A Framework for Collaborative Pattern-based Model Driven Security

A Prototypical Implementation

Simon Forster B.Sc.

Supervisor: Dr. Michael Hafner

Innsbruck, September 1, 2011
Abstract

This thesis implements the pattern-driven approach *Model Driven Security (MDS)* as specified in the SECTET Framework [HBAN06]. The goal is the model driven configuration of a security architecture based on security services (SeAAS [HMB09a]) through UML models. The models are gradually refined to executable code that configures a decentralized Service Oriented Architecture (SOA) scenario.
Acknowledgment

It is a pleasure to thank those who made this thesis achievable. Without their guidance and help it would not have been possible to conduct this thesis.

First, I want to thank my supervisor Dr. Michael Hafner for his active and dedicated support. He always had a sympathetic ear for my problems and provided me with positive feedback throughout this thesis.

I owe my deepest gratitude to my parents who made my computer science studies possible. They always supported me in my decisions and backed me up.

Most of all I want to thank one of the most important persons in my life, Victoria. During my whole studies she was a patient listener and encouraged me in difficult times.
Declaration of Authorship

I, B.Sc. Simon Forster, declare that this thesis and the work presented in it are my own and have been generated by me as the result of my own original research. I confirm that:

• This work was done mainly while in candidature for a research degree at the Leopold-Franzens University Innsbruck.

• Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated.

• Where I have consulted the published work of others, this is always clearly attributed.

• Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

• I have acknowledged all main sources of help.

• Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Date __________________________ Signature __________________________ (Simon Forster)
Contents

Abstract ii
Acknowledgment iii
Declaration of Authorship iv

1. Introduction 1
   1.1. Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
   1.2. Structure of the Thesis . . . . . . . . . . . . . . . . . . . . . . . . . 2

2. Background 3
   2.1. Technologies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
      2.1.1. Eclipse Plug-in Development Environment . . . . . . . . . . . 3
      2.1.2. ATL Transformation Language . . . . . . . . . . . . . . . . 4
      2.1.3. Java Emitter Templates . . . . . . . . . . . . . . . . . . . . 5
   2.2. Model Driven Security Engineering . . . . . . . . . . . . . . . . . . 5
   2.3. Target Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   2.4. Multi-layer Model Refinement . . . . . . . . . . . . . . . . . . . . . 8
      2.4.1. Roles within SECTET . . . . . . . . . . . . . . . . . . . . . 9
   2.5. Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3. Problem Description 12
   3.1. Use Case Scenario . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
   3.2. Problem 1: Achieving Security Interoperability . . . . . . . . . . . 13
   3.3. Problem 2: Resolving Policy Dependencies . . . . . . . . . . . . . . 15
   3.4. SeAAS Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 16

4. Framework Overview 18
   4.1. Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
   4.2. Input Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
   4.3. Output Policies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
   4.4. Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
      4.4.1. Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
      4.4.2. 1st Refinement . . . . . . . . . . . . . . . . . . . . . . . . . 29
      4.4.3. 2nd Refinement . . . . . . . . . . . . . . . . . . . . . . . . . 31
1. Introduction

The first section of this thesis motivates the topic. After that the structure of the rest is presented.

1.1. Motivation

The overwhelming complexity of modern software systems not only challenges engineers during conception and development but also poses great problems to those in charge of maintenance and configuration. As these systems mostly realize security critical scenarios, security concerns play an important role.

As a result, such systems are not easy and intuitive to use. Hence, up to now there was usually a developer with deep technical expertise needed in order to maintain or change those systems. As Changes in business workflows occur frequently.

With the rise of Service Oriented Architecture (SOA) environments where the single partners are decoupled from each other it is evidently not easily possible to have a developer who is familiar with the systems each single partner is using.

With this thesis we want to lift some of this burden of maintenance from the developer of this scenario. Ideally, we want to make it possible for a domain expert or security engineer to perform workflow level changes in the business workflow themselves.

So, we are trying to make the configuration of such security architectures more convenient. Therefore, we leverage an existing service oriented security architecture and enhance it with a configuration framework that provides a user-friendly interface. Once a domain expert contributed his know how about the business workflow with the help of diagrams, this piece of software can be used by security experts to configure the target architecture.

Furthermore, it should be possible to configure these target architectures in a collaborative and decentralized way.
1.2. Structure of the Thesis

This thesis is organized as follows.

Chapter 2 introduces the technologies needed for the realization of our prototype (which will be presented in chapter 5). Additionally, the main concepts which are followed in this thesis are also introduced in this chapter. In a final section some related work is presented.

Chapter 3 describes the problems we are trying to tackle in this thesis. Therefore, first a Use Case Scenario is introduced. Throughout this thesis this Use Case illustrates the solution as well as the implementation.

Chapter 4 introduces our solution to the problems defined in chapter 3.

Chapter 5 illustrates the prototypical implementation.

Chapter 6 concludes this thesis and discusses open issues for further research.

The list of figures, listings and acronyms as well as the bibliography are attached at the end of this thesis. Additionally, appendix A contains technical details.
2. Background

In this section we provide some background information. First we introduce various technologies which represent the technical foundation of our prototype. Then we explain the concept of Model Driven Security in detail, as this is the fundamental approach we are building upon. Afterwards, we present the target architecture for our prototype. We also explain the concept of Multi-layer Model Refinement. In a final section we present some related work.

2.1. Technologies

We start with a brief introduction of miscellaneous technologies we utilized for our prototype. As the prototype that is introduced in chapter 5 is an Eclipse Plug-in, we first give an overview of Plug-in Development Environment (PDE). Afterwards we present the model transformation language ATL Transformation Language (ATL) and the text generation framework Java Emitter Templates (JET).

2.1.1. Eclipse Plug-in Development Environment

The PDE\(^1\) is a light weight development environment/framework that provides the functionality of loading and handling all kind of plug-ins. Basically everything in Eclipse is a plug-in. Furthermore, Eclipse is an open platform. This makes it very easy to develop own plug-ins for it. Therefore it provides a separate layer to connect to. This layer is called PDE and provides all necessary tools for plug-in development.

In most cases a plug-in is delivered as a jar file. In this file all necessary information for the plug-in is stored. This not only includes the code itself but also the required resources like images.

\(^{1}\)http://www.eclipse.org/pde/
2.1.2. ATL Transformation Language

ATL is a language that is capable of transforming input models into output models. The main field of application of ATL is Model Driven Development (MDD). There, transformation languages like ATL are used to transform a Platform Independent Model (PIM) into a Platform Specific Model (PSM) (figure 2.1).

![Figure 2.1: The Transformation Process within MDD](For11)

First, meta-models for both, the PIM and PSM have to be created. Then the transformation between these two meta-models can be specified. After that, instances of the PIM can automatically be transformed into instances of the PSM. This transformation is often called Model2Model transformation, as both the input and output of it are models.

Model2Model transformations generate the elements of the target model out of source model elements. In general the source model is at least one level above the target model regarding the level of abstraction. Thus it is possible to create a concrete application out of model elements. This can be achieved through construction rules, which are integrated into the transformation process. Usually, these rules are defined by software architects who have expertise in this area.

Figure 2.2 shows the relationship between all models involved in the transformation process. Even the transformation itself conforms to a meta-model. Additionally, even the meta-models themselves have to be conform to a meta-model, the meta-meta-model (e.g. Ecore).

The ATL Integrated Development Environment (IDE)\(^2\) provides a number of standard development tools. This Eclipse plug-in includes a transformation engine that is able to interpret the rules written using ATL.

\(^2\)http://www.eclipse.org/atl/
2.1.3. Java Emitter Templates

JET provides a convenient way to add automatic code generation to a project [FL10]. With the help of JET it is possible to generate a static skeleton in advance during development, dynamic information is provided later on during runtime.

The syntax of JET is similar to the syntax of Java Server Pages (JSP). As with JSP files it is possible to provide JET templates with parameters. Furthermore, plain Java code is also supported within JET templates. So it is possible to include high level programming constructs (e.g. while-loops).

The templates are automatically transformed into Java source code that can easily be integrated into existing projects.

Here we introduced basic technologies that were used for the prototype presented in chapter 5. In the upcoming section we introduce key concepts of this thesis. We first introduce the concept of Model Driven Security Engineering (MDSE).

2.2. Model Driven Security Engineering

Models have a long tradition in software engineering. Before the rise of MDD, models were only used for documentation purposes. This usage of models in a development process is called model based engineering/development. But this has several disadvantages. For one, software systems are not static but most of the time dynamic. And this not only during development time but also during runtime of a software
system. That means that a software system often has to be adapted to changes of business processes even after the system was finished. Another disadvantage is that these models only contribute little to the development because there is still a software developer needed to interpret these models and write the code out of this interpretation. Therefore modeling is seen as a big overhead by most of the developers out there.

The aim of MDD is to link the models to implementation code. This concept can already be seen by the word “driven”. This reveals the importance of a model within this approach. So the aim is not to produce just program skeletons but also actual implementation code out of the models.

As a result there should be no need for a developer anymore to make changes to the system. Ideally a domain expert should be able to apply changes to the system by himself. Even, the domain expert is not required to have any programming skills.

Further information on the topic of MDD can be found in [SVEH07].

Now, the idea of MDSE is to integrate the security related requirements into these models (mostly Unified Modeling Language (UML)). For example, the authors of [HB09] specialized the concept of MDD to MDSE “by providing a framework in which security concepts in an application domain are modelled using UML at the PIM abstraction level and are merged with business requirements models”.

As a result the high-level security requirements are realized at the model level and are separated from the underlying security architecture [HMB09a].

For more details on MDSE the reader may refer to [HB09].

Here we presented the approach of MDD and extended it with security related aspects. In the upcoming section we introduce the security architecture that is utilized by our prototype as the target architecture.

2.3. Target Architecture

Just like the approach of MDD the concept of MDSE also needs a target architecture. This target architecture reflects the PSM. Later, in chapter 4.3 we will see that the artifacts we generate are the ones used by this security architecture. Of course, it is possible to replace the target architecture with the corresponding PSM without a need to change the PIM. This is one of the advantages of MDSE. In chapter 4.2 we will introduce a Domain Specific Modeling Language (DSML) that specifies our PIM. In this section we introduce the security architecture we used as target platform for our prototype.
The target architecture follows the concept of Security As A Service (SeAAS). The key idea is to provide security related functions within a Service Oriented Architecture (SOA) environment as dedicated services. Web Services within a domain do not have to take care of any security related task. This burden is taken by a dedicated SeAAS server.

The architectural blueprint for such an architecture was introduced in [HMB09a] and summarized here.

Figure 2.3: SeAAS Conceptual Architecture [HMB09a].

Figure 2.3 shows the conceptual architecture. Within one domain there are three components:

- An Enterprise Service Bus (ESB) (1),
Background

- the Business Services and (2)
- a SeAAS Component (3).

In this scenario the business services are taken from the Elektronische Gesundheits-Akte (ELGA) healthcare scenario. “These service endpoints are decoupled from the security and messaging components”, [HMB09a]. The ESB is the message entry point of the domain and responsible to mediate the incoming messages first to the SeAAS component and as a second step to the right business service endpoint. The SeAAS component itself is responsible to perform the security related tasks of a message flow. Therefore, it retrieves, parses and evaluates the security policies for a particular request/response. In order to achieve this goal, the SeAAS does have a bunch of security services available to use. These security services are divided into two categories:

- Primitive Security Services (e.g. Encryption, Signature) (4)
- Advanced Security Services (e.g. Authentication, Non-Repudiation) (5)

In this section we introduced the target architecture used by our prototype. This corresponds to the PSM in the concept of MDSE. The upcoming section will bridge the gap between the PIM and the PSM. We therefore introduce the concept of Multi-layer Model Refinement.

2.4. Multi-layer Model Refinement

In order to get the PSM out of the PIM the MDD concept includes a transformation step. Common approaches of MDD and MDSE only consider two layers with a single transformation. The transformation rules within this transformation are bound to a specific target architecture. This has the disadvantage that new transformation rules have to be written, as soon as either the security requirements or the target architecture changes. In this section we introduce the framework (SECTET) of Memon (et al.) [HMB09b] that is following a three layer approach instead in order to overcome this shortcoming.

SECTET adds another layer of abstraction between the security enhanced models and the implementation technology. “In this abstraction layer, security requirements are modeled with more fine-grained details using Security Protocols, Security Controls and Composition Rules” [HMB09b].

Figure [2.4] shows the three layers of SECTET. The Secure Business Process Models in the PIM (Security-enhanced Functional Models) splits up the functional view (Business Process Models) from the security aspects (Abstract Security Policies).
The Platform Independent Transformation maps the security requirements (defined in models) to the security patterns which can solve those requirements. As an example, the security requirement Authentication can be mapped to the security protocol for Brokered Authentication. These Abstract Security Patterns are then mapped to concrete Executable Security Services within the target platform. This mapping is done by the transformation rules in the Platform Specific Transformation.

![Diagram of SECTET Framework](image)

**Figure 2.4:** SECTET Framework [Mem11].

In [Mem11] Mukthiar Memon additionally introduced roles associated to the models and transformations. The upcoming subsection explains these roles in more detail.

### 2.4.1. Roles within SECTET

When using the concept of MDD it is common practice to divide the involved tasks over different people that have expertise in this specific area. Here we will now present the different roles involved in the SECTET framework.
Figure 2.5 shows the three categories of roles that correspond to the three layers within the SECTET framework.

<table>
<thead>
<tr>
<th>Role</th>
<th>Domain Expert/Domain Engineer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use cases</td>
<td>- Model business processes</td>
</tr>
<tr>
<td></td>
<td>- Define security requirements with business processes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role</th>
<th>Framework Engineer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use cases</td>
<td>Create and update:</td>
</tr>
<tr>
<td></td>
<td>- Security Patterns &amp; Refinements Catalogue</td>
</tr>
<tr>
<td></td>
<td>- Security Policy Models</td>
</tr>
<tr>
<td></td>
<td>- Model Transformation Templates</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role</th>
<th>Security Expert/Security Engineer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use cases</td>
<td>- Retrieve Security Requirements from Security-enhanced Business Process Models</td>
</tr>
<tr>
<td></td>
<td>- Select Security Patterns and Refinements from the Patterns Catalogue</td>
</tr>
<tr>
<td></td>
<td>- Generate code after Pattern Refinement using Templates</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role</th>
<th>Application Administrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use cases</td>
<td>- Deploy and Update Security Components and Artifacts</td>
</tr>
</tbody>
</table>

**Figure 2.5**: Roles in the SECTET Framework [Mem11].

The *Domain Expert or Domain Engineer* has the knowledge of the domain (e.g. processes in healthcare applications). Therefore, he is responsible to design the business model of the workflow. In addition, he is also responsible to assign the security requirements to these models using a DSML (see chapter 4.2).

The *Framework Engineer* is responsible to create and update the security patterns (predefined sequences of security mechanisms that can be interpreted by the target architecture) and protocols. Furthermore, he creates a catalogue with model transformation templates from which the *Security Expert* will subsequently choose one.

During model refinement the *Security Expert or Security Engineer* refines the abstract security requirements to actual code. Therefore, he selects patterns or protocols out of the catalogue created by the *Framework Engineer*. Using code generation templates he then generates the policy files that can be parsed and interpreted by the target architecture.

The *Application Administrator* is responsible to deploy and update the security components (created by the *Framework Engineer*) and policy files (generated by the *Security Expert*).
In this section we introduced the concept of the three layer model refinement of the SECTET framework. Additionally, we gave a brief overview of the roles that are involved in security engineering with this framework. More information can be found at [Mem11] and [HMB09b]. The upcoming section presents some related work in this area.

2.5. Related Work

As already mentioned most implementations of MDSE only use a two layer approach with one transformation (PIM → PSM). An example is the SECTINO framework introduced in [HBBN05]. There the authors present a concept “...for the realization and high-level management of security-critical workflow based on the paradigm of Model Driven Security” [HBBN05].
3. Problem Description

This chapter introduces and scopes the problems which this thesis is trying to solve. We introduce a simple use case to illustrate the key challenges.

3.1. Use Case Scenario

This section introduces the use case scenario. The use case deals with a web application for ordering items. One can think of Amazon where it is possible to buy products which are not directly provided by Amazon itself. With the help of our fictitious example it is possible for a retailer, as it is also possible at Amazon, to make his products available on the web. The web company, Amazon, provides products of various retailers and acts as an intermediate regarding the payment.

In figure 3.1 one can see the process from a customer’s view of ordering an item from a retailer. Therefore the client only has to communicate with the intermediate store, which then handles the process of validating the credit card information as well as the actual ordering of the selected items by the retailer.

First the customer sends an ordering request to the intermediate online store (Store) which then contacts the credit card company (CC Company) in order to validate the credit card information sent by the customer. If successfully validated, the Store subsequently orders the requested items at the retailer (Warehouse). A message reporting the status of the ordering request is sent to the customer after reception of the response from the retailer.

Beside the message workflow also the security requirements within the process of ordering an item are illustrated within this figure. It is stated that the initial request from the customer has to be secured, authenticated and can’t be denied. The same requirements apply on the request message from the Store to the Warehouse. All other messages only have to be secured in terms of confidentiality and integrity.

In order to enforce such security requirements in target architectures, machine readable policies have to be generated. The next section outlines the dependencies between policies of different actors.
3.2. Problem 1: Achieving Security Interoperability

Once the security requirements are specified at the business level the system implementation has to enforce them. In case of web applications such security requirements are defined with the help of executable policy files.

These policy files instruct the web application how to handle incoming and outgoing messages sent by or sent to other actors.

Figure 3.2 depicts the use case with the correlating policies. Here one can see that the security requirements of both, the request message and the response message, are combined within the same policy file. It is important to notice that the same policy file is capable of configuring the service requester as well as the service provider. The advantage of being able to configure both endpoints with the same file has a major drawback. These two files do always have to match each other. With only a minor change within one policy file the whole communication is disrupted. Preserving consistency for configuration files is a hard task to achieve manually.

One possible solution to this problem would be to store these policy files centrally either with one of the interacting companies or with a trusted third party. This way
only one copy of the policy file would have to exist to which both parties would have access. When storing it with one company, this one has obviously more rights to control these policy files. In order to be fair only the solution with a trusted third party would work. But another problem arises when using this solution. To make things easier there should only be one trusted third party where to store these policy files. Otherwise the policies of one company are spread over multiple trusted third parties. This would make the handling of these files again error prone and confusing. Using just one trusted third party to store all policy files obviously does not scale very well.

Because of the latter two disadvantages

- scalability when using a Trusted Third Party
- unbalanced rights when storing it with one company

we decided to follow the approach of decentralized storing the policy files with all participating companies. This solves the two problems from above. But at the same time that means we have to find a practical way to create the policy files for all parties in a consistent way.

Figure 3.2: Policy Dependencies between the Actors.
The aim of this thesis is trying to solve the problem of creating these policy files in a consistent way. This will be done for all participants in the business process in a decentralized way. Therefore it should be possible to generate the policy files for all participants out of the workflow diagrams automatically. Nothing should have to be adapted manually. Changes can be made on model level only in order to preserve consistency.

Chapter 6 will give an outlook on how the problems of managing or even changing the policy files, created with the introduced framework, can be solved.

But this is not the only problem that this thesis tries to tackle. There is also a second one which is introduced in the upcoming section.

3.3. Problem 2: Resolving Policy Dependencies

Another big issue that can easily be seen within this use case scenario is that a policy does not only depend on the security requirements for the communication of two partners but also on policies for communication between other parties. This fact is illustrated in figure 3.3. The policy that handles the security requirements of the customer and the Store depends on the policy which handles the security requirements for the Store and the CC Company.

This dependency exists because the Store should not be able to decrypt the credit card information of the customer but has to pass the information on. So this information has to be encrypted using the requirements within the policy of the CC Company. The requirement that the Store should not be able to read the credit card information is usually created in order to prevent fraud. There are two possibilities to betray the owner of the credit card. For one, the Store itself could charge the credit card. And even if one could trust the owner of the Store it could be possible that an attacker of the Store has successfully circumvented it’s security facilities. In this case he would also have access to the credit card information. So, when the Store is not able to encrypt the information also an attacker of it would not be able to encrypt it.

This small example already requires a lot of expertise in this area and work to be done in order to preserve consistency between all policies. Real world business processes are definitely more complex than this simple use case example and thus the task to generate the policy files by hand is almost infeasible or at least very time consuming and error-prone. Usually all these information, such as who should be allowed to read specific information like the credit card number, are already available during the design phase of such a business process and thus can be easily integrated.
into models. Having stored all these information within a model it is then possible to generate the policies automatically.

Before diving into the solution for these two problems and consequently into the main part of this thesis we illustrate how the use case scenario is deployed on the Security As A Service (SeAAS) architecture in the next section. As we follow the Model Driven Development (MDD) paradigm within our framework it is necessary to have a unique target architecture. Chapter 2.3 introduced this target architecture. There it was also discussed why this platform was chosen.

### 3.4. SeAAS Architecture

In figure 3.4 the underlying architecture of this business process is shown.

There one can see that the *Warehouse*, the *Store* as well as the *CC Company* utilize the SeAAS architecture which was introduced in chapter 2.3. Within these three domains there are the business services deployed which are needed for this use case scenario. The business services of the *Warehouse* and the *CC Company* are Apache Axis web services. The business service of the *Store* is implemented using Web
Services Business Process Execution Language (WS-BPEL). Additionally, also the security services which are responsible to meet the policy defined security requirements are deployed within each domain separately. Except for the non-repudiation service, all security services are Apache Axis Web Services. The non-repudiation protocols are implemented with WS-BPEL. The communication between the Enterprise Service Bus (ESB) and the SeAAS engine takes place using Java Message Service (JMS). The rest of the communication uses Hypertext Transfer Protocol (HTTP). Further information about these technologies was given in chapter 2.1.

This chapter introduced the problems this thesis is trying to tackle. Therefore we introduced a simple business use case as a running example. This use case will also be used to demonstrate the solution to the problems mentioned throughout this chapter. In the upcoming chapter will discuss the solution of these problems.
4. Framework Overview

As the reader now has an understanding of the key problems, this chapter proposes a solution. This is the heart of the thesis as it outlines the main concepts which were elaborated.

This chapter starts with a short overview. Then the input and outcome of the transformation process are illustrated. The transformation process itself will be introduced afterwards.

4.1. Overview

The main idea in order to overcome the problems mentioned in chapter 3 is to provide the Security Engineers with a framework that facilitates the configuration of local Security As A Service (SeAAS) infrastructures (within the area of one Security Engineer’s authority) as well as remote ones (within the same business workflow but within the area of authority of another Security Engineer) in a decentralized way. This is possible by creating models that illustrate the business process. Additionally, the security requirements have to be integrated into these models. Ideally there should not be a developer needed anymore to make changes to the security requirements of the business process.

In order to achieve this goal, the information needed within a policy file have to be integrated into these models. As we follow the multi-layered transformation approach that was presented in chapter 2.4 we are able to provide additional information to the generation of the policy files. In upcoming sections we will see examples of such information that could not have been saved within the input models. In the next section an example of such a model will be shown. How the transformation process is designed can then be seen in section 4.4.

In figure 4.1 the life cycle of the policy generation process is illustrated. There one can see that the output of the transformation process are policies which configure all participating SeAAS instances and corresponding security services.

Before going into more detail of the transformation process the next two sections exemplify the input models and output policies.
4.2. Input Models

The transformation process introduced in section 4.4 basically needs the following three models as an input:

- workflow (activity diagram)
- interface model (class diagram)
- document model (class diagram)

This section explains the relevance of each of them.

The most important information for generating the required policy files are modelled in a Unified Modeling Language (UML) activity diagram. This model defines not only the whole business process between all actors but also specifies the security requirements.

The framework proposed by this thesis supports the following high level security requirements:

- integrity
- confidentiality
- authentication
- authorization
Framework Overview

- non-repudiation

Within the activity diagram these requirements can be specified at message level but as we will see in chapter 5.3.2 it will be possible to even specify them at element/parameter level.

Only the authentication requirement is an exception as it can only be specified at service level, as we will see later on.

All actors and the interactions between them are defined in an activity diagram (figure 4.2). This example illustrates the task of ordering an item, as it was already introduced in chapter 3.1. Here a customer wants to place an order and therefore sends the corresponding request (PlaceOrderRequest) to the Store.

The security requirements have to be specified on the data objects, which are flowing between the particular activities.

The next model that serves as input for the transformation process is the so called Interface Model. Basically, this is a class diagram defining services used within the business workflow. Operations of the services have to be named like the activities defined in the workflow model (activity diagram). Figure 4.3 illustrates the interface model of our use case scenario.

Here one can see that the authentication requirement has to be specified on the interface as it can only be specified on service level. In this example it should be the case that the operations from both services, Store and Warehouse, can only be performed by authenticated users, whereas the operations from the service CreditCardCompany may be performed by everybody.

It is important to specify the service classes with the stereotype interface because the interface model is not the only class diagram that serves as input to the transformation process. With the help of this stereotype it is possible to distinguish between them.

As already mentioned the security requirements within the workflow diagram can only be specified at message level but can be refined further within the transformation process. In order to be able to decrease the granularity (from message level down to parameter/element level) within the transformation another input model is necessary. This is the so called Document Model. This class diagram describes the body (payload) of the messages sent between the actors. Figure 4.4 illustrates the structures of the messages of our simple example.

This information has to be provided in order to be able to encrypt parts of a message for example. It could be the case that one wants to encrypt only the credit card information in the PlaceOrder request.
Here it is also necessary to assign a stereotype to the classes within this class diagram to be able to distinguish them between the classes from the interface model. For this purpose the stereotype *message* is used.

Chapter 5.3.2 discusses how the information from the document model is used during the transformation process.
After studying all these models one could ask why the security requirements are stored within the workflow model and not within the document model as it would then be possible to define them directly on the elements instead of the message. The decision to define them in the workflow is based on an important reason: This way it is possible to model the security requirements for various interactions independently. Figure 4.3 illustrates an example for different interactions. Domain 1 offers various services to users. These users are located either in Domain 2 or 3. The connection between Domain 2 and Domain 1 can not be tapped. So it can be argued that messages sent over this connection do not have to be encrypted. But it can not be guaranteed that also the connection from Domain 3 to Domain can not be tapped. So the security requirements for these two distinct connections may be different (with or without the security requirement confidentiality). Again, that means the
security requirements are defined per interaction and not per operation.

The reason why another path was followed regarding the authentication requirement is obvious. Once it is necessary to authenticate in advance of service consumption it does not make any sense that only specific users have to authenticate themselves. Because the authentication of the origin of a message is needed anyway in order to distinguish between them.

![Diagram](image)

**Figure 4.5:** Example for different Interactions.

Figure 4.6 illustrates the activity diagram that would serve as input to the transformation process. This diagram includes the two interactions explained in the last paragraph.

In order to ease the task of modeling all these diagrams the user is provided with a UML profile. This profile (figure 4.7) supports the user with predefined stereotypes that were introduced throughout this chapter.

As we now have shown what the input to the transformation process is, we will present the output in the next section. Specifically we will illustrate the structure of the produced policy files. Chapter 5.5 will then show what exactly has to be done in order to adapt these policy files to the own needs.

### 4.3. Output Policies

When using the SeAAS concept there are basically two policies necessary to configure the security requirements. One that configures the SeAAS engine itself and another
one for configuring the security services within each domain.

The SeAAS policy file basically tells the engine which high level security requirements have to be met. An example can be seen in listing 4.1.

```xml
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    wsu:Id="SeAASPolicy">
```

---

Figure 4.6: Activity Diagram with Different Interactions.

Figure 4.7: The Sectissimo UML Profile.
Framework Overview

Listing 4.1: SeAAS Policy File

This policy file conforms to the Web Services Policy (WS-Policy) standard and describes the security requirements which were defined within the use case scenario for the Store service. There the stereotypes integrity, confidentiality and non-repudiation were assigned to the Place Order request and response in the workflow model. These requirements are stated by the lines 5, 6 and 8 in listing 4.1. In addition the stereotype authentication was assigned to the service within the interface model and is stated in line 9. These security requirement elements have an unique and own namespace, http://www.sectissimo.info/security/services/seaas/ws-policy. This namespace is necessary to reliably identify the requirements.

The other policy file is responsible to advise the security services with technical information. Listing 4.2 shows an example of such a policy file.
In addition to the WS-Policy standard, this policy also conforms to the Web Services Security Policy (WS-Sec-Policy) standard. Like the SeAAS policy it also states the security requirements for the Place Order operation within our use case scenario. The difference is that this policy file actually provides technical information for the security services. Lines 42 - 45 show the instruction responsible for authentication. The security requirements integrity and confidentiality are defined in the lines 47 - 49 and 50 - 52.

In this policy file there are more information than could be extracted out of the input models. These technical details, like what element to sign or encrypt, are added during the refinement process. This is explained further in section 4.4.

As we do now have these policy files the next step is to find a way to bind these policy files to a service. In other words, we need a mechanism to link a policy to a service. In order to achieve this goal the SeAAS engine uses a simple concept. As
the SeAAS engine is designed for Web Services it is able to use the Web Services Description Language (WSDL) files for this purpose. So in addition to the policy files also a part of the corresponding WSDL file will be generated. Specifically only the binding part will be generated automatically. Other information, like the structure of a message, does not have to change when a security requirement needs to be adapted to new needs. Within these WSDL files the policies will be referenced using the Web Services Security (WS-Security) standard. With this it is possible to save a policy in a file and refer to it from another one. This has the advantage that the files remain more readable. It is possible to save the bindings for the SeAAS engine and the security services within the same file as it is illustrated in listing 4.3. Here the namespaces are omitted for a better readability.

```xml
<wsdl:import location="http://sectissimo.info/Resources/wsdl/portType/OrderService.wsdl" namespace="http://sectissimo.info/service/order"/>

<wsdl:binding name="StoreSeAASBinding" type="tns:StoreInterface">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="PlaceOrder">
    <soap:operation soapAction="http://sectissimo.info/service/order/PlaceOrder"/>
    <wsdl:input>
      <wsdl:part name="body" use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsdl:part name="body" use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:binding name="StoreBSSBinding" type="tns:StoreInterface">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="PlaceOrder">
    <soap:operation soapAction="http://sectissimo.info/service/order/PlaceOrder"/>
    <wsdl:input>
      <wsdl:part name="body" use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsdl:part name="body" use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Listing 4.3: Binding with Policy Reference
Framework Overview

Because this binding is a standard WSDL binding the file also has to be conform to the WSDL standard. In order to fulfill this, line 1 in listing 4.3 includes the port type and all other required information into this file. As already explained this information is static and thus it is not required to change it as soon as security requirements are changed. So it makes more sense to just reference them and to not include the whole content.

The interesting part is shown in line 8 where the policy file for the SeAAS engine is referenced. As soon as the SeAAS engine recognizes such a policy reference, it extracts the information out of this file and replaces the references by its content. In other words, the content of the referenced file, in our case the policy, overwrites the reference. It is not only possible to reference the whole file, but also to reference parts of a file only. This can be achieved by the use of the WS-Security standard. With that a name can be assigned to a policy to which the reference is then able to point using # as a separator between the location of the file and the name of the policy.

The policy reference in line 8 defines the security requirements for the request of the operation whereas the policy reference in line 12 defines the security requirements for the response. Both are read by the SeAAS engine. Line 23 and 27 are read by the security services.

For the reason of simplicity in this example there was just one policy file generated which included all security requirements. Usually it would be the case that a separate file is generated for each message. Additionally, if authentication is required when calling a service, there is also a separate policy file generated which can then be referenced earlier - outside of the definition of the operations - in the binding. As a result the reference has to be inserted only once but will be used for all operations and messages within this service.

Again all information that is necessary to create these binding parts of the WSDL file can be retrieved out of the input models. An exception is the location of the WSDL file that defines the port type and other required information. This location can be provided during the transformation process. This leads already to the next section.

4.4. Transformation

In this section the main concept of this thesis is introduced. The transformation generates the desired policies out of the input models as suggested within section 4.2. After a general overview of the transformation process is given, the two refinement steps are explained in more detail within a separate section for each of them.
4.4.1. Overview

As already mentioned, the transformation process is the key concept of this thesis. It produces the policy files. During the transformation process the security requirements which were assigned to the input models are refined. This refinement consists of two main steps which are performed consecutively:

- make an architectural decision
- provide key technical details

The architectural decision is the first refinement step. For example, there the decision for a specific protocol has to be made.

Afterwards, the second refinement step is taken. The technical details provide the required information to the protocol which was chosen for example during refinement step one. We implemented the transformation process with the help of these two steps primarily to follow the pattern driven approach of [HMB09b] as introduced in chapter 2.4. This way it would be easily possible to exchange the target architecture with no need to change our whole framework. Because the architectural pattern or protocol is independent of the used target architecture.

The next two sections treat these two refinements in more detail. In chapter 5 the technical details on how the transformation process was implemented will be presented.

4.4.2. 1st Refinement

During this refinement step an architectural decision has to be taken. That means a Security Expert or Security Engineer has to choose a specific architectural protocol (e.g. Brokered Authentication) for each security requirement (e.g. Authentication) and message. Chapter 2.4.1 presented all roles which were introduced in [Mem11]. This way it is possible to have one protocol for the request and another one for the response of an operation. When making this refinement the user is provided with a list of available patterns where one pattern corresponds to one protocol. After refining the non-repudiation security requirement, a policy file (listing 4.4) is generated. This file asserts which protocol (line 6) to use. The Application Administrator, introduced in chapter 2.4.1 then has to deploy this policy file on the SeAAS architecture. This can either be done by copying the file onto a web server that is dedicated for saving all policy files or to a local directory within the SeAAS engine. Chapter 5.3 will show how this file is used within the non-repudiation protocol.
Listing 4.4: Generic Non-Repudiation Policy.

Figures 4.8 and 4.9 show the authentication and non-repudiation patterns which are available in the prototype. For the sake of completeness the decision patterns for the other security requirements were added in the appendix in section A.1.

In our use case example the Security Expert or Security Engineer is asked for this architectural decision regarding the authentication requirement twice because the Store and the Warehouse do have the security requirement authentication assigned. In this case the decision has to be done once per service and not per message, as the security requirement authentication is an exception and only assigned to a whole service and not to a message within one interaction.
Framework Overview

So basically all that has to be done during this refinement step of authentication is to choose the right pattern for either direct authentication or brokered authentication. As direct authentication is quite self-explanatory, brokered authentication is examined here further.

Once the brokered authentication pattern is chosen, the SeAAS engine uses the protocol specified in the sequence diagram in figure 4.10. After the SeAAS engine of the service requester’s domain processed the policy, a token is requested from the identity provider. Once the request is successfully validated, the identity provider sends a token back to the service requester’s domain. The SeAAS engine appends this token to the actual service request and sends it to the service provider. After the service provider successfully validated the token, the request is processed and the result sent back to the requester.

![Sequence Diagram](image)

**Figure 4.10:** The Brokered Authentication Protocol implemented by the SeAAS Engine.

After the decision for a specific pattern was made the next step is to provide the technical details to the transformation process. Using brokered authentication such information would be the address of the identity provider. The details of this transformation task are explained in the next section.

### 4.4.3. 2nd Refinement

The second refinement step is about technical details. Here all additional information that could not be modelled and provided within the input models is provided in order to generate the policies. Again, taking the brokered authentication example from the last section the following information has to be supplied:

- policy of the identity provider
• address of the identity provider
• type of token

The policy file of the identity provider is the first information that is necessary in order to execute the brokered authentication protocol as this is responsible to tell the SeAAS engine in the service requester’s domain what security requirements have to be met to get a valid token. Of course also the address of the identity provider has to be given. This way it is possible to replace one identity provider by another. The type of the token to request is also an important information.

This is just an excerpt of the technical information that is necessary when choosing brokered authentication. The complete list will be shown and discussed in chapter 5.2.3.

In this chapter a general overview of the concepts which were elaborated during this thesis was given. The transformation process as well as the input and its output was explained. The next chapter illustrates the specific details of this concept which were worked out during its implementation.
5. Prototype

This chapter presents the prototypical implementation of the concepts which were elaborated in the context of this thesis. In order to explain the implementation specific concepts we rely on the running example from chapter 4. In this example we concentrate on the two security requirements authentication and non-repudiation. On one hand these are the most important and interesting requirements and on the other hand they are different in the way that the first one is applicable at service level only and the latter one at message level.

The first section of this chapter introduces the graphical user interface which guides the user through the transformation process. We then elaborate the meta-models, which were created during this thesis. We then examine the transformation process in more detail, as this is the main part of the implementation. In a final section we present the process of generating the actual code. The last section discusses the extensibility of the prototype.

5.1. Graphical User Interface

Here we present the graphical user interface of the prototype. The prototype is implemented as an Eclipse plug-in. The GUI basically consists of two parts: an input screen for importing the models and one for performing the refinement. We introduce these two input screens in detail in the upcoming two sections.

5.1.1. Refinement Steps

After successfully having imported the Unified Modeling Language (UML) model which defines the processes and security requirements, the Security Engineer has to perform the refinements. The goal of these refinement steps is to provide additional information to the process of transforming the input models into policies. Using these refinements is an efficient way to provide this process with information (e.g. technical details) that could not have been stored within the source models by the Domain Expert. This task performs the Security Engineer who possesses the required expertise. Therefore we also generated an input screen, more specifically, we generated an eclipse wizard.
We implemented an own wizard for each security requirement. That means for every security requirement the Domain Engineer can assign to a message in the workflow model, there exists a corresponding wizard in our prototype. The Security Engineer can then choose the security requirement he wants to refine in the context menu illustrated in figure 5.1. This context menu pops up when the Security Expert clicks with the right mouse button on the Sectissimo-MDS-Meta-Model instance. Under the group Sectissimo he can choose a security requirement to refine.

![Figure 5.1: The Context Menu in Eclipse with our Plug-in.](image)

We decided to combine (have to be refined at once) the two security requirements integrity and confidentiality as they are technically related to each other. That means just the refinement steps have to be performed for these two security requirements together. Of course it is still possible for the Domain Engineer to assign either integrity or confidentiality. In this case the Security Engineer would also have to refine only one of them. Just in case the Domain Engineer assigned both requirements on the same message in the workflow model the Security Engineer has to perform the refinement steps for both at the same time. This way we circumvent the task of combining or merging the security bindings of them. Because for web services the encryption and signature share the same security binding. If we would allow to refine these two separately from each other we would end up with two security bindings. Merging these two would not be that trivial.
As soon as the Security Expert decides to refine one of the security requirements, the corresponding wizard shows up. The wizard consist of several pages. These pages vary from one security requirement to another. But the general structure is the following:

- On the first page the Security Engineer has to choose a specific pattern or protocol for the security requirement he is refining.
- On the second page he has to specify the technical details that are required for the pattern chosen on the first page.
- The third page collects the information for the security binding of the requirement.

Figure 5.2 illustrates the first page of the wizard that treats *authentication*. This page corresponds to the first refinement step, where the architectural pattern is chosen. Here the Security Expert is provided with the corresponding patterns that are available for each security requirement. In case of *authentication* these two patterns are *Direct Authentication* and *Brokered Authentication* as already explained in chapter 4.4.2.

![Figure 5.2: Wizard Page for Authentication.](image)

After the Security Engineer has made his decision, the next page in this wizard is shown to him. Figure 5.3 illustrates the page that corresponds to the *Direct Authentication* pattern. This page is responsible for the second refinement step. So here the Security Expert has to provide the technical details for the chosen pattern. Again, the Security Expert is supported with values to choose for the elements within this pattern. In this figure one can see elements that are required and others are optional. The optional ones can be activated by selecting the corresponding radio button with the label *Yes*. Additionally we also implemented an error detection. Basically this is necessary to check whether all required fields are filled out or not.
In the latter case an error message is shown to the Security Engineer stating what has to be done in order to be able to get to the next page. An error does not only occur if a required element is not filled out but also when the radio button with label Yes is selected but the field is still empty, as it is the case in figure 5.3.

![Wizard Page for Direct Authentication.](image)

**Figure 5.3:** Wizard Page for Direct Authentication.

Both patterns for authentication do need a Security Binding. The purpose of this Security Binding is to define how the security (e.g. encryption) is realized in technical terms by the endpoints. The type of binding has also to be chosen on this wizard page. As the Security Binding itself needs some additional information we created a third page for this purpose where this information can be entered.

So in total there are three wizard pages for authentication. These three pages have to be completed for each of the involved services, as the security requirement authentication is assigned to a service. The wizard shows these three pages in this order for each service. In figures 5.2 and 5.3 one can see in the heading of the wizard pages there is the name of the service (Store) which the Security Engineer is refining with this page. As all other security requirements can be assigned per interaction, the wizard shows the corresponding pages for each of them. And in the heading of these pages instead of the name of the service the interaction (source, target, service, and operation) is displayed there.

Once the Security Engineer has entered all required information, the policy files are generated and saved within the folders of the actors’ domains which were created
during the import phase. In the next section we will present the graphical user interface of this import phase.

### 5.1.2. UML Model Import

The model import input screen is responsible to import the UML models which were created by the Domain Engineer. During this step the information stored within these models is transformed into a single model that conforms to our *MDS-Meta-Model* which will be presented in section 5.2.2. We integrated this input screen into the standard eclipse creation wizard which can be reached by *File → New → Other…* within Eclipse. There the *Sectissimo - Model Driven Security* import wizard is located in an own folder (*Sectissimo*) as depicted in figure 5.4.

![Select a wizard](image)

**Figure 5.4:** The Integration of our Wizard within Eclipse.

By choosing this wizard the actual input screen (figure 5.5) is revealed. There the Security Engineer has to specify three items:

- **UML-Model** - this field points to the UML file that stores the input models.
- **Container** - a project (or folder) has to be selected that will subsequently save the generated policies
- **File name** - the name of the *Sectissimo-MDS-Meta-Model* instance
In order to import a model that was created with a UML Case Tool, the file has to be exported as a XML Metadata Interchange (XMI) file within the used modeling tool (e.g. MagicDraw\(^1\)). This task has to be carried out by the Domain Engineer, who initially created these models. XMI is a standard created by the Object Management Group (OMG) to support tool independence and interoperability. When exporting the models from MagicDraw, beside the main file, that stores the information modeled by the Domain Expert, there are also files created which save all UML profiles used within the model. It is necessary that all files are located within the same folder when the Security Engineer imports them using our framework.

During this import step the UML models are transformed into our Sectissimo-MDS-Meta-Model. When finished, a new file is created in the selected container. Section 5.2.2 illustrates the instance of our use case models conforming to this meta model. Additionally, folders for each partner of the business process are generated. Subsequently the policy files will be stored separately within these folders.

Here we presented the graphical user interface of our prototype. In the next section the meta-models - the conceptual underpinning of our framework - are introduced. In section 5.3 the relationship between the GUI and the meta-models is presented, as this is a key concept within our prototype. The reason why we coupled these two elements together is that it will then be easy for Framework Engineers to extend our

\(^1\)http://www.magicdraw.com/
prototype with protocols of their choice. How this can be done will be presented in section 5.5.

5.2. Meta-Models

The framework introduced in chapter 4 leverages an approach based on the Model Driven Development (MDD) approach. Therefore it is necessary to have a set of underlying meta-models. This section presents the core meta-models. Before we introduce them we show the language that was used to create these meta-models.

5.2.1. Type of Meta-Models

Regardless of whether one wants to create a program, model or meta-model it is necessary to make the decision on the language to use. In case of creating meta-models there are basically two paths which could be followed:

- top down creation using UML
- bottom up creation using ecore

Creating meta-models using the UML language is a task of restriction. Starting with the whole scope of UML, one has to constrain and restrict the parts which are essential for the resulting meta-model. Therefore it is called top down.

Ecore on the other hand follows the opposite path. Starting from scratch one is able to create the desired language by oneself. That is why it is called bottom up. This language is lightweight in the sense that it provides exactly what was specified and nothing more. The main component of an ecore meta-model are eclasses. They are comparable to classes in object oriented languages and shape the scaffold of modeled languages.

For our prototype we made the decision to use ecore as our meta-modeling language. Because our prototype is an Eclipse plug-in and ecore is well integrated into Eclipse as well. Ecore is a component of the Eclipse Modeling Framework (EMF).

The next two sections introduce the meta-models which were elaborated as part of this thesis.

\[http://www.eclipse.org/modeling/emf/\]
5.2.2. Working Model

In this section we introduce the meta-model which serves as an intermediate for all further transformations and code generations.

The first step is to import the UML diagrams as introduced in chapter 4.2. During this process the information modeled within these diagrams is transformed into a single model. This model conforms to our Sectissimo-MDS-Meta-Model (figure 5.6).

Figure 5.6: The Sectissimo-MDS-Meta-Model.

The root eclass of this meta-model is MDSMetaModel. This eclass has a containment relation to the following three eclasses:

- **Type** - responsible to save the structure of the messages
- **Partner** - saves all involved partners
• Service - models the services

An instance of the MDSMetaModel can model an unlimited number of services. The eclass Service does have the attributes name, namespace and prefix. It does also have a containment relation to the eclass ServicePolicy which again has a containment relation to the eclass Authentication. This states that the security requirement authentication can be applied to a service. Additionally the Service has a containment relation to the eclass Operation. This eclass does have a single attribute (name) and containment relations to the eclasses Request and Response. Both of these eclasses do have a reference to the eclass Type. So far that means a service can have an unlimited number of operations. These operations do have a request type and can have a response type. Both eclasses Request and Response do have a containment relation to the eclass MessagePolicy which again has a containment relation to the eclasses Integrity, Confidentiality, Authorization and NonRepudiation. This states that these security requirements can be applied to the request and response message of an operation separately. Additionally, the eclass MessagePolicy does have two references to the eclass Partner. This way it is possible to store the sending and receiving partner to the message policy. This information is necessary to generate the policy files for the right partners.

The reason why we transform the UML input models into this meta-model is that this way it becomes easier to retrieve and work with the information stored in this model.

ATL Transformation Language (ATL) is used for this transformation. Due to peculiarities of this transformation engine there were two distinct transformations necessary. Chapter A.2 in the appendix illustrates these two transformations.

Listing 5.1 shows an instance of the Sectissimo-MDS-Meta-Model corresponding to the use case scenario. We present the structure of this model further by explaining the Warehouse service in more detail. Like all other services, the Warehouse is defined within a service element (line 38). One will notice that beside the name there is also the namespace and prefix defined. This information can also be extracted from the UML diagrams during initial import and transformation. Therefore the Domain Expert or Domain Engineer has to additionally provide this information within the interface model (figure 5.7) when creating the input models. Once these two attributes are added within the interface model they are also exported from MagicDraw and recognized within our framework during the import phase. The namespace and prefix have their own attributes in the service class within the interface model. In the service element the operation OrderItem (lines 39 - 53) and the authentication security requirement (line 55) are defined. All other security requirements, like non-repudiation, are defined within an own policy element in the request
and response element. Additionally, this policy element does have two attributes: source and target. They store the involved partners for each interaction.
Prototype

Listing 5.1: Sectissimo-MDS-Meta-Model Instance from the Use Case Scenario.

```xml
<operation>
<policy>
  <authentication/>
</policy>
</service>
<type name="OrderItemRequest">
  <variable name="item" type="Item"/>
  <variable name="quantity" type="int"/>
</type>
<type name="PlaceOrderResponse">
  <variable name="status" type="Status"/>
</type>
<type name="ValidateCCRequest">
  <variable name="creditCard" type="CreditCard"/>
</type>
<type name="OrderItemResponse">
  <variable name="status" type="Status"/>
</type>
<type name="PlaceOrderRequest">
  <variable name="item" type="Item"/>
  <variable name="quantity" type="int"/>
  <variable name="creditCard" type="CreditCard"/>
</type>
<partner name="Customer"/>
<partner name="CreditCardCompany"/>
<partner name="Warehouse"/>
<partner name="Store"/>
</mdsmm:MDSMetaModel>
```

Figure 5.7: The Class for the Warehouse within the Interface Model.

So it is possible to save all information that can be retrieved out of the UML diagrams within an instance of this meta-model. But, as already explained, this information does not suffice to generate the policies. Therefore the transformation process, where additional information is provided, is needed. For that purpose we also generated a set of meta-models which will be examined next.
5.2.3. Decision Models

This section introduces the decision models that were elaborated for the transformation process. As the transformation process consists of two refinement steps, we created meta-models for each of these refinement steps.

**Architectural Decision**

The architectural decision is the first refinement step. Therefore we generated a meta-model for each security requirement. Within these meta models the available patterns are modeled.

Figures 5.8 and 5.12 illustrate the available patterns for authentication and non-repudiation. They correspond exactly to the class diagrams from chapter 4.4.2. Based on our decision to use ecore models, we had to model these diagrams again using ecore. The ecore classes Authentication and NonRepudiation are abstract and do have a single attribute called isRequired. This attribute states that these classes are necessary for the transformation. What impact this attribute has in our framework will be discussed in section 5.3. The available patterns for both, authentication and non-repudiation are non abstract subclasses of them.

![Figure 5.8: The Meta-Model for the Authentication Security Requirement.](image)

Of course there are many other available protocols for both, authentication and non-repudiation. Centralized or distributed authentication would be other existing protocols for authentication. An additional protocol for non-repudiation would be probabilistic non-repudiation [MR99]. During the implementation of our prototype we had to adjust the amount of patterns to the available amount of resources (time to code). So we decided to implement only the patterns for DirectAuthentication and BrokeredAuthentication (figure 5.8) and NaiveNonRepudiation and FairNonRepudiation (figure 5.12). Chapter 4.4.2 already explained how we implemented the protocol for brokered authentication. Section 5.3 will present the protocols for naive- and fair-non-repudiation.
Next we will exemplify the remaining meta-models. They are used for the technical details within the second refinement step.

**Technical Details**

The meta-models presented in this section specify the information necessary for the realization of each pattern. Therefore we created a meta-model for each of the available patterns. So in the end we came up with meta-models for each security requirement and each pattern available as realization options for these security requirements.

![Diagram of the Meta-Model for the Direct Authentication Pattern.](image)

**Figure 5.9:** The Meta-Model for the Direct Authentication Pattern.

Figure 5.9 shows the meta-model for direct authentication. All information necessary to provide this pattern is modeled within this meta-model. So the policy file will be generated (section 5.4) out of this model. For example, this pattern requires a security binding. This is modeled with the abstract eclass *SecurityBinding.*
The supported patterns for the security binding are defined using the non abstract sub eclasses `SymmetricBinding` and `AsymmetricBinding`. The version of the used token would be another example. There the Security Engineer can either choose `WssUsernameToken10` or `WssUsernameToken11`.

Another example can be seen in figure 5.10. It shows the meta-model for naive non-repudiation. Again, this meta-model illustrates the necessary information for the underlying pattern or protocol, naive non-repudiation. This protocol does also need a security binding, indicated by the abstract eclass `SecurityBinding`. Additionally, it is possible to provide the protocol with the information which part of the message to sign or encrypt. This is depicted by the two eclasses `SignedElements` and `EncryptedElements`. Within these two eclasses a new attribute is introduced, `queryModel`. This indicates to our framework that the document model has to be queried for that information. Specifically, the `Sectissimo-MDS-Meta-Model` is searched for the information that was extracted out of the document model. As a result the Security Expert is provided with this information when making the decision which elements to sign and encrypt.

In this section we introduced the meta-models we created for our prototype. Furthermore, we examined the models for direct authentication and naive non-repudiation in more detail. The meta-model for the fair non-repudiation pattern is discussed in the next section. The meta-model for brokered authentication is illustrated in A.3 in the appendix. The upcoming section will now treat the transformation process. We also reveal the relationship between the meta-models and the graphical user interface. Additionally we will also introduce the behaviour of the non-repudiation protocols within our prototype.
5.3. Transformation

This section explains the implementation specific details of the two refinement steps within our framework. Specifically, the relationship between the meta-models and the graphical user interface will be explained. Therefore we will concentrate on the non-repudiation security requirement and show the steps on basis of our use case scenario. In section 5.4 we will then show how the policy files are generated after the Security Engineer has performed these refinement steps.

5.3.1. 1st Refinement

As already explained in chapter 4.4.2 the first refinement step is making the decision for the architectural pattern. Therefore a wizard page is generated where the Security Expert has to choose one of the available patterns. Figure 5.11 shows this wizard page for non-repudiation. This wizard page as well as the one already presented in section 5.1.1 is generated entirely automatic.

![Figure 5.11: Wizard Page for Non-Repudiation.](image)

That means we just implemented the basic skeleton, or in other words the rough structure of these pages. The actual elements that have to be specified and the corresponding values are retrieved out of the meta-models introduced in section 5.2.3. So these meta-models are used to create the input forms. Figure 5.12 illustrates the meta-model that is responsible to configure the form from figure 5.11.

So we do not use these meta-models to instantiate them as it would usually be the case, but we use the information stored within them to fill the wizard pages with this information.
During design time, in the creation step of the wizard, the corresponding meta models for the chosen security requirement are queried. Therefore, first all abstract eclasses are retrieved. They define the elements which can be specified by the Security Engineer. Additionally, these abstract eclasses can have the attribute `isRequired`, indicating that a value has to be assigned to this specific element. Otherwise the wizard page shows an error message and the Next button is disabled. The values that can be assigned to an element are defined by non-abstract sub eclasses. In our case these two eclasses for `NonRepudiation` are `NaiveNonRepudiation` and `FairNonRepudiation`. Therefore the wizard page (figure 5.11) creates as first step a label with the text of the abstract class and as a second one a combo box with these two values. When the Security Experts makes his decision for one of these patterns the corresponding protocol is implicitly chosen. These patterns or protocols do need some additional information. This information is provided by the second refinement step that will be discussed in the upcoming section. First we want to examine one of these protocols further. Figure 5.13 shows the protocol for fair non-repudiation modeled as a sequence diagram.

This protocol executes between the following three domains:

- Domain 1 - business service requester’s domain
- Domain 2 - business service provider’s domain
- Domain 4 - the trusted third party

Domain 1 (Store) has a Security As A Service (SeAAS) engine, Basic Security Services (BSS) and the Non-Repudiation Service. This service is implemented with the help of Web Services Business Process Execution Language (WS-BPEL). The Non-Repudiation Service inside Domain 1 has the role of the origin of the message (NRO). Domain 2 (Warehouse) does have the same services with the only difference that the Non-Repudiation Service has the role of the recipient (NRR) here. Domain
Figure 5.13: Our Implementation of the Fair Non-Repudiation Protocol

3 implements the trusted third party (TTP) also with the help of WS-BPEL. Additionally it also has the basic security services (BSS) deployed. The SeAAS engines of domain 1 and 2 initialize the protocol with the corresponding message.

The basic flow of this protocol is implemented statically. Dynamic information like the address of the trusted third party are retrieved dynamically from a metadata exchange service and omitted in this diagram. This information is referred to the second refinement step, the technical details. This will be discussed in the next section. The sequence diagram of the remaining non-repudiation protocol, naive non-repudiation, is illustrated in chapter A.4 in the appendix.
5.3.2. 2nd Refinement

The second refinement step is about technical details. Once the Security Engineer has chosen an architectural pattern within the wizard, the next page will be shown. In this wizard page he then has to specify the technical details corresponding to the chosen pattern. Figure 5.14 shows the page for fair non-repudiation.

![Wizard Page for Fair Non Repudiation.](image)

Instead of the service name the interaction is displayed in the heading of the page. The first thing the fair non-repudiation protocol needs is the type of binding, either symmetric or asymmetric. As usual this information is retrieved out of the corresponding meta-model (figure 5.15). The protocol does also need a trusted third party as indicated by the abstract eclass `TrustedThirdParty`. This abstract eclass does not have any sub eclasses. This is because the framework can not make any suggestions for the trusted third party. The framework can not provide the Security Engineer with a set of trusted third parties to choose from. But of course, this information is necessary using a fair non-repudiation protocol. That is why it has to be in the meta-model. So, abstract eclasses without any subtypes offer more flexibility to the Security Expert as the value can be directly inserted by him.

Additionally, in the wizard page it is possible to specify the elements to sign or encrypt during the protocol execution. These two elements are optional. Therefore the attribute `isRequired` is not assigned to them within the meta-model. Instead, the attribute `queryModel` is assigned to these eclasses. This indicates that the values have to be retrieved from the document model. So in our running example (figure 4.2), where non-repudiation is specified between the `Store` and the `Warehouse` on the
message $OrderItemRequest$, the Security Engineer can choose one of the following elements:

- item
- quantity

as illustrated in figure 5.14. During the modeling phase of the security enhanced workflow diagram it was only possible to specify a security requirement on message level. Now, using the concept of refinement, it became even possible to decrease the granularity from message level to element/parameter level as suggested in chapter 4.2.

Providing these technical details is the last step in the refinement phase of our framework. Right after that, the policies will be generated. The process of generating these policy files will be discussed in the upcoming section.

This section discussed the transformation which was initially introduced in chapter 4.4 in more detail. To be more precise, we elaborated on implementation specific aspects and the relation between the graphical user interface and the meta-models. The next section will now present the generation of the Web Services Security Policy (WS-Sec-Policy) files.
5.4. Code Generation

The last step in our prototype is the generation of the policy files which will subsequently configure the SeAAS engines. We decided to use Java Emitter Templates (JET) for this task.

We defined a template for each policy that can be generated using our prototype. Essentially, there is a template for each of the meta-models that define the second refinement step (technical details). Figure 5.2 illustrates such a JET template. For practical reasons the namespaces are omitted. This template is responsible to generate the policy file for the fair non-repudiation protocol.

```xml
<% @ jet package="info.sectissimo.mds.transformation" imports="java.util.HashMap" class="FairNonRepudiation" %>

HashMap input = (HashMap) argument;
<xml version="1.0" encoding="UTF-8">
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp:All>
      <%=%input.get("Binding")%>
      if (input.containsKey("SignedElements")) {
        <sp:SignedElements>
          <sp:XPath xmlns:data="http://sectissimo.info/security/service/nrp/common" data:<%=%input.get("SignedElements")%>/*/</sp:XPath>
        </sp:SignedElements>
      }
      <%>
      if (input.containsKey("EncryptedElements")) {
        <sp:EncryptedElements>
          <sp:XPath xmlns:data="http://sectissimo.info/security/service/nrp/common" data:<%=%input.get("EncryptedElements")%>/*/</sp:XPath>
        </sp:EncryptedElements>
      }
      <%>
      </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 5.2: The JET Template for Fair Non-Repudiation.

Basically, these templates define the static structure of the resulting policy file. Dynamic information, like which element to encrypt, is provided by a HashMap. This stores the values chosen or entered by the Security Expert. The keys of the

---

<http://www.eclipse.org/modeling/m2t/?project=jet>
Prototype

HashMap again correspond exactly to the abstract eclasses from the corresponding meta-model.

In chapter 4.4.2 we mentioned that the framework needs an additional policy file for the non-repudiation security requirement. This policy file is responsible to instruct the SeAAS engine which protocol (fair or naive) to trigger. This policy file (figure 5.3) is also generated using a JET template.

```xml
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp:All>
      <nrp:NRProtocolType xmlns:nrp="http://sectissimo.info/services/security/nrp/protocolttype">
        <if input.get("NonRepudiation").equals("NaiveNonRepudiation"){
          <nrp:NonFairNRProtocol/>
        }
        <else if input.get("NonRepudiation").equals("FairNonRepudiation"){
          <nrp:FairNRProtocol/>
        }
      </nrp:NRProtocolType>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 5.3: The JET Template for Non-Repudiation.

So for non-repudiation we implemented a “generic” protocol which will subsequently call the concrete protocol based on this policy file. Figure 5.16 illustrates this behaviour using a sequence diagram.

Like the fair and naive non-repudiation protocols, we implemented the generic one also with the help of WS-BPEL. As a result we ended up with a service for the origin of the message and the recipient. In figure 5.16 the message flow from the origin is illustrated. The message flow at the recipients side is similar. First, the SeAAS engine initializes the Generic Non-Repudiation Service with the payload of the message that was sent from one domain to another. The Generic Non-Repudiation Service then queries the Metadata Exchange Service in order to retrieve
the policy stating which protocol to trigger. Based on this policy file the \textit{Generic Non-Repudiation Service} then initializes the corresponding protocol.

In this section the generation of the actual policy files was presented. So now we explained the whole prototype starting with the graphical user interface, the meta-models and the transformation process. In the remaining section we now want to emphasize the extensibility of our prototype.

5.5. Extensibility

The building blocks of our prototype are the meta-models. They are responsible to generate the graphical user interface and forward the information to the JET templates. In order to extend our framework the main part is to adapt these policy files to own needs.
In this section we go through those tasks that have to be performed to extend the prototype. Therefore we explain them by means of an example. Suppose the Framework Engineer wants to add a third non-repudiation protocol to the architecture. In a first step he has to add another subclass to the abstract eclass NonRepudiation in the meta-model from figure 5.12. Then he has to create a new meta-model describing the details of this new protocol following the conventions introduced in this chapter (section 5.2). Corresponding to this meta-model and the requirements of his protocols he then has to create a JET template to actually generate the required policy file. Additionally, he also has to change the JET template that creates the policy file for the generic non-repudiation service. In there he simply has to add the option for his new protocol. As a last step he has to implement the actual protocol using WS-BPEL for example and register it to the SeAAS architecture.

With these changes the Security Engineer is subsequently able to choose this new protocol, provide the technical details and generate the policy files.

Now we presented the whole prototype we implemented on the basis of the concepts from chapter 4. Additionally we introduced the possibility to extend this prototype to the own needs in a fairly simple way. The upcoming chapter will summarize these things up and give an outlook on things that could be added to this framework and prototype.
6. Summary

The aim of this thesis is to implement the concept of Model Driven Security Engineering (MDSE) within a Service Oriented Architecture (SOA) environment. Moreover, we tried to solve this problem in a decentralized way. This is very important in order to preserve scalability as there are usually various business partners (and thus business services) involved in a single workflow when dealing with SOA environments.

When following the concept of MDSE there is always a specific target architecture needed. For this purpose we selected the SECTISSIMO [HMB09b] approach. There, a dedicated security engine exists that handles all security related tasks. The business services are not even aware of any security. Hence, only a single component has to be configured for performing the security operations. Using this architecture it is possible to keep the amount of the generated and required artefacts small. However, this does not imply that the policies that are responsible to configure the security engine have to be stored in this centralized place.

The target architecture depicts the Platform Specific Model (PSM) within the concept of MDSE. In addition to that also a Platform Independent Model (PIM) is needed. Therefore, we generated our own Unified Modeling Language (UML) profile that is equipped with the security requirements which can interpreted by our prototype as well as the target architecture.

Finally, the MDSE concept needs some kind of transformation. In this thesis we utilized the transformation concept from [Mem11]. There the concept of two distinct transformations is introduced.

In this thesis we also presented a prototype that implements all these different concepts. During the development of this prototype we detected three main issues that could further be worked on.

- In this prototype we only generate the policies that configure the Security As A Service (SeAAS) engine. It would make sense to generate the Web Services Description Language (WSDL) files for the services as well. As all information necessary to create those files are already mapped to the input models this would be a convenient way to provide the Application Administrator with the latest interface files. Furthermore, it would be useful to generate the deployment descriptors for the Enterprise Service Buss (ESBs) too. The big
benefit of automatically creating the deployment descriptors as well as the WSDL files is a reduced error-proneness. Because this information has to be provided in the non human-readable Extensible Markup Language (XML) format.

- Another issue is the problem of efficiently managing and changing the policy files. They have to be adapted as soon as the business workflow or the security requirements change. It is not easy to manually manage different versions of configuration files, so an automatically management is desired which yields us to the third issues we encountered during this thesis.

- Integrating a way to automatically deploy all resources would make the life of the Application Administrator a lot easier and solve the problem from above (2nd bullet).
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Transformation Process within MDD [For11]</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Model Correspondence regarding the Transformation [The10]</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>SeAAS Conceptual Architecture [HMB09a]</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>SECTET Framework [Mem11]</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Roles in the SECTET Framework [Mem11]</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Message Flow within the Use Case Scenario</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Policy Dependencies between the Actors</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Policy Dependencies between Policies</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Use Case Scenario deployed on the SeAAS Architecture</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>The Life Cycle of Policy Generation</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Workflow Model with Security Requirements enhanced</td>
<td>21</td>
</tr>
<tr>
<td>4.3</td>
<td>Interface Model with Security Requirements enhanced</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Document Model with Security Requirements enhanced</td>
<td>22</td>
</tr>
<tr>
<td>4.5</td>
<td>Example for different Interactions</td>
<td>23</td>
</tr>
<tr>
<td>4.6</td>
<td>Activity Diagram with Different Interactions</td>
<td>24</td>
</tr>
<tr>
<td>4.7</td>
<td>The Sectissimo UML Profile</td>
<td>24</td>
</tr>
<tr>
<td>4.8</td>
<td>Patterns available for the Authentication Requirement</td>
<td>30</td>
</tr>
<tr>
<td>4.9</td>
<td>Patterns available for the Non-Repudiation Requirement</td>
<td>30</td>
</tr>
<tr>
<td>4.10</td>
<td>The Brokered Authentication Protocol implemented by the SeAAS Engine</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>The Context Menu in Eclipse with our Plug-in</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Wizard Page for Authentication</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Wizard Page for Direct Authentication</td>
<td>36</td>
</tr>
<tr>
<td>5.4</td>
<td>The Integration of our Wizard within Eclipse</td>
<td>37</td>
</tr>
<tr>
<td>5.5</td>
<td>The Import Screen</td>
<td>38</td>
</tr>
<tr>
<td>5.6</td>
<td>The Sectissimo-MDS-Meta-Model</td>
<td>40</td>
</tr>
<tr>
<td>5.7</td>
<td>The Class for the Warehouse within the Interface Model</td>
<td>43</td>
</tr>
<tr>
<td>5.8</td>
<td>The Meta-Model for the Authentication Security Requirement</td>
<td>44</td>
</tr>
<tr>
<td>5.9</td>
<td>The Meta-Model for the Direct Authentication Pattern</td>
<td>45</td>
</tr>
<tr>
<td>5.10</td>
<td>The Meta-Model for the Naive Non-Repudiation Pattern</td>
<td>46</td>
</tr>
<tr>
<td>5.11</td>
<td>Wizard Page for Non-Repudiation</td>
<td>47</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.12</td>
<td>The Meta-Model for the Non-Repudiation Security Requirement</td>
<td>48</td>
</tr>
<tr>
<td>5.13</td>
<td>Our Implementation of the Fair Non-Repudiation Protocol</td>
<td>49</td>
</tr>
<tr>
<td>5.14</td>
<td>Wizard Page for Fair Non Repudiation</td>
<td>50</td>
</tr>
<tr>
<td>5.15</td>
<td>The Meta-Model for the Fair Non-Repudiation Pattern</td>
<td>51</td>
</tr>
<tr>
<td>5.16</td>
<td>The Initialization Phase of the Non-Repudiation Protocol</td>
<td>54</td>
</tr>
<tr>
<td>A.1</td>
<td>Available patterns for the authorization requirement</td>
<td>65</td>
</tr>
<tr>
<td>A.2</td>
<td>Available patterns for the confidentiality requirement</td>
<td>65</td>
</tr>
<tr>
<td>A.3</td>
<td>Available patterns for the integrity requirement</td>
<td>65</td>
</tr>
<tr>
<td>A.4</td>
<td>The meta-model for the brokered authentication pattern</td>
<td>71</td>
</tr>
<tr>
<td>A.5</td>
<td>The protocol implemented for naive non repudiation</td>
<td>72</td>
</tr>
</tbody>
</table>
List of Listings

4.1. SeAAS Policy File .................................................. 24
4.2. BSS Policy File ..................................................... 25
4.3. Binding with Policy Reference .................................... 27
4.4. Generic Non-Repudiation Policy ................................. 30

5.1. Sectissimo-MDS-Meta-Model Instance from the Use Case Scenario. . 42
5.2. The JET Template for Fair Non-Repudiation ..................... 52
5.3. The JET Template for Non-Repudiation .......................... 53

A.1. First ATL transformation ............................................ 66
A.2. Second ATL transformation ........................................ 67
List of Acronyms

ATL ........... ATL Transformation Language

DSML .......... Domain Specific Modeling Language

ELGA .......... Elektronische Gesundheits - Akte

EMF .......... Eclipse Modeling Framework

ESB .......... Enterprise Service Bus

HTTP .......... Hypertext Transfer Protocol

IDE .......... Integrated Development Environment

JET .......... Java Emitter Templates

JMS .......... Java Message Service

JSP .......... Java Server Pages

MDD .......... Model Driven Development

MDS .......... Model Driven Security

MDSE .......... Model Driven Security Engineering

OMG .......... Object Management Group

PDE .......... Plug-in Development Environment

PIM .......... Platform Independent Model

PSM .......... Platform Specific Model

SeAAS .......... Security As A Service

SOA .......... Service Oriented Architecture
List of Acronyms

UML ............ Unified Modeling Language
WS-BPEL ........ Web Services Business Process Execution Language
WS-Policy ........ Web Services Policy
WS-Sec-Policy ... Web Services Security Policy
WS-Security ...... Web Services Security
WSDL ............. Web Services Description Language
XMI .............. XML Metadata Interchange
XML .............. Extensible Markup Language
Bibliography


A. Technical Details

A.1. Architectural Decision Patterns

Figure A.1: Available patterns for the authorization requirement.

Figure A.2: Available patterns for the confidentiality requirement.

Figure A.3: Available patterns for the integrity requirement.
Technical Details

A.2. ATL Transformations

```
@path MDS=/info.sectissimo.transformation.m2m/metamodel/MDS-Meta-Model.ecore

@nsURI UML=http://www.eclipse.orguml2/2.0.0/UML

module step1;

create OUT: MDS from IN: UML;

helper context UML!Element def: hasStereotype(name : String) : Boolean =
    not self.getAppliedStereotype(name).oclIsUndefined();

rule model2mds {
    from s: UML!Model
to
t: MDS!MDSMetaModel {
    type <- s.packagedElement->select(e | e.oclIsTypeOf(UML!Class))->
        select(f | f.hasStereotype('MDS-Profile::message'))->collect(g |
            thisModule.class2type(g)),
    partner <- UML!ActivityGroup.allInstancesFrom('IN')->collect(h |
            thisModule.group2partner(h))
}

lazy rule class2type {
    from s: UML!Class
to
t: MDS!Class {
    name <- s.name,
    variable <- s.ownedAttribute->collect(e | thisModule.
        property2variable(e))
}

lazy rule property2variable {
    from s: UML!Property
to
t: MDS!Variable {
    name <- s.name,
    type <- s.type.name
}

lazy rule group2partner {
    from u: UML!ActivityGroup
to
t: MDS!Partner {
    name <- u.name
}
}
```

Listing A.1: First ATL transformation.
Technical Details

1. @path MDS=info.sectissimo.transformation.m2m/metamodel/MDS-Meta-Model.ecore
2. @nsURI UML=http://www.eclipse.org/uml2/2.0.0/UML
3. module step2;
4. create OUT: MDS from UMLIN: UML, MDSIN: MDS;
5. helper context UML!Element def: hasStereotype(name : String) : Boolean =
   not self.getAppliedStereotype(name).oclIsUndefined();
6. helper context UML!Element def: hasNamespace() : Boolean =
   not self.ownedAttribute->any(e | e.name = 'namespace').oclIsUndefined();
7. helper context UML!Element def: hasPrefix() : Boolean =
   not self.ownedAttribute->any(e | e.name = 'prefix').oclIsUndefined();
8. rule model2mds {
   from
   u: UML!Model, m: MDS!MDSMetaModel
to
t: MDS!MDSMetaModel {
   type <= m.type,
   partner <= m.partner,
   service <= u.packagedElement->select(e | e.oclIsTypeOf(UML!Class))->
      select(f | f.hasStereotype('MDS-Profile::interface'))->collect(g |
         thisModule.class2service(g))
}
9. rule Type {
   from
   m: MDS!Type
to
t: MDS!Type{
   name <= m.name,
   variable <= m.variable
}
10. rule Partner {
   from
   m: MDS!Partner
to
t: MDS!Partner{
   name <= m.name
}
11. rule Variable {
   from
   m: MDS!Variable
to
t: MDS!Variable{
   name <= m.name,
   type <= m.type
}
lazy rule class2service {
  from u: UML! Class
to t: MDS! Service {
  name <- u.name,
  operation <- u.ownedOperation->collect (g | thisModule. operation2operation (g))
}
do {
  if (u.hasStereotype ('MDS-Profile::authentication')) {
    t.policy <- thisModule.node2policy();
  }
  if (u.hasNamespace()) {
    t.namespace <- u.ownedAttribute->any (e | e.name = 'namespace').defaultValue.value;
  }
  if (u.hasPrefix()) {
    t.prefix <- u.ownedAttribute->any (f | f.name = 'prefix').defaultValue.value;
  }
}
}
lazy rule operation2operation {
  from u: UML! Operation
to t: MDS! Operation {
  name <- u.name,
  request <- thisModule.parameter2request (u.ownedParameter->any (e | e.direction.toString () = 'in')),
  response <- thisModule.parameter2response (u.ownedParameter->any (e | e.direction.toString () = 'return'))
}
}
lazy rule parameter2request {
  from u: UML! Parameter
to t: MDS! Request {
  requestValue <- MDS! Type.allInstancesFrom ('MDSIN') -> any (e | e.name = u.type.name)
}
do {
  if (UML! CentralBufferNode.allInstancesFrom ('UMLIN') -> select (f | f.name = u.type.name)->size () > 0) {
    t.policy <- UML! CentralBufferNode.allInstancesFrom ('UMLIN') -> select (f | f.name = u.type.name)->collect (g | thisModule.node2mpolicy (g))
  }
}
}
lazy rule parameter2response {
  from
u: UML! Parameter
t: MDS! Response

\[
\text{responseValue} \leftarrow \text{MDS! Type.allInstancesFrom ('MDSIN')}\rightarrow \text{any}(e \mid e.\text{name} = u.\text{type.name})
\]

do

\[
\text{if (UML! CentralBufferNode.allInstancesFrom ('UMLIN')}\rightarrow \text{select}(f \mid f.\text{name} = u.\text{type.name})\rightarrow \text{size()} > 0)
\]

\[
\text{t.policy} \leftarrow \text{UML! CentralBufferNode.allInstancesFrom ('UMLIN')}\rightarrow \text{select}(f \mid f.\text{name} = u.\text{type.name})\rightarrow \text{collect}(g \mid \text{thisModule.node2mpolicy}(g))
\]

lazy rule node2mpolicy

from

u: UML! CentralBufferNode
to

t: MDS! MessagePolicy

\[
\text{source} \leftarrow \text{MDS! Partner.allInstancesFrom ('MDSIN')}\rightarrow \text{any}(e \mid e.\text{name} = u.\text{incoming}\rightarrow \text{first()}\rightarrow \text{source.inPartition}\rightarrow \text{first()}\rightarrow \text{name})
\]

\[
\text{target} \leftarrow \text{MDS! Partner.allInstancesFrom ('MDSIN')}\rightarrow \text{any}(e \mid e.\text{name} = u.\text{outgoing}\rightarrow \text{first()}\rightarrow \text{target.inPartition}\rightarrow \text{first()}\rightarrow \text{name})
\]

do

\[
\text{if (u.\text{hasStereotype ('MDS-Profile:authorization'))}
\]

\[
\text{t.authorization} \leftarrow \text{thisModule.authorization}(u);
\]

\[
\text{if (u.\text{hasStereotype ('MDS-Profile:integrity'))}
\]

\[
\text{t.integrity} \leftarrow \text{thisModule.integrity}(u);
\]

\[
\text{if (u.\text{hasStereotype ('MDS-Profile:confidentiality'))}
\]

\[
\text{t.confidentiality} \leftarrow \text{thisModule.confidentiality}(u);
\]

\[
\text{if (u.\text{hasStereotype ('MDS-Profile:non-repudiation'))}
\]

\[
\text{t.nonrepudiation} \leftarrow \text{thisModule.nonrepudiation}(u);
\]

lazy rule node2spolicy

from

u: UML! Class
to

t: MDS! ServicePolicy

\[
\text{authentication} \leftarrow \text{thisModule.authentication}(u)
\]

do

lazy rule authorization

from

u: UML! CentralBufferNode
to

t: MDS! Authorization

}
Technical Details

Listing A.2: Second ATL transformation.
A.3. Meta-Models

Figure A.4: The meta-model for the brokered authentication pattern.
Figure A.5: The protocol implemented for naive non repudiation.