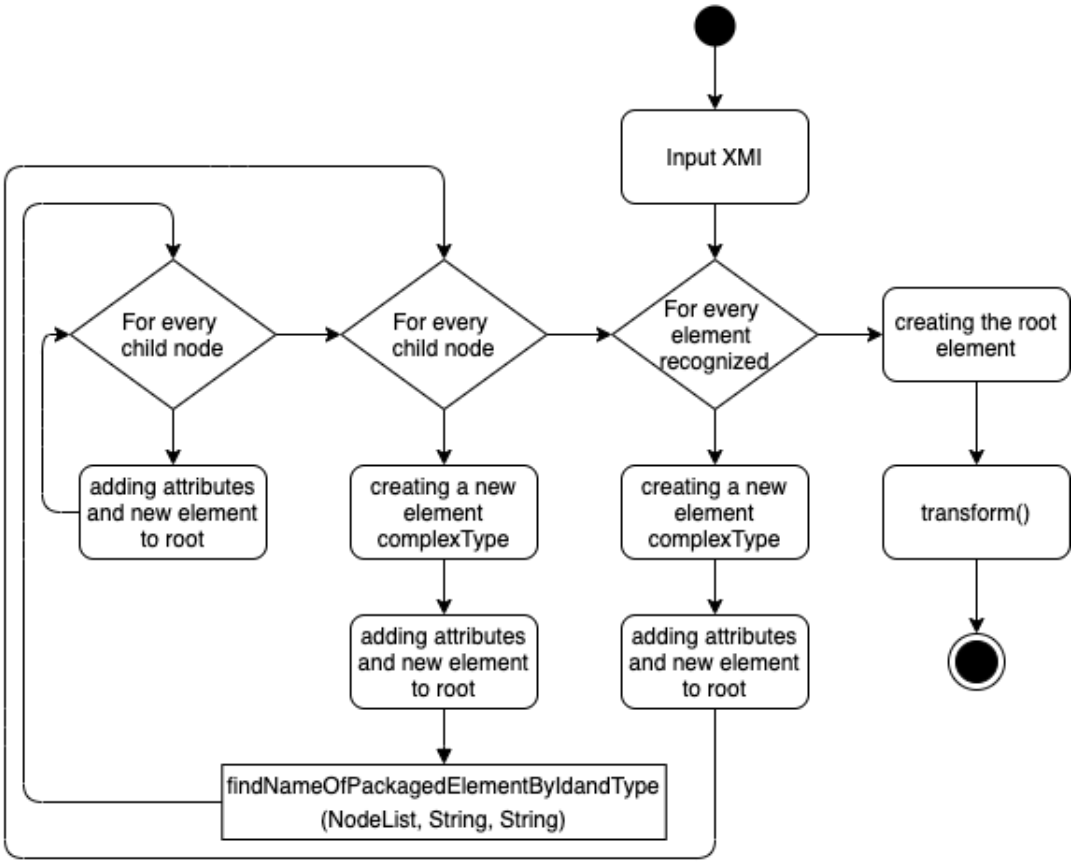

Figure B.2: XMI to XSD Transformation tool Workflow

- once the translation of each element in the input file has been completed, a concrete element is generated that functions as the root element of the XML-Schema file;
- once the whole structure has been created, the "transform()" function takes care of the concrete generation of the output file.

Tool architecture

The architecture of the proposed tool is structured in two main classes, represented in Figure B.3, and a support class for the creation of objects to be included in the document; these classes are composed as explained below:

- Converter: class that deals with the management of the input file and recognizes the details useful for creating the elements for the output file. This class does not define instance variables. This class consists of the following functions:
  - **public** Converter():

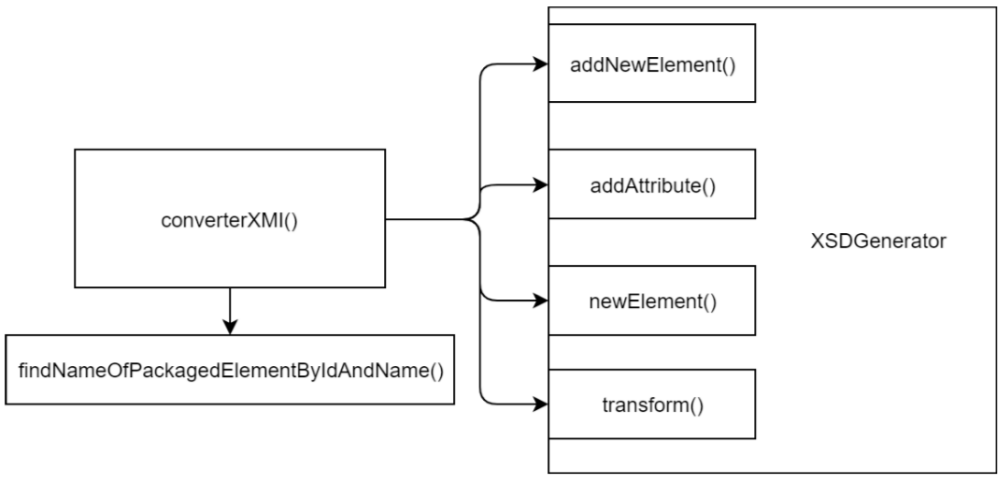

Figure B.3: Design of the XMI to XSD Transformation Tool.

constructor function that allows you to instantiate an object corresponding to the "Converter" class. It takes care of reading the input file, recognizing the elements useful for generating the output and interacts with the other functions for the purpose of the tool;

- **private** String findNameOfPackagedElementByIdandType(NodeList nl, String id, String classe):
  this function deals in detail with finding the name, in string format, of an element belonging to the list of nodes passed as a parameter. To find the correct name, the element that has the value "class" in the "xmi: type" attribute and the "id" value in the "xmi: id" attribute is searched in the list of nodes passed. If the search is not successful, "null" is returned.

- XSDgenerator: class that deals with the management of elements recognized and converted with the XMLSchema format, also deals with the actual transformation of the element generated when reading the input file and the actual generation of the output file. It has a main element which forms the root of the output file. This class defines the following instance variables:
  - **private final static** String NS_PREFIX = "xs:":
    this field allows you to define with which prefix all the elements will be generated;
  - **private** Document doc:
    this field contains the output document to which this instance refers;
  - **private** Element schemaRoot:
    this field contains the root element of the instance output document;
  - **private** NameTypeElementMaker elMaker:

this field contains the element that allows you to generate new elements compatible with the output document.

This class consists of the following functions:

- **public** XSDgenerator():
  constructor function that allows you to instantiate an object corresponding to the "XSDgenerator" class and takes care of creating the main objects of the class;
- **public boolean** transform(String outputName):
  this function deals with the generation of the output relative to the new XSD file using the "outputName" parameter as a reference for the name and destination path of the output file;
- **public** Element newElement(String name):
  this function generates a new element, the "name" parameter is used to give the name to the element to be generated;
- **public void** addAttribute(Element element, String nameAttr, String attrValue):
  this function is responsible for adding to the element "element" an attribute named "nameAttr" and the value "attrValue";
- **public void** addNewElement(Element e):
  this function adds a new element to the root of the schema to be generated;

- NameTypeElementMaker: support class for the "XSDgenerator" class that provides two functions for managing objects in the desired document. This class defines the following instance variables:
  - **private** String nsPrefix:
    this field allows you to define with which prefix all the elements will be generated;
  - **private** Document doc:
    this field contains the output document to which this instance refers.

  This class consists of the following functions:
  - **public** NameTypeElementMaker(String nsPrefix, Document doc):
    this constructor allows you to define which document the class refers to and if a prefix string exists;
  - **public** Element createElement(String elementName):
    this function creates an element in the document assigned to the class

correctly, adding the prefix assigned to the class to the "elementName" parameter;

– **public void** setAttribute(Element element, String nameAttr, String attrValue):
this function is responsible for adding to the element "element" an attribute named "nameAttr" and the value "attrValue".

B.3 NSF language generation tool

This tool allows the generation of an XMLSchema file containing the language of all the NSF instanced. The output file is generated according to the security capabilities assigned to the NSF using a valid XML format file with respect to the XMLSchema file containing all the assignable security capabilities. In the architecture of the thesis, this tool is used every time we add, remove or modify the security capabilities of a NSF or instantiate a new NSF, so as to have the new language for the specific NSF always updated. This tool can make use of files with metadata capabilities. The folder containing such metadata must be called "metadata" and must be in the same folder that contains the tool or the tool source. The metadata consists of the relationship between integer and string value of any enumerations that require this relationship, and are structured so that for each row there is first the numeric value and, separated by a space, the alphabetic value. The proposed metadata files concern protocols and service ports.

The tool develops the output following the workflow below (Figure B.4).

The main steps in the operation of this tool are as follows:

- validation of input files;
- creation of the complex type "Policy" which will be used to instantiate the policies in the rules generation file;
- cycle for each security capability present in the NSF configuration file. The reference point of the tool is the tag called "securityCapability" which allows to recognize each security capability defined;
- for each securityCapability element, the new element used as an identifier to create the relationship between the NSF and the detail classes is generated;
- creation of the complex type "Rule" which will be used to indicate where starts a new rule in the rules generation file;
- cycle for each security capability present in the NSF configuration file;
- creation of the complex type for each instantiated security capability;

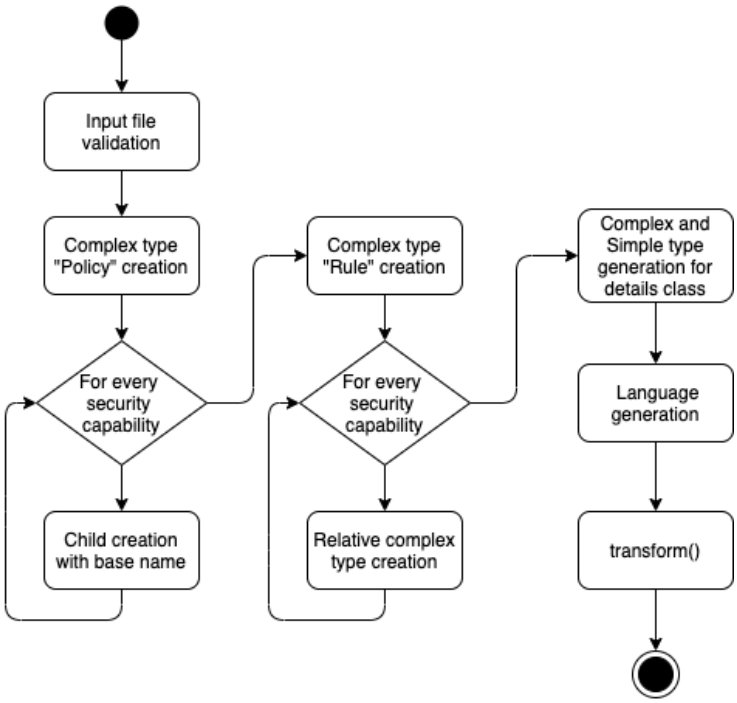

Figure B.4: NSF Language Generation Tool Workflow.

- creation of complex and simple types related to the TranslationDetails class;
- generation of the specific language;
- the "transform ()" function is used which allows the generation of the output.

B.3.1 Tool architecture

The architecture of the proposed tool is structured in two main classes, and is represented in Figure B.5 and in Figure B.6. The architecture is divided into several classes to allow the first class "LanguageModelGenerator" to manage the interpretation of the input file, to manage the structure and all the elements that will be used to generate the output itself, in the second class "XSDgenerator" and a further class for managing the creation of the actual elements belonging to the output document. Furthermore, the separate images allow to divide the functions involved during the generation of the generic language (Figure B.5), and the functions used during the management of the "LanguageConstraint" class of a security capability (Figure B.6).

The architecture of the tool is characterized by a main function "genearateLanguage()" which uses all the other functions. The architecture is developed in the following way:

- the "LanguageModelGenerator" class takes care of reading the input file and

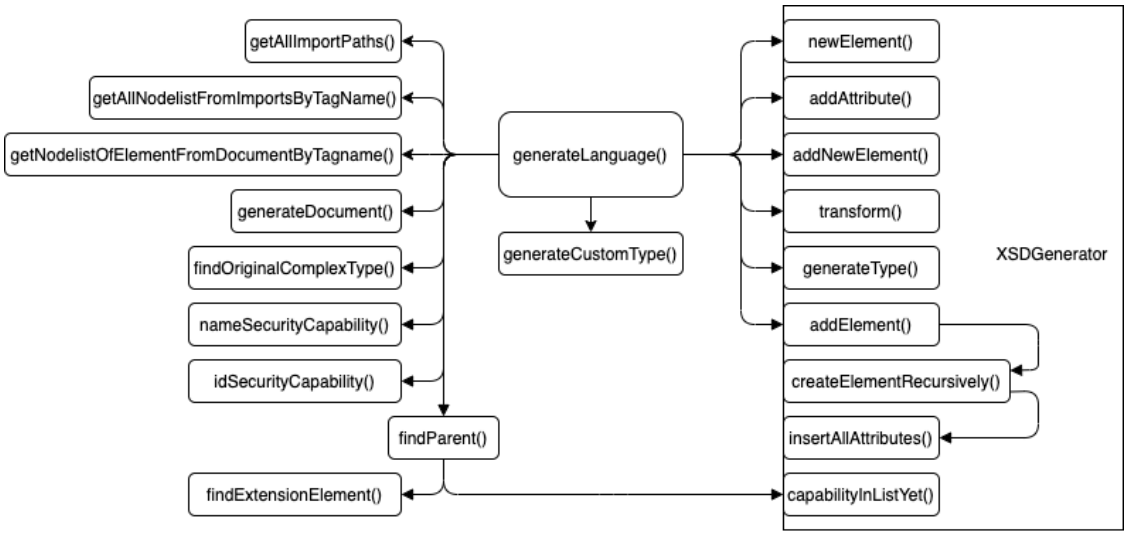

Figure B.5: NSF Language Generation Tool Architecture, first part.

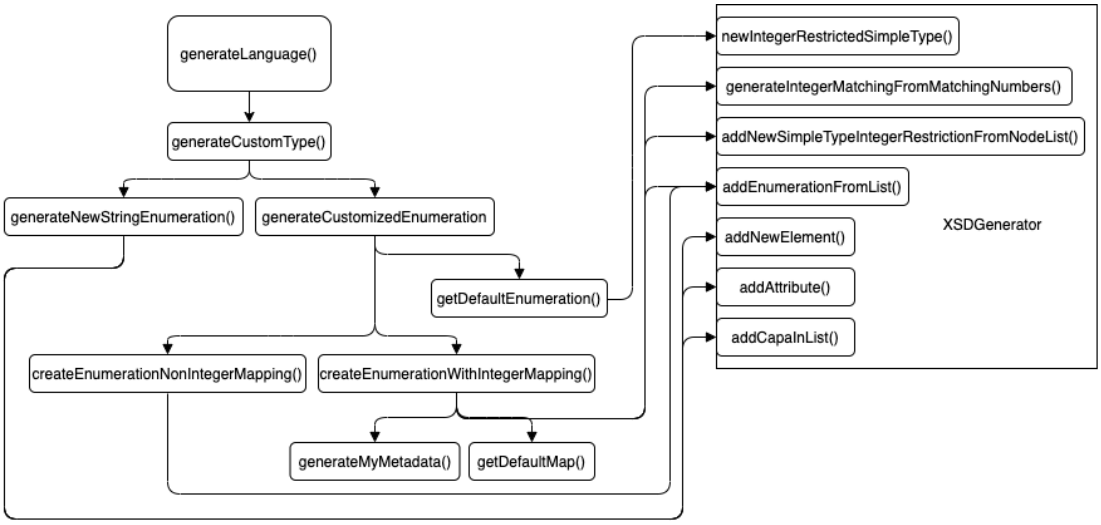

Figure B.6: NSF Language Generation Tool Architecture, second part.

managing the received structures.
This class defines the following instance variables:

- **private** String xsd;
  field that contains the path of the reference XMLSchema file, file that contains the XMLSchema representation of the CapDM;
- **private** String xml;
  field that contains the path of the reference XML file, file that contains the list of assigned security capabilities and any "languageGenerationDetails" and "capabilityTranslationDetails" classes;
- **private** String outputName;

field that contains the path where the output file will be generated and therefore the name of the file itself.

- **private** List<String> imports;
  field that contains the list of paths to the files that make up the CapDM;
- **private** List<NodeList> complexTypeNodeLists;
  field that contains the list of complex nodes belonging to all the files that make up the CapDM;
- **private** List<NodeList> simpleTypeNodeLists;
  field that contains the list of simple nodes belonging to all the files that make up the CapDM;
- **private** XSDgenerator gen;
  field that contains the instance object of the "XSDgenerator" class;
- **private** List<String> metadataPath;
  field that contains the list of paths referring to the metadata files generated during the generation of the language;

This class consists of the following functions:

- **public** LanguageModelGenerator(String xsd, String xml, String outputName)
  constructor function allows you to instantiate objects of the "LanguageModelGenerator" class, instantiating the variables of the class itself, including those referring to the parameters passed with the constructor;
- **public boolean** generateLanguage()
  this function is in charge of generating the language of the NSF. It deals with the generation of the elements necessary for the management of the policy, then creates the "policy" element and the complex type "Policy" derived from it, with the "nsfName" attribute to indicate the name of the NSF we are creating the rules and the "rule" element to indicate the start of a new rule. It also takes care of reading all the security capabilities and the related "details" class in the input file and transforming them into complex types to be inserted in the output file.
- **private** List<String> getAllImportPaths()
  this function generates a list containing all the paths of the "xs: import" elements present in the input file. This function is necessary in the event that the CapDM translated as an XMLSchema file should be divided into multiple XSD files, in this way it is sufficient that the main XSD file contains the details of the paths necessary to reach all the further XSD files;

- **private** List<NodeList> getAllNodelistFromImportsByTagName (List<String> imports, String tag)
  this function generates a list of nodeList elements containing the list of nodes containing elements with that particular "tag" for each document in the "imports" list;
- **private static** NodeList getNodelistOfElementFromDocumentByTagname (Document d, String tagname)
  this function returns the list of nodes containing the elements with that specific "tagname" belonging to the document "d";
- **private static** Document generateDocument (String path)
  this function generates a "Document" type object based on the path specified in the "path" variable;
- **private static** Element findOriginalComplexType (NodeList complextype, String capa)
  this function looks for an element that has as attribute "name" the value contained in "capa" belonging to a list of "complextype" nodes and returns this element of type "Element" or "null" if it has not been found;
- **private** Element findParent (Element complextype)
  this function recursively goes up the chain of elements from which the "complextype" element derives, until it reaches the generic element "SecurityCapability". 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)
  this function searches if an "xs: extension" element exists in the various nodes inside the element "e" passed, if it exists then it returns this element, otherwise it returns "null";
- **private** String nameSecurityCapability(Element item)
  Function to receive the name of the referred security capability;
- **private** String idSecurityCapability(Element item)
  Function to receive the ID of the security Capability referred to.

- the "LanguageModelGenerator" class also deals with the management of the details defined in the "languageGenerationDetails" type elements of the input file and the management of the received structures (Figure B.6). The "LanguageModelGenerator" class is managed by the following functions:

- **private boolean** generateCustomType(Element element, NodeList n)
  this function allows the generation of the NSF language based on the details exposed in the "LanguageConstraint" class. It is called only if "language" tag elements have been recognized. It recognizes the elements contained in the "LanguageConstraint" class and manages them using the appropriate functions;

- **private boolean** generateCustomizedEnumeration(String enumerationName, Element modifyDefaultEnumeration, NodeList modeledNL)
  this function generates elements that respect the details defined in the "language" elements encountered. This function is strictly based on the structure of the "LanguageConstraint" class. The operations performed by this function depend on the content of the "languageGenerationDetails" element being analyzed. Also in this function are defined the methods of using any metadata as in the case of protocol types or port types. If you want to change the behavior of the tool towards the "LanguageConstraint" class, this is the function from which you have to start;

- **private boolean** createEnumerationWithIntegerMapping(String enumerationName, NodeList addNewValueNodeList, NodeList renameValueNodeList, NodeList removeValueNodeList, NodeList addExistingValueNodeList, Element defaultEnumeration, NodeList setNumericRangeNodeList, NodeList generateIntegerMatching, String metadata, List<String> capabilityAndAttributesToBeChanged)
  this function is responsible for creating the enumeration with the parameters defined in the "language" element, any integer type and is responsible for modifying the type of the affected attribute if any. It also deals with the generation of any metadata due to the relationship between the integer and the string value of the enumeration;

- **private void** generateMyMetadata(Map<String, Integer> nameValueMap, String metadata)
  this function actually takes care of generating the metadata output file. The metadata is sorted in ascending numerical order;

- **private static** Map<String, Integer> sortByValue(Map<String, Integer> unsortMap)
  this function takes care of sorting a map according to its integer values;

- **private** Map<String, Integer> getDefaultMap(String metadata)
  this function generates a default map built using the values contained in

the metadata file passed as a parameter;

- **private boolean** createEnumerationNonIntegerMapping(String name, NodeList addNewValueNodeList, NodeList renameValueNodeList, NodeList removeValueNodeList, NodeList addExistingValueNodeList, Element defaultEnumeration)
  this function generates the enumeration type without the relationship between integer and string value. Initially it generates a list of string values which is passed to another function belonging to the "XSDgenerator" class which will create the actual object suitable for the output;
- **private** Element getDefaultEnumeration(String name, NodeList modeledNL)
  this function searches a list of "modeledNL" nodes for the element with the "name" attribute equivalent to the content of the "name" function parameter;
- **private boolean** generateNewStringEnumeration(String name, NodeList valueNL)
  this function generates a new name enumeration the content of the "name" parameter, if it is not yet in the list of elements already generated. This enumeration is generated exclusively on the basis of the values contained in each node of the "valueNL" list;

- the "XSDgenerator" class deals with the management of elements recognized and converted with the XMLSchema format, it also deals with the actual transformation of the element generated when reading the input file and the actual generation of the output file. It has a main element which forms the root of the output file.

  This class defines the following instance variables:

  - **private final static** String NS_PREFIX = "xs:";
    this field allows you to define with which prefix all the elements will be generated;
  - **private** Document doc;
    this field contains the output document to which this instance refers;
  - **private** Element schemaRoot;
    this field contains the root element of the instance output document;
  - **private** NameTypeElementMaker elMaker;
    this field contains the element that allows you to generate new elements compatible with the output document;

- **private** TreeSet<String> capability;
  this field contains the list of the names of the security capabilities encountered;
- **private** List<String> type;
  this field contains the list of the names of the complex or simple types encountered;

This class consists of the following functions:

- **public** XSDgenerator()
  this constructor allows you to instantiate an object corresponding to the "XSDgenerator" class and takes care of creating the main objects of the class;
- **public boolean** transform(String outputName)
  this function is responsible for generating the output relating to the new XSD file using the "outputName" parameter as a reference for the name and destination path of the output file;
- **public** Element newElement(String name)
  this function generates a new element, the "name" parameter is used to give the name to the element to be generated;
- **public void** addAttribute(Element element, String nameAttr,
  String attrValue)
  this function is responsible for adding to the element "element" an attribute named "nameAttr" and the value "attrValue";
- **public void** addNewElement(Element e)
  this function adds a new element to the root of the schema to be generated;
- **public void** addElement(Element e)
  this function adds a new element to the root of the schema to be generated using a copy of the element passed as a parameter;
- **private** Element createElementRecursively(Element e)
  this function recursively generates a copy of the element passed as a parameter, therefore of all its internal characteristics. This function avoids generating the classes or attributes relating to "languageGenerationDetails" or "capabilityTranslationDetails" elements, which are not necessary in the NSF language;
- **private void** insertAllAttributes(Element fromElement, Element toElement)
  this function adds all the attributes of one element, "fromElement", to

another “toElement” element. Furthermore, if a “type” attribute is recognized then it is added to the list of types encountered, a list necessary to generate the complex types of the NSF language;

- **public void** addCapaInList (String s)
  this function inserts the name passed as a parameter in a list of strings without duplicates;
- **public boolean** capabilityInListYet (String s)
  this function evaluates if the value passed as parameter is already present in the list. This function is used together with the list of capabilities that have already been encountered during the translation;
- **public void** generateType(NodeList typeComplex)
  this function uses the nodes belonging to the list of nodes passed as a parameter to generate a new element that can be added to the root that will generate the output. This function uses “addElement()” to generate each recognized element;
- **public void** addEnumerationFromList (String name, List<String> valueList)
  this function creates the enumeration element with the XMLSchema format, using the “name” parameter as the name of the enumeration and entering all the values of the passed list as the “valueList” parameter;
- **public void** addNewSimpleTypeIntegerRestrictionFromNodeList(List<String> capabilityAndAttributesToBeChanged, NodeList setNumericRangeNodeList)
  this function generates a new element using the values present in the list of nodes “setNumericRangeNodeList”. The name is decided with the first value of the “capabilityAndAttributesToBeChanged” list. This list is composed of a first value that indicates the name of the class containing the attributes to which you want to change the type, the further values of the list are the names of the attributes to which you want to change the type, these attributes must belong to the class whose name is in position 0 of the list;
- **private void** changeElementTypeInExistingComplexType (List<String> capabilityAndAttributesToBeChanged, String newTypeName)
  this function looks for the element whose name is in the first element of the “capabilityAndAttributesToBeChanged” parameter. Once this element has been found, the function modifies the type of attributes whose names are the next elements in the list by inserting the value of

the "newTypeName" parameter.

- **public void** generateIntegerMatchingFromMatchingNumbers (List<String> capabilityAndAttributesToBeChanged, Map<String, Integer> nameValueMapSortedByIntegerValue)
  this function generates a new integer type restriction using the values contained in the "nameValueMapSortedByIntegerValue" map, the name of the new type is generated with the value in the first position of the "capabilityAndAttributesToBeChanged" list by adding the string "IntegerRestriction". To complete the generation, this function calls the "changeElementTypeInExistingComplexType" function to change the type of the attributes involved;
- **public void** newIntegerRestrictedSimpleType(NodeList setNumericRangeNodeList, List<String> capabilityAndAttributesToBeChanged)
  this function generates a new integer type restriction using the values contained in the "capabilityAndAttributesToBeChanged" list, the name of the new type is generated with the value in the first position of the "capabilityAndAttributesToBeChanged" list by adding the string "IntegerRestriction". To complete the generation, this function calls the "changeElementTypeInExistingComplexType" function to change the type of the attributes involved;

- the "NameTypeElementMaker" class is a helper class for generating elements in a specific document. This class is used exclusively by functions belonging to the "XSDgenerator" class, since they are responsible for creating the output document.

  This class defines the following instance variables:

  - **private** String nsPrefix;
    this field allows you to define with which prefix all elements will be generated;
  - **private** Document doc;
    this field contains the output document to which this instance refers.

  This class formed by the following functions:

  - **public** NameTypeElementMaker(String nsPrefix, Document doc)
    this constructor allows you to define which document refers to the class and if a prefix string exists;

- **public** Element createElement(String elementName)
  this function creates an element in the document assigned to the class correctly, adding the prefix assigned to the class to the "elementName" parameter;
- **public void** setAttribute(Element element, String nameAttr,String attrValue)
  this function is responsible for adding an attribute named "nameAttr" and the value "attrValue" to the "element" element. In this case, however, it is checked that the attribute names do not contain references to other XMLSchema, in case only the real name of the attribute is used.

B.3.2 Possible changes

If you want to add functionality to the "LanguageConstraint" class, the functions from "generateCustomType()" are involved following the figure of architecture (Figure B.5). To manage a new parameter of the "LanguageConstraint" class, the relative construct "getElementsByTagName()" must be added and then the resolution procedure of the new parameter must be added by adding the relative function.

B.4 NSF low-level language translation tool

This tool allows the generation of a text file containing the policy expressed in the NSF low-level language. The output file is generated according to the security capabilities present in the security policy expressed in the generic language of the NSF. In the thesis scope architecture, this tool is used every time you want to translate a policy for the NSF date. The tool develops the output following the workflow in Figure B.7.

The main steps in the operation of this tool are as follows:

- input file validation;
- cycle for every rule in the file containing the policy;
- cycle for every security capability in the rule being considered;
- the element with the key tag "capabilityTranslationDetails" relating to the security capability currently considered is taken from the file for assigning the capabilities;
- once the security capabilities relating to a rule have been concluded, the dependencies of each security capability used for the rule are checked;

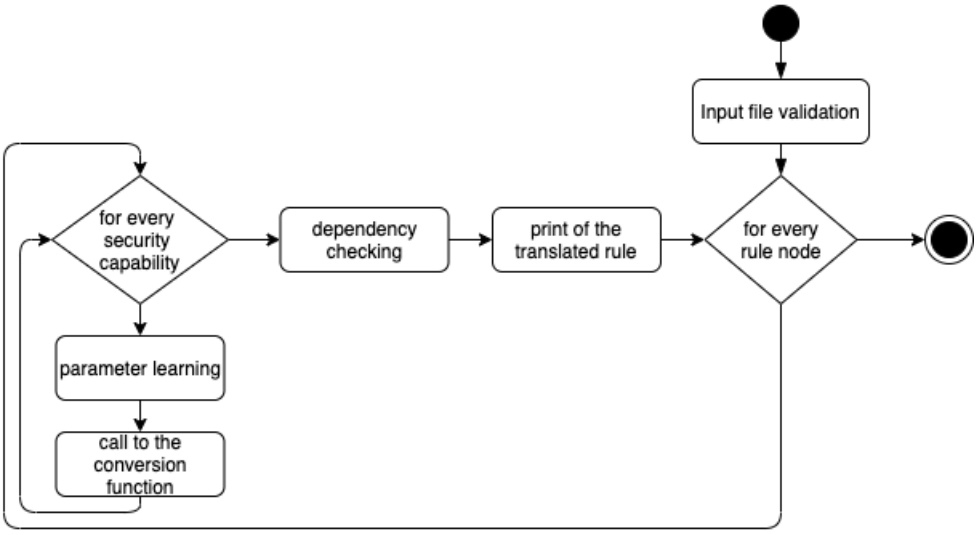

Figure B.7: Translation Tool Workflow

- if the rule is correct it is written to the output file.

Tool architecture

The architecture of the proposed tool is shown in Figure B.8, in this structure the main function is "translate()" which deals with the translation at a macroscopic level, but entrusts the translation of the detail to the "clauseConverter()" function, the which deals in detail with each element that makes up the rule. To carry out the translation, the "NSFTranslator" class uses the information contained in Figure 5.4.

This class defines the following instance variables:

- **private** String xsdLanguage;
  field that contains the path of the XMLSchema file that contains the representation of the language of the NSF;

- **private** String xmlRule;
  field that contains the path of the XML file that contains the policy, expressed in the generic language, to be translated into the low-level language of the NSFs;

- **private** String xmlCatalogue;
  field that contains the path of the XML file that contains the catalogue of security capabilities assigned to the NSF and any "languageGenerationDetails" and "capabilityTranslationDetails" classes;

- **private** String outputName;

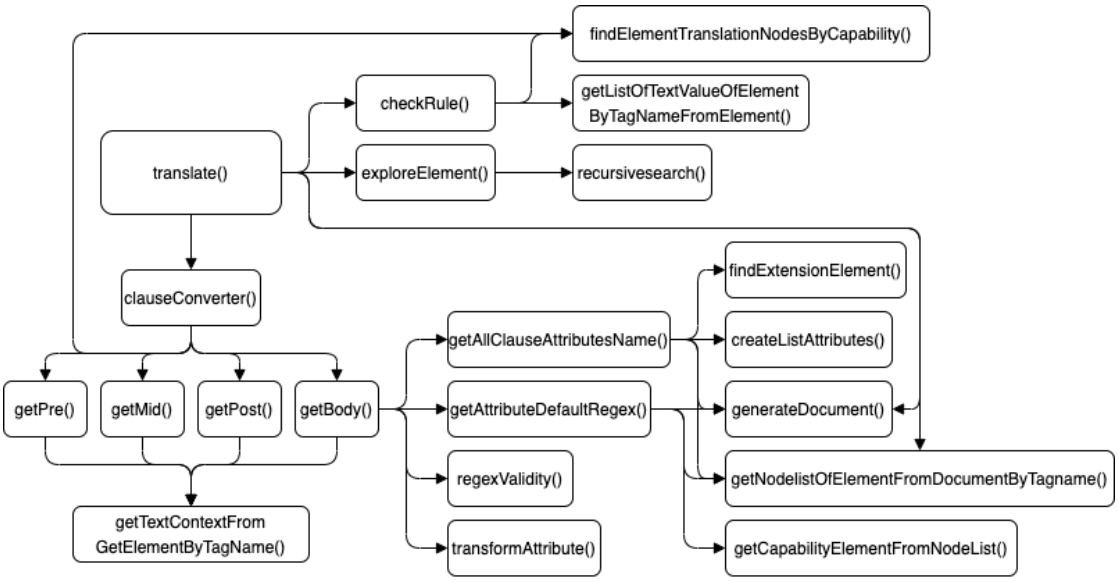

Figure B.8: Translation Tool Architecture

field that contains the path and name of the output file where the NSF low-level policy will be generated;

- **private** String temporaryRule;
  field used to insert the parts of the rule translated into string format. Some parameters passed to the "translate ()" function act directly on this field. Once the security capabilities related to this rule are over, this field will be used to verify the correctness of the rule itself and in case of a positive result the content will be written in the output file;

- **private** String temporaryCapabilityAndAttributes;
  string field that contains the name of the currently considered security capability and a sequence of attribute names each followed by the value of that attribute. Each component of this field is separated from the next by the space character "␣".
  For example: "nameCapacity␣attribute1␣value1␣attribute2␣value2".

- **private** String temporaryCapability;
  field containing the name of the currently managed security capability;

- **private** List<String> temporaryListCapabilityOfRule;
  field that contains the list of the names of the security capabilities encountered during the translation of the current rule;

- **private** NodeList translationNodes;
  field that contains the list of the nodes of the security capabilities that contain information about the "capabilityTranslationDetails" classes for each security

capability;

- **private** String nsfName;
  field that contains the name of the NSF whose policy we want to translate. It is indicated as there can be multiple NSFs with the same security capability;
- **private** String nextCapabilityTemp;
  temporary support string to check if the condition on which to perform the cycle is verified
- **private** NodeList nodes;
  List that contains all the rule nodes
- **private** Element nextCapabilityElement;
  variable in which to save the next capability to perform checks on some dependencies
- **private** Element myCapabilityTranslation;
  field that contains the instance of the "capabilityTranslationDetails" class relating to the security capacity currently analyzed.

This class consists of the following functions:

- **public** NSFTranslator()
  constructor that allows you to instantiate an object corresponding to the "NSFTranslator" class;
- **public boolean** translate(String xsd, String xmlCatalogue, StringxmlRule, String outputName, String startString, String endString, String forced, boolean scr, boolean ecr)
  this function reads the input policy and manages each element using the correct "capabilityTranslationDetails" class instance. It takes care of writing the output file every time a rule is translated using the customization parameters that were passed when this function was called, it also calls the appropriate function to check the correctness of each rule. The analysis of the policy contained in the input file is based on the reference to the security capacity inside the "capabilityTranslationDetails" class and in the considered NSF.
- **private static** Document generateDocument(String path)
  this function generates a "Document" object based on the path specified in the "path" variable;
- **private static** NodeList getNodelistOfElementFromDocumentByTagname(Document d, String tagname)

this function returns the list of nodes containing the elements with that specific "tagname" belonging to the document "d";

- **private void** exploreElement(Element e)
  function that checks if the security capability in parsing has any child nodes. If the recursive method is called it has the print function;

- **private void** recursivesearch(Element e)
  this recursive function parses within an element, node by node to look up the node and attribute name. Being recursive, it analyzes up to the last child and then saves all the nodes with their respective attributes;

- **private** String clauseConverter()
  this function deals with the conversion of the single element of a rule. For the management of the single element, the breakdown of this element into four parts is proposed:
  - pre: part that contains the portion of the string that indicates which command is being used;
  - mid: part that contains the concatenator between the pre and the body;
  - body: part that contains the portion of the string that concretely indicates the possible value of the command;
  - post: part that contains the concatenator between the current rule portion and the next.

  Each of which parts is developed by a specific function. This function takes care of returning a string where the 4 parts just described are concatenated;

- **private** String getPre()
  this function considers the name of the security capability relative to the current rule portion and the "capabilityTranslationDetails" class relative to this security capability. From the conditions of the rule portion it is recognized which command to use, defined in the "capabilityTranslationDetails" class;

- **private** String getMid()
  this function looks for the "capabilityTranslationDetails" element relating to the internal concatenation of the portion of the rule; if in the "capabilityTranslationDetails" class it has not been specified then a space is considered as a predetermined concatenator;

- **private** String getBody()
  this function considers the attributes of the security capability and their values relative to the current rule portion and the "capabilityTranslationDetails"

class relative to the security capability considered. By evaluating the attributes of the rule portion, the use cases of any concatenators to be applied, defined in the "capabilityTranslationDetails" class, are recognized. This function also takes care of evaluating if correct values have been entered, in fact it can evaluate these values by applying any regular expressions or explicit numerical restrictions;

- **private** String getPost()
  this function searches for the "capabilityTranslationDetails" element relating to the concatenation between the current portion of the rule and the next; if in the "capabilityTranslationDetails" class it has not been specified then a space is considered as a predetermined concatenator;

- **private** Element findElementTranslationNodesByCapability()
  this function looks for an element based on the name of the security capability that is being considered at the moment, the name is contained in a variable of the class. This function returns null if there should be no instance of the relative "capabilityTranslationDetails" class.

- **private** String getTextContextFromGetElementByTagName(Element e, String s)
  this function returns the textual value of the element with a name equivalent to the content of the "s" parameter;

- **private** List<String> getAllClauseAttributesName()
  this function is responsible for generating a list containing all the names of the attributes of the security capability that is currently being considered. The list is filled using the "createListAttributes()" function;

- **private void** createListAttributes (NodeList elementNL, List<String> ls,
  Element e)
  this function recursively creates a list that contains the attributes belonging to the element "e", if the element is a complex type then the same function is called until all the names of the parameters are obtained;

- **private** String getAttributeDefaultRegex(String attributeName)
  this function searches if the security capability being considered at the moment, in the attribute whose name corresponds to the value of the "attributeName" parameter, has a preset regular expression;

- **private** Element getCapabilityElementFromNodeList(String capa, NodeList nl)
  this function returns, if present, the element whose "name" attribute has the same value as the content of the "capa" parameter;

- **private boolean** checkRule()
  this function checks whether the dependency rules contained in the instance of each component of the rule are satisfied. To obtain this result, the function uses the information contained in the "capabilityTranslationDetails" instance of each security capability that makes up the rule. In particular, it considers the instances of the "dependency" elements. Structures containing the capabilities used by the rule and the rule already translated but not yet validated are used to check dependencies;

- **private** List<String> getListOfTextValueOfElementByTagNameFromElement(
  Element e, String tagName)
  this function, given an "e" element, finds the "tagName" tag element and generates a list of strings containing all the textual values belonging to these elements;

- **private boolean** regexValidity(String value, String regex)
  this function verifies the validity of the value contained in the "value" parameter with respect to the regular expression contained in the "regex" parameter.

- **private** String transformAttribute(String value, String transform)
  method that allows you to perform a transformation on the specified attribute "value". First a check is made if the value of "transform" is configured, then if this transformation is implemented or not;

- **private** Element findExtensionElement (Element e)
  it searches if in the various elements inside the element passed "e" exists that "xs: extension" and in case it returns that.

B.5 Validation tool

This tool allows the validation of an XML file against an XMLSchema file both passed as parameters to the tool. In the architecture of the thesis, this tool is used whenever an XML file necessary for one of the proposed tools needs to be instantiated.

The architecture of this tool consists of a single class. This class consists of the following functions:

- **public** Validation()
  constructor that allows you to instantiate an object corresponding to the "Validation" class;

- **public boolean** validate(String xsd, String xml)

this function allows you to validate the file defined in the path contained in the “xml” parameter with respect to the file defined in the path contained in the “xsd” parameter.

Bibliography

[1] Marco Scutari; JKorbinian Strimmer. *Introduction to Graphical Modelling*. URL: https://arxiv.org/pdf/1005.1036.pdf.

[2] World Wide Web Consortium. *XML Schema: Formal Description*. September 2001. URL: https://www.w3.org/TR/2001/WD-xmlschema-formal-20010925/.

[3] A. Pras; J. Schoenwaelder. *On the Difference between Information Models and Data Models*. January 2003. URL: https://www.rfc-editor.org/rfc/rfc3444.txt.

[4] C.Alexander; S.Ishikawa; M.Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[5] Gamma Erich; Helm Richard; Johnson Ralph; Vlissides John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] K. Balasubramanian; A. Gokhale; G. Karsai; J. Sztipanovits; S. Neema. *Developing Applications Using Model-drivenDesign Environments*. 21 February 2006. URL: https://ieeexplore.ieee.org/document/1597085.

[7] Parastoo Mohagheghi; Jan Aagedal. *Evaluating Quality in Model-Driven Engineering*. 21 February 2006. URL: https://www.omg.org/ocsmp/MiSE2007-QualityMDE.pdf.

[8] *MDA - THE ARCHITECTURE OF CHOICE FOR A CHANGING WORLD*. URL: https://www.omg.org/mda/.

[9] *Interface to Network Security Functions (I2NSF)*. October 2014. URL: https://datatracker.ietf.org/wg/i2nsf/about/.

[10] Ellen Messmer. *Gartner: Cloud-based security as a service set to take off*. October 2013. URL: https://www.networkworld.com/article/2171424/gartner--cloud-based-security-as-a-service-set-to-take-off.html.

[11] S. Hares; D. Lopez; M. Zarny; C. Jacquenet; R. Kumar; J. Jeong. *Interface to Network Security Functions (I2NSF): Problem Statement and Use Cases*. July 2017. URL: https://www.rfc-editor.org/rfc/rfc8192.txt.

[12] D. Lopez; E. Lopez; L. Dunbar; J. Strassner; R. Kumar. *Framework for Interface to Network Security Functions*. February 2018. URL: https://www.rfc-editor.org/rfc/rfc8329.txt.

[13] L. Xia; J. Strassner; C. Basile; D. Lopez. *Information Model of NSFs Capabilities*. April 2019. URL: https://www.rfc-editor.org/rfc/rfc8329.txt.

[14] N. Noceti. *Rappresentazione astratta delle funzionalità di controlli di sicurezza*. 2019. URL: https://webthesis.biblio.polito.it/13181/1/tesi.pdf.

[15] S. Frankel; S. Krishnan. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. February 2011. URL: https://datatracker.ietf.org/doc/html/rfc6071.

Acknowledgements

Questo spazio lo dedico alle persone che, con il loro supporto, mi hanno aiutato in questo meraviglioso percorso di approfondimento delle conoscenze acquisite durante questi anni universitari.

Un ringraziamento particolare va al mio relatore Cataldo Basile che mi ha seguito, con la sua infinita disponibilità, in ogni passo della realizzazione del progetto, fin dalla scelta dell'argomento.

Ringrazio i miei genitori che sono il pilastro della mia vita, le fondamenta dei miei giorni. Questa tesi è per loro e a loro dedico la gioia che il tagliare il traguardo della laurea accende nel mio cuore. Con gratitudine sconfinata. Senza il supporto morale dei miei genitori, non sarei mai potuto arrivare fin qui. Grazie per esserci sempre stati soprattutto nei momenti di sconforto.

A te che sei sempre stata con me, non mi hai mai lasciato solo e mi hai sempre fatto sentire quanto tu credessi in me, ogni giorno, ogni minuto, dopo ogni caduta, prima di ogni vittoria. Il tuo amore così forte mi ha dato l'energia e l'entusiasmo che mi hanno portato a raggiungere l'obiettivo. Oggi la mia laurea la condivido con te, che ne sei artefice almeno quanto me perché senza di te nella mia vita nulla di quello che ho vissuto sarebbe mai accaduto. Grazie Cristina ♡.

A un uomo ancor prima che a un Nonno. Ringrazio una persona speciale che non ho mai smesso di amare dal giorno in cui è iniziato questo nuovo viaggio. Questo traguardo lo dedico anche a te che per me sei stato una guida e non smetterai mai di esserlo.

Grazie ai miei Nonni per l'amore che mi hanno saputo donare e per l'appoggio che non mi hanno mai fatto mancare.

Un ringraziamento generale, ma non per questo meno importante, va a tutti i miei parenti, zii, cugini che nel mio cuore hanno sempre un posto. Li ringrazio per il loro interesse e per l'appoggio avuto nei miei confronti.

Ringrazio Gianluca che mi è sempre stato vicino, il mio migliore amico, e ti voglio ringraziare per tutto quello che fai per me ogni giorno. Sei unico.. il migliore!

Ringrazio Giovanni che con le sue disavventure ha saputo rallegrare tutti i momenti di questo percorso.

Ringrazio Roberto, il mio PT e nutrizionista, che mi ha sempre incoraggiato e supportato per l'università e soprattutto sopportato con il mio mal di schiena.

Allargo i ringraziamenti a tutti i miei amici Dario, Giovanni, Davide, Ferdinando, Giulio, Alessio e tanti altri che con i loro consigli, critiche, suggerimenti, indicazioni e sostegno morale mi hanno aiutato in questo percorso.

Grazie a tutti i miei colleghi di corso Nino, Angelo, Miriam, Simone, Sabatino, Gennaro e Giuseppe per avermi sempre incoraggiato, aiutato e consigliato fin dall'inizio del percorso universitario.