

Hasso-Plattner-Institut für IT-Systems Engineering
Universität Potsdam, Germany

Model-driven Security in Service-oriented Architectures

Leveraging Security Patterns to Transform
High-level Security Requirements to Technical Policies

Dissertation

zur Erlangung des akademischen Grades

”Doctor rerum naturalium”

(Dr. rer. nat.)

am Fachgebiet Internet-Technologien und -Systeme

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von

Dipl. Inf. Michael Menzel

Potsdam, August 2011

This work is licensed under a Creative Commons License:
Attribution - Noncommercial - Share Alike 3.0 Germany
To view a copy of this license visit
<http://creativecommons.org/licenses/by-nc-sa/3.0/de/>

Published online at the
Institutional Repository of the University of Potsdam:
URL <http://opus.kobv.de/ubp/volltexte/2012/5905/>
URN <urn:nbn:de:kobv:517-opus-59058>
<http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-59058>

Disclaimer

I confirm that the work submitted is my own and that it does not contain unreferenced or unacknowledged material.

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und nur unter Zuhilfenahme der angegebenen Literatur.

Bonn, den 20. August 2011



Michael Menzel

Acknowledgements

Die vorliegende Dissertation ist am Hasso-Plattner-Institut in Potsdam entstanden und ist das Ergebnis eines jahrelangen Weges der Forschung im Rahmen des Forschungskollegs "Service-Oriented System Engineering". Ein herzlicher Dank gilt daher den Professoren des Hasso-Plattner-Instituts, die meine Forschung in all den Jahren begleitet haben. Insbesondere möchte ich meinem Doktorvater, Herrn Professor Dr. Christoph Meinel, danken, dass ich diesen Weg an seinem Lehrstuhl beschreiten konnte. Seine Unterstützung und Förderung gaben mir die Möglichkeit, den Forschungsbereich SOA Security mitzugestalten und Erfahrungen in interessanten Projekten mit externen Partnern zu sammeln.

Den Mitgliedern des Forschungskollegs und des Lehrstuhls Internet-Technologien und -Systeme möchte ich für die hilfreichen Diskussionen und die Unterstützung danken, die ich in den Jahren erfahren habe. Ein besonderer Dank gilt meinen unermüdlichen Kollegen Ivonne Thomas und Robert Warschofsky, die mit ihren Anmerkungen und Korrekturvorschlägen viel zur finalen Version der Dissertation beigetragen haben. Die in fruchtbarer Zusammenarbeit durchgeführten Lehrveranstaltungen, Vorträge und Arbeitsgruppentreffen zum Forschungsgebiet SOA Security haben geholfen, das Fundament für diese Dissertation zu legen. Im Rahmen eines dieser Seminare hat, gemeinsam mit Ivonne Thomas, die Arbeit am SOA Security LAB begonnen, welches die Verwendung des in dieser Dissertation vorgestellten modellgetriebenen Ansatzes demonstriert. Das SOA Security LAB in seiner jetzigen Form und der Erfolg beim IEEE Services Cup 2010 in Miami wären aber ohne die Unterstützung und Mitarbeit von Robert Warschofsky nicht möglich gewesen. Erst seine Umsetzung einer Abstraktionsschicht für Webservice-Policies im Rahmen seiner Masterarbeit und die kontinuierliche Zusammenarbeit an dem SOA Security LAB ermöglichten die Bereitstellung eines so komplexen Systems.

Meine Eltern Cornelia Menzel und Wilhelm Menzel sowie meine Verlobte Vera Cornel haben all die Jahre die Schwierigkeiten und Entbehrungen des wissenschaftlichen Weges mitgetragen und es immer gut verstanden, mich zu motivieren und Kraft zu geben. Dafür danke ich Ihnen vom ganzen Herzen.

Abstract

Service-oriented Architectures (SOA) facilitate the provision and orchestration of business services to enable a faster adoption to changing business demands. Web Services provide a technical foundation to implement this paradigm on the basis of XML-messaging. However, the enhanced flexibility of message-based systems comes along with new threats and risks. To face these issues, a variety of security mechanisms and approaches is supported by the Web Service specifications. The usage of these security mechanisms and protocols is configured by stating security requirements in security policies. However, security policy languages for SOA are complex and difficult to create due to the expressiveness of these languages.

To facilitate and simplify the creation of security policies, this thesis presents a model-driven approach that enables the generation of complex security policies on the basis of simple security intentions. SOA architects can specify these intentions in system design models and are not required to deal with complex technical security concepts.

The approach introduced in this thesis enables the enhancement of any system design modelling languages – for example FMC or BPMN – with security modelling elements. The syntax, semantics, and notion of these elements is defined by our security modelling language SecureSOA. The meta-model of this language provides extension points to enable the integration into system design modelling languages. In particular, this thesis demonstrates the enhancement of FMC block diagrams with SecureSOA.

To enable the model-driven generation of security policies, a domain-independent policy model is introduced in this thesis. This model provides an abstraction layer for security policies. Mappings are used to perform the transformation from our model to security policy languages. However, expert knowledge is required to generate instances of this model on the basis of simple security intentions. Appropriate security mechanisms, protocols and options must be chosen and combined to fulfil these security intentions. In this thesis, a formalised system of security patterns is used to represent this knowledge and to enable an automated transformation process. Moreover, a domain-specific language is introduced to state security patterns in an accessible way. On the basis of this language, a system of security configuration patterns is provided to transform security intentions related to data protection and identity management. The formal semantics of the security pattern language enable the verification of the transformation process introduced in this thesis and prove the correctness of the pattern application.

Finally, our SOA Security LAB is presented that demonstrates the application of our model-driven approach to facilitate a dynamic creation, configuration, and execution of secure Web Service-based composed applications.

Zusammenfassung

Service-orientierte Architekturen ermöglichen eine dynamische Bereitstellung und Orchestrierung von Diensten, um eine schnelle Anpassung an Geschäftsanforderungen zu gewährleisten. Webservices bieten die technologische Grundlage zur Umsetzung dieses Paradigmas auf der Basis einer nachrichtenbasierten Kommunikation. Um die neuen Risiken und Gefahren zu adressieren, die mit diesem dezentralen Ansatz einhergehen, unterstützen die Webservice-Spezifikationen eine Vielzahl von Sicherheitsmechanismen und Protokollen. Die Verwendung dieser Mechanismen wird, gemäß den Sicherheitsanforderungen, deklarativ in Sicherheitspolicies spezifiziert. Allerdings weisen Polycysprachen für SOA eine hohe Komplexität auf und sind fehleranfällig in der Verwendung.

Um die Generierung von Sicherheitskonfigurationen in komplexen Systemen zu vereinfachen, wird in dieser Arbeit ein modellgetriebener Ansatz vorgestellt, der eine Modellierung von Sicherheitsanforderungen in Architekturmodellen ermöglicht und eine Generierung von Sicherheitspolicies auf Grundlage dieser Anforderungen unterstützt. Die Modellierungsebene ermöglicht eine einfache und abstrakte Darstellung von Sicherheitsanforderungen, die sich auch Systemarchitekten erschließt, welche keine Sicherheitsexperten sind.

Der in dieser Arbeit vorgestellte Ansatz ermöglicht die Erweiterung beliebiger Systemmodellierungssprachen - beispielsweise FMC oder BPMN - mit sicherheitsbezogenen Modellierungselementen. Die Syntax, die Semantik und die Darstellung dieser Elemente werden durch unsere Sicherheitsmodellierungssprache SecureSOA spezifiziert. Erweiterungspunkte ermöglichen die Integration dieser Sprache in beliebige Systemmodellierungssprachen. Insbesondere wird in dieser Arbeit die Erweiterung von FMC-Blockdiagrammen mit SecureSOA demonstriert.

Um eine modellgetriebene Generierung von Sicherheitspolicies zu ermöglichen, spezifiziert diese Arbeit ein domänenunabhängiges Policymodell, das eine Abstraktionsschicht zu Sicherheitspolicies bildet. Allerdings kann eine Generierung von Policymodellinstanzen auf Grundlage der modellierten Anforderungen nur erfolgen, wenn im System Expertenwissen hinterlegt ist, das die Auswahl von Sicherheitsprotokollen, -mechanismen und -optionen bestimmt. Im Rahmen dieser Arbeit werden Entwurfsmuster für SOA-Sicherheit zur Transformation herangezogen, die dieses Wissen repräsentieren. Dazu wird ein Katalog von Entwurfsmustern eingeführt, der die Abbildung von Sicherheitsintentionen auf unser domänenunabhängiges Modell unterstützt. Diese Muster sind mittels einer domänenspezifischen Sprache (DSL) definiert, um eine einfache Spezifikation der Entwurfsmuster zu ermöglichen. Die formale Semantik dieser Entwurfsmustersprache ermöglicht die formale Verifikation des Transformationsprozesses, um die Korrektheit der Entwurfsmusteranwendung nachzuweisen.

Die Anwendbarkeit unseres Ansatzes wird durch das SOA Security LAB unter Beweis gestellt, das eine dynamische Konfiguration und Ausführung von sicheren Webservice-Szenarien ermöglicht.

Contents

1	Introduction	1
1.1	State of the Art in Model-driven Security	3
1.2	Contribution: A Framework for Model-driven Security	4
1.2.1	Modelling Security Requirements	4
1.2.2	A Domain-independent SOA Security Model	5
1.2.3	A Model-driven Transformation	6
1.3	Outline	8
2	SOA Security Concepts and Standards	9
2.1	Web Service Technology	9
2.1.1	Messaging with SOAP	10
2.1.2	WSDL - Describing Web Services	11
2.2	Secure Messaging	13
2.2.1	WS-Security	13
2.3	Identity Management	15
2.3.1	SAML	18
2.3.2	WS-Trust	19
2.4	Policy Management	19
2.4.1	WS-Policy	21
2.4.2	WS-SecurityPolicy	21
3	A Security Meta-Model for SOA	23
3.1	A Model for Service Interactions	23
3.1.1	The SOA Interaction Model	23
3.1.2	Modelling Digital Identities	25

3.2	Modelling Security Requirements	27
3.2.1	Security Policy Structure	27
3.2.2	Security Constraints for Authentication and Data Protection	28
3.3	Meta-Model Formalisation	30
3.3.1	Domain-Independent Relational Model	30
3.3.2	Security Constraints Formal Semantics	32
4	SecureSOA: A Language to Model Security Intentions	35
4.1	Providing Security Design Languages - Overview	35
4.1.1	Enhancing Modelling Languages	35
4.1.2	Defining Modelling Design Languages	36
4.1.3	Merging Security and System Design Languages	37
4.2	SecureSOA - A Security Design Language for SOA	38
4.2.1	SecureSOA Abstract Syntax	38
4.2.2	SecureSOA Concrete Syntax	42
4.2.3	SecureSOA Formal Semantics	43
4.3	A SecureSOA Dialect based on FMC	46
4.3.1	Fundamental Modeling Concepts	46
4.3.2	Merging SecureSOA and FMC	47
4.4	SecureSOA Modelling Example	48
5	Security Configuration Patterns for SOA Security	51
5.1	Security Patterns – State of the Art	51
5.2	Web Service Security Configuration Patterns	53
5.3	A Domain-specific Language for Security Configuration Patterns	54
5.3.1	Security Configuration Pattern Operations and Functions	54
5.3.2	Security Configuration Pattern DSL Formal Syntax	56
5.3.3	Security Configuration Pattern DSL Formal Semantics	59
5.4	Security Ontology	69
6	A System of Security Configuration Patterns	71
6.1	Patterns for Identification and Authentication	71
6.1.1	Basic Implementation Schemes	72
6.1.2	Security Ontology	73

6.1.3	Pattern Definitions	74
6.2	Patterns for Data Protection	82
6.2.1	Basic Implementation Schemes	82
6.2.2	Security Ontology	82
6.2.3	Pattern Definitions	84
7	A Pattern-driven Transformation Process	89
7.1	Security Intention Transformation	90
7.1.1	Execution Context Creation	90
7.1.2	Applying Security Patterns	91
7.1.3	Security Policy Model Generation	97
7.2	Formal Verification of the Transformation Process	100
8	Proof of Concept: The SOA Security LAB	105
8.1	SOA Security LAB Overview	105
8.2	SOA Security LAB Architecture	108
8.3	Security Pattern Engine Implementation	110
8.3.1	Security Pattern Engine Package Structure	110
8.3.2	Package 'dataTypes'	111
8.3.3	Package 'model'	112
8.3.4	Package 'dataAccess'	113
8.3.5	Package 'mapping'	113
8.3.6	Package 'dsl'	114
8.3.7	Package 'core'	116
8.4	SOA Security LAB Usage	117
9	Conclusion	125
A	Glossary	129
	Bibliography	137

Chapter 1

Introduction

In recent years, IT enterprise infrastructures evolved into distributed and loosely coupled system landscapes that expose the assets and resources in an organisation as business services. The paradigm of Service-oriented Architectures (SOA) facilitates the flexible provision and reuse of these services to enable a faster adoption to changing business requirements. OASIS has published a reference model [MLM⁺06] that defines SOA as '*a paradigm for organising and utilising distributed capabilities that may be under the control of different ownership domains.*' Services expose capabilities to address the needs of service consumers. The interactions between a service provider and a service consumer are introduced in this model as a core concept to use a capability by exchanging information.

The basic concepts of service orientation as described in the OASIS reference model [MLM⁺06] lead to key aspects that are widely used to describe service-oriented computing. Thomas Erl [Erl05] described these key aspects as *Loose coupling*, *Service Contract*, *Autonomy*, *Abstraction*, *Reusability*, *Composability*, *Statelessness*, and *Discoverability*. Services encapsulate and expose logic (autonomy) as described in the service contract (abstraction). The contract defines the structure of exchanged information and minimizes the dependency between client and service to facilitate a loose coupling. A client has to conform to the contract and is not statically bound to a binary interface. The independent nature of services, with respect to operating systems, programming languages and system architectures, facilitates the creation of complex service orchestrations. In the scope of organisational workflows, SOA provides a suitable foundation to execute business processes as an orchestration of multiple independent services.

Altogether, Service-oriented Architectures deliver a flexible infrastructure to allow independently developed software components to communicate in a seamless manner. However, this flexibility comes along with new security risks and threats. Messages exchanged in a decentralized environment across system borders are vulnerable to information tampering and disclosure. To address these risks, exchanged information must be protected in terms of confidentiality and integrity by applying encryption mechanisms and digital signatures. Furthermore, the authentication of participants in an interaction is required to restrict the service access to authorised users. While monolithic applications can manage their users in an isolated fashion, this approach is not applicable in the scope of service computing. The composition and dynamic selection of services require a seamless usage and integration of services - even across organisational borders. Identity management models for decentralised environments have been developed and provide the foundation to authenticate and identify users across administrative domains.

Security requirements concerning the identification and authentication of users and the confidentiality and integrity of exchanged information must be enforced by interacting participants (services and clients). The enforcement of these requirements transforms messages that are exchanged in the scope of these interactions. For example, exchanged messages might be encrypted, signed, or convey identity information due to security requirements stated by a service. Security policies enable services to express and group these security capabilities and requirements and represent an important concept to enable interoperability at runtime. Policies are provided with the interface description of a service and define a list of supported security options. Clients can retrieve this policy and select security mechanisms and protocols to invoke the service securely.

In the scope of the Web Service specifications, WS-SecurityPolicy can be used to specify requirements regarding the protection of exchanged messages (e.g. algorithms, key strength, and protected message parts) and the provision of identity information (e.g. authentication options and trusted parties). This policy language provides a complex structure and a variety of options to specify and restrict the usage of the Web Service security specifications. However, the broad range of options and security mechanisms complicates the creation of secure and consistent policies. Security policies are hard to understand and even harder to create, due to the expressiveness of this language. Since various security mechanisms and options can be required and combined in a policy, strong security expert knowledge is needed.

To simplify the creation of security configurations for Web Services, tool support is offered by all major Web Service platforms and development environments. These tools provide policy editors and preconfigured policies that can be used by selecting a profile or binding. However, strong security knowledge is still required to choose appropriate bindings and to provide additional security related configurations. In addition, these tools do not take the overall system architecture into account. The designer of a policy has to consider available security services, trust relationships, communication channels and the security capabilities of the participants involved in the interactions with services.

Model-driven security approaches promise to assist the policy generation process by providing a conceptual layer to model, verify, and transform security requirements in a simplified way. The enhancement of system design models with security requirements provides a comprehensive view on security aspects to facilitate the generation of consistent security policies that comply with these requirements. Various approaches have been specified to enhance different types of modelling languages such as UML [Jue02, BDL06] or BMNP [RFMP07, WS07]. Some of these approaches provide an enhancement of system design models to express and verify security requirements, while other approaches enable a transformation of authorisation annotations to generate access control policies. However, these approaches are not applicable in the scope of service-based systems that require the modelling and transformation of requirements related to secure messaging.

To overcome these limitations, this thesis presents a methodology to enable the modelling of security intentions in system design models and a transformation process to generate Web Service security policies. Our modelling approach provides a simple and high level notion of security requirements that can be used by enterprise architects who are not required to have a strong security background. Furthermore, our approach enables an automated transformation of simple security intentions to complex, customised security configurations. To perform this transformation, a catalogue of security patterns for service-based systems is introduced in this work. The specification of a security design language to enhance system models, a formalised security pattern system, and a pattern-based transformation process are the main contributions of this thesis.

1.1 State of the Art in Model-driven Security

The domain of model-driven security is an emerging research area. The need to describe security policies referring to an application scenario on an abstract level is discussed in [TIN04]. A tool is presented that provides business-oriented views to configure secure Web Services on the basis of interaction models and related threats.

Jürjens introduces the UMLsec extension in [Jue02] to express and verify security aspects within UML diagrams. UML profiles, tags, and stereotypes are used to express requirements such as confidentiality, access control and non-repudiation. This approach provides formal semantics to enable a verification of security requirements. For example, security protocols can be modelled and verified. However, to perform such a formal verification, all security-related aspects such as cryptographic data must be specified in the system model. As a consequence, UMLsec models have a high degree of complexity and tend to be difficult to understand without a strong security background. Although this approach could be adapted to model and verify Web Service security protocols, it does not provide a simple, high-level notion for security intentions.

Breu and Haffner propose a methodology for security engineering in Service-oriented Architectures [HB08] that is based on a model-driven approach. Security requirements are modelled in a domain-specific language and transformed to a domain-independent language that is used to generate security policies. In particular, they outline a mapping to authorisation constraints. Although messaging-related security goals can be expressed, a mapping to a security policy language is not described. In addition, specific Web Service concepts such as claim-based identities are not considered that would be required to configure identity management systems.

In [BDL06], Basin and Lodderstedt introduce SecureUML that provides a security design language to describe role-based access control and authorisation constraints. In addition, they describe a general schema to integrate this language in different types of system design languages. Using this schema, they specify the modelling language ComponentUML to illustrate the integration of SecureUML. The transformation into executable EJB and .NET systems with configured access control infrastructures is described as well.

Previous work done by Rodriguez et al. [RFMP06], [RFMP07] discusses an approach to express security requirements in the context of business processes by defining security requirement stereotypes that link to activity elements of a business process. In addition, they propose graphical annotations to visually enrich the process model with related security requirements. Although they support several security requirements, they neither describe a schema to integrate these requirements in other modelling languages nor provide a model-driven transformation. A model-driven scenario based on their annotations is considered as future work.

Wolter [WS07] fosters a model-driven approach to enable a generation of XACML access control policies based on enhanced business process models. Therefore, he describes an extension for BPMN to visualise authorisation requirements and related constraints. Similar to Wolter, Klarl [KMW⁺09] introduced a model-driven approach based on the enhancement of process models as well.

Jensen and Feja describe a model-driven generation of Web Service security policies based on the annotation of security requirements in business process models [JF09]. In particular, their approach is focused on the generation of WS-SecurityPolicy documents to ensure a secure messaging in terms of confidentiality and integrity. Security requirements and aspects related to identity management concepts (e.g. identity provisioning requirements, trust relationships, or identity providers) are not considered.

Using security patterns, Delessy describe a pattern-driven process for secure SOAs [Del08]. This approach intends to use patterns for a semi-automated translation to support system architects. An automated transformation to security policies is not provided.

In summary, most of the related approaches have been focusing on the modelling of authorisation requirements or the specification of security requirements in the scope of business processes. A generic framework that enables an integration schema to express security requirements in any system design modelling language and that provides the generation of messaging-related security policies for service-based systems has not been provided yet.

1.2 Contribution: A Framework for Model-driven Security

The contribution of this thesis is a model-driven approach that simplifies the design of security policies by enabling the visual modelling of high-level security intentions in system design models. Security policies are generated on the basis of these intentions using the transformation process provided by our approach. As illustrated in Figure 1.1, our model-driven approach consists of three layers. Security requirements, expressed at the modelling layer are translated to a domain-independent SOA security model. This model constitutes the foundation to generate WS-Security policies. The modelling of security requirements, the structure of the domain-independent model and the transformation process across these layers are outlined in this section.

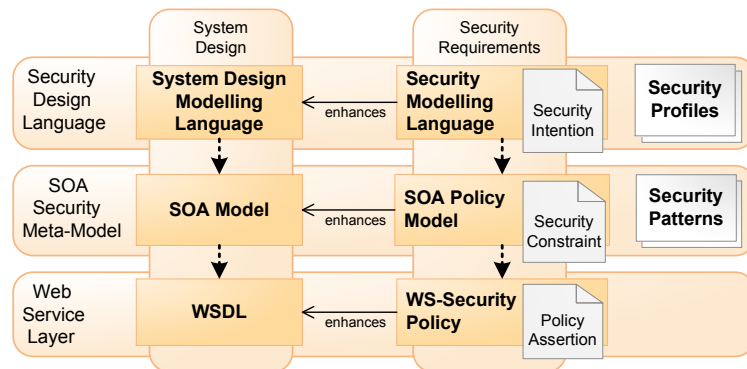


Figure 1.1: Model-driven Security in SOA

1.2.1 Modelling Security Requirements

System design models such as FMC block diagrams [TAW05] or UML models [Gro09b] form the foundation to enable system designers to state security aspects in an easy accessible way. We use the integration schema provided by SecureUML [BDL06] to enable the enhancement of these system design models with our security design language SecureSOA. SecureSOA specifies security-related modelling elements by providing the abstract syntax, notion, and formal semantics of these elements. In particular, this language specifies security intentions, security annotations, and trust relationships. Security intentions are used to represent basic requirements (e.g. authentication or confidentiality), while security annotations represent capabilities (such as a user directory facilitating the management of users). To enable the annotation of security re-

quirements in system models, we foster the integration of SecureSOA in Fundamental Modelling Concept (FMC) Block Diagrams.

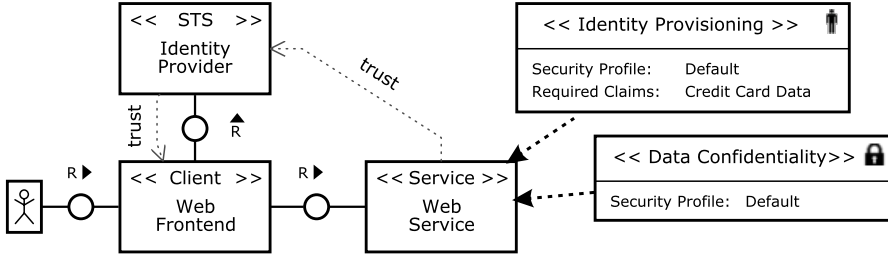


Figure 1.2: Modelling Security Intentions

For instance, an FMC block diagrams enhanced with SecureSOA is shown in Figure 1.2. A user leverages a web frontend to access a service. Moreover, a Security Token Service (STS) is deployed that performs the user management. The trust relationship to the client indicates that users of the web frontend are registered at the identity provider, while the trust relationship from the service to the STS indicates that this service relies on statements made by the identity provider. The identity provider authenticates users – e.g. by validating user name and password entered at the web front end – and issues security tokens. These tokens can be sent along with the request message to access the service. In addition to the system structure, Figure 1.2 depicts two security intentions representing security goals that must be enforced by the security infrastructure.

Profile	Security Mechanisms
high	X509-Token
default	UserName-Token, X509-Token

Table 1.1: Examples of Security Profiles

To enable the transformation of security intentions to a technical layer, a set of security mechanisms and protocols must be listed for each intention that can be used to enforce it. Instead of specifying the algorithms, key strength and other technical details at the modelling layer, a modelled security intention refers to a security profile that provides this information. Profiles are used to abstract from technical details that should be hidden from the modeller and depend on the integration platform that is used to provide and secure the services. Basically, the profiles list security mechanisms that are supported by the platform to enforce security intentions. As an example, Table 1.1 lists two profiles that provide security mechanisms to implement the authentication of users.

1.2.2 A Domain-independent SOA Security Model

Security policy languages provide complex and expressive grammars to state security requirements at a technical layer. For example, WS-SecurityPolicy specifies an XML syntax to represent messaging-related requirements that are structured in policy assertions and policy alternatives. However, other security policy languages may define a grammar that organises requirements in another way.

To facilitate the generation of security configurations in different security policy languages, we have specified a domain-independent SOA security model that provides an abstraction layer to technical security policies for service-based systems. This abstraction layer provides a consistent interface to simplify the handling and creation of security policies and supports the expression of security requirements concerning communication related security goals.

Our SOA security policy model provides a separation of concerns. This model facilitates a transformation of high-level security intentions to technical security requirements independent of any security policy language. Security policy languages supported by specific platforms and frameworks - for instance WS-SecurityPolicy or Apache Rampart configuration files - can be integrated easily by providing mappings from our domain-independent model to these languages.

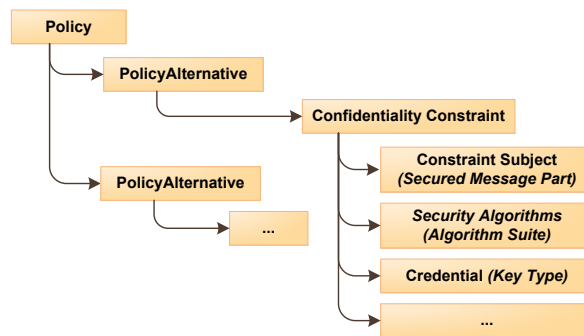


Figure 1.3: Policy Model Example

A policy in our domain-independent policy meta-model consists of several *Policy Alternatives* that contain a list of *Security Constraints*. In general, a *Security Constraint* describes a set of requirements to fulfil a *Security Goal*. An example of a security policy model instance is shown in Figure 1.3. A *Confidentiality Constraint* is specified stating that a specific piece of information (message parts) must be protected using specific algorithms and key types.

1.2.3 A Model-driven Transformation

The generation of enforceable security policies based on security intentions is a challenging task that has not been fully addressed in related work. A simple mapping is not sufficient to implement the transformation process, since context information must be considered that is specified at the modelling layer (e.g. provided security services, trust relationships, etc.).

Using our domain-independent policy model, the transformation of security intentions to security configurations works as follows: First of all, security constraints are generated based on the modelled security intentions and combined in policy alternatives. In a second step, this policy model instance is transformed to a security policy document that is stated in a specific security policy language.

1.2.3.1 Pattern-based Transformation of Security Intentions

The creation of security policy model instances on the basis of a SecureSOA model is the first step of our model-driven transformation. However, a simple mapping is not sufficient to perform

the transformation from abstract security intentions to an instance of our security model that describes complex technical requirements. Security expert knowledge is required to determine an appropriate strategy to secure services and resources, since multiple solutions might exist to satisfy a security goal. For example, confidentiality can be implemented by securing a channel using SSL or by securing parts of transferred messages using WS-Security. To describe these strategies and their preconditions in a standardised way, we foster the usage of security configuration patterns. Security patterns have been introduced by Yoder and Barcalow in 1997 [YB97] and are based on the idea of design patterns as described by Christopher Alexander in 1977: *'A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that pattern'* [AIS⁺77]. This thesis provides a formalised system of security configuration patterns to enable an automated application of security patterns in the transformation process. Each security configuration pattern creates and configures a set of policy constraints (*solution*) for a security intention (*problem*). In addition, conditions (*forces*) are defined for each pattern that determine its applicability. To state the definition of the forces and the solution of security configuration patterns, we have specified a domain-specific language (DSL). The syntax and the formal semantics of our security configuration pattern DSL are presented in this thesis and enable a formal verification of the transformation process.

Overall, our security configuration patterns enable an automated creation of security policy constraints based on simple security intentions. In addition, the security pattern system is supported by a security ontology that provides security-specific knowledge as shown in Figure 1.1. It is used by our transformation process and enables a mapping of security concepts referenced in a security pattern's solution to security mechanisms listed in the profiles.

1.2.3.2 Generation of System Configurations

The transformation of security policy model instances to security policy documents is the final step in our model-driven approach. A security policy document is expressed in a security policy language that is supported by the integration platform.

To enable a generation of security policies for Web Service-based systems, we have specified a mapping to WS-Policy and WS-SecurityPolicy. WS-Policy [VOH⁺07] defines a grammar to group and express requirements and consists of a set of policy alternatives representing a disjunction of requirement sets. Each alternative groups a set of policy assertions or additional alternatives and represents a conjunctive combination. Policy assertions represent concrete requirements that are defined by additional specifications. The WS-SecurityPolicy specification has been developed to express requirements related to Web Service security and provides assertions to describe requirements regarding the provision of security tokens and the use of encryption/signature algorithms and options [NGG⁺07a].

Since our policy model supports the concept of policy alternatives, a mapping to WS-Policy is straightforward. However, the generation of WS-SecurityPolicy assertions is more complicated due to the complexity of this specification. The transformation of our security policy model to WS-SecurityPolicy assertions must be performed in multiple phases that generate the different types of WS-SecurityPolicy assertions (binding assertions, protection assertions, and the supporting token assertions) in each step. This transformation process, which is not discussed in the scope of this thesis, has been developed and implemented by Warschofsky et al. in [WMM10, War10].

1.3 Outline

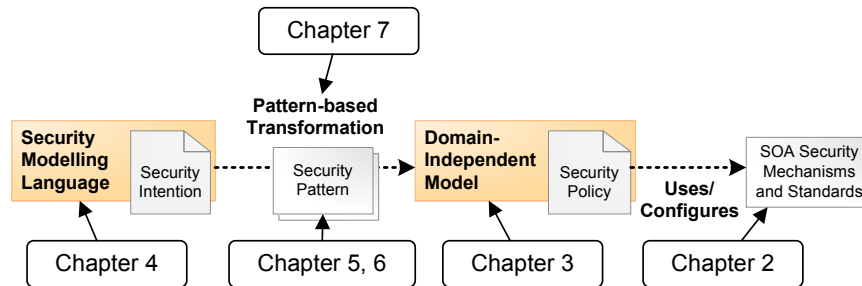


Figure 1.4: The Structure of the Thesis

The structure of this thesis is illustrated in Figure 1.4. Chapter 2 provides an overview about security in service-based systems and introduces the concepts secure messaging, identity management and policy management. In the following chapter, our meta-model for SOA security is introduced that provides an SOA interaction model to describe the basic entities and relations in an SOA. Moreover, a meta-model is presented specifying our policy model. These meta-models define our domain-independent model used in our model-driven approach and constitutes the foundation to specify our security modelling language SecureSOA. This language and its integration into FMC block diagrams is described in chapter 4. In the next chapter, the structure and the semantics of security configuration patterns are introduced that are used to generate and modify security constraints. Security configuration patterns are stated in a domain-specific language that will be introduced in this chapter as well. This language is used in chapter 6 to define our pattern system for identity management and data protection. Our pattern system provides the foundation to perform the transformation process of visual models to security policies as introduced in chapter 7. In this chapter, the basic concepts of our pattern engine are introduced that applies security patterns to security intentions. The correctness of this transformation step is proven on the basis of the patterns' formal semantics. Chapter 8 describes our SOA Security LAB that uses our model-driven approach to provide a test platform for service security, while chapter 9 concludes this thesis.

Chapter 2

SOA Security Concepts and Standards

This chapter provides an overview of key concepts and standards facilitating the design and implementation of secure service-based systems. As a starting point, the Web Service standards are introduced that define a flexible messaging framework for SOA. Next, secure messaging and identity management approaches are described that provide the foundation to realise authentication, confidentiality, and integrity in decentralised systems. The technical specifications used to implement these concepts are presented as well. Finally, the concepts and standards related to security policy management are introduced that facilitate the statement of security requirements related to secure messaging and identity management.

2.1 Web Service Technology

The concept of service orientation encourages the vision of self-descriptive services that enable a loose coupling of enterprise systems to facilitate a faster adaption to changing business demands. To implement this approach, the Web Service standards provide a framework for interoperable machine-to-machine interactions. As defined by D. Foggon et al. in [FMUW03], *a Web Service is a remotely accessible application component that listens for certain text-based requests, usually made over HTTP, and reacts to them.* These application components are also referred to as Web Methods. Although Web Services are protocol-independent, most Web Service implementations expect their Web Methods to be invoked using HTTP-requests conveying SOAP messages based on XML. Due to the usage of these standards, Web Services are independent of operating systems and programming languages.

A broad range of Web Service specifications has been defined by the Organization for the Advancement of Structured Information Standards (OASIS). These specifications provide a framework that enables a secure and reliable messaging in decentralised systems. Figure 2.1 illustrates and groups selected Web Service specifications. Based on *XML*, *SOAP* and *WS-Addressing* provide a message structure to implement a message exchange. Meta information concerning a Web Service is provided by the standards *WSDL* and *WS-Policy*. *WSDL* defines an interface description language, while *WS-Policy* provides a structure to express and group capabilities and

requirements as assertions in policies. These policies can enhance WSDL documents as described by *WS-PolicyAttachment*. *WS-MetadataExchange* specifies a simple message structure to query meta information from a Web Service. Furthermore, multiple specifications enable the implementation of security mechanisms to secure the message exchange. *WS-Security* enhances SOAP to enable the usage of XML encryption and XML signature. In addition, the exchange of credentials with SOAP is described to facilitate the authentication of users. These credentials are represented as an XML structure and are denoted as *Security Tokens*. A Security Token Service (STS) provides a *WS-Trust* interface and is used to perform the authentication of users and to issue security tokens. To optimise the exchange of multiple secure messages, *WS-SecureConversation* enables the establishment of a secure channel. *WS-SecurityPolicy* defines security assertions that can be used with *WS-Policy*. These assertions define requirements concerning the usage of the specifications *WS-Security*, *WS-Trust*, and *WS-SecureConversation*.

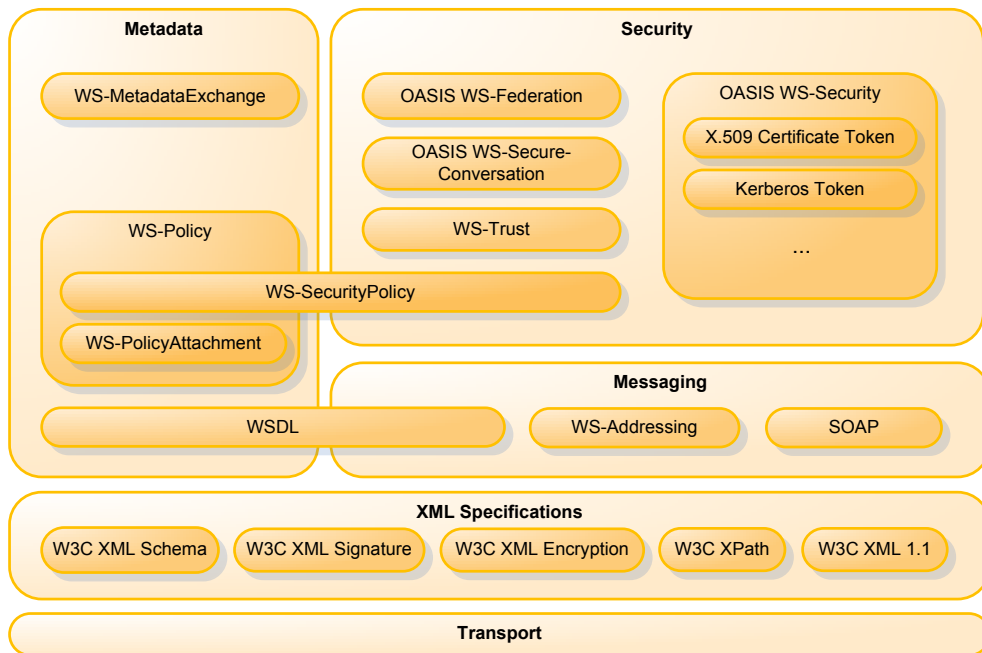


Figure 2.1: Web Service Specifications (Extract)

2.1.1 Messaging with SOAP

SOAP is an XML-based message format to exchange information using HTTP or other protocols. Initially, it has been designed to implement an XML-based remote procedure call. With the advent of additional Web Service specifications, SOAP evolved into a generic messaging framework. While SOAP was an acronym for Simple Object Access Protocol, this denotation is not used anymore, since it does not correspond to the current purpose of SOAP.

The message structure defined by SOAP consists of an envelope as shown in Figure 2.2. Within this envelope, two additional sections are provided to describe processing information and message content: An optional header element and a mandatory body element.

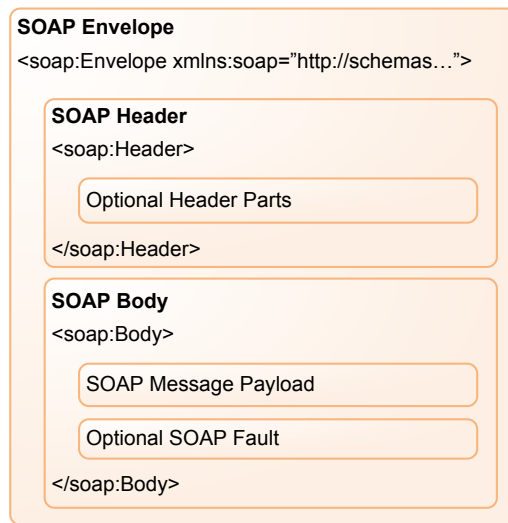


Figure 2.2: SOAP Message Structure

A SOAP header provides the possibility to extend the message with meta information. For instance, a SOAP header can be used to store security information such as security keys and digital signatures, or routing information to pass a message through multiple intermediaries. These headers are defined by additional Web Service specifications. The attribute *mustUnderstand* can be used with each XML element in the SOAP header section to indicate that the receiver of a message must be able to process this header. The body of a SOAP message encapsulates the payload of the message such as a method name and invocation parameters. In addition, fault information can be conveyed in case of errors.

To route messages over intermediaries, addressing information must be included in exchanged messages. Therefore, WS-Addressing has been specified to define additional message headers that identify endpoints and messages. In addition, these headers can be used to implement an asynchronous service invocation.

2.1.2 WSDL - Describing Web Services

The Web Service Definition Language (WSDL) specification provides an XML-based language to describe service interfaces in a platform and protocol independent way. WSDL represents services as a set of endpoints, which offer multiple operations. For each operation, incoming and outgoing messages, error messages, and a message exchange pattern are specified. An operation can be invoked by a client using the address provided by the WSDL description. A WSDL document is structured hierarchically, whereas the `<description>` element is used to represent the root element. To facilitate the reuse of service descriptions, WSDL documents are separated in an abstract and a concrete part.

The abstract service definition describes the interface of a service in terms of input and output messages, while the concrete service definition binds this interface to protocols and an address. As illustrated in Figure 2.3, the Web Service's abstract definition of a WSDL 1.2 document is defined by the *portType* element that contains several *operation* elements. These operations reference

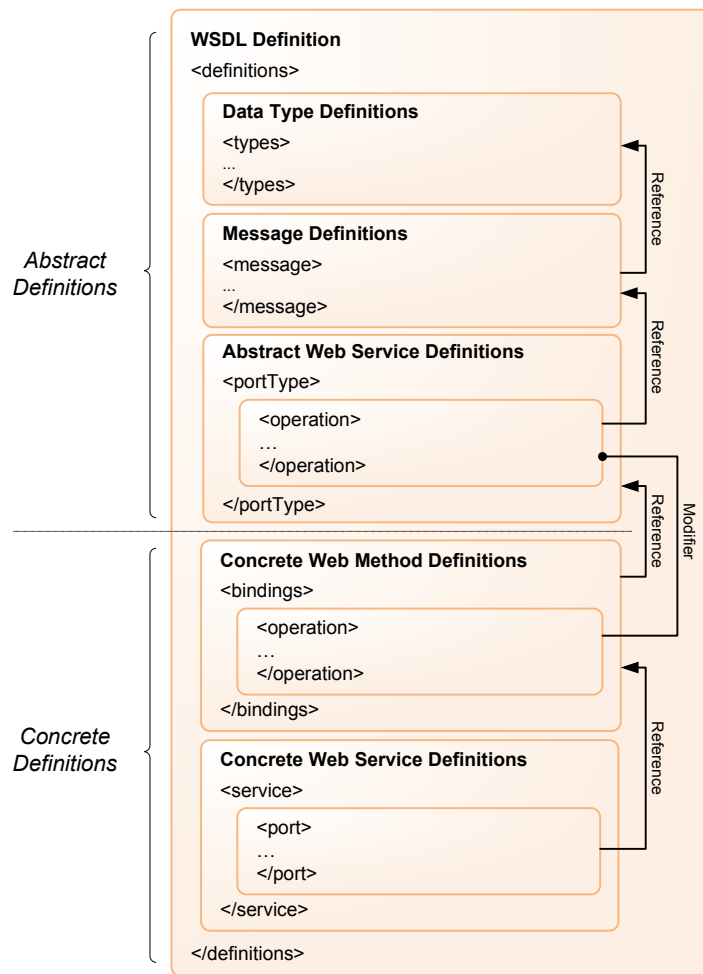


Figure 2.3: WSDL 1.2 Structure

messages used for input, output, or fault transmission. Each message can be defined by either an XML Schema Definition (XSD) or a *type* definition. The concrete definition appends protocol-specific information to the *portType* by defining a binding. For each operation in *portType*, there is also a protocol-specific *operation* element in *binding*. A series of *binding* elements are composed to create a service. Altogether, a WSDL 1.2 document is based on five basic elements:

- **types** - specifies basic data types on the basis of XSD definitions.
- **message** - defines data that is exchanged between a consumer and a provider during service invocations. Each message consists of multiple *part* elements that represent the conveyed data. Therefore, each part is associated with an XSD type or a *type* element.
- **portType** - represents an abstract definition of a service. There are multiple operations defined in each *portType* specifying the exposed Web Methods. Within these operations, messages are referenced to represent input, output or fault information.

- **binding** - each binding is mapped to a *portType* element to specify the required transport protocol and contains multiple operations to specify invocation options for each Web Method such as the required encoding.
- **service** - represents the concrete definition of a Web Service. A *service* element is composed of several ports. Each *port* references a *binding* and defines an address.

WSDL 2.0 introduced some minor changes to this structure. *PortType* has been renamed to *Interface* and the data type definitions are referenced by *input*, *output* and *fault* elements without using the *message* element.

2.2 Secure Messaging

The enforcement of the security goals confidentiality, integrity, and availability represents the core principle of information security. In the scope of secure messaging, confidentiality and integrity must be ensured to protect conveyed information. Confidentiality prevents the disclosure of information to unauthorized individuals, whereas integrity facilitates the detection of modification performed by unauthorized parties. However, message integrity requires the authentication of message senders to verify the identity of authorized parties (cf. [Jue03], p. 44).

Cryptographic methods and digital certificates provide the foundation to implement these security goals. The application of these methods can be performed at the transport layer by establishing a secure channel that enables the authentication of communication partners and the enforcement of confidentiality and integrity of transferred information. However, this approach is not appropriate in a system that passes messages over multiple intermediaries. Although secure channels can be established between actors in such a system, information stored and processed by the intermediaries would not be protected. Moreover, the receiver of a message would not be able to verify the identity of the message sender. Therefore, messages passed across multiple intermediaries must be protected using cryptographic mechanisms to enable a protection of information in rest, transit and processing.

In the scope of the XML-based Web Service specifications, XML Signature and XML Encryption provide the fundamental building blocks to enable the application of encryption and signature mechanisms to SOAP messages.

2.2.1 WS-Security

WS-Security has been proposed as a standard by Microsoft and IBM [IBM02] in 2002 and was established as an OASIS standard in 2004. This standard defines enhancements to SOAP to enable a secure messaging in terms of integrity, confidentiality, and authentication. WS-Security provides a framework to integrate digital signature and encryption methods. In addition, it enables the exchange of key and authentication information. WS-Security is based on XML-Signature and XML-Encryption to enable an end-to-end protection of messages.

WS-Security provides a security header for SOAP that serves as a standardised place to store security information. As illustrated in Figure 2.4, this additional header can be composed of three major elements: security tokens, a digital signature, and encryption meta data.

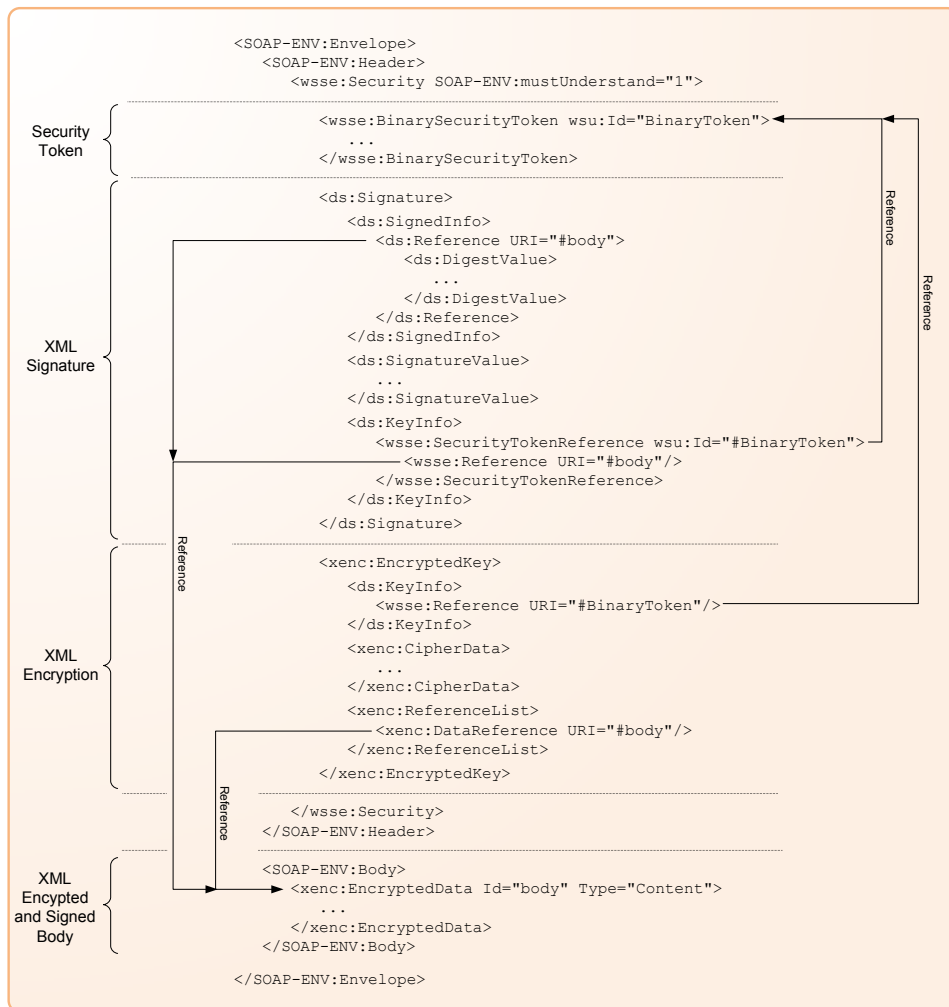


Figure 2.4: WS-Security Structure

Security tokens represent credentials that provide identity information for authentication and authorisation. In addition, a token might provide information concerning the keys that have been used to apply digital signature and encryption mechanisms to a message. WS-Security defines two types of security tokens: an unsigned token (`UsernameToken`), which is used to provide user name and password information to a service, and a signed token (`BinarySecurityToken`) that has been endorsed by a third party. Figure 2.4 illustrates the usage of a binary security token that represents an X.509 certificate. The attribute *ValueType* is used to specify the content type (X.509 certificate or a Kerberos), while the attribute *EncodingType* defines the representation of the binary data.

Moreover, a WS-Security header can contain the tag *Signature* to encapsulate XML signature information. XML Signature enables the application of digital signature technology to XML and is itself described with XML. Figure 2.4 illustrates the usage of the following key elements:

- *SignedInfo* – specifies the signed message parts and the algorithms that have been used. *SignedInfo* encapsulates the *Reference* element that provides a set of references to signed message elements as well as the digests of these elements. Please note that message signatures based on the usage of simple id references are vulnerable to signature wrapping attacks. This issue and related countermeasures are discussed in [MA05, GJLS09].
- *SignatureValue* – contains a Base-64 encoded value that represents the digital signature. This value is the result of encrypting a digest of the *SignedInfo* element.
- *KeyInfo* – references a key to enable the verification of the signature. In case of WS-Security, *KeyInfo* points to the used security token.

Finally, WS-Security facilitates the encryption of message parts by leveraging XML encryption. Encrypted message parts are replaced by *EncryptedData* elements that wrap encrypted data and provide meta data information. Figure 2.4 illustrates the encryption of the message body that is based on a combined usage of symmetric and asymmetric encryption. Since an asymmetric encryption on the basis of a public key infrastructure is a time consuming task compared to symmetric encryption, a symmetric session key is generated for each message exchange. This session key is stored in the element *CipherData* that is wrapped in the *EncryptedKey* element of the WS-Security message header. In addition, the *EncryptedKey* element provides the element *ReferenceList* that points to all *EncryptedData* items that have been encrypted with the shared key. As illustrated in Figure 2.4, the element *KeyInfo* provides a reference to the public key that has been used to encrypt the shared key.

2.3 Identity Management

The identification and authentication of users are important security requirements to ensure a trustworthy communication in decentralised systems and provide the foundation to restrict access to services. The enforcement of these security goals operate on a representation of an user's identity in the digital world - a user's digital identity. A digital identity consist of a set of attributes and is managed in an account.

Identity Management describes the process of establishing, representing, maintaining and provisioning a person's identity as digital identities in IT systems as shown in Figure 2.5.

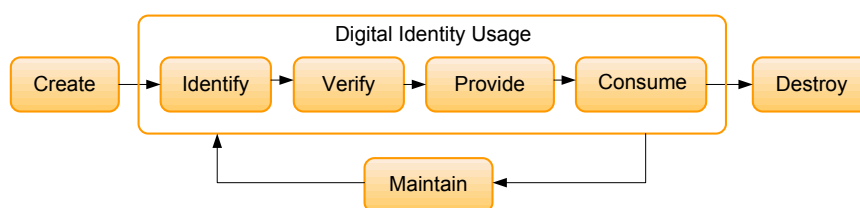


Figure 2.5: Life Cycle of a Digital Identity

The first step in this process represents the registration of users. In order to create a user's digital identity, identity information is stored in an account that is created for the user. The usage of a digital identity at a service provider is based on four steps. A user is identified and authenticated to verify that a specific digital identity belongs to this user. Then, required identity information

is provided to a service provider to enable the consumption of this information. At the end of the life cycle, a digital identity is destroyed by deleting the user's account.

The digital identity life cycle identifies basic steps that are performed by identity management systems. The concepts and architecture behind these systems are classified and described by identity management models. We distinguish four models, while each model implements a specific identity management approach. As described by Thomas et al. in [TM11], an identity management model can be based on an domain-based approach or on an open environment approach.

The *domain-based approach* represents traditional models that bind a digital identity to a specific security domain (e.g. a company). The consumption of identity information by services is limited to this domain. Figure 2.6 illustrates identity management models implementing the domain-based approach.

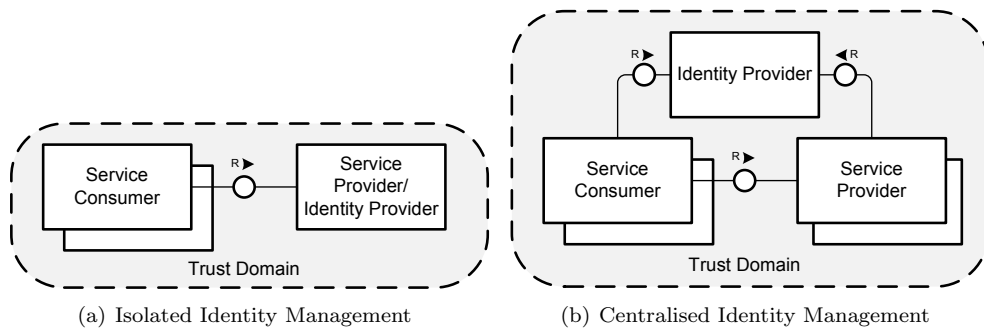


Figure 2.6: Domain-based Identity Management Models

The *isolated identity management* model is shown in Figure 2.6(a) and implements a service specific management of digital identities. A service is attached to a user directory that provides the identity information for this service and enables the authentication of users. Each service that is based on this model has full control over the users. However, users have to be registered at each service independently. Since the identification and representation of users is service-specific, the usage of this model prevents the orchestration and composition of independent services. To enable a single-sign-on across multiple services in a domain, the *centralised identity management* model can be used as illustrated in Figure 2.6(b). The services are connected to a single identity provider to organise the identity management in a centralised way. The identity provider is responsible to manage the digital identities of users and to perform their authentication. Authentication decisions can be brokered to services that rely on this information. In addition, identity information required by the services can be provided as well. The usage of an identity provider requires trust relationships between the relying services and the centralised identity provider. Since the identity provider is used in a single trust domain, these relationships are established by default.

Although domain-based identity management approaches enable service providers to control the identity information of their users, these approaches prevent the usage of identities across domain boundaries. Since users have to be registered in each domain independently, an increasing number of digital identities and accounts have to be managed. Consider the usage of web applications in the internet: users have to deal with a multitude of user name and password combinations and tend to select the same password for each account. This example illustrates that the application of domain-based identity management approaches in open environments results in an increasing

number of security risks. Furthermore, identity information (e.g. the user's address) has to be updated in multiple accounts to keep all identity information consistent.

Identity management models based on the *open environment approach* address these issues by enabling the usage of digital identities across trust domains. These models are based on the usage of multiple identity providers that share and broker identity information. Since identity providers can be implemented on the basis of different technologies and protocols, an abstraction layer is required to enable the interoperable exchange of identity information.

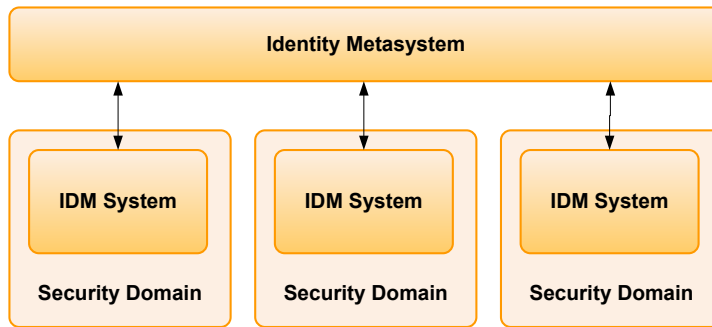


Figure 2.7: Identity Metasystem

This abstraction layer is defined by the *identity metasystem* [CJ06] as illustrated in Figure 2.7. Standards such as WS-Trust and SAML provide interoperable interfaces and token formats. The identity metasystem facilitates the integration of identity management systems to avoid the replacement of existing solutions.

The identity management models based on the open environment approach are shown in Figure 2.8. Both models are based on the identity metasystem to integrate multiple identity providers. Service providers rely on identity information and authentication decisions that are asserted by trustworthy identity providers. The brokering of identity information enables a single-sign-on across organisational borders. Both models differ from each other in the establishment of the underlying trust relations.

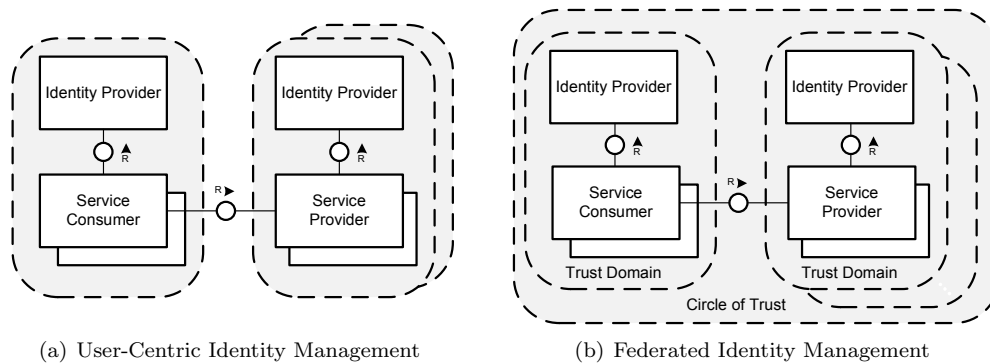


Figure 2.8: Open Environment Identity Management Models

The *Federated Identity Management* model illustrated in Figure 2.8(b) forms a federation of multiple organisations and is based on a circle of trust. Underlying contracts define the obligations

of each participating organisation concerning the identity management and the authentication of users. Service providers rely on assertions issued by any identity provider in the federation.

The *User-centric Identity Management* model shown in Figure 2.8(a) is not based on contracts and predefined trust relations. A user can select any identity provider that is able to provide authentication assertions or required identity information. The security tokens issued by these identity providers can be used to access a service. The service provider has to decide, whether a token can be accepted from this source. For instance, Information Card [Cha06], which is implemented as CardSpace in Microsoft Windows, is based on this model.

2.3.1 SAML

SAML (Security Assertion Markup Language) is an XML-based framework standardised by the OASIS Security Services Technical Committee that enables the description, issuing, and exchange of identity information.

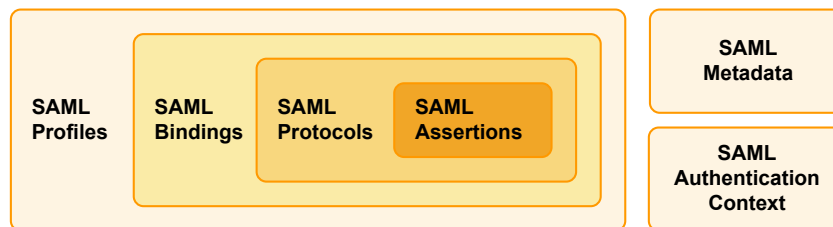


Figure 2.9: SAML Components

As shown in Figure 2.9, SAML provides the following components. *SAML Assertions* are used to make statements about an identity, while *SAML Protocols* provide protocols to request and exchange these statements. To enable the exchange of assertions using standard messaging or transport protocols, a mapping is provided by *SAML Bindings*. Finally, *SAML Profiles* guide the application of these protocols in the scope of specific use cases.

SAML assertions (denoted as SAML tokens in the scope of the Web Service specifications) contain one or more statements made by a SAML authority. This authority vouches for the correctness of the information in the statements. Each assertion contains issuer information and identifies the subject the statements are made about. A digital signature of the issuer is included in the assertions to enable the verifiability of the statements' integrity. In particular, SAML specifies three types of statements: authentication statements, authorisation statements, and attribute statements.

Authentication statements assert that a user (subject) has been authenticated by the issuer of the assertion using a specific authentication method at a particular point in time. An authorisation assertion is used to assert access control decisions, while attribute statements convey identity attributes.

Figure 2.10 illustrated a simple SAML assertion. This assertion contains an authentication statement and specifies that the subject Michael.Menzel has been authenticated by the identity provider of the Hasso Plattner Institute. In addition, the authentication context states that a username/password credential has been used to perform the authentication that was submitted over a secure channel.



Figure 2.10: SAML Assertion Example

2.3.2 WS-Trust

WS-Trust defines a Web Service interface to issue, renew, verify, and cancel security tokens. A service that provides this interface is denoted as Security Token Service (STS). Security tokens can be requested from an STS using a request for security token (RST) message structure. These tokens are returned by a request for security token response (RSTR). The RST request message can specify requirements concerning the desired token. For example, the token type (e.g. SAML 2.0), the conveyed identity information (claims), and encryption/signature options can be specified. In order to issue a token, an STS must authenticate the message sender on the basis of a security token that has been sent with the message request.

WS-Trust provides an important interface to implement an identity provider that supports the open environment identity management models. The conversion of security tokens is the primary task of a security token service implementing a federated or decentralised identity management approach. Either security tokens that enable a user authentication (e.g. username/password token) are converted to SAML tokens or SAML tokens issued by other security token services are converted to new SAML tokens.

2.4 Policy Management

While interface definition languages (e.g. WSDL) are used to describe the exposed functionality, security policy languages enable the enhancement of these interface descriptions to describe requirements and capabilities a service consumer must comply to. In particular, security requirements can be expressed as policies to configure secure interactions between service consumers and service providers. These requirements refer to the protection of exchanged data and the provisioning of identity information as described in the previous sections.

Security Policy Management provides a process to ensure that all participants adhere to common guidelines and regulations. The security policy management process contains several steps to map these regulations to the technical layer as illustrated in Figure 2.11.



Figure 2.11: Policy Management Process

The definition and creation of security policies is the first step in this process. Since strong security knowledge is required to transform common regulations to technical policies, the policy creation step represents the most complex task. A model-driven approach to facilitate this step is the contribution of this thesis. In the second step, security policies are distributed and deployed at the services. These policies configure the security modules of the services and facilitate the negotiation of security requirements between service consumers and service providers. The services have to enforce the security configurations stated by their policies by applying the requirements to incoming service requests and outgoing service responses. Finally, the policy monitoring step verifies that policy requirements have been applied correctly.

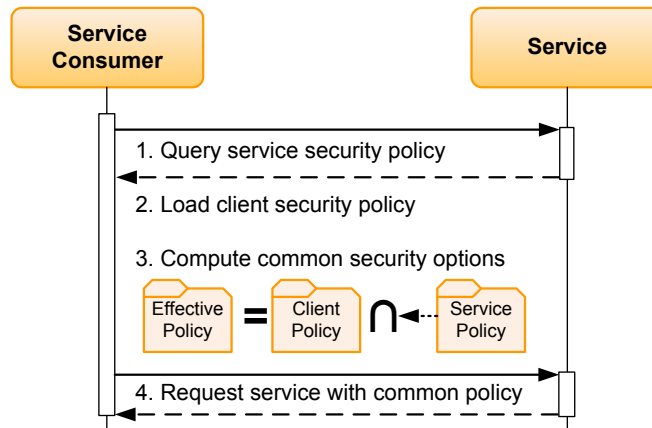


Figure 2.12: Negotiation of Security Requirements

Figure 2.12 illustrates the policy negotiation step that represents an important concept in the scope of the Web Service specifications. Web Service security policies organise security requirements in policy alternatives. Each alternative represents a set of valid security configurations to interact with a service in a secure manner. Clients can retrieve a service’s policy and compare the offered security alternatives with their own security configurations. Finally, a policy alternative that is supported by the client can be selected to invoke the service.

The provision of multiple alternatives facilitates interoperability at run-time by enabling the dynamic configuration of actors in a service-based system. An example is illustrated in Figure 2.13. The security policy of the service requires that AES 256 or AES 192 has to be used for encryption, while the client policy states that AES 168 and AES 192 are supported. The client retrieves the service policy and calculates the effective policy that requires AES 192. Finally, this policy is used by the client to secure the service invocation with AES 192. In the scope of the Web Service specifications, WS-Policy and WS-SecurityPolicy provide a grammar to group and express security requirements and capabilities as policy alternatives.

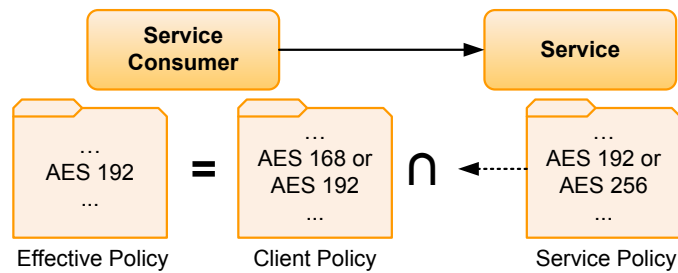


Figure 2.13: Policy Negotiation Example

2.4.1 WS-Policy

The WS-Policy W3C standard [VOH⁺07] describes an extensible and flexible XML-based grammar to express general characteristics, capabilities and requirements of actors in an XML Web Service-based system as policies. A WS-Policy document provides a series of policy alternatives, while each policy alternative describes a collection of policy assertions specifying requirements that must be fulfilled by service consumers. However, the definition of specific policy assertions representing concrete requirements (e.g. required encryption method) is not in the scope of WS-Policy, since WS-Policy just provides a general framework to structure any type of requirements in a consistent way. Additional specifications such as WS-PolicyAssertions provide policy assertions for a specific application domain. Security specific assertions are defined by the WS-SecurityPolicy standard.

The root element of a WS-Policy document is represented by the `<wsp:Policy>` tag that can contain the assertions `<wsp:All>`, `<wsp:ExactlyOne>`, and `<wsp:OneOrMore>` provided by WS-Policy. These assertions require that all, exactly one, or at least one assertion contained in this element must be fulfilled by service consumers.

These policy assertions provided by WS-Policy can be nested to group policy assertions stating specific requirements. Alternative sets of requirements can be expressed by using the assertions `<wsp:ExactlyOne>` and `<wsp:OneOrMore>`. Since any disjunctive and conjunctive combination of the assertion listed above can be used to structure requirements, WS-Policy provides an algorithm to convert a WS-Policy expression in a disjunctive normal form.

In addition, an algorithm is described by WS-Policy to compare policy documents. Clients can implement this algorithm to select policy assertions for service invocation by intersecting their policies with a policy retrieved from a service. Next to WS-Policy, the specifications *WS-MetadataExchange* and *WS-PolicyAttachment* are used to facilitate policy negotiation in Web Service-based systems. *WS-MetadataExchange* provides a protocol to retrieve metadata from Web Service endpoints and enables clients to retrieve a WS-Policy document from a service prior to the invocation of this service. *WS-PolicyAttachment* facilitates the association of WS-Policy documents with subjects of Web Services. These policy subjects (e.g. endpoints, operations, or messages) identify specific Web Service parts that can be associated with policy requirements.

2.4.2 WS-SecurityPolicy

As mentioned above, additional specifications are required to define policy assertions for specific application domains. Therefore, WS-SecurityPolicy has been defined to provide a set of policy

assertions that express security-related requirements and capabilities concerning the usage of WS-Security, WS-Trust, and WS-SecureConversation. In particular, WS-SecurityPolicy specifies the following key assertions:

- *Security Binding Assertions* provide requirements to secure an exchange of Web Service messages. Three types of binding assertions are specified by WS-SecurityPolicy that represent different security patterns. A *TransportBinding* specifies requirements for a message transfer across a secure channel (e.g. based on SSL), while *SymmetricBinding* and *AsymmetricBinding* state requirements to protect conveyed information at the message layer. Messages can be secured with the same security token for both message exchange directions (*SymmetricBinding*) or with different security tokens (*AsymmetricBinding*).

Additional assertions can be nested in security binding assertions to state specific requirements: *Token Assertions* specify type and properties of security tokens that must be used in the scope of a security binding, while *Algorithm Suite Assertions* define a set of required algorithms.

- *WS-Security* and *WS-Trust Assertions* require the support and compliance with WS-Security and WS-Trust options. These assertions are used to ensure the interoperability of participants concerning optional elements and different versions of these specifications.
- *Supporting Token Assertions* specify requirements regarding security tokens that must be included in a message (in addition to the token that is specified by the binding assertion). For example, this assertions can be used to require a SAML token to assert the authentication of users.
- *Protection Assertions* specify message parts that must be protected using signature or encryption mechanisms.
- *Required Element Assertions* require the existence of message parts in exchanged messages that are specified using XPath expressions.

Chapter 3

A Security Meta-Model for SOA

This chapter introduces a meta-model for security in service-based systems that consists of two parts. A basic SOA interaction model facilitates the description of participants and their relations in SOA, while an extension of this model represents security policies stating security requirements. This model provides the foundation for our model-driven approach and supports:

1. the specification of our security design language SecureSOA. This language is introduced in the next chapter and is used to state security requirements for SOA at the modelling layer. The definition of this language is based on a meta-model that extend our SOA interaction model with security intentions.
2. the specification of our domain-independent model. As introduced in section 1.2, a system design model enhanced with SecureSOA is transformed to our domain-independent model that can be translated to a specific security policy language. The meta-model for SOA security introduced in this chapter defines the elements of our domain-independent model.

In addition, an approach is introduced to formalise our meta-model. This formalisation is used to specify the formal semantics of our model.

3.1 A Model for Service Interactions

In this section, we will introduce the basic entities of our model and their relationships to describe an interaction in a Service-oriented Architecture. Further, we will show how these entities can be mapped to Web Service specifications such as SOAP.

3.1.1 The SOA Interaction Model

The basic actors (e.g. services and clients) participating in a service-based communication are represented as *objects* in our model. Objects consist of a set of *attributes* and can participate in an *interaction*, see Figure 3.1. An interaction is always performed on a *medium* that is connected to objects. For instance in the scope of Web Services, an object could be a Web Service client or a Web Service itself. A Web Service is bound to a medium – for example a TCP/IP network – and

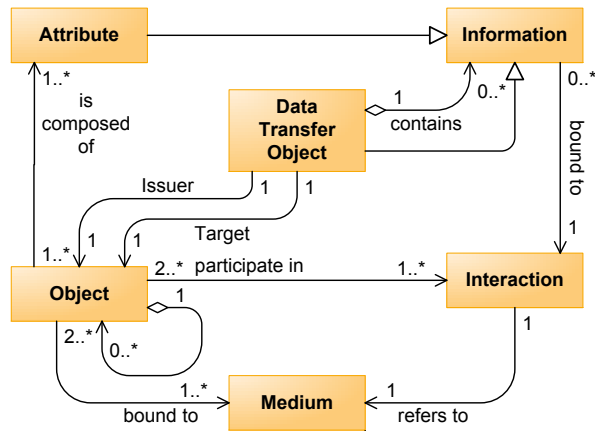


Figure 3.1: SOA Interaction Model

can interact with Web Service clients by exchanging SOAP-Messages. Therefore, an interaction also involves the exchange of *information*.

To enable a detailed description of Web Service messaging, we model transferred information as *data transfer objects* as introduced by Fowler in [Fow03]. A data transfer object is '[...] little more than a bunch of fields and the getters and setters for them. [...] it allows you to move several pieces of information over a network in a single call. [...] the data transfer object is responsible for serializing itself into some format that will go over the wire.'

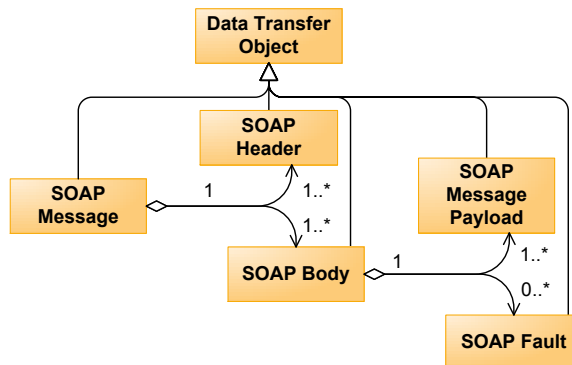


Figure 3.2: Modelling SOAP-based Message Structures

Figure 3.1 shows the adaptation of this concept to our model. A *data transfer object* represents serialised information and is an information itself. However, it can also contain information. This recursive structure facilitates the description of SOAP messages and encapsulated message parts. Figure 3.2 visualizes the mapping of our model to the SOAP message structure (cf. SOAP messaging framework specification [GHM⁺07]). A SOAP envelope is a data transfer object that can contain different message parts that are data transfer objects itself.

Moreover, a data transfer object has a *target* and an *issuer*. This reflects that a data transfer

object can be sent over several objects acting as intermediaries. Therefore, issuer and target do not have to correspond necessarily to the objects that are involved in an interaction exchanging a data transfer object. In the scope of Web Service technology, WS-Addressing [GHR06] would be used to represent issuer and target information in a SOAP-message by including a WS-Addressing header.

3.1.2 Modelling Digital Identities

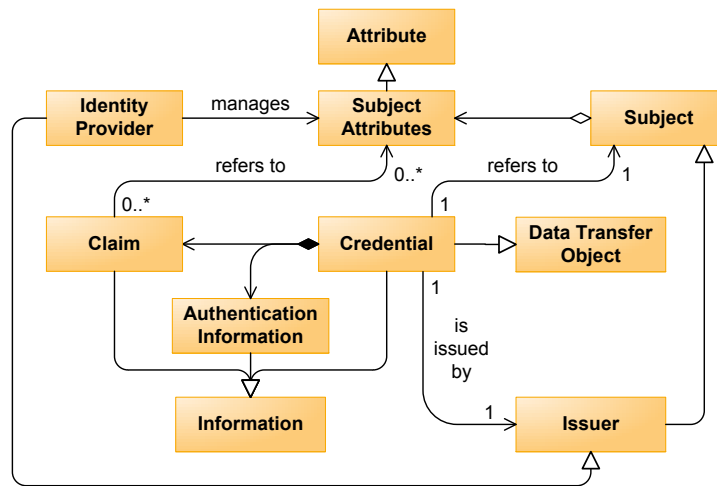


Figure 3.3: Digital Identity Model

In our SOA interaction model, objects such as Web Service clients are entities in a technical system that operate on behalf of an organisation or person. We refer to this organisation or person as *subject*. In the digital world, a subject is described by a set of *subject attributes* that are stored in an account managed by an *identity provider*, as shown in Figure 3.3. In reality, it is quite common that people (subjects) have multiple digital identities registered with different identity providers, for instance their employer, their email provider or various shopping sites in the internet.

In order to use services – whether Web Services or services in the internet – the exchange of identity information is required to identify, authenticate, and authorise a subject. This information is represented by a *credential* that contains a set of *claims* [Jon06] about the subject and an *authentication information*. A credential is created and asserted by an *issuer*, whose identity can be verified by the authentication information in the credential.

In a simple scenario, a user has to provide a username and a password to access a service. This credential (username/password) is issued by the user himself, contains a claim 'my user name is ...' and provides a password as authentication information. Furthermore, to enable single sign-on in a more complex scenario, the credential might be a SAML token [RHPM06] issued by the user's employer (acting as identity provider). Such a token could contain arbitrary claims about the user (e.g. his role in the organisation) and includes a signature as authentication information to enable a verification of the token and its claims.

In summary, claims are made by an issuer about a subject and represent a set of subject attributes. While in a closed, administered, and trustworthy security domain the term security assertion is commonly used, the term claim has been introduced in the scope of the Web Service specifications and is described by the identity metasystem [Jon06]. This term represents a degree of doubt regarding the brokering of identity information across trust domains in a loosely coupled system. The trustworthiness of a claim about a subject depends on the identity of the issuer.

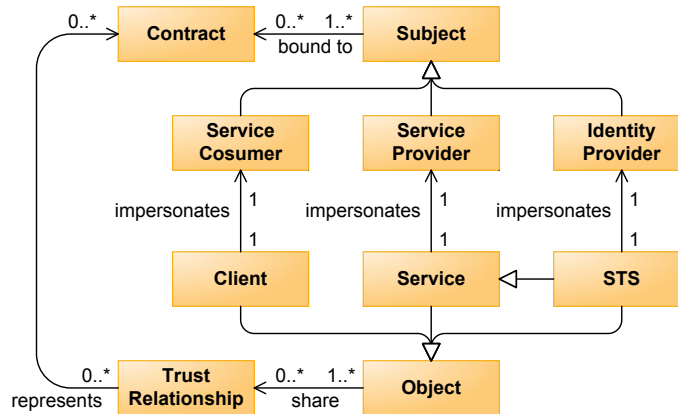


Figure 3.4: Roles in our SOA Interaction Model

As aforementioned, objects act in the digital world on behalf of subjects and, therefore, impersonate them. As shown in Figure 3.4, subjects and objects can act in different roles. *Services* are objects that offer capabilities on behalf of *service providers*, whereas a *service consumer* can interact with these services impersonating a *user*. An *identity provider* is a subject that manages a digital identity of other subjects and is impersonated by a *security token service* (service interfaces are defined in [NGG⁺07b, RHPM06]), which is a specialisation of a service.

Trust relationships exist between objects and provide the foundation to enforce security requirements in service-based systems. As defined by Jøsang et. al. [JFH⁺05], trust can be defined as *the extent to which one party is willing to depend on something or somebody in a given situation with a feeling of relative security, even though negative consequences are possible*. Trust relationships enable objects to rely on information provided by another party. In the scope of our meta-model, the meaning of this relationship depends on the context and the roles of the participating objects as follows:

1. **Trust relationship from a Service to an STS** - indicates that a services relies on statements made by a security token service. In particular, this service relies on authentication decisions and identity information provided by the STS. Moreover, this relation indicates that the service is able to verify the identity of the STS. In general, certificates provide the foundation for this authentication step. For example, a security token can be issued by an STS. The verification of the signature enables the service to rely on the information conveyed in this token.
2. **Trust relationship from an STS to a Client** - this relation is used to represent trust relations that are established between an STS and all users that are impersonated by this client. A trust relation between a user and an STS indicates that this user is registered at the identity provider that provides the STS. A credential (e.g. username/password)

facilitates the security token service to authenticate this user and enables the STS to rely on the identification of users. Since the user has to handle his credentials in a secure manner to avoid misuse, trust is required between the identity provider and the user.

3. **Trust relationship from a Service to a Client** - the meaning of this relation depends on the context. First of all, this relation indicates that a service is willing to rely on information provided by the client. For example, this information could represent a request message that will be processed by the service. This relation requires that the service is able to verify the identity of the client. In the scope of Web Service messaging, certificates provide the foundation to implement this requirement. On the other hand, this relation might indicate that the users impersonated by a client are registered at this service. In this case, this trust relationship is specified in accordance with the trust relationship 'STS to Client'. Since the service must be able to manage the identities of users, a user directory must be attached to the service. In the next chapter, security annotations will be introduced in the scope of SecureSOA to annotate this capability.

3.2 Modelling Security Requirements

In the previous section, we introduced the basic entities and their relations to model participants and interactions in a Service-oriented Architecture. Based on this general model, we will describe our approach to model security requirements in this section. Therefore, we will start with a basic model that will reveal the general structure of a security policy and its relation to other entities in our model. This generic description captures the essential policy elements to enable a mapping to any policy language. Based on this structure we will describe security constraints for specific security goals such as authentication and confidentiality.

3.2.1 Security Policy Structure

As we have outlined in the previous section, the interaction between objects and the exchange of data transfer objects are important concepts to model communication in distributed and loosely coupled systems. Security policies in such systems define requirements to restrict the communication between participants in order to comply to predefined security intentions and organisational regulations.

A *policy*, as shown in Figure 3.5, is stated by a *policy subject* (e.g. a service) to express the requirements of this object concerning the interaction with other objects. Therefore, we do not consider policies that relate to the internal functioning of an object. In our model, policy requirements always refer to interactions and related data transfer objects.

A policy consists of a set of *policy alternatives*. Each policy alternative requires a set of *security constraints* that describe requirements for a specific *security goal*. A policy will be fulfilled, if all security constraints of one policy alternative are enforced. Therefore, the usage of policy alternatives enables a disjunctive and conjunctive combination of security constraints and represents a disjunctive normal form. Since all logical formulas can be converted into a disjunctive normal form, any security policy structure based on a nesting of disjunctive and conjunctive combinations of policy requirements can be normalised and mapped to our model. Security requirements stated by security constraints refer to data transfer objects sent and received by the policy subject. These data transfer objects are denoted as constraint subjects. Two different

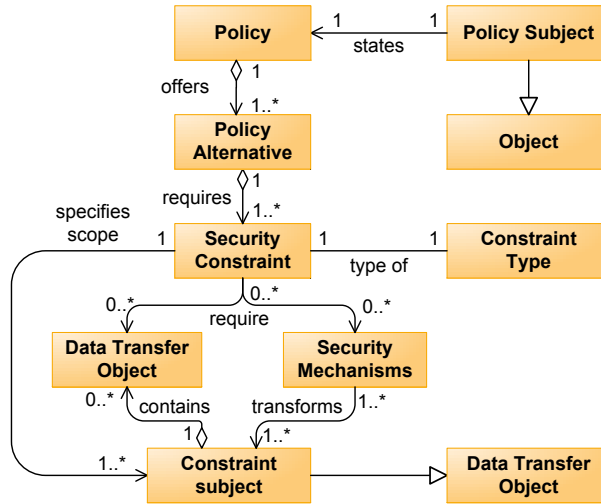


Figure 3.5: Security Policy Model

types of requirements can be stated by security constraints that must be applied to the constraint subjects. First of all, a constraint can specify the *security mechanisms* that must be applied to the constraint subjects. A security mechanism specifies a protocol or an algorithm – e.g. for encryption and signature – that is used to transform data transfer objects. In addition, a security constraint can specify a set of data transfer objects that are required to be included in the constraint subject. For example, a constraint could state that a SOAP request message is the constraint subject that must contain a specific type of credential for authentication.

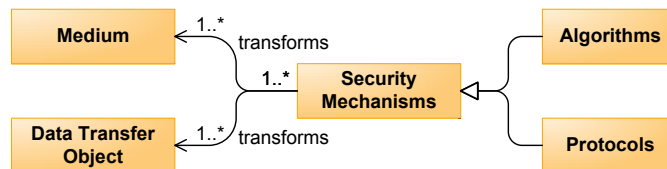


Figure 3.6: Security Mechanisms

3.2.2 Security Constraints for Authentication and Data Protection

Our security policy model defines a general structure to group and describe security constraints for distributed systems. Based on this model, we have specified specialised constraints that define precisely the required information and security mechanisms regarding the security goals authentication, confidentiality, and integrity.

Figure 3.7 illustrates the structure of an *authentication constraint*. This constraint requires a specific type of credential (such as SAML or username/password) that must be included in the data transfer objects. As aforementioned, these data transfer objects are identified by the constraint subject as shown in Figure 3.5. An authentication constraint specifies a set of required

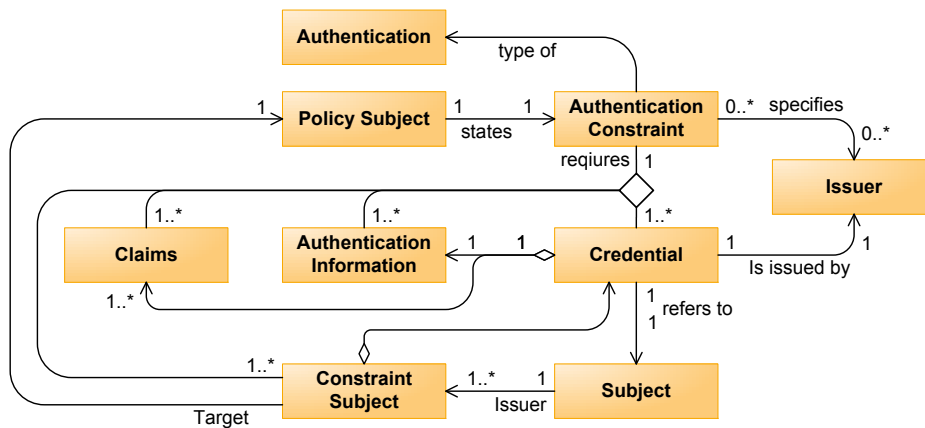


Figure 3.7: Authentication Constraint

claim types to identify the attributes that must be provided to the policy subject. In addition, an issuer can be specified (for instance a specific identity provider) that must have asserted this credential.

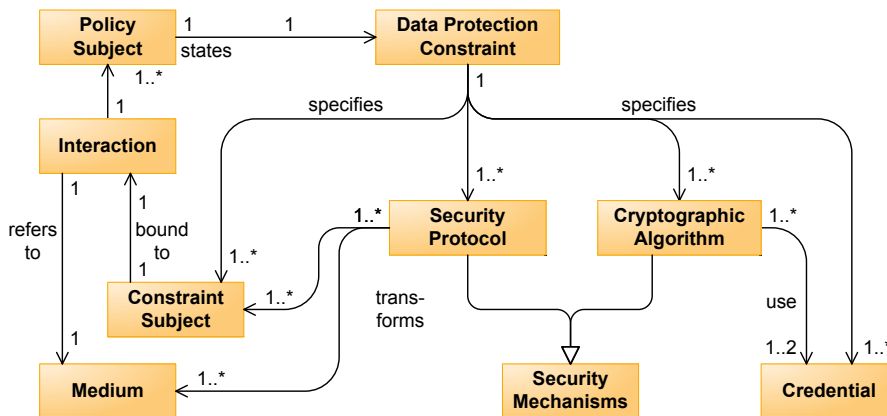


Figure 3.8: Data Protection Constraint

The data protection constraint is shown in Figure 3.8 and enables the specification of requirements concerning the integrity and confidentiality of exchanged information. This constraint specifies the following elements:

- a *security protocol* – this requirement identifies the protocol that is used to implement the protection of exchanged information (e.g. SSL to require a secure channel at the transport layer or WS-Security to secure data transfer objects itself).
- one or more *cryptographic algorithms* – one of these algorithms must be used by the protocol to protect information (e.g. AES or DES).
- one or two *credentials* – identifies the type of credential that is used as a key to secure information. Depending on the specified algorithm, there must be a single credential type

defined (e.g. symmetric encryption) or two types of credentials for incoming and outgoing data transfer objects (e.g. asymmetric encryption).

- one or more *constraint subjects* – define the data transfer objects that must be encrypted or signed (e.g. a credit card number or a message header).

As shown in Figure 3.9, the data protection constraint is used to specify the confidentiality constraint and integrity constraint. A specific constraint type is assigned to each subclassed constraint.

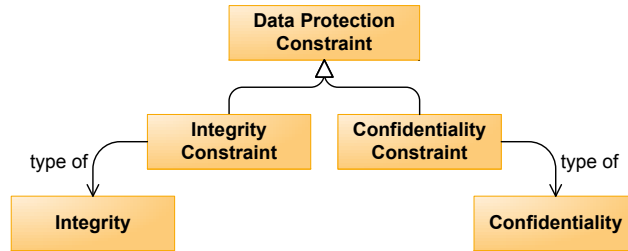


Figure 3.9: Confidentiality and Integrity Constraints

3.3 Meta-Model Formalisation

We can formalize our security meta-model as a relational model (based on sets and relations) as described by Lodderstedt [Lod04]. This formalisation provides a foundation to specify formal semantics for security constraints and facilitates the verification of the transformation steps in our model-driven approach.

A MOF-based model m (M1 level) described by a language l is expressed as a n -tuple $m_l := (c_1, \dots, c_n)$. Each object in the n -tuple represents a set containing instances of a specific class or relation in the meta-model. Subclasses in the meta-model are translated to sets that share elements with sets representing the parent classes. If c_1, \dots, c_n represent subclasses of c , then $\bigcup_{i=1}^n c_i = c$. To create a relational model, each class and association in the meta-model is mapped to a set c_i in a model m_l . The sets in the model m_l that represent a meta-model class are named after this class, while the names of sets representing an association are created using the capital letters of the related classes.

For instance, a simple example that connects a client to a service contains two participants (1: Web Frontend, 2: Web Service). These instances can be expressed as the following sets: $Object = \{1, 2\}$, $Client = \{1\}$, and $Service = \{2\}$. In addition, the following relations can be defined that represent the interactions and trust relationships in the example: $OO_{Interaction} = \{(1, 2)\}$ and $OO_{Trust} = \{(2, 1)\}$

3.3.1 Domain-Independent Relational Model

As introduced in this chapter, our SOA security meta-model is composed of our SOA interaction model and the policy model. These models define the entities of our domain-independent model

that facilitates the transformation of security intentions to enforceable security policies. Using the formalisation approach we can express the domain-independent model as a relational model.

Therefore, we can translate a domain-independent model based on our SOA security meta-model to a relational model m_{dim} :

$$\mathbf{m}_{dim} := (\textit{Object}, \textit{Client}, \textit{Service}, \textit{STS}, \textit{Interaction}, \\ \textit{TrustRelationship}, \textit{DataTransferObject}, \textit{Policy}, \\ \textit{PolicyAlternative}, \textit{SecurityConstraint}, \textit{AuthenticationConstraint}, \\ \textit{ConfidentialityConstraint}, \textit{IntegrityConstraint}, \textit{Claims}, \\ \textit{OI}, \textit{OT}, \textit{DI}, \textit{DD}, \textit{DO}_{\textit{Issuer}}, \textit{DO}_{\textit{Target}}, \textit{PO}, \textit{AP}, \textit{CA}, \textit{CD}, \textit{CC})$$

The model m_{dim} is composed of the following elements:

1. Sets for each class in the SOA interaction part of the SOA security meta-model:
Object, *Client*, *Service*, *STS*, *Interaction*, *TrustRelationship* and *DataTransferObject*.
2. Sets representing security policies and security constraints:
Policy, *PolicyAlternative*, *SecurityConstraint*, *AuthenticationConstraint*,
ConfidentialityConstraint, *IntegrityConstraint*, *Claim*
3. Sets that represent relations between the classes in the meta-model as shown in Table 3.1 and in Table 3.2.

As aforementioned, subclasses in the meta-model are translated to sets that share elements with sets representing the parent classes. It follows that $\textit{Object} = \textit{Client} \cup \textit{Service} \cup \textit{STS}$

Relation	Description
$\textit{OI} \subseteq \textit{Object} \times \textit{Interaction}$	objects that participate in interactions
$\textit{OT} \subseteq \textit{Object} \times \textit{TrustRelationship}$	objects that participate in trust relationships
$\textit{DI} \subseteq \textit{DataTransferObject} \times \textit{Interaction}$	information bound to an interaction
$\textit{DD} \subseteq \textit{DataTransferObject} \times \textit{DataTransferObject}$	used to describe the composition of data transfer objects
$\textit{DO}_{\textit{Issuer}} \subseteq \textit{DataTransferObject} \times \textit{Object}$	assigns an issuer to a data transfer object
$\textit{DO}_{\textit{Target}} \subseteq \textit{DataTransferObject} \times \textit{Object}$	assigns a target to a data transfer object

Table 3.1: Formalising Basic Relations in the Meta-Model

Relation	Description
$CD \subseteq SecurityConstraint \times DataTransferObject$	assigns a security constraint to a data transfer object
$PO \subseteq Policy \times Object$	assigns a policy to an object
$AP \subseteq PolicyAlternative \times Policy$	assigns alternatives to a policy
$CA \subseteq SecurityConstraint \times PolicyAlternative$	assigns constraints to an alternative
$CC \subseteq AuthenticationConstraint \times Claims$	assigns a required set of claims to an authentication constraint

Table 3.2: Formalising Relations in the Policy Meta-Model

In addition, the following relations are used:

1. X^* denotes the reflexive and transitive closure of a relation X .
2. X^{-1} represents the inverse relation of X with $X^{-1} := \{(y, x) \mid (x, y) \in X\}$.
3. $X \circ Y$ denotes the composition of the relations X and Y with $a X \circ Y c \Leftrightarrow \exists b : a X b \wedge b Y c$.

To describe the interactions and trust relationships between objects, we provide the following relations:

$$\begin{aligned}
 OO_{Interaction} &\subseteq Object \times Object & (3.1) \\
 &= \{(x, y) \in Object^2 \mid \exists i \in Interaction : x OIi \wedge y OIi\}
 \end{aligned}$$

$$\begin{aligned}
 OO_{Trust} &\subseteq Object \times Object & (3.2) \\
 &= \{(x, y) \in Object^2 \mid \exists t \in TrustRelationship : x OTt \wedge y OTt\}
 \end{aligned}$$

Based on this foundation, predications can be made concerning the structure of data transfer objects. Each data transfer object must be bound to an interaction:

$$\forall d \in DataTransferObject \exists i \in Interaction : (d, i) \in DI \quad (3.3)$$

Furthermore, each data transfer object must be associated to an issuer and a target. An interaction path must exist between the issuer and the target to ensure that this data transfer object can be exchanged. Since a data transfer object can be sent across multiple intermediaries, the reflexive and transitive closure of $OO_{Interaction}$ is used to express this predicate:

$$\begin{aligned}
 \forall d \in DataTransferObject \exists o_{Issuer} \in Objects \exists o_{Target} \in Objects : & \quad (3.4) \\
 & (d, o_{Issuer}) \in DO_{Issuer} \\
 \wedge & (d, o_{Target}) \in DO_{Target} \\
 \wedge & o_{Issuer} OO_{Interaction}^* o_{Target}
 \end{aligned}$$

3.3.2 Security Constraints Formal Semantics

Security constraints represent requirements concerning the structure and the properties of data transfer objects that are sent to a specific object. We define the functions sig_{dim} , enc_{dim} , $auth_{dim}$, and id_{dim} that evaluate a particular property of a data transfer object:

Data Integrity -

The function $sig_{dim} : DataTransferObject \mapsto \{true, false\}$ returns true, if a signature is attached to a data transfer object $d \in DataTransferObject$. This function can be specified in dependency to the *IntegrityConstraint* set:

$$\begin{aligned} sig_{dim}(d) \Leftrightarrow \exists a \in SecurityAlternative : d DO_{Target} \circ PO^{-1} \circ AP^{-1} a \wedge & \quad (3.5) \\ \forall a \in SecurityAlternative \text{ where } d DO_{Target} \circ PO^{-1} \circ AP^{-1} a : & \\ \exists c \in IntegrityConstraint : c CD d \wedge c CA a & \end{aligned}$$

A data transfer object d is signed, if

1. all policy alternatives that are offered by the target of the data transfer object d (line 2) contain an integrity constraint that refers to d (line 3).
2. at least one security alternative is offered by the target of d (line 1). In combination, both conditions guarantee the existence of an integrity constraint that requires the protection of d .

Data Confidentiality -

The function $enc_{dim} : DataTransferObject \mapsto \{true, false\}$ returns true, if a data transfer object $d \in DataTransferObject$ is encrypted. This function can be specified in dependency to the *ConfidentialityConstraint* set:

$$\begin{aligned} enc_{dim}(d) \Leftrightarrow \exists a \in SecurityAlternative : d DO_{Target} \circ PO^{-1} \circ AP^{-1} a \wedge & \quad (3.6) \\ \forall a \in SecurityAlternative \text{ where } d DO_{Target} \circ PO^{-1} \circ AP^{-1} a : & \\ \exists c \in ConfidentialityConstraint : c CD d \wedge c CA a & \end{aligned}$$

A data transfer object d is encrypted, if

1. all policy alternatives that are offered by the target of the data transfer object d (line 2) contain an confidentiality constraint that refers to d (line 3).
2. at least one security alternative is offered by the target of d (line 1). In combination, both conditions guarantee the existence of an confidentiality constraint that requires the protection of d .

User Authentication -

The function $auth_{dim} : DataTransferObject \mapsto \{true, false\}$ returns true, if a credential is attached to a data transfer object $d \in DataTransferObject$. This credential enables the authentication of users. The function $auth_{dim}$ can be specified in dependency to the *AuthenticationConstraint* set:

$$\begin{aligned} auth_{dim}(d) \Leftrightarrow \exists a \in SecurityAlternative : d DO_{Target} \circ PO^{-1} \circ AP^{-1} a \wedge & \quad (3.7) \\ \forall a \in SecurityAlternative \text{ where } d DO_{Target} \circ PO^{-1} \circ AP^{-1} a : & \\ \exists c \in AuthenticationConstraint : c CD d \wedge c CA a & \end{aligned}$$

A data transfer object d conveys a credential that authenticates an user, if

1. all policy alternatives that are offered by the target of the data transfer object d (line 2) contain an authentication constraint that refers to d (line 3).
2. at least one security alternative is offered by the target of d (line 1). In combination, both conditions guarantee the existence of an authentication constraint that requires the provision of a credential.

Identity Provisioning -

The function $id_{dim}^{claim} : DataTransferObject \mapsto \{true, false\}$ returns true, if a credential with a specific claim is attached to a data transfer object. This function can be specified in dependency to the *AuthenticationConstraint* set:

$$\begin{aligned}
 id_{dim}^{claim}(d) \Leftrightarrow & \exists a \in SecurityAlternative : d DO_{Target} \circ PO^{-1} \circ AP^{-1} a \wedge & (3.8) \\
 & \forall a \in SecurityAlternative \text{ where } d DO_{Target} \circ PO^{-1} \circ AP^{-1} a : \\
 & \exists c \in AuthenticationConstraint : c CD d \wedge c CA a \wedge a CC claim
 \end{aligned}$$

A data transfer object d conveys a credential that provides a specific claim, if

1. all policy alternatives that are offered by the target of the data transfer object d (line 2) contain an authentication constraint that refers to d and requires the specified claim (line 3).
2. at least one security alternative is offered by the target of d (line 1). In combination, both conditions guarantee the existence of an authentication constraint that requires the provision of the claim.

Chapter 4

SecureSOA: A Language to Model Security Intentions

The SOA security meta-model introduced in the previous chapter provides the foundation to define our security modelling language SecureSOA. As outlined in section 1.2, SecureSOA enables the integration of basic security intentions in system design models to facilitate an automated generation of security policies. In this chapter, we introduce SecureSOA by enhancing our SOA interaction model with additional artefacts to enable the representation of security intentions and capabilities. In addition to the meta-model, we specify the visualisation of the SecureSOA modelling elements and the formal semantics of the security intentions. Moreover, we discuss different strategies in this chapter to enhance arbitrary system design models with our security modelling language and provide an integration schema that is based on the schema used by SecureUML [BDL06]. A modelling dialect on the basis of FMC block diagrams is introduced to illustrate the integration of SecureSOA in a design modelling language.

4.1 Providing Security Design Languages - Overview

Various modelling languages and dialects have been defined that can be used to model different aspects in an SOA. For instance, the system structure can be visualised in UML or FMC, while the processes executed by this system can be modelled with BPMN. Each modelling language provides a specific view on a particular aspect of the system that can be used to annotate security intentions. To aggregate and enforce intentions from different types of modelling languages, security intentions must be defined consistently and independent from any modelling language. In this section we discuss strategies to integrate security intentions into modelling languages and outline the steps of our approach.

4.1.1 Enhancing Modelling Languages

To integrate security modelling elements in system design languages, an enhancement of these languages is required. In general, three approaches can be distinguished to implement such an enhancement:

1. **Light-weight extensions** – The easiest way to enhance a particular system design language is the usage of extension points provided by the language itself. For instance, UML provides stereotypes and tags to extend UML modelling elements. Light-weight UML extensions are used by UMLsec to express security requirements. However, the visualisation of complicated security requirements might get confusing. Moreover, not all modelling languages define extension points to enhance modelling elements. For example, FMC diagrams do not offer such points.
2. **Heavy-weight extensions** – Another approach to enhance modelling languages is based on the extension of its meta-model. For example, this approach is used by Rodríguez to define his security extensions for BPMN and UML [RFMP07] process models. A major disadvantage of this approach, however, is that the definition and integration of security requirements is done specifically for a particular system design modelling language based on its meta-model.
3. **Defining a new language** – To avoid the drawbacks mentioned above, a new modelling language can be defined. This modelling language integrates security elements and contains specific redefined elements of a system design modelling language. SecureUML uses this approach to model security requirements as an integral part of system models. Therefore, Basin and Lodderstedt described a generic approach to create a new security design languages by integrating security modeling languages into system design modelling languages as described in [BDL06].

4.1.2 Defining Modelling Design Languages

The schema described by Basin and Lodderstedt provides a flexible approach to construct security design languages. A security modelling language is defined once with certain extension points that enable an integration into different design modelling languages for service-based systems. The resulting languages are denoted as modelling dialects. Moreover, formal semantics can be provided for the security modelling language that enable the verification of the requirements modelled in any dialect. We have adopted this approach as shown in Figure 4.1.

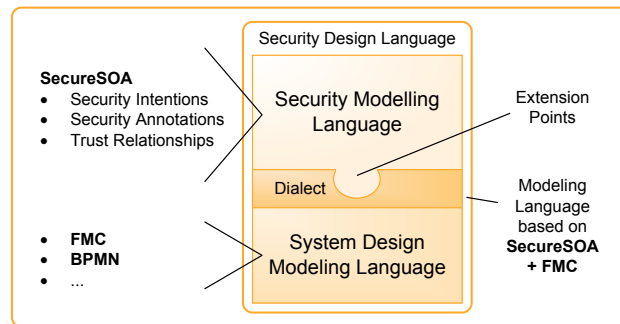


Figure 4.1: Schema for constructing Security Design Languages

The schema consists of the following parts:

1. A security modelling language is used to express security requirements for a specific purpose. We have defined SecureSOA that enables the modelling of security intentions and capabilities for services-based systems.

2. The structure of a system is described by a system design modelling language. While different types of modelling languages can be used, our approach is based on FMC Block Diagrams that enable the visualisation of system architectures.
3. Both languages are integrated by merging their vocabulary using the extension points of the security modelling language. The resulting language is called a dialect.

1) Subclassing extension points	
	<p>The easiest way to perform the integration is to map a design modelling language class m_j to its corresponding extension point s_i in SecureSOA that represents an abstraction of this class. This dependency can be represented as an inheritance relationship in the dialect between the classes s_i and m_j.</p>
2) Enhancing the dialect with new classes	
	<p>This schema enables the mapping of SecureSOA subclasses. A SecureSOA class s_i is inherited by a class $s_{i'}$ that models a specific aspect of service-based systems. Although a class s_i can be mapped to a class m_j as described by schema 1, it is unlikely to find a class in a general purpose modelling language that corresponds to the specialised class $s_{i'}$. To associate the extension point $s_{i'}$ with the design modelling language, it is necessary to enhance the dialect with a new class d_i that inherits the class $s_{i'}$ and m_j.</p>
3) Defining associations and OCL constraints	
	<p>However, there might not be a straight mapping for each extension point of SecureSOA, since certain aspects might be modelled on different levels of abstraction in both languages. In this case, a class s_i has to be mapped to multiple classes m_j and m_k in the other language. To integrate these classes into the dialect, associations have to be defined between the corresponding classes. OCL constraints can be used to capture additional semantics of these dependencies.</p>

Table 4.1: Schemas for Creating the Modelling Dialect

4.1.3 Merging Security and System Design Languages

In SecureSOA, extension points are represented by basic classes that are defined by our SOA interaction model. These extension points can be mapped to entities in any system design model. For example, our model represents participants in an SOA such as services as objects that participate in an interaction by exchanging information. FMC visualises system architectures that are composed of agents communicating over a channel. Therefore, an object is an extension point and can be mapped to an agent.

As stated by Lodderstedt in [Lod04], there is no universal approach to perform an integration of arbitrary security and design modelling languages. The integration technique depends on the structure of the security modelling language. In the scope of SecureSOA, we have identified

three integration patterns that are listed in Table 4.1. The definition of these patterns is based on classes and relationships in the meta-models of the security design language and the system design language. We denote the set of classes in the SecureSOA meta-model as $s = \{s_1, \dots, s_{n_1}\}$, classes in the meta-model of the system design language as $m = \{m_1, \dots, m_{n_2}\}$ and classes in the meta-model of the dialect as $d = \{d_1, \dots, d_{n_3}\}$.

4.2 SecureSOA - A Security Design Language for SOA

SecureSOA is our security modelling language that enables a modelling of security requirements and capabilities for service-based systems and is defined by a MOF-based meta-model (abstract syntax). In addition, we will introduce the notion of these elements (concrete Syntax) as UML profiles and the definition of the formal semantics of SecureSOA.

4.2.1 SecureSOA Abstract Syntax

Our security modelling language SecureSOA is based on the SOA interaction model that has been introduced in the previous chapter. The elements in this model provide basic entities and relations to describe interactions in a Service-oriented Architecture. These elements constitute the extension points of our model that are mapped to entities in the meta-model of the system design model as described by the integration schema introduced above.

In addition to this set of basic entities, we have to specify additional elements to represent security requirements and security capabilities. Therefore, we introduce security intentions and security annotations as new modelling elements in this section.

4.2.1.1 Modelling of Security Intentions

The security intention meta-model is illustrated in Figure 4.2. A security intention states security requirements (e.g. the enforcement of a specific security goal or the provision of security related information) for a security intention subject (*Intention Subject*) that is either an object or a data transfer object.

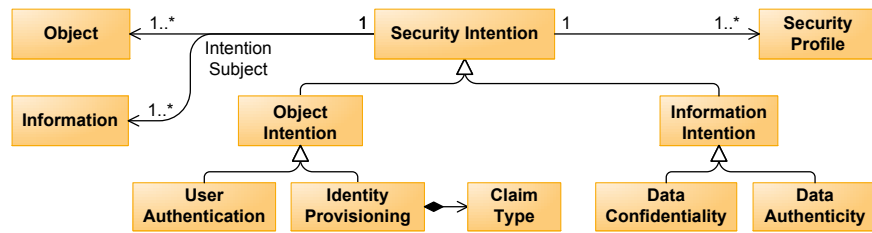


Figure 4.2: Modelling Security Intentions

In the scope of this thesis, we provide the following basic set of security intentions that are introduced in Table 4.2: *User Authentication*, *Identity Provisioning*, *Data Authenticity*, and *Data Confidentiality*. As illustrated in Figure 4.2, these security intention classes are derived from the class *Security Intention*. SecureSOA provides an open and extensible model that is not

limited to these intentions. Custom security intentions can be defined by deriving additional security intention classes. For example, a non-repudiation intention could be defined.

Security Intention	Intention Type	Description
User Authentication	Object Intention	requires the trustworthy identification and authentication of users that access the <i>Intention Subject</i> . Data Transfer Objects (e.g. messages) sent to the subject must contain a credential that proves the identity of these users.
Identity Provisioning	Object Intention	requires the provisioning of identity information to the <i>Intention Subject</i> . Data Transfer Objects (e.g. messages) sent to the subject must contain a credential providing the required set of claims.
Data Confidentiality	Information Intention	indicates that the confidentiality of the information conveyed by the subject must be guaranteed.
Data Authenticity	Information Intention	indicates that the integrity of the information conveyed by the subject must be ensured. In addition, the trustworthiness of the sender must be ensured.

Table 4.2: SecureSOA Security Intentions

One of the key components of our security modelling language SecureSOA are data transfer objects that enable interactions in a service-based system. Security intentions state requirements that affect the structure of exchanged data transfer objects. For example, the security intention *Data Authenticity* requires that a data transfer object must contain a signature, while the intention *User Authentication* requires data transfer objects to contain a credential. The requirements stated by an intention must be enforced by the receivers of these data transfer objects. Therefore, security intentions always have an impact on an object and on a set of data transfer objects.

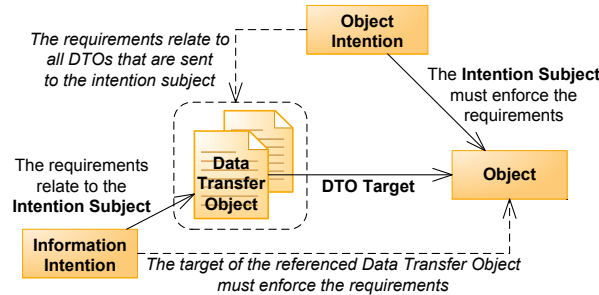


Figure 4.3: Security Intention Types and their Dependencies

As shown in Figure 4.2, we distinguish two types of security intentions. An *Information Intention* is attached to a data transfer object that is the subject of this intention. For example, a data transfer object representing a credit card information can be annotated in a BPMN diagram using a *Data Confidentiality* intention. However, it is not feasible for all security requirements to be represented as annotations on data transfer objects. For instance, *Identity Provisioning* requires the provisioning of identity information to a service. Since this security intention refers to a service, the subject of this intention is of type object. We denote this class of security intentions

as *Object Intention*. All data transfer objects that are sent to the subject of an object intention must comply with the requirements specified by this intention. Figure 4.3 illustrates the two types of security intentions and their dependencies to objects and data transfer objects. The *Object Intention* and the *Information Intention* are connected to the corresponding subjects, while the dotted line represents the implicit relationship to data transfer objects or objects.

The annotation of data transfer objects to specify security requirements presumes that there are modelling elements provided by the system design language to represent data transfer objects. This is not the case in each system design modelling language. While BPMN enables the modelling of data objects, FMC block diagrams visualises the static structure of a system, but does not provide a view on data objects. To enable the specification of information intentions in these diagram types as well, we allow information intentions to be connected to objects. In this case, an information intention applies to all data transfer object that are exchanged with this object. This aspect will be considered in the definition of the formal semantics of SecureSOA. An example will be provided in section 4.4.

Use	Security mechanism	Version	
✓	Secure Sockets Layer (SSL)	2.0	remove
✓	Web Services security protocol (WSS)	1.0	remove
✓	Security Assertion Markup Language (SAML)	2.1	remove
✓	Username/password token (UsernamePasswordToken)	-	remove
✓	Basic 256 RSA Algorithm Suite (Basic256)	-	remove

Figure 4.4: Security Profile Example

In addition, further information must be provided at the modelling layer to enable a model-driven transformation. Security mechanisms and protocols must be selected for each security intention to enable the generation of security policies. However, this choice depends on

1. the capabilities of the platform or runtime environment that enforces the generated security policies. For example, different Web Service frameworks support different sets of algorithm suites and protocol versions.
2. the required security level. The security level determines the minimal key strength and the type of authentication mechanisms that must be used.
3. requirements of the modeller. The modeller might have reasons to require the usage of specific protocol versions and security mechanisms to enforce a security intention.

However, our modelling approach intends to hide technical details at the modelling layer. The modeller should not be bothered with details such as security algorithms and mechanisms that

are used to enforce a specific intention. Nevertheless, the modeller should be able to influence the selection of security mechanisms if this is required.

To solve this issue, our approach is based on the usage of *Security Profiles* that provide predefined sets of security mechanisms and protocols. Security profiles are labelled by a name and are referenced by the security intentions as shown in the security intention meta-model in Figure 4.2. An example of a security profile denoted as 'default' is illustrated in Figure 4.4. The modeller of the system can assign such a default profile providing the most common security mechanisms to an intention or he can select another predefined profile. Since a security profile is just a set of security mechanisms and a name, a modeller can easily define custom profiles to enforce a specific security intention.

A basic sets of security profiles must be provided by the implementation of our model-driven approach according to the capabilities of the run-time environment. The name of a profile can indicate a security level, but this is just an informal association that is specific for this security profile. We chose not to define and use security levels explicitly in our approach, since this would require the definition of all-encompassing, holistically metrics to enable an automated assignment of security mechanisms and protocols to these security levels.

To facilitate the assignment of security mechanisms to security profiles, common security patterns can be used. For example, Schumacher provided a pattern catalogue in [SFBH⁺06] that defines the pattern 'automated I&A Design Alternatives'. This pattern provides a process to choose among authentication techniques alternatives. In particular, a process is described to collect requirements that are used to evaluate authentication techniques. This evaluation can be used as a basis to create identification and authentication profiles.

4.2.1.2 Modelling of Security Annotations

While security intentions facilitate the specification of security requirements, we need additional modelling elements to represent capabilities of objects. A *Security Annotation* is a modelling element that associates a set of security attributes with a subject. Similar to security intentions, security annotations can relate to data transfer objects or to objects as illustrated in Figure 4.5.

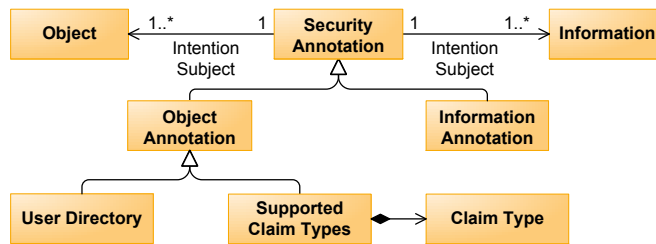


Figure 4.5: Modelling Security Annotations

In the scope of this thesis, we provide two security annotations that are related to identity management capabilities:

1. **User Directory** – this annotation element indicates that an user directory is attached to an object. An user directory stores the digital identities of users and enables attached objects the authentication of registered users. The capabilities described by the annotation *User Directory* are considered in the selection phase of security patterns.

To illustrate the usage of this annotation, consider a service that is connected to a client. An *User Authentication* intention is attached to the service to require the authentication of users. Therefore, all service requests that are sent to this service have to provide a credential. To select security patterns that configure the requirements concerning the provision of security credentials, the SecureSOA model must identify the objects in the system that are capable to authenticate users. This information is provided by the annotation *User Directory*. Since each STS provides a user directory by default, the *User Directory* annotation is specified implicitly for this type of object.

2. **Supported Claim Types** – Next to the authentication of users, services might require the provisioning of identity information. The set of identity claims required at a service can be specified using the security intention *Identity Provisioning*. To identify the objects in SecureSOA that are capable to provide a specific set of claim types, the security annotation *SupportedClaimTypes* can be used. In particular, this annotation can be used to specify the claim types supported by an security token services and clients. For instance, a client can request the user to enter the required identity information that are provided to the relying party as a SAML-Token.

4.2.2 SecureSOA Concrete Syntax

The concrete syntax specifies the visualisation of elements that are defined by the abstract syntax. Since the extension points of the SecureSOA meta-model provided by our SOA interaction model are mapped to classes in the design modelling language, their notion is already defined by this language. Therefore, we just have to define the notion of security intentions and annotations.





UML Stereotype	Symbol
<< User Authentication >>	
<< Identity Provisioning >>	
<< Data Authenticity >>	
<< Data Confidentiality >>	

Table 4.3: SecureSOA Concrete Syntax

In general, there are two possibilities to define a concrete syntax (notion) for these elements. The first option is to express them as a property of the subject of the element. For example, if the notion of the design modelling language is based on UML, then the subject of an intention might be visualised as an UML class. In this case, the notion of this class could be extended to contain an additional property 'user identification'. This property would offer the possibility to specify a set of claim types that are required by an activity. Another option to visualise security requirements is the definition of artefacts for each element that can be used to annotate the element's subject. We have chosen this approach to define the concrete syntax of security intentions specified by SecureSOA.

Our notion for security intentions is based on an UML concrete syntax using UML classes and stereotypes. Each intention is visualised as an UML class that is connected to the intention's subject using an UML association. The mapping between SecureSOA intentions and UML stereotypes is listed in Table 4.3. Figure 4.6 illustrates the notion of the User Authentication and Identity Provisioning intentions.

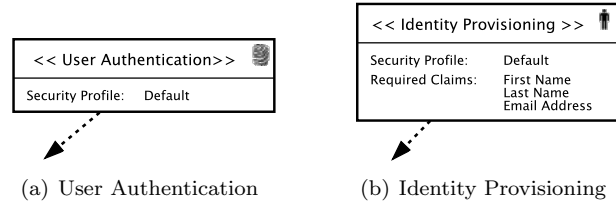


Figure 4.6: Notion of Security Intentions

In addition to security intentions, we have to define the notion of security annotations. Since a user directory and a set of claim types represent data storages, we use the concrete syntax for storages provided by FMC block diagrams to represent these elements. The visualisations of these annotations are shown in Figure 4.7.

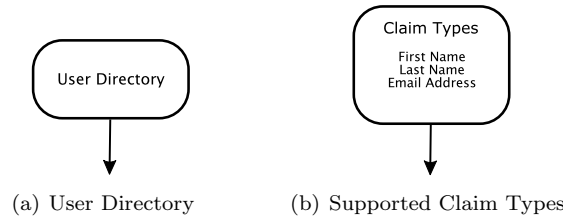


Figure 4.7: Notion of Security Annotations

4.2.3 SecureSOA Formal Semantics

As described in section 3.3, our formalisation approach is based on a mapping of a MOF-based model described by a language l to a relational model m_l that is based on sets and relations.

Our security modelling language SecureSOA enhances our SOA interaction model with security intentions and security annotations. Security intentions and security annotations refer to a specific subject (e.g. a service) and can specify multiple parameters such as required claim types. Using our formalisation approach, we can translate a security design modelling language based on SecureSOA to a relational model $m_{SecureSOA}$ that is defined as follows:

$$\mathbf{m}_{SecureSOA} := (\textit{Object}, \textit{Client}, \textit{Service}, \textit{STS}, \textit{Interaction}, \textit{DataTransferObject}, \textit{TrustRelationship}, \textit{Claim}, \textit{SecurityIntention}, \textit{ObjectIntention}, \textit{InformationIntention}, \textit{DataConfidentiality}, \textit{DataAuthenticity}, \textit{UserAuthentication}, \textit{IdentityProvisioning}, \textit{SecurityAnnotation}, \textit{ObjectAnnotation}, \textit{InformationAnnotation}, \textit{UserDirectory}, \textit{SupportedClaimTypes}, \textit{SecurityProfile}, \textit{SecurityProfileLabel}, \textit{SecurityMecanism}, \textit{OI}, \textit{OT}, \textit{DI}, \textit{DD}, \textit{DO}_{Issuer}, \textit{DO}_{Target}, \textit{SO}, \textit{SD}, \textit{SL}, \textit{ML}, \textit{LL}, \textit{IC}, \textit{AO}, \textit{AD}, \textit{TC})$$

The model $m_{SecureSOA}$ is composed of the following elements:

1. Sets and relations specified by the SOA interaction model as specified in section 3.3.
2. Sets for each security intention and associated data: *SecurityIntention*, *ObjectIntention*, *InformationIntention*, *DataConfidentiality*, *DataAuthenticity*, *UserAuthentication*, *IdentityProvisioning*, and *Claim*.
3. Sets for each security annotation: *SecurityAnnotation*, *ObjectAnnotation*, *UserDirectory*, *InformationAnnotation*, *SupportedClaimTypes*.
4. Sets that represent relations between the classes in the SecureSOA meta-model. In addition to the relations DD , OI , OT , DI , DO_{Issuer} , DO_{Target} that are defined in Table 3.1, relations referring to security intentions and annotations are required as listed in Table 4.4.
5. The set *SecurityProfile* is used to represent security profiles that are referenced by security intentions. In addition, the set *SecurityProfileLabel* $\subseteq \Sigma^*$ contains all labels that are used to identify security profiles, while *SecurityMechanism* $\subseteq \Sigma^*$ represents security algorithms and protocols that can be used in the profiles.

Relation	Description
$SO \subseteq SecurityIntention \times Object$	assigns the subject to an object intention (cf. 4.2.1.1).
$SD \subseteq SecurityIntention \times DataTransferObject$	assigns the subject to an information intention (cf. 4.2.1.1).
$SL \subseteq SecurityIntention \times SecurityProfile$	assigns a security intention to a security profile.
$ML \subseteq SecurityMechanism \times SecurityProfile$	assigns security mechanisms to security profiles.
$LL \subseteq SecurityProfileLabel \times SecurityProfile$	assigns labels to security profiles.
$IC \subseteq IdentityProvisioning \times Claim$	assigns a required set of claims to the security intention <i>Identity Provisioning</i> .
$AO \subseteq SecurityAnnotation \times Object$	assigns the subject to an object annotation 4.2.1.2.
$AD \subseteq SecurityAnnotation \times DataTransferObject$	assigns the subject to an information annotation 4.2.1.2.
$TC \subseteq SupportedClaimTypes \times Claim$	assigns a required set of claims to the annotation <i>Supported Claim Types</i> .

Table 4.4: Formalising Relations in the SecureSOA Meta-Model

Since subclasses in the meta-model are translated to sets that share elements with sets representing parent classes, it follows that

$$\begin{aligned}
 SecurityIntention &= ObjectIntention \cup InformationIntention \\
 ObjectIntention &= UserAuthentication \cup IdentityProvisioning \\
 InformationIntention &= DataConfidentiality \cup DataAuthenticity \\
 SecurityAnnotation &= ObjectAnnotation \cup InformationAnnotation \\
 ObjectAnnotation &= UserDirectory \cup SupportedClaimTypes
 \end{aligned}$$

The relations SO and SD in the meta-model are used to assign a subject (object or information) to a security intention. In particular, the relation SO will be used, if the intention is an object intention, while SD is used to assign a subject to an information intention. As described in section 4.2.1.1, an information intention can also be assigned to an object using the relation SO . In this case, this intention applies to all data transfer objects that are exchanged with this object. Therefore, it follows that

$$\begin{aligned} & \forall i \in \text{InformationIntention} : \\ (\exists o \in \text{Objects} : (i \text{ SO } o) \Rightarrow \forall d \in \{d \mid d \text{ DO}_{\text{Target}} o \vee d \text{ DO}_{\text{Issuer}} o\} : (i \text{ SD } d)) \end{aligned} \quad (4.1)$$

In the following, $SO_x = \{(s, d) \in SO \mid s \in X\}$ represents a subset of SO for a specific security intention X . SD_X is defined correspondingly.

Security intentions represent requirements concerning the structure and the properties of data transfer objects that are sent to a specific object. To define the semantics of the security intentions, we define the functions $sig_{\text{SecureSOA}}$, $enc_{\text{SecureSOA}}$, $cred_{\text{SecureSOA}}$, and $id_{\text{SecureSOA}}$ that evaluate a particular property of a data transfer object:

Data Authenticity -

The function $sig_{\text{SecureSOA}} : \text{DataTransferObject} \mapsto \{true, false\}$ returns true, if a signature is attached to a data transfer object $d \in \text{DataTransferObject}$. This function can be specified in dependency to the security intention *Data Authenticity*:

$$sig_{\text{SecureSOA}}(d) \Leftrightarrow \exists i \in \text{DataAuthenticity} : (i \text{ SD } d) \quad (4.2)$$

A data transfer object d is signed, if a *Data Authenticity* intention is attached to a data transfer object.

Data Confidentiality -

The function $enc_{\text{SecureSOA}} : \text{DataTransferObject} \mapsto \{true, false\}$ returns true, if a data transfer object $d \in \text{DataTransferObject}$ is encrypted. This function can be specified in dependency to the security intention *Data Confidentiality*:

$$enc_{\text{SecureSOA}}(d) \Leftrightarrow \exists i \in \text{DataConfidentiality} : (i \text{ SD } d) \quad (4.3)$$

A data transfer object d is encrypted, if a *Data Confidentiality* intention is attached to a data transfer object.

Authentication -

The function $auth_{\text{SecureSOA}} : \text{DataTransferObject} \mapsto \{true, false\}$ returns true, if a credential is attached to a data transfer object $d \in \text{DataTransferObject}$. This function can be specified in dependency to the security intention *User Authentication*:

$$\begin{aligned} auth_{\text{SecureSOA}}(d) \Leftrightarrow & \exists i \in \text{UserAuthentication} : \\ & (d \text{ DO}_{\text{Target}} \circ SO^{-1} i) \vee (d \text{ DO}_{\text{Issuer}} \circ SO^{-1} i) \end{aligned} \quad (4.4)$$

A data transfer object d contains a credential, if it is sent or received by an object that is attached to an *User Authentication* intention i .

Identity Provisioning -

The function $id_{SecureSOA}^c : DataTransferObject \mapsto \{true, false\}$ returns true, if a credential with a claim c is attached to a data transfer object $d \in DataTransferObject$. This function can be specified in dependency to the security intention *Identity Provisioning*:

$$id_{SecureSOA}^c(d) \Leftrightarrow \exists i \in IdentityProvisioning : ((d DO_{Target} \circ SO^{-1} i) \vee (d DO_{Issuer} \circ SO^{-1} i)) \wedge (i IC c) \quad (4.5)$$

A data transfer object d contains a credential with a claim c , if it is sent or received by an object that is attached to an *Identity Provisioning* intention i that requires this claim.

In addition, we have to state the following predicates concerning the trust relationships to ensure that an object can authenticate its users. An interaction between a client and another object implies that there must be a trust relationship if a security intention *User Authentication* or *Identity Provisioning* is attached to the object. Let $SO_{OI} := SO_{UserAuthentication} \cup IdentityProvisioning$, then it follows that

$$\forall c \in Client \forall o \in Object : (c OO_{Interaction}^* o \wedge \exists i : i SO_{OI} o) \Rightarrow o OO_{Trust}^* c \quad (4.6)$$

4.3 A SecureSOA Dialect based on FMC

SecureSOA offers the possibility to express security intentions in various modelling languages. We have chosen FMC Compositional Structure Diagrams (Block Diagrams) as a system design modelling language, since FMC offers a suitable foundation to describe an SOA on a technical layer in terms of involved participants and their communication channels.

4.3.1 Fundamental Modeling Concepts

Fundamental Modeling Concepts (FMC) provides an approach to describe software systems. It can be used to model the structure, processes, and value domains of a system.

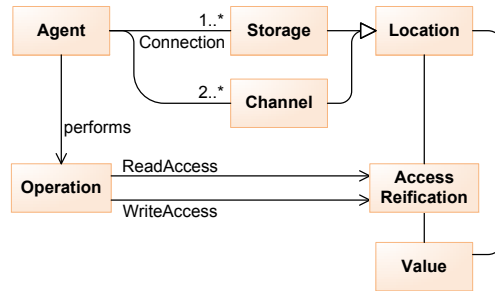


Figure 4.8: MOF-based FMC Meta-Model

FMC Compositional Structure Diagrams (also known as FMC Block Diagrams) depict the static structure of a system and the relationships between system components. This diagram type distinguishes between active and passive components. Agents are active system components

that are capable to communicate via channels and to perform activities in the system. Channels and storages are passive components used to transmit or store information.

The FMC meta-model [TAW05] describes the abstract syntax for all diagram types and is specified using FMC entity relationship diagrams. We have translated the FMC meta-model to a MOF-based meta-model. Figure 4.8 depicts the part of the MOF-based meta-model that describes FMC block diagrams. Agents are connected to storages or channels and interact by performing read or write operations.

4.3.2 Merging SecureSOA and FMC

To integrate SecureSOA in FMC, the vocabularies of both languages have to be merged and the entities in FMC have to be mapped to corresponding extension points in SecureSOA. The meta-model of the dialect is shown in Figure 4.9.

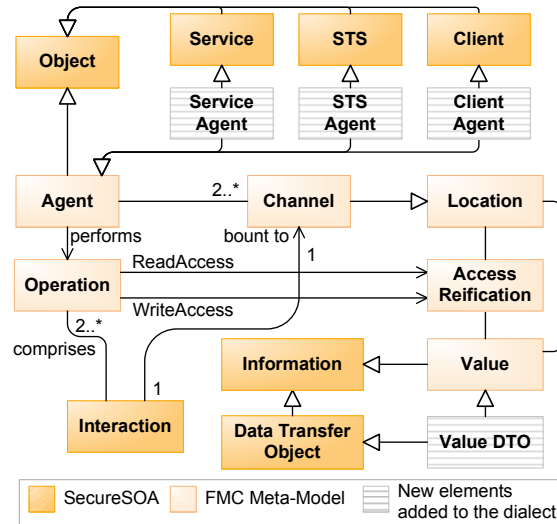


Figure 4.9: SecureSOA-based FMC Dialect

As aforementioned in section 4.1.3, the easiest way to perform the integration is to subclass elements of SecureSOA. *Object* is subclassed by *Agent*, while *Information* is subclassed by *Value*. However, there is no class in FMC that can be mapped to *Service*, *Client*, *STS* and *Data Transfer Object*. As described by our integration pattern 2 in section 4.1.3, these extension points can be mapped by adding new elements (c.f. grey coloured elements in Figure 4.9) to the dialect that subclass related elements in FMC and SecureSOA.

Finally, the SecureSOA class *Interaction* has to be mapped to FMC. Subclassing will not work as integration technique, since interaction is not just a channel in FMC. It is composed of a *Channel* in combination with an *Operation* that is performed on this channel. Therefore, we specified associations to perform the integration as defined by pattern 3 in section 4.1.3.

The notion of the classes in the meta-model of the dialect is provided by the concrete syntax of FMC and SecureSOA. However, a notion must be defined for the elements *Service Agent*, *STS Agent* and *Client Agent* that have been added to the dialect. Since these elements inherit from *FMC Agent*, their notion is based on the notion of this class. To indicate the agent's

type (*Service*, *Client*, or *STS*) we enhanced the concrete syntax with a notion for stereotypes as defined by UML.

4.4 SecureSOA Modelling Example

This section illustrates the usage of the FMC-based SecureSOA modelling dialect. We have added this dialect as a stencil set to the web-based modelling tool Oryx [DOW08]. Due to limitations in Oryx concerning the visualisation of edges, our stencil set represents a simplified version of FMC block diagrams. We visualise a channel between two actors as a plain arrow and omit the visualisation of a channel as a small circle.

Figure 4.10 illustrates a web shop scenario modelled with SecureSOA. This scenario implements an order process that handles the selection, payment and shipping of items bought in an online store. Therefore, the web application integrates three services: The Music Shop Service that manages the items of the store, the Money Transfer Service, which handles the payment step in the order process, and the Order Processing Service that initiates the shipping of the goods using the recipients address. Note that each agent indicates its type (*Client*, *Service* or *STS*) using stereotypes. The order and the shipping service are operated by external services providers and belong to isolated trust domains.

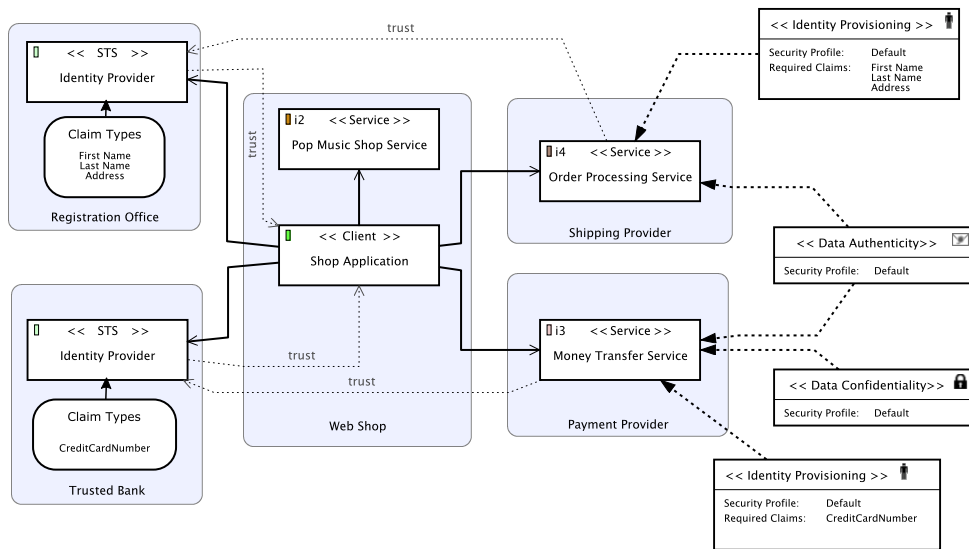


Figure 4.10: SecureSOA Web Shop Example

There are two additional organisations modelled in this scenario. The Trusted Bank and the Registration Office act as identity providers managing their user’s digital identities and provide a Security Token Service (STS). The trust relations from Shop Application to the security token services indicate that the users impersonated by the shop application have an account at each identity provider.

Furthermore, SecureSOA is used to annotate security intentions to various actors in this scenario. The security intention *Identity Provisioning* indicates that the Money Transfer Service needs payment information including a credit card number, while the shipping service requires the

provision of the user's name and address. Since these attributes must be asserted and provided by trustworthy sources, the payment service has established a trust relationship with the trusted bank, while the shipping service relies on information from the registration office. The security annotation *Supported Claim Types* is used to indicate that the identity providers are able to provide the sets of claims required by the services. To secure the exchanged information, the intentions *Data Authenticity* and *Data Confidentiality* are used as well in this example. Each intention relates to the default profile.

The entities in the scenario map to objects and interactions in our SOA interaction model. There are six objects in this scenario (2 x STS, 1 x Client, 3 x Service) that participate in five interactions with other objects. The objects interact by exchanging data transfer objects that are message requests and responses.

Modelling Element	Id
Shop Application	1
Music Shop Service	2
Order Processing Service	3
Money Transfer Service	4
Registration Office Identity Provider	5
Trusted bank Identity Provider	6

Table 4.5: Formalised Modelling Elements (Extract)

An identifier is assigned to each modelling element to formalise the model as listed in Table 4.5. Next to the elements modelled explicitly in Figure 4.10, data transfer objects must be represented. Therefore, two data transfer objects are created for each interaction representing request and response messages. Finally, we can formalise this scenario as follows:

<i>Object</i>	= {1, 2, 3, 4, 5, 6}
<i>Client</i>	= {1}
<i>Service</i>	= {2, 3, 4}
<i>STS</i>	= {5, 6}
<i>DataTransferObject</i>	= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
<i>SecurityIntention</i>	= {1, 2, 3, 4}
<i>ObjectIntention</i>	= {1, 2}
<i>InformationIntention</i>	= {3, 4}
<i>IdentityProvisioning</i>	= {1, 2}
<i>DataConfidentiality</i>	= {3}
<i>DataAuthenticity</i>	= {4}
<i>OOInteraction</i>	= {(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)}
<i>OOTrust</i>	= {(5, 1), (6, 1), (3, 5), (4, 6)}
<i>DOTarget</i>	= {(1, 2), (2, 1), (3, 3), (4, 1), (5, 4), (6, 1), (7, 5), (8, 1), (9, 6), (10, 1)}
<i>DOIssuer</i>	= {(1, 1), (2, 2), (3, 1), (4, 3), (5, 1), (6, 4), (7, 1), (8, 5), (9, 1), (10, 6)}
<i>SO</i>	= {(1, 3), (2, 4), (4, 3), (3, 4), (4, 4)}
<i>SD</i>	= {(4, 3), (4, 4), (3, 5), (3, 6), (4, 5), (4, 6)}

In the following, we will inspect the security intention *Data Confidentiality* (Id 3). Since it is attached to the service 'Money Transfer Service' (Id 4), this intention refers to all data transfer objects that are exchanged with this service as stated in Formula 4.1. Since $3 \in$

InformationIntention, $4 \in \text{Objects}$, and $3 \text{ SO } 4$, it must hold that

$$\begin{aligned} & \forall d \in \{d \mid d \text{ DO}_{\text{Target } 4} \vee d \text{ DO}_{\text{Issuer } 4}\} : (3 \text{ SD } d) \\ \Leftrightarrow & \forall d \in \{5, 6\} : (3 \text{ SD } d) \\ \Leftrightarrow & \text{true} \end{aligned}$$

The function $enc_{\text{SecureSOA}}$ was defined in Formula 4.3. From $3 \in \text{DataConfidentiality}$, $3 \text{ SD } 5$, and $3 \text{ SD } 6$ we obtain:

$$\begin{aligned} enc_{\text{SecureSOA}}(5) &= \text{true} \\ enc_{\text{SecureSOA}}(6) &= \text{true} \end{aligned}$$

According to the definition of the function $enc_{\text{SecureSOA}}$, the data transfer objects 5 and 6 are encrypted in the scope of our model.

Chapter 5

Security Configuration Patterns for SOA Security

SecureSOA facilitates the enhancement of system design modelling languages to enable the specification of security intentions and additional security-related aspects. As outlined in section 1.2, security intentions are transformed to security constraints that provide a detailed model of technical requirements (e.g. required protocols, algorithms, or credentials). However, a broad range of different solutions and configuration options exists that can be used to configure security constraints to satisfy a security intention. For example, confidentiality can be enforced at the transport layer or at the message layer and requires the provision of cryptographic keys. A suitable choice of appropriate security mechanisms depends on the overall architecture of the service-based system. Therefore, expertise knowledge is required to determine an appropriate strategy that specifies how to secure a service by enforcing an intention. To describe these strategies and their preconditions in a standardised way, we foster the usage of security patterns.

This chapter provides an introduction to security patterns that are used to represent security expert knowledge. To enable an automated application of security patterns in the scope of our model-driven approach, we introduce a formalised pattern structure denoted as security configuration pattern. A domain-specific language is provided to define the preconditions and the solution for this class of patterns. The syntax and formal semantics of this language are specified in this chapter.

5.1 Security Patterns – State of the Art

Security patterns are based on the idea of design patterns that has been introduced by Christopher Alexander in 1977: *'A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that pattern'* [AIS⁺77]. This approach has been applied to software development in 1987 by Cunningham and Beck [BC87]. In general, design patterns are defined in an informal way, usually in the natural language, to enable programmers and system designers to adapt the solution described by a pattern to their own specific problem. Patterns are described in documents that have a specific structure as listed in Table 5.1. As described in [MD96], the mandatory elements of a pattern are *Name*, *Context*,

Forces, Problem, and Solution. Security patterns have been introduced by Yoder and Barcalow [YB97] in 1997. Based on this work, various security patterns and pattern systems have been defined that refer to different phases in the development process. An overview about recent work is given in [YWM08] by Yoshioka et al.

Element	Description
Name	is a label that identifies the pattern and reflects the intention of this pattern.
Context	describes the environment before the application of this pattern.
Forces	are conditions that exist within the context. They affect the problem and might represent trade-offs or preconditions.
Problem	describes a problem that occurs within the context.
Solution	is a proven solution for the problem within the context.
Dependencies	might exist between patterns. As described by Zimmer [Zim95] there are three basic dependencies that might occur between patterns: <i>Usage</i> , <i>Refinement</i> or <i>Conflict</i> .

Table 5.1: Design Pattern Structure

Delessy and Fernandez defined several security patterns for SOA and Web Service security [DFLP07, FDLF06] that describe best practices and concepts such as *identity provider* and *identity federation*. Their work was enhanced by [ESP07] to describe authentication and authorisation infrastructures. These patterns provide an informal description, although parts of the pattern's solution are formalised using UML diagrams.

Microsoft published the book 'Web Service Security - Scenarios, Patterns, and Implementation Guidance' [SS05]. This book presents a catalogue of security patterns for Web Services and discusses the usage and preconditions for each pattern. In accordance with the design pattern structure, these patterns are described in an informal way.

The need of a formalisation of security patterns has been addressed by M. Schumacher [Sch03] by providing a classification. Schumacher proposed an approach to define a knowledge base using a security ontology that provides mappings between security concepts and security pattern elements. This knowledge base was used to implement a security pattern search engine that enables developers to retrieve security patterns that meet given queries. This approach offers a classification of elements in the pattern structure (e.g. the context or the problem), but is not suitable to enable an automated application of security patterns.

In fact, Yoshioka et al. describe the unification of security patterns and system models as important future work: '*For modelling security concerns, models such as UMLsec and SecureUML have been proposed, so we need to adapt such models to security patterns*'.

5.2 Web Service Security Configuration Patterns

Security patterns provide reusable expert knowledge that can be used by system designers. As outlined in the previous section, these patterns are represented in an informal way and do not support a model-driven transformation on the basis of an automated pattern application. Therefore, a formalisation of the pattern structure is required. Existing approaches that provide a formalisation – e.g. as proposed by Schumacher [Sch03] – offer a classification of the elements in the pattern structure (e.g. the context or the problem) and do not enable an automated reasoning and application of these patterns.

Moreover, the solutions provided by common security patterns describe a generic process that can be applied by programmers and security engineers to a concrete platform. The different steps in such a process could require the implementation of security-related mechanisms, the provisioning of configuration data such as certificates, or the configuration of the platform itself. However, patterns describing the implementation of security mechanisms and protocols are not applicable in the scope of our approach. Instead, security patterns are needed that describe the usage and configuration of security mechanisms and protocols concerning a particular security intention in order to enable an automated model-driven transformation. A platform that supports our model-driven approach has to support a basic set of security mechanisms and protocols that can be configured by applying our system of security patterns for SOA. Therefore, our approach requires a restricted type of security patterns. We denote this class of security patterns as *security configuration patterns*.

A security configuration pattern addresses a specific problem that is described by a security intention and facilitates the generation and modification of security constraints. The applicability of a pattern depends on the forces of this pattern that specify conditions in the scope of a SecureSOA model.

The structure of a security pattern is illustrated in Figure 5.1. A pattern is applied to an environment that is described by the **Context** of the pattern. The context is a relational SecureSOA model as introduced in chapter 4. Each security configuration pattern has a **Name** (a string) that identifies the pattern and addresses a **Problem**. As shown in Figure 5.1, the problem of a security configuration pattern is identified by a security intention.

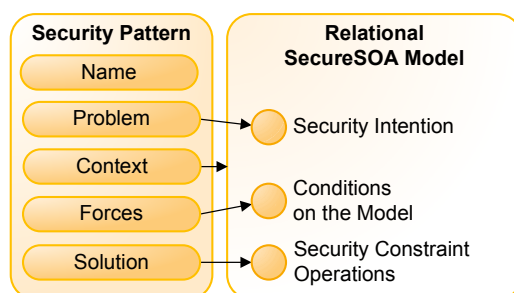


Figure 5.1: Security Configuration Pattern Structure

The **Forces** of our security configuration patterns determine the applicability of a pattern in a specific context by defining conditions over the entities and their relationships (e.g. a service must have a trust relationship to a client), while the **Solution** instantiates new entities and relationships in this model (e.g. security constraints). To specify the conditions and operations

on the data model in an accessible way, we provide a Domain Specific Language (DSL) that is used to specify the forces and the solution of a pattern. The syntax and the semantics of our security configuration pattern DSL will be introduced in the next section.

To sum up, a security configuration patterns defines a solution that provides a set of constraint operations. The applicability of a pattern is determined by the context and related forces. We can formalise security configuration patterns as follows:

Definition: Let $\Sigma_{Intensions} = \left\{ \begin{array}{l} UserAuthentication, IdentityProvisioning, \\ DataAuthenticity, DataConfidentiality \end{array} \right\}$ denote the security intentions specified by SecureSOA, \mathbb{F} and \mathbb{S} represent sets of all syntactically well-formed forces and solutions that can be formulated using our security configuration pattern DSL, and $M_{SecureSOA}$ denote the set of all SecureSOA models. A security configuration pattern is a tuple

$$(name, problem, forces, solution) \in \Sigma^* \times \Sigma_{Intensions} \times \mathbb{F} \times \mathbb{S} \quad (5.1)$$

We provide a system of multiple security patterns to address the security intentions defined by SecureSOA. As described by Schumacher [Sch03], a pattern system is a set of patterns that are defined using a consistent, uniform and precisely defined structure. Therefore, we can formalise a pattern system as a set $patternSystem \subseteq \Sigma^* \times \Sigma_{Intensions} \times \mathbb{F} \times \mathbb{S}$ with $patternSystem = \{pattern_1, \dots, pattern_{\#patternSystem}\}$. We denote the components of $pattern_i$ as $name_i$, $problem_i$, $forces_i$, and $solution_i$.

5.3 A Domain-specific Language for Security Configuration Patterns

To specify the forces and the solution of a pattern, a language is required that provides a syntax to state conditions and operations on the data model. Languages such as QVT [Gro08] or ATL [gro09a] have been specified in the scope of the Model-Driven Architectures (MDA) approach and provide an expressive and standardized syntax to define model transformations. However, the flexibility of these languages comes along with an increased complexity that would complicate the definition of security patterns. Therefore, we propagate the usage of a concise domain-specific language (DSL) that is defined specifically for the usage in our security pattern system.

A security configuration pattern DSL operates on a relational SecureSOA model. In the following, the operations and functions of our domain-specific pattern language will be introduced. Furthermore, the formal syntax and semantics of our domain-specific language will be described as well.

5.3.1 Security Configuration Pattern Operations and Functions

Our DSL provides a set of operations and functions that can be used to state the forces and the solution. These operations and functions operate on the pattern's context (sets and relations of SecureSOA). The fundamental sets in SecureSOA are *Client*, *Service*, and *STS* that represent actors. Therefore, these keywords are used in our domain-specific pattern language to represent the corresponding sets. In addition, the keyword *intention* is reserved to represent the concrete instance of a security intention a pattern is applied to. *intention* provides a set of properties

5.3. A DOMAIN-SPECIFIC LANGUAGE FOR SECURITY CONFIGURATION PATTERNS

that represent the parameters and the subject of the intention (e.g. *intention.subject* represents the subject the intentions refers to).

Besides security intentions, SecureSOA provides security annotations to express security-related capabilities of objects. As described in Table 5.2, we represent annotations as properties of the objects they are attached to.

Operators	Description
<i>(boolean) object.UserDirectory</i>	returns a boolean value that indicates whether an <i>UserDirectory</i> annotation is attached to this object.
<i>(set) object.SupportedClaimTypes</i>	returns the set of claims that are provided by <i>SupportedClaimTypes</i> annotations attached to this object.

Table 5.2: Object Properties Representing Annotations

Table 5.3 lists functions that evaluate interactions and trust relations between actors in the relational model. The parameters *object1* and *object2* represent a single object (expressed as a number) or multiple objects (expressed as a set of numbers).

Function	Description
<i>(boolean) Interaction(object1,object2)</i> <i>(boolean) Trust(object1,object2)</i>	determine whether there is a trust or interaction relationship between two objects. If <i>object1</i> or <i>object2</i> is a set of numbers, then it is analysed whether there is a relationship from/to all objects in this set
<i>(set) InteractionPath(object1,object2)</i> <i>(set) TrustPath(object1,object2)</i>	determine whether two objects or set of objects relate concerning the transitive closure of the respective relations. These functions return a set of numbers that represent the objects on the path.

Table 5.3: Functions to Evaluate Relations in SecureSOA

Moreover, boolean operators are provided by our DSL as illustrated in Table 5.4. These functions provide the foundation to express the condition of the forces in a pattern. Each function operates on boolean values. However, a set or an integer value can be passed to these functions as well. A set will evaluate as true if this set has elements, while an integer value will be evaluated as true if it is non-zero. For instance, $\{1,2\} \text{ AND } \{ \}$ is equal to *false*, while $0 \text{ IMPLIES } 10$ is *true*.

Operators	Description
<i>(boolean) value AND/OR value</i>	these operators correspond to boolean and/or assertions.
<i>(boolean) value IMPLIES value</i>	represents an implication.

Table 5.4: Boolean Operators

Functions operating on sets are listed in Table 5.5. The function *CONTAINS* checks a containment relationship (e.g. $\{1,2,3,4\} \text{ CONTAINS } (\{1,2\}) \equiv \text{true}$), while the *WHERE* function

returns a restricted set of elements. To illustrate this function, consider the following example: $\{1, 2, 3, 4\}.WHERE(\{it > 2\}) = \{3, 4\}$

Function	Description
<i>(boolean) set CONTAINS value</i>	checks whether a value is contained in a set. If this value is a set, then it is verified that each element of the second set is contained within the first set.
<i>(set) set INTERSECTION set</i>	calculates the intersection of two sets that is made up of the objects contained in both sets.
<i>(set) set.WHERE({condition})</i>	returns a set with elements of <i>set</i> that comply with the condition. The condition passed to this function has to evaluate as <i>true</i> or <i>false</i> . The variable labelled <i>it</i> can be used in the condition to represent the element in the set.

Table 5.5: Functions Operating on Sets

A sequence of operations can be applied to all elements of a set using the *FORALL* operation (see Table 5.6). The current element of the set the sequence is applied to is denoted as *it*.

Operation	Description
<i>FORALL (set){operation*}</i>	executes a list of operations for each element in a given set.

Table 5.6: ForAll Operation

Finally, specific operations are required that support the expression of a pattern's forces and solution. Table 5.7 lists the *ASSERT* operation that can be used to state the forces, while the operations that can be used to specify solutions are shown in Table 5.8.

Operation	Description
<i>ASSERT (value)</i>	is used to define the forces of a pattern. A pattern can be applied in a certain context, if all <i>ASSERT</i> operations evaluate as <i>true</i> . An <i>ASSERT</i> operation will return <i>true</i> , if the value passed to this function is equals true, is a non-empty set, or is a non-zero number.

Table 5.7: Forces Operation

5.3.2 Security Configuration Pattern DSL Formal Syntax

Our DSL can be used to specify the forces and the solution of a security configuration pattern. We provide the formal grammars $G_{Forces} = (N_{Forces}, \Sigma_{Forces}, P_{Forces}, \langle Forces \rangle)$ and $G_{Solution} = (N_{Solution}, \Sigma_{Solution}, P_{Solution}, \langle Solution \rangle)$ to specify the syntax of the forces and the solution. Both grammars share a common set of symbols and rules that are defined as follows:

Σ_{Forces} and $\Sigma_{Solution}$ denote finite sets of terminal symbols that appear in the string formed from the grammar. These sets are defined as

Operation	Description
<i>REQUIRE</i> (<i>securityGoal</i>)	this operation results in the creation of security constraints for a specific security goal that are returned by the solution.
<i>SET</i> (<i>key</i> , <i>value</i>)	assigns a value to specific properties (identified by a key) of the constraints that have been created during the execution of the solution.
<i>USE</i> (<i>key</i>)	similar to the <i>SET-operation</i> , this operation assigns a value to specific properties of the constraints. The value is resolved from the profiles as introduced in section 1.2.1.
<i>ENFORCE</i> (<i>intention</i>)	this operation can be used in the scope of a solution and indicates that a specific security intention must be enforced. A set of security constraints is returned that is created by applying security patterns matching to the intention.
<i>SCOPE</i> (<i>subject</i>){ <i>operation</i> *}	limits the application of SET and USE operations to constraints that refer to the specified subject.

Table 5.8: Solution Operations

$$\begin{aligned} \Sigma_{Forces} &= \Sigma_{IntentionProperties} \cup \Sigma_{Actors} \cup \Sigma_{Intentions} \cup \Sigma_{Annotations} \cup \\ &\quad \Sigma_{SecurityGoals} \cup \Sigma_{Functions} \cup \Sigma_{SecurityOntology} \cup \Sigma_{OperationsForces} \\ \Sigma_{Solution} &= \Sigma_{IntentionProperties} \cup \Sigma_{Actors} \cup \Sigma_{Intentions} \cup \Sigma_{Annotations} \cup \\ &\quad \Sigma_{SecurityGoals} \cup \Sigma_{Functions} \cup \Sigma_{SecurityOntology} \cup \Sigma_{OperationsSolution} \end{aligned}$$

The basic sets defining the terminal symbols are specified as follows:

$$\begin{aligned} \Sigma_{Token} &= \{ \cdot, \cdot, \{, \}, (,) \} \\ \Sigma_{IntentionProperties} &= \{ intention, subject, claims \} \\ \Sigma_{Actors} &= \{ Service, Client, STS \} \\ \Sigma_{Intentions} &= \{ UserAuthentication, IdentityProvisioning, \\ &\quad DataAuthenticity, DataConfidentiality \} \\ \Sigma_{Annotations} &= \{ UserDirectory, SupportedClaimTypes \} \\ \Sigma_{SecurityGoals} &= \{ Confidentiality, Integrity, Authentication \} \\ \Sigma_{OntologyConcepts} &\subseteq \Sigma^* \\ \Sigma_{Functions} &= \{ Interaction, Trust, InteractionPath, TrustPath, \\ &\quad AND, OR, CONTAINS, INTERSECTION, \\ &\quad IMPLIES, FORALL, WHERE \} \\ \Sigma_{OperationsForces} &= \{ ASSERT \} \\ \Sigma_{OperationsSolution} &= \{ REQUIRE, SET, USE, ENFORCE, SCOPE \} \end{aligned}$$

The terminal symbols in the set $\Sigma_{IntentionProperties}$ are used to construct references to properties of the intention the pattern is applied to, while Σ_{Actors} is used to specify symbols representing different types of actors. The elements in $\Sigma_{Intentions}$ and $\Sigma_{Annotations}$ symbolise the security intentions and annotations specified by SecureSOA and $\Sigma_{SecurityGoals}$ denotes security goals that identify required security constraints. $\Sigma_{OntologyConcepts}$ provides symbols that represent concepts

specified by a security ontology. A security ontology provides a set of concepts and relationships for a specific security configuration pattern system. Its structure will be introduced in section 5.4. Finally, we use the terminal symbols in the set $\Sigma_{Functions}$ to denote common functions and operations that can be used in the forces as well as in the solution. Operations that are specific for the forces or the solution are specified in the sets $\Sigma_{OperationsForces}$ and $\Sigma_{OperationsSolution}$.

The components N_{Forces} in G_{Forces} and $N_{Solution}$ in $G_{Solution}$ denote finite sets of nonterminal symbols that do not appear in strings formed from the grammars:

$$\begin{aligned} N &= \{ \langle Set \rangle, \langle Number \rangle, \langle Boolean \rangle, \langle AnyType \rangle, \langle NumberOrSet \rangle \} \\ N_{Forces} &= N \cup \{ \langle Forces \rangle, \langle OperationForces \rangle \} \\ N_{Solution} &= N \cup \{ \langle Solution \rangle, \langle OperationSolution \rangle, \langle Concepts \rangle, \\ &\quad \langle Intention \rangle, \langle SecurityGoal \rangle \} \end{aligned}$$

The symbols $\langle Forces \rangle$ respectively $\langle Solution \rangle$ of G_{Forces} and $G_{Solution}$ represent the start symbols. In addition, the formal grammars contain the finite sets $P_{Solution} \subseteq (N_{Solution})^* \times (\Sigma_{Solution} \cup N_{Solution})^*$ and $P_{Forces} \subseteq (N_{Forces})^* \times (\Sigma_{Forces} \cup N_{Forces})^*$ of production rules. In particular, $P_{Solution}$ and P_{Forces} contain the following basic set of production rules:

$$\begin{aligned} \langle Solution \rangle &\rightarrow \langle OperationSolution \rangle; \langle Solution \rangle \mid \epsilon \\ \langle Forces \rangle &\rightarrow \langle OperationForces \rangle; \langle Forces \rangle \mid \epsilon \end{aligned}$$

$P_{Solution}$ and P_{Forces} share a set of rules to construct basic functions and operations. These functions operate on basic data types that are denoted as $\langle Boolean \rangle$, $\langle Number \rangle$, and $\langle Set \rangle$. Since some functions accept different types of parameters, we define

$$\begin{aligned} \langle AnyType \rangle &\rightarrow \langle Boolean \rangle \mid \langle Number \rangle \mid \langle Set \rangle \\ \langle NumberOrSet \rangle &\rightarrow \langle Number \rangle \mid \langle Set \rangle \end{aligned}$$

The nonterminal symbols representing data types can be replaced by variables or functions returning the corresponding type. Therefore, the following production rules are defined for the different data types:

$$\begin{aligned} \langle Number \rangle &\rightarrow \textit{intention.subject} \\ &\quad \mid \textit{it} \\ \langle Set \rangle &\rightarrow \textit{intention.claims} \\ &\quad \mid \textit{STS} \mid \textit{Client} \mid \textit{Service} \\ &\quad \mid \langle Set \rangle . \textit{WHERE}(\langle Boolean \rangle) \\ &\quad \mid \langle Set \rangle \textit{ INTERSECTION } \langle Set \rangle \\ &\quad \mid \textit{InteractionPath}(\langle NumberOrSet \rangle, \langle NumberOrSet \rangle) \\ &\quad \mid \textit{TrustPath}(\langle NumberOrSet \rangle, \langle NumberOrSet \rangle) \\ &\quad \mid \langle Number \rangle . \textit{SupportedClaimTypes} \end{aligned}$$

$$\begin{aligned}
 \langle \text{Boolean} \rangle &\rightarrow \langle \text{AnyType} \rangle \text{ AND } \langle \text{AnyType} \rangle \mid \langle \text{AnyType} \rangle \text{ OR } \langle \text{AnyType} \rangle \\
 &\mid \langle \text{AnyType} \rangle \text{ IMPLIES } \langle \text{AnyType} \rangle \\
 &\mid \langle \text{Set} \rangle \text{ CONTAINS } \langle \text{NumberOrSet} \rangle \\
 &\mid \text{Interaction}(\langle \text{NumberOrSet} \rangle, \langle \text{NumberOrSet} \rangle) \\
 &\mid \text{Trust}(\langle \text{NumberOrSet} \rangle, \langle \text{NumberOrSet} \rangle) \\
 &\mid \langle \text{Number} \rangle . \text{UserDirectory}
 \end{aligned}$$

In addition, the following rules for security intentions and security goals are defined:

$$\begin{aligned}
 \langle \text{Intention} \rangle &\rightarrow \text{UserAuthentication} \mid \text{IdentityProvisioning} \\
 &\mid \text{DataConfidentiality} \mid \text{DataAuthenticity} \\
 \langle \text{SecurityGoal} \rangle &\rightarrow \text{Authentication} \mid \text{Confidentiality} \mid \text{Integrity}
 \end{aligned}$$

Finally, rules are defined for operations that can be used to specify the forces and the solutions:

$$\begin{aligned}
 \langle \text{OperationForces} \rangle &\rightarrow \text{ASSERT}(\langle \text{AnyType} \rangle) \\
 &\mid \text{FORALL}(\langle \text{Set} \rangle)\{\langle \text{Forces} \rangle\} \\
 \langle \text{OperationSolution} \rangle &\rightarrow \text{REQUIRE}(\langle \text{SecurityGoal} \rangle) \\
 &\mid \text{SET}(\langle \text{Concept} \rangle, \langle \text{AnyType} \rangle) \\
 &\mid \text{USE}(\langle \text{Concept} \rangle) \\
 &\mid \text{ENFORCE}(\langle \text{Intention} \rangle) \\
 &\mid \text{FORALL}(\langle \text{Set} \rangle)\{\langle \text{Solution} \rangle\} \\
 &\mid \text{SCOPE}(\langle \text{NumberOrSet} \rangle)\{\langle \text{Solution} \rangle\}
 \end{aligned}$$

The nonterminal symbol $\langle \text{Concept} \rangle$ represents concepts defined by an ontology. Therefore, it can be replaced using production rules that map this symbol to a symbol in $\Sigma_{\text{ontologyConcept}}$:

$$\langle \text{Concept} \rangle \rightarrow c, \text{ where } c \in \Sigma_{\text{ontologyConcept}}$$

5.3.3 Security Configuration Pattern DSL Formal Semantics

As outlined in section 5.2, a security configuration pattern operates on a relational SecureSOA model and is applied to a security intention contained in the intention set of this model. The forces determine the patterns applicability, while the solution results in the generation of our security constraint model. Therefore, the meaning of the programs *Forces* and *Solution* of a pattern stated in our security configuration pattern DSL can be described by their corresponding input-output-functions:

$$\begin{aligned}
 \llbracket \text{Forces} \rrbracket &: M_{\text{SecureSOA}} \times \text{Intention} \rightarrow \mathbb{B} \\
 \llbracket \text{Solution} \rrbracket &: M_{\text{SecureSOA}} \times \text{Intention} \rightarrow M_{\text{dim}}
 \end{aligned} \tag{5.2}$$

The forces of a pattern map the input to a boolean value that indicates the applicability of this

pattern, while the solution transforms a model instance of $M_{SecureSOA}$ into an instance of the domain-independent model of M_{dim} . To standardize the definition of the input-output-functions for operations, functions, and variables of our security configuration pattern DSL, we represent the input as a set of variables. Therefore, let $Vars$ be an union of all syntactic domains that are used to represent variables. Moreover, this set contains elements for each input variable of the function's *Forces* and *Solution*. In particular, each component name of a relational SecureSOA model m_{Secure} (e.g. *Service*, *Client*, $OO_{Interaction}$, ...) is an element of $Vars$ and a variable *Intention* is used to represent the security intention a pattern is applied to. We define a variable assignment function that is used to assign values to variables in $Vars$ as follows:

Definition: Let $\mathbb{D} = \mathbb{B} \cup \mathbb{N} \cup 2^{\mathbb{N}}$ be a value set. We define a function $\sigma : Vars \rightarrow \mathbb{D}$ as a variable assignment that maps a variable in $Vars$ to a value in \mathbb{D} .

The value set \mathbb{D} represent the basic values used in our approach that are either of data type boolean, number, or set. For instance, let $Service = \{1, 2, 3\}$ be a component of a relational SecureSOA model $m_{SecureSOA}$. Therefore, we obtain $Service \in Vars$ and $\sigma(Service) = \{1, 2, 3\}$. $\sigma(Service)$ returns the value of the variable labelled *Service*.

Using the variable assignment function we can restate the semantics of the program's *Forces* and *Solution* as

$$\begin{aligned} \llbracket Forces \rrbracket &: (Vars \rightarrow \mathbb{D}) \rightarrow \mathbb{B} \\ \llbracket Solution \rrbracket &: (Vars \rightarrow \mathbb{D}) \rightarrow M_{dim} \end{aligned} \tag{5.3}$$

5.3.3.1 Formal Semantics of Basic Data Types and Functions

We use the variable assignment function σ to define the semantics of basic data types and functions. The semantics of a boolean expression $b \in Vars$ is defined by a function $\llbracket b \rrbracket : (Vars \rightarrow \mathbb{D}) \rightarrow \mathbb{B}$ as follows:

$$\begin{aligned} \llbracket true \rrbracket (\sigma) &:: \equiv true \\ \llbracket false \rrbracket (\sigma) &:: \equiv false \\ \llbracket b \rrbracket (\sigma) &:: \equiv \sigma(b) \\ \llbracket \neg b \rrbracket (\sigma) &:: \equiv \neg \llbracket b \rrbracket (\sigma) \end{aligned} \tag{5.4}$$

In addition, we can define the semantics of the operators *AND* and *OR*. Let $x_1, x_2 \in Vars$ and $\tau : \mathbb{D} \rightarrow \mathbb{B}$ a mapping from the value set to a boolean value. The function τ is used to support the definition of these operators to enable the provision of numbers and sets as parameters in addition to boolean values.

$$\tau(x) ::= \begin{cases} x & \text{if } x \in \mathbb{B} \\ x > 0 & \text{if } x \in \mathbb{N} \\ x \neq \{\} & \text{if } x \in 2^{\mathbb{N}} \end{cases} \tag{5.5}$$

$$\begin{aligned} \llbracket x_1 \text{ AND } x_2 \rrbracket (\sigma) &:: \equiv \tau(\llbracket x_1 \rrbracket (\sigma)) \wedge \tau(\llbracket x_2 \rrbracket (\sigma)) \\ \llbracket x_1 \text{ OR } x_2 \rrbracket (\sigma) &:: \equiv \tau(\llbracket x_1 \rrbracket (\sigma)) \vee \tau(\llbracket x_2 \rrbracket (\sigma)) \end{aligned} \tag{5.6}$$

We use the function $\llbracket x \rrbracket : (Vars \rightarrow \mathbb{D}) \rightarrow \mathbb{N}$ to define the semantics of an integer expression

5.3. A DOMAIN-SPECIFIC LANGUAGE FOR SECURITY CONFIGURATION PATTERNS

$x \in Vars$ as follows:

$$\begin{aligned}
\llbracket x \rrbracket (\sigma) &:= \sigma(x) & (5.7) \\
\llbracket -x \rrbracket (\sigma) &:= -\llbracket x \rrbracket (\sigma) \\
\llbracket x_1 \square x_2 \rrbracket (\sigma) &:= \llbracket x_1 \rrbracket (\sigma) \square \llbracket x_2 \rrbracket (\sigma), \square : \mathbb{N}^2 \rightarrow \mathbb{B}, \square \in \{+, -, *\} \\
\llbracket x_1 \sqsubseteq x_2 \rrbracket (\sigma) &:= \llbracket x_1 \rrbracket (\sigma) \sqsubseteq \llbracket x_2 \rrbracket (\sigma), \sqsubseteq : \mathbb{N}^2 \rightarrow \mathbb{B}, \sqsubseteq \in \{=, <, \leq, \geq, >\}
\end{aligned}$$

The semantics of a set expression $s \in Vars$ are defined by a function $\llbracket s \rrbracket : (Vars \rightarrow \mathbb{D}) \rightarrow 2^{\mathbb{N}}$ as follows:

$$\llbracket s \rrbracket (\sigma) := \sigma(s) \quad (5.8)$$

In the next step, we define the semantics of the *INTERSECTION* operation. Let s_1 and s_2 be set expressions in $Vars$, then

$$\llbracket s_1 \text{ INTERSECTION } s_2 \rrbracket (\sigma) := \llbracket s_1 \rrbracket (\sigma) \cap \llbracket s_2 \rrbracket (\sigma) \quad (5.9)$$

Furthermore, we define the semantics of the *CONTAINS* function that evaluates a containment relationship. Let $s \in Vars$ be a set expression, while $x \in Vars$ is a set or number expression.

$$\llbracket s \text{ CONTAINS } x \rrbracket (\sigma) := \begin{cases} \llbracket x \rrbracket (\sigma) \in \llbracket s \rrbracket (\sigma) & \text{if } \llbracket x \rrbracket (\sigma) \in \mathbb{N} \vee \llbracket x \rrbracket (\sigma) \in \mathbb{B} \\ \llbracket x \rrbracket (\sigma) \subseteq \llbracket s \rrbracket (\sigma) & \text{if } \llbracket x \rrbracket (\sigma) \in 2^{\mathbb{N}} \end{cases} \quad (5.10)$$

Next, we define the semantic of the functions *Interaction* and *Trust* that are used to evaluate the relationships between actors. Let $x_1, x_2 \in Vars$ be a set or number expression. Since these functions accept numbers and sets as parameters, we define the auxiliary function $\varsigma : \mathbb{D} \rightarrow 2^{\mathbb{N}}$. This function expresses elements of the value set as a set to simplify the definition of the functions *Interaction* and *Trust*.

$$\varsigma(x) := \begin{cases} x & \text{if } x \in 2^{\mathbb{N}} \\ \{x\} & \text{if } x \in \mathbb{N} \vee x \in \mathbb{B} \end{cases} \quad (5.11)$$

$$\begin{aligned} \llbracket \text{Interaction}(x_1, x_2) \rrbracket (\sigma) &:= \forall e_1 \in \varsigma(\llbracket x_1 \rrbracket (\sigma)) \forall e_2 \in \varsigma(\llbracket x_2 \rrbracket (\sigma)): \\ &\quad (e_1, e_2) \in \sigma(OO_{\text{Interaction}}) \end{aligned} \quad (5.12)$$

$$\begin{aligned} \llbracket \text{Trust}(x_1, x_2) \rrbracket (\sigma) &:= \forall e_1 \in \varsigma(\llbracket x_1 \rrbracket (\sigma)) \forall e_2 \in \varsigma(\llbracket x_2 \rrbracket (\sigma)): \\ &\quad (e_1, e_2) \in \sigma(OO_{\text{Trust}}) \end{aligned} \quad (5.13)$$

To determine whether there is a transitive relation between two objects, we provide the functions *InteractionPath* and *TrustPath*. These functions return the set of all objects on paths from the first to the second object. If there is no transitive relation between two objects, an empty set is returned. Let σ be a variable assignment function, $\sigma(Rel) \subseteq \sigma(Object) \times \sigma(Object)$ a relation between Objects (e.g. $OO_{\text{Interaction}}$ or OO_{Trust}), and let x_1 and x_2 denote number expressions in $Vars$. We represent the paths from an object x_1 to x_2 connected via relations in $\sigma(Rel)$ as a graph (V_{Rel}, E_{Rel}) that is returned by a function $Paths_{Rel} : \mathbb{D} \times \mathbb{D} \times (Vars \rightarrow \mathbb{D}) \rightarrow 2^{\sigma(Object)} \times 2^{\sigma(Object) \times \sigma(Object)}$. Let $Paths_{Rel}(x_1, x_2, \sigma) = (V_{Rel}, E_{Rel})$ where $V_{Rel} \subseteq \sigma(Objects)$

and $E_{Rel} \subseteq \sigma(Rel)$. We define the sets of edges (E_{Rel}) and vertices (V_{Rel}) as follows. Let $x, y \in \sigma(Objects)$, then

$$(x, y) \in E_{Rel} \text{ iff there is a sequence } p_0, \dots, p_n \text{ with } p_i \in \sigma(Object) \text{ for all } 0 \leq i \leq n \quad (5.14)$$

$$\begin{aligned} \text{where } & (p_i, p_{i+1}) \in \sigma(Rel) \text{ for all } 0 \leq i < n \\ \text{and } & \llbracket x_1 \rrbracket(\sigma) = p_0, \llbracket x_2 \rrbracket(\sigma) = p_n \\ \text{and } & \exists z \in \{0, \dots, n-1\} : p_z = x \wedge p_{z+1} = y \end{aligned}$$

$$x \in V_{Rel} \Leftrightarrow \exists y \in \sigma(Objects) : (x, y) \in E_{Rel} \vee (y, x) \in E_{Rel} \quad (5.15)$$

Using the function $Paths_{Rel}$, we can specify the input-output function of the functions $InteractionPath$ and $TrustPath$.

$$\llbracket InteractionPath(x_1, x_2) \rrbracket(\sigma) := V_{Rel} \quad (5.16)$$

$$\begin{aligned} \text{with } Paths_{Rel}(x_1, x_2, \sigma) &= (V_{Rel}, E_{Rel}) \\ \text{and } Rel &= OO_{Interaction} \end{aligned}$$

$$\llbracket TrustPath(x_1, x_2) \rrbracket(\sigma) := V_{Rel} \quad (5.17)$$

$$\begin{aligned} \text{with } Paths_{Rel}(x_1, x_2, \sigma) &= (V_{Rel}, E_{Rel}) \\ \text{and } Rel &= OO_{Trust} \end{aligned}$$

Moreover, we have to specify the semantics of the expressions that are used to reference the properties of the security intention the pattern is applied to (e.g. *intention.subject*). As introduced in chapter 4, an intention subject is either the object that is assigned to an intention using the relation SO or a data transfer object associated with the relation SD . In addition, a security intention relates to an object that must enforce this intention (*intention.policySubject*) and the data transfer objects (*intention.DTOSubject*) that must be transformed due to the intention. If a subject represents a data transfer object, the properties *intention.subject.issuer* and *intention.subject.target* can be used to resolve the issuer and the target of this data transfer object. Claims are assigned to an intention using the relation IC . To simplify the following definitions, we use the auxiliary function $\varphi : \sigma(SecurityIntention) \rightarrow \mathbb{B}$ that evaluates whether an intention is of type *object intention*.

$$\varphi(i) := \begin{cases} true & \text{if } i \in \sigma(UserAuthentication) \cup \sigma(IdentityProvisioning) \\ false & \text{else} \end{cases} \quad (5.18)$$

Let $intention \in Vars$ be a number expression, then

$$\llbracket intention.subject \rrbracket(\sigma) := \begin{cases} o \text{ with } (\sigma(intention), o) \in \sigma(SO) \\ \quad \text{if } \varphi(\sigma(intention)) \\ d \text{ with } (\sigma(intention), d) \in \sigma(SD) \\ \quad \text{else} \end{cases} \quad (5.19)$$

$$\llbracket \text{intention.DTOSubject} \rrbracket (\sigma) := \begin{cases} d \text{ with } \exists o : \sigma(\text{intention}) \sigma(SO) o \wedge d DO_{\text{Target}} o \\ \text{and } \exists i : d DI i \\ \text{if } \varphi(\sigma(\text{intention})) \\ d \text{ with } (\sigma(\text{intention}), d) \in \sigma(SD) \\ \text{else} \end{cases} \quad (5.20)$$

$$\llbracket \text{intention.policySubject} \rrbracket (\sigma) := \begin{cases} o \text{ with } (\sigma(\text{intention}), o) \in \sigma(SO) \\ \text{if } \varphi(\sigma(\text{intention})) \\ o \text{ with } (\sigma(\text{intention}), o) \in \sigma(SD) \circ \sigma(DO_{\text{Target}}) \\ \text{else} \end{cases} \quad (5.21)$$

$$\llbracket \text{intention.subject.issuer} \rrbracket (\sigma) := o \text{ with } (\sigma(\text{intention}), o) \in \sigma(SD) \circ \sigma(DO_{\text{Issuer}}) \quad (5.22)$$

$$\llbracket \text{intention.subject.target} \rrbracket (\sigma) := o \text{ with } (\sigma(\text{intention}), o) \in \sigma(SD) \circ \sigma(DO_{\text{Target}}) \quad (5.23)$$

$$\llbracket \text{intention.claims} \rrbracket (\sigma) := \{c \mid (\sigma(\text{intention}), c) \in \sigma(IC)\} \quad (5.24)$$

Besides security intentions, SecureSOA provides security annotations. Let obj denote a number expression in $Vars$, then

$$\llbracket \text{obj.UserDirectory} \rrbracket (\sigma) := \exists a \in \text{UserDirectory} : (a, \llbracket \text{obj} \rrbracket) \in AO \quad (5.25)$$

$$\llbracket \text{obj.SupportedClaimTypes} \rrbracket (\sigma) := \{c \in \text{Claim} \mid \exists a \in \text{SupportedClaimTypes} : (a, \llbracket \text{obj} \rrbracket) \in AO \wedge (a, c) \in TC\} \quad (5.26)$$

The semantics of the syntactic expressions representing the actors in an SOA are defined as follows:

$$\begin{aligned} \llbracket \text{Client} \rrbracket (\sigma) &:= \sigma(\text{Client}) & (5.27) \\ \llbracket \text{Service} \rrbracket (\sigma) &:= \sigma(\text{Service}) \\ \llbracket \text{STS} \rrbracket (\sigma) &:= \sigma(\text{STS}) \\ \llbracket \text{RelyingParty} \rrbracket (\sigma) &:= \sigma(\text{STS}) \cup \sigma(\text{Service}) \end{aligned}$$

5.3.3.2 Operational Small Step Semantics

As described in the previous section, a pattern's forces and solution can be specified syntactically as a sequence of operations o_1, \dots, o_n . The processing of these operations can be represented as a sequence of execution steps.

Definition: Let $os_1 = (o_1, \dots, o_n)$ be a sequence of operations and σ_1 the variable assignment for os_1 . An execution step is a relation $os_i | \sigma_i \rightarrow os_{i+1} | \sigma_{i+1}$, where $os_i = (o_i, \dots, o_n)$ and $os_{i+1} = (o_{i+1}, \dots, o_n)$. os_{i+1} denotes the sequence of operations after the execution of operation o_i .

Since each execution step alters the assignment function σ_i due to the assignment of new values to variables in $Vars$, a variable assignment function σ_i is transformed to a function σ_{i+1} . We model the relationship between both functions using the operator \oplus that is used to replace the assignment of variable values:

Definition: Let $\sigma : Vars \rightarrow \mathbb{D}$ be a variable assignment function and $\{x_1, \dots, x_k\} \subseteq Vars$ a set of variables. $\sigma \oplus \{x_1 \rightarrow d_1, \dots, x_k \rightarrow d_k\}$ is a variable assignment function, where each variable x_i is assigned a value d_i with $(1 \leq i \leq k)$

Let $s \in Vars$ be a set expression. We use the function \oplus to define the semantics of the expression $s.where(\{condition\})$ that returns a set of elements that comply with the condition.

$$\llbracket s.where(\{condition\}) \rrbracket (\sigma) := \{e \in \llbracket s \rrbracket (\sigma) \mid \llbracket condition \rrbracket (\sigma \oplus \{it \rightarrow e\})\} \quad (5.28)$$

To define the semantics of an execution step $os_i | \sigma_i \rightarrow os_{i+1} | \sigma_{i+1}$, we have to specify os_{i+1} and σ_{i+1} in dependence on the first operation o_i in os_i .

The assignment operation is the simplest operation provided by our DSL. Let $x \in Vars$ and $o_i \equiv x := e$. The semantics of this operation can be defined as follows:

$$o_i; os_{i+1} \mid \sigma_i \rightarrow os_{i+1} | \sigma_{i+1} \text{ with } \sigma_{i+1} = \sigma_i \oplus \{x \rightarrow \llbracket e \rrbracket (\sigma_i)\} \quad (5.29)$$

Another basic operation is the $FORALL(s)\{os\}$ operation that can be used in the definition of the forces and the solution to repeat a set of operations os for each element in the set $\llbracket s \rrbracket = (s_1, \dots, s_{\#s})$. The current instance of the element is denoted as it and added to the set $Vars$. The semantics of an execution step with $o_i \equiv FORALL(s)\{os\}$ are defined as:

$$\begin{aligned} o_i; os_{i+1} \mid \sigma_i &\rightarrow it := s_1; os; \\ &\vdots \\ &it := s_{\#s}; os; os_{i+1} | \sigma_{i+1} \text{ with } \sigma_{i+1} = \sigma_i \end{aligned} \quad (5.30)$$

5.3.3.3 Specifying the Semantics of the Forces

The result of a pattern's forces depends on the evaluation of the operation $ASSERT(expr)$. The semantics of the execution step with $o_i \equiv ASSERT(expr)$ can be defined as follows:

$$\begin{aligned} o_i; os_{i+1} | \sigma_i &\rightarrow os_{i+1} | \sigma_{i+1} \text{ with} \\ &\sigma_{i+1} = \sigma_i \oplus \{ForcesResult \rightarrow \tau(\llbracket expr \rrbracket (\sigma_i)) \wedge \sigma_i(ForcesResult)\} \end{aligned} \quad (5.31)$$

$ForcesResult$ is a variable that is used to represent the result of a pattern's forces. The execution of an $ASSERT$ operation will set this variable to true if the expression passed to this operation evaluates as true and if $ForcesResult$ is not false due to a previous $ASSERT$ evaluation.

Let σ_1 be the initial variable assignment function generated from the pattern context, whereas $ForcesResult \rightarrow true \in \sigma_1$. Using the variable $ForcesResult$, we can specify the forces of a pattern as follows:

$$\llbracket Forces \rrbracket := \sigma_n(ForcesResult), \text{ with } Forces | \sigma_1 \rightarrow^* \epsilon | \sigma_n \quad (5.32)$$

5.3.3.4 Specifying the Semantics of the Solution

The operations in a pattern's solution create or modify security constraints. To define the semantics of a pattern's solution, we will start with the definition of the semantics of the operation

5.3. A DOMAIN-SPECIFIC LANGUAGE FOR SECURITY CONFIGURATION PATTERNS

$ENFORCE(intention)$. Let $P(\sigma_1)$ be the set of applicable patterns regarding a variable assignment function σ_1 :

$$P(\sigma_1) := \{j \in \{1, \dots, \#patternSystem\} \mid \llbracket Forces_j \rrbracket(\sigma_1) \wedge \sigma_1(intention) = SecurityIntention_j\} \quad (5.33)$$

$P(\sigma_1)$ contains all patterns whose intentions are equal to $intention$ and whose forces evaluate as true. This set might contain multiple security patterns that represent alternatives. Since the solutions of each alternative pattern might require the enforcement of additional security intentions, the application of security intentions result in a tree of applied security patterns. To enable the generation of security alternatives after the execution of the solution, it is necessary to construct the pattern application tree and to associate security constraints to this tree during the execution of the patterns solution. Therefore, we define a Pattern Application Tree (PAT) as a graph $PAT = (V_{PAT}, E_{PAT})$, where $V_{PAT} \subseteq \mathbb{N}$ and $E_{PAT} \subseteq \mathbb{N} \times \mathbb{N}$. Each element in V_{PAT} represents the application of a security pattern to fulfil a specific security intention. We associate the policy subject of the security intention with the corresponding node in the tree using the relation $PatO \subseteq V_{PAT} \times Objects$. This object represents the entity (e.g. a service) that requires the enforcement of the related security constraints. The relation $CPAT \subseteq SecurityConstraint \times V_{PAT}$ is used to relate security constraints to nodes in the pattern application tree.

An example is shown in Figure 5.2. The pattern application tree resulted from the application of four security patterns. The initial pattern v_1 required the enforcement of an additional security intention that triggered the application of a single pattern in v_2 . This pattern created the security constraint c_1 and required the enforcement of an additional security intention, too. In this case, two pattern were applied that created the security intentions c_2 and c_3 . The nodes v_1 and v_2 relate to object o_1 , while the pattern v_3 and v_4 refer to object o_2 .

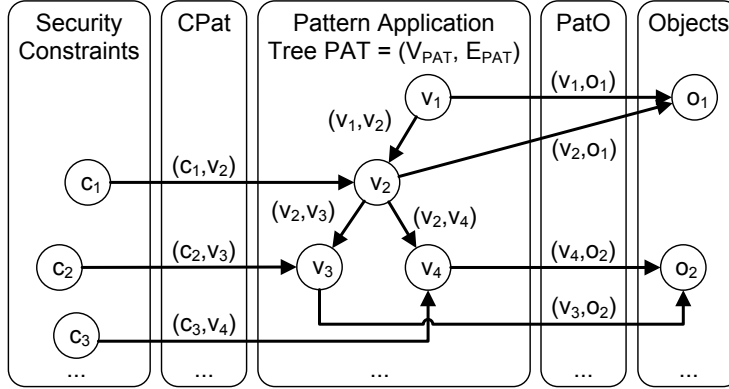


Figure 5.2: Pattern Application Tree Example

We use the variable $PAT \in Vars$ to denote the pattern application tree, while $PID \in Vars$ represents the current element in the pattern application tree during the execution of the solution. Let σ_i be a variable assignment function with $PAT \rightarrow (V_{PAT}, E_{PAT}) \in \sigma_i$, let $PID \rightarrow pid \in \sigma_i$ with $pid \in V_{PAT}$, and let $subj = \llbracket intention.policySubject \rrbracket(\sigma_i)$. The semantic of an execution step with $(o_i \equiv ENFORCE(intention))$ that applies additional security patterns for the specified security intention is defined in Formula 5.34.

The execution of this operation results in the insertion of the solutions of the applicable patterns

into the sequence of operations. In addition, the pattern application tree stored in σ_{i+1} is enhanced with new states for the additionally applied patterns that are linked to the policy subject using the relation $PatO$. Assignment operations are inserted into the sequence of operations to set the variable PID to the node in the pattern application tree that is associated with a pattern whose solution is executed next. After applying the solutions of all applicable patterns, PID is set back to its original value $\sigma_i(PID)$ and the remaining sequence of operations o_{i+1} ; will be executed. Altogether, the semantics of an *ASSERT* operation are defined as follows:

$$\begin{aligned}
 o_i; os_{i+1} | \sigma_i \rightarrow PID := \#V_{PAT} + 1; solution_{j_1}; & \quad (5.34) \\
 \vdots & \\
 PID := \#V_{PAT} + \#P(\sigma_i); solution_{j_{\#P(\sigma_i)}}; PID := pid; os_{i+1} | \sigma_{i+1} & \\
 \text{where } j_1, \dots, j_{\#P(\sigma_1)} \in P(\sigma_1), & \\
 \text{and } \sigma_{i+1} = \sigma_i \oplus \{PAT \rightarrow (V_{PAT} \cup \{\#V_{PAT} + 1, \dots, \#V_{PAT} + \#P(\sigma_i)\}, & \\
 E_{PAT} \cup \{(pid, \#V_{PAT} + 1), & \\
 \vdots & \\
 (pid, \#V_{PAT} + \#P(\sigma_i)\}), & \\
 PatO \rightarrow PatO \cup \{(\#V_{PAT} + 1, subj), & \\
 \vdots & \\
 (\#V_{PAT} + \#P(\sigma_i), subj)\}\} &
 \end{aligned}$$

The operation $REQUIRE(g)$ is another important operation that is used to generate new security constraints for a security goal $g \in Vars$. These constraints are stored in the constraint sets *AuthenticationConstraint*, *ConfidentialityConstraint*, and *IntegrityConstraint* of *Vars* and represent components of M_{dim} .

To enhance these constraint sets with a new constraint, we define a set of functions $v_X : (Vars \rightarrow \mathbb{D}) \rightarrow 2^{\mathbb{N}}$ that returns an enhanced set of security constraints as a variable assignment. Since the sets *AuthenticationConstraint*, *ConfidentialityConstraint*, and *IntegrityConstraint* are subsets of *SecurityConstraint*, the function v_X enhances these sets with $\#SecurityConstraint + 1$ as a new and unique constraint number. In addition, v_{SC} relates the new constraint to the data transfer object it is referring to.

$$v_{AC}(\sigma) = \{AuthenticationConstraint \rightarrow \sigma(AuthenticationConstraint) \cup \{\#\sigma(SecurityConstraint) + 1\}\} \quad (5.35)$$

$$v_{CC}(\sigma) = \{ConfidentialityConstraint \rightarrow \sigma(ConfidentialityConstraint) \cup \{\#\sigma(SecurityConstraint) + 1\}\} \quad (5.36)$$

$$v_{IC}(\sigma) = \{IntegrityConstraint \rightarrow \sigma(IntegrityConstraint) \cup \{\#\sigma(SecurityConstraint) + 1\}\} \quad (5.37)$$

$$\begin{aligned}
 v_{SC}(\sigma) = \{SecurityConstraint \rightarrow & \quad (5.38) \\
 \sigma(SecurityConstraint) \cup \{\#\sigma(SecurityConstraint) + 1\}, & \\
 CD \rightarrow & \\
 \sigma(CD) \cup \{\#\sigma(SecurityConstraint) + 1, \llbracket intention.DTOsubject \rrbracket\} &
 \end{aligned}$$

Let $n_{SC} := \#\sigma_i(SecurityConstraint)$. An execution step with ($o_i \equiv REQUIRE(g)$) alters the variable assignment function in dependency to the specified security goal g . In particular, a new

security constraint is added to the constraint sets and the relation $CPat$ is enhanced to associate the new constraint with the current node PID in the pattern application tree. Therefore, the semantics of an execution step are defined as follows:

$$\begin{aligned}
 & o_i; os_{i+1} | \sigma_i \rightarrow os_{i+1} | \sigma_{i+1} & (5.39) \\
 & \text{where } \sigma_{i+1} = \sigma_i \oplus v_{AC}(\sigma_i) \cup v_{SC}(\sigma_i) \cup \{CPAT \rightarrow \sigma_i(CPAT) \cup (n_{SC} + 1, \sigma_i(PID))\} \\
 & \quad \text{if } \sigma_i(g) = \textit{Authentication} \\
 & \text{and } \sigma_{i+1} = \sigma_i \oplus v_{CC}(\sigma_i) \cup v_{SC}(\sigma_i) \cup \{CPAT \rightarrow \sigma_i(CPAT) \cup (n_{SC} + 1, \sigma_i(PID))\} \\
 & \quad \text{if } \sigma_i(g) = \textit{Confidentiality} \\
 & \text{and } \sigma_{i+1} = \sigma_i \oplus v_{IC}(\sigma_i) \cup v_{SC}(\sigma_i) \cup \{CPAT \rightarrow \sigma_i(CPAT) \cup (n_{SC} + 1, \sigma_i(PID))\} \\
 & \quad \text{if } \sigma_i(g) = \textit{Integrity}
 \end{aligned}$$

Finally, we can specify the semantics of a pattern's solution as follows. The execution of a patterns solution results in the generation of security constraints that are described by components of our domain-independent model. Therefore, we can define the semantics of the solution as a function that transforms a model $m_{SecureSOA}$ into a model m_{dim} .

$$\begin{aligned}
 \llbracket \textit{Solution} \rrbracket & := m_{dim}, & (5.40) \\
 \text{with } m_{dim} & := (\sigma_n(\textit{Object}), \sigma_n(\textit{User}), \sigma_n(\textit{Service}), \sigma_n(\textit{STS}), \\
 & \sigma_n(\textit{Interaction}), \sigma_n(\textit{TrustRelationship}), \\
 & \sigma_n(\textit{DataTransferObject}), \textit{Policy}, \textit{PolicyAlternative}, \\
 & \sigma_n(\textit{SecurityConstraint}), \sigma_n(\textit{AuthenticationConstraint}), \\
 & \sigma_n(\textit{ConfidentialityConstraint}), \sigma_n(\textit{IntegrityConstraint}), \\
 & \sigma_n(\textit{Claims}), \sigma_n(\textit{OI}), \sigma_n(\textit{OT}), \sigma_n(\textit{DI}), \sigma_n(\textit{DD}), \sigma_n(\textit{DO}_{Issuer}), \\
 & \sigma_n(\textit{DO}_{Target}), \textit{PO}, \textit{AP}, \textit{CA}, \sigma_n(\textit{CD}), \sigma_n(\textit{CC})) \\
 \text{where } \textit{Solution} & | \sigma_1 \rightarrow^* \epsilon | \sigma_n
 \end{aligned}$$

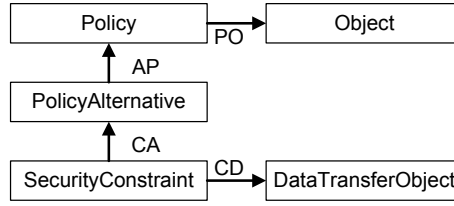
As mentioned above, the semantics of a solution are defined by a function that maps input variables representing a SecureSOA model to a domain-independent model m_{dim} . The components of this model describing the basic SOA entities and their relations (such as Object, Interaction, OI, ...) are provided by the SecureSOA input model $m_{SecureSOA}$ and are stored by the corresponding variables in $Vars$. The values of these variables are determined by the variable assignment function σ_n after the execution and termination of the solution in step n . In addition, this function provides the constraint sets *AuthenticationConstraint*, *ConfidentialityConstraint*, *IntegrityConstraint*, and *SecurityConstraint* containing the constraints that have been generated by executing the function *REQUIRE* as described above.

Next to the components stored in variables of $Vars$, m_{dim} contains the sets *PolicyAlternative*, *Policy*, *AC*, *AP*, and *PO* that are used to group constraints in security policies. In the following part, we define these sets on the basis of the pattern application tree that has been generated during the execution of this solution. Figure 5.3 illustrates these sets and their relationships.

A policy consists of multiple security alternatives that provide a set of security alternatives. Each policy is assigned to an object (e.g. a Service) using the relation *PO*. Since each Service or STS provides a single policy, we can identify a policy with the number of the object it is assigned to. Therefore, we define *Policy* and *PO* as follows:

$$\textit{Policy} := \sigma_n(\textit{Service}) \cup \sigma_n(\textit{STS}) \quad (5.41)$$

$$\textit{PO} := \{(x, x) \mid x \in \textit{Policy}\} \quad (5.42)$$


 Figure 5.3: Policy Sets and Relations in m_{dim}

Next, we have to specify the policy alternatives in dependency to the pattern application tree. As mentioned above, a node in this tree represents a pattern application step. Since nodes with the same parent node have been applied to the same security intention, the security constraints associated with each node form security alternatives. For example, the nodes v_3 and v_4 in Example 5.2 represent the application of alternative patterns. In particular, all security constraints that (1) relate to the same object and that (2) are on a path from the root node to a leaf of the tree form one security alternative.

The pattern application tree shown in Figure 5.2 provides three policy alternatives: $\{c_1\}$ on the path v_1, v_2 , $\{c_2\}$ on the path v_1, v_2, v_3 , and $\{c_3\}$ on the path v_1, v_2, v_4 . The first alternative provides security constraints for object o_1 , while the other security constraints relate to o_2 . These objects relate to the last node in the corresponding path and are denoted as the subject of the path.

To specify the security alternatives, we define the set *alternativePaths* that contains all these paths in the pattern application tree representing policy alternatives. Let $\sigma_n(PAT) = (V_{PAT}, E_{PAT})$, $V_{PAT} = \{v_0, \dots, v_n\}$ and $alternativePaths \subseteq V_{PAT}^*$ with

$$\begin{aligned}
 (p_1, \dots, p_n) \in alternativePaths &\Leftrightarrow (p_i, p_{i+1}) \in E_{PAT} \text{ for all } 1 \leq i \leq n & (5.43) \\
 &\wedge q_1 = v_0 \\
 &\wedge \# (q_1, \dots, q_m) \in V_{PAT}^* \\
 &\text{where } q_1 = p_n \\
 &\text{and } (q_j, q_{j+1}) \in E_{PAT} \text{ for all } 1 \leq j \leq m \\
 &\text{and } \exists z : (q_z, o) \in \sigma_n(PatO) \\
 &\quad \text{where } o \in \sigma_n(Objects) \\
 &\quad \text{with } (v_n, o) \in \sigma_n(PatO)
 \end{aligned}$$

alternativePaths contains all paths in the pattern application tree from the root node to a node p_n so that there is no path starting from p_n with nodes that relate to same object as p_n . For example, the path (v_1, v_2) in Figure 5.2 is such a path, since there is no path from v_2 to a leaf node of the tree (v_3 or v_4) with a node that relate to o_1 (v_3 and v_4 relate to o_2). However, (v_1) is not an element of *alternativePaths*, since there is a path (v_1, v_2, v_3) with v_2 relating to o_2 .

Since each element in *alternativePaths* represents a policy alternative, we define the constraint set *PolicyAlternative* as

$$PolicyAlternative := \{1, \dots, \#alternativePaths\} \quad (5.44)$$

In the next step, the security constraints in the constraint sets must be associated with the security alternatives using the relation *CA*. This relation is defined as follows. Let *alternativePaths* =

$\{path_1, \dots, path_n\}$ with $path_x = (p_1^x, \dots, p_k^x)$, then

$$(c, a) \in CA \Leftrightarrow \exists i : (c, p_i^a) \in \sigma_n(CPAT) \wedge (p_i^a, o) \in \sigma_n(PatO) \Rightarrow (p_k^a, o) \in \sigma_n(PatO) \quad (5.45)$$

A pair (c, a) with $c \in SecurityConstraints$ and $a \in SecurityAlternative$ is an element of CA , if c relates to a node in $path_a$ of *alternativePaths*. The second condition ensures that the constraint and its node relate to the subject of this path.

Finally, we have to associate each alternatives with a policy. A policy alternative belongs to a policy, if the subject of the corresponding alternative path is equal to the subject of the policy. Let $path_a = (p_1, \dots, p_k)$, then

$$(a, p) \in AP \Leftrightarrow \exists o \in Object : (p_k, o) \in \sigma_n(PatO) \wedge (p, o) \in PO \quad (5.46)$$

In this section, we have specified the formal semantics of our security configuration pattern DSL that has been used to define the semantics of a pattern's forces and solution. These definitions provide the foundation to enable a verification of the transformation process that will be presented in section 7.2. In particular, this formal verification proves that a specific security constraint has been created for a security intention. Although we prove the existence of security constraints, we do not consider the values of the constraint properties (e.g. *Security Mechanisms*). Therefore, the corresponding elements are not represented in our relational domain-independent model. Since the operations *SET* and *USE* modify the properties of security constraints, we did not specify the the semantics of these operations in the scope of this thesis. The specification of the semantics of these operations would require an enhancement of the relational models to express these properties.

5.4 Security Ontology

The security configuration pattern structure enables the specification of a solution that is based on a generic set of operations to generate and manipulate security constraints. In particular, the *USE* operation enables the declaration of a specific security concept (e.g. *MessageProtectionProtocol*) that must be resolved from the security profiles and assigned to properties of the security constraints. Since a formalisation of the relationships between the technologies listed in the profiles and the high level security concept referenced in the pattern is required, a security ontology has to be defined for each security pattern system that provides this knowledge. An ontology provides a formal, explicit specification of a shared conceptualisation and defines the vocabulary that can used to define security configuration patterns.

As an example, consider the security ontology that is shown in Figure 5.4. The operation *USE('MessageProtectionProtocol')* can be stated in a security pattern solution to select such a protocol from the profiles and to set this protocol in the confidentiality constraint. The relationship between the concept referenced by the *USE* operation, the profile entry and the security constraint property is described by a security ontology. As shown in Figure 5.4, the ontology reveals that 'WSS' is a 'MessageProtectionProtocol'. Since 'MessageProtectionProtocol' is a 'SecurityProtocol', the property 'SecurityProtocol' can be identified in the constraint. Finally, the policy type of 'WSS' described by the policy is set in the constraint.

Our security ontologies are based on four types of relations. The '*is-a*' relationship describes a sub classification of a concept, while '*contain*' describes a containment relationship. A policy

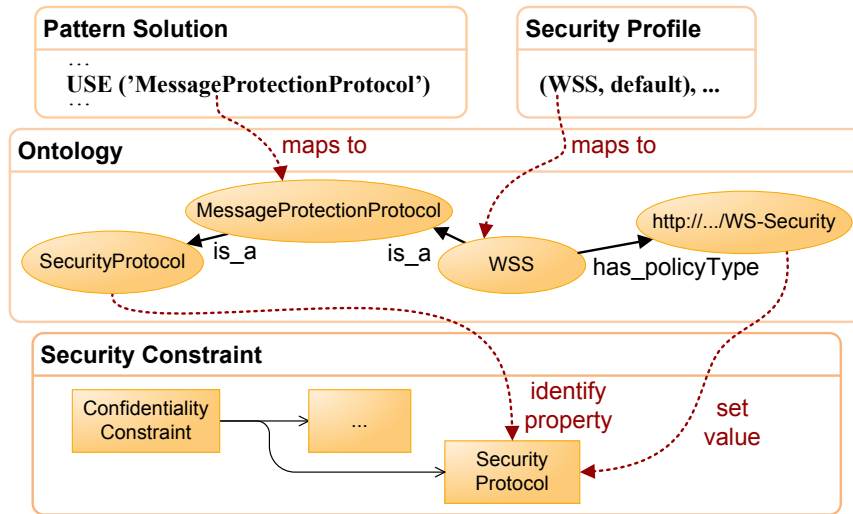


Figure 5.4: Ontology Usage Example

type can be assigned to concepts representing a concrete security mechanism using the relation *has_policyType*. Moreover, the *require* relation expresses that a concept requires the implementation of another security concept. For example, *MessageProtectionProtocol* requires a concept *MessageProtectionToken* that must be resolved from the profile and set in the constraint.

Chapter 6

A System of Security Configuration Patterns

The domain-specific language for security configuration patterns introduced in the previous chapter constitutes the foundation to define our pattern system that supports a transformation of security intentions. In this chapter, we introduce our patterns system that provides a set of solutions for the security intentions *User Authentication*, *Identity Provisioning*, *Data Confidentiality*, and *Data Authenticity*, which have been specified by SecureSOA in section 4.2. We can distinguish two classes of security intentions: The security intentions *User Authentication* and *Identity Provisioning* result in the generation of an authentication constraint, while *Data Confidentiality* and *Data Authenticity* support the generation of a data protection constraint. For each intention class, we introduce basic implementation schemes, a security ontology and security patterns that address related security intentions.

6.1 Patterns for Identification and Authentication

A broad range of security patterns have been defined to determine requirements and design decisions concerning the identification and authentication of users. For example, Schumacher provides a catalogue of identification and authentication patterns in [SFBH⁺06]. These patterns can be applied to evaluate security mechanisms and are useful to facilitate the creation of security profiles as described in section 4.2.1. However, patterns of this type do not consider the system architecture and are not applicable in our approach to configure the usage of security mechanisms and protocols at a technical layer. Security Patterns for service-based systems have been defined by Delessy [FDLP06, DFLP07]. These patterns support the selection of components (e.g. an identity provider), but they do not describe the usage and configuration of these components. This has been addressed by Microsoft in [SS05]. Their pattern catalogue defines basic patterns to configure the authentication in Web Service-based systems. In particular, the patterns *direct authentication* and *brokered authentication* are provided that enable selection of appropriate credential types. The pattern *brokered authentication* configures a decentralised authentication on the basis of a security token service.

The web service security patterns introduced by Microsoft provide the basis for our pattern catalogue [QMT⁺10], which has been developed and published in the scope of our participation in the TeleTrust SOA Security working group. This catalogue provides informal patterns and describes a general approach to map security requirements to security mechanisms in dependency to basic use cases. The concepts that have been used to define the identification and authentication patterns in this catalogue provide the foundation to define a pattern system based on our security configuration pattern DSL. These implementation schemes and related security configuration patterns are presented in this section.

6.1.1 Basic Implementation Schemes

The specification of requirements concerning the authentication of users and the provisioning of identity information is based on our digital identity model introduced in section 3.1.2. Users are registered at one or more identity providers, which manage their digital identities in an account. We consider an identity provider as a participant that is able to broker identity information to relying parties. In the scope of the Web Service specifications, an identity provider is a security token service that implements WS-Trust. However, we use this term in a wider sense. As described in section 4.2.1.2, SecureSOA provides the annotation *User Directory* that can be attached to objects. A user directory enables services and clients to authenticate users. Since clients and services can broker authentication decisions and identity information to composed services, we consider them as identity providers as well. Authentication decisions and identity information are conveyed in credentials that are issued by identity providers. The receiver of a credential has to rely on the conveyed claims on the basis of a trust relationship to the sender.

Security configuration patterns for identification and authentication generate authentication constraints for all entities (Service or STS) in a service-based system that are needed to assert, convert and provide identity information to a relying party. Security constraints state requirements concerning the types and the properties of exchanged credentials. In general, we can distinguish two types of credentials: *User credentials* issued by the subject itself ($issuer = subject$) that are used by a subject to make claims about himself (e.g. Username/Password token) and *issued credentials* that are asserted by a third party ($issuer \neq subject$) such as an issued SAML token.

Therefore, we can derive two basic authentication schemes as shown in Figure 6.1. The *subject authentication* schema (cf. Figure 6.1(a)) states that a *user credential* must be required by an identity provider to enable the authentication of registered users. The trust relationships between identity provider and users indicate that these users are registered at this identity provider.

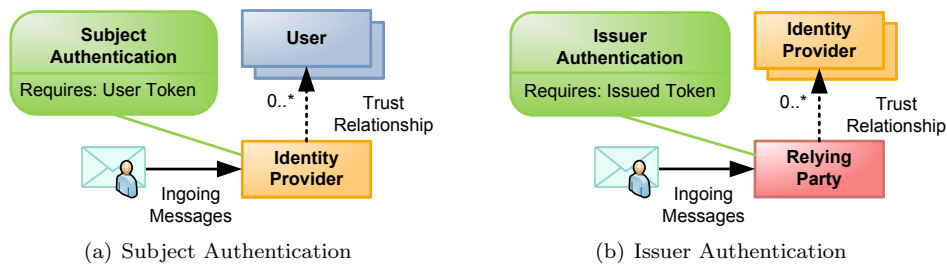


Figure 6.1: Basic Authentication Schemes

On the other hand, services and identity providers that do not manage the identities of users

themselves, have to rely on the authentication and provisioning of identity information performed by trusted identity providers. Therefore, these relying parties have to require an issued token as described by the *issuer authentication* schema illustrated in Figure 6.1(b). To ensure the trustworthiness of the claims conveyed in the credential, the relying party has to authenticate the issuing identity provider.

In chapter 2.3 four identity management models have been introduced. While the isolated identity management model is based on subject authentication, the *brokered authentication* schema is applicable to the other models. As illustrated in Figure 6.2, this schema is based on the combined use of subject and issuer authentication. The user authentication is performed by an identity provider that has a trust relationship to the user, while the service performs an issuer authentication. There is a trust path between this identity provider and the service that might involve additional identity providers that perform an issuer authentication to convert the credential.

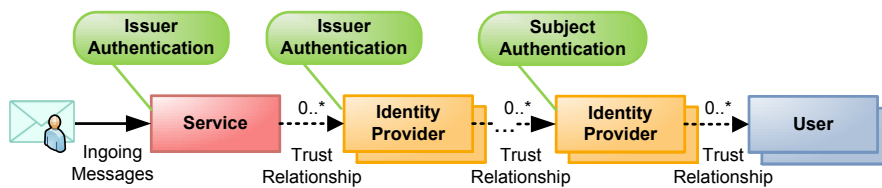


Figure 6.2: Brokered Authentication

Figure 6.2 illustrates the brokered authentication schema, which can be used to configure entities in the identity management models. The centralised identity management model performs a brokered authentication with a single centralised identity provider, while the open identity management models might involve additional identity providers. In the scope of the federated identity management, a brokered authentication is performed based on multiple identity providers, since a federation is based on the association of independent administrative domains.

6.1.2 Security Ontology

The ontology for identity management in SOA provides a classification of credentials and relates these classes (ontology concepts) to concrete technologies. For each ontology concept that can be required in a WS-SecurityPolicy, the corresponding WS-SecurityPolicy type is specified. As mentioned in the section above, we distinguish credentials used to perform an issuer authentication from credentials that enable a direct authentication. These types of credentials are named *IssuedToken* and *UserToken* as shown in Figure 6.3 and refer to the parent concepts *credential* using an 'is-a'-relationship. These concepts are further subdivided in concepts representing credential types. The *UserToken* is a superior concept of *UserNameToken* and *SelfSignedCertificate*. Since these concepts denote credential types that can be required in a WS-Policy, they are assigned to a policy type.

The concept *IssuedToken* is subdivided into *STSIssuedToken* and *ClientIssuedToken*, since WS-SecurityPolicy distinguishes between two classes of issued credentials. An *STSIssuedToken* type must be required by a policy, if a token should be issued by an STS. The usage of the related policy type enables the Web Service framework used by the client to retrieve the security token in an automated manner. On the other hand, requiring a specific *ClientIssuedToken* such as SAML enables the client logic to control the creation and provisioning of this token. Both concepts rely on the usage of an issued credential such as *SAML*.

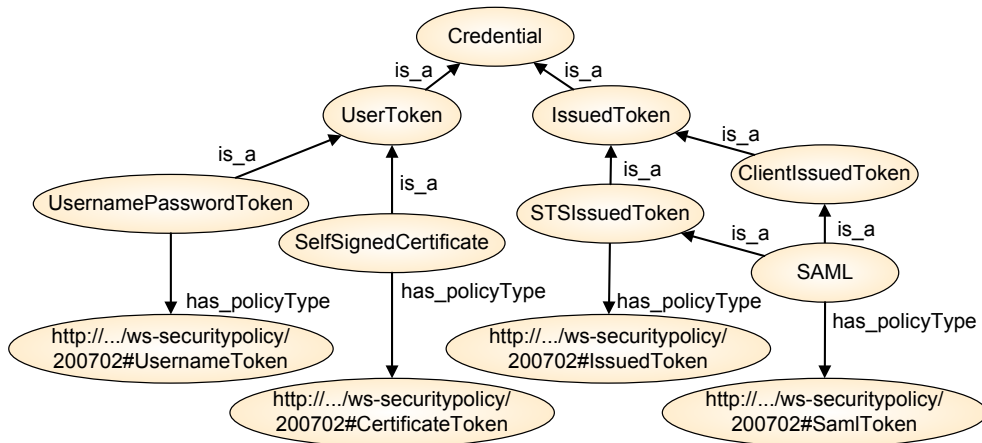


Figure 6.3: Identity Management Ontology

6.1.3 Pattern Definitions

The authentication schemes introduced above provide the foundation to define our system of security configuration patterns for authentication and identity provisioning. A security intention can be fulfilled by one or more patterns, while each pattern might require the enforcement of additional security intentions. These dependencies result in a graph structure that is shown in Figure 6.4.

The *User Authentication* intention is enforced by two security configuration patterns: The pattern *Client Issuer Authentication* enables the authentication of clients that are able to broker authentication decisions, while the pattern *Brokered Authentication* enables the integration of security token services to perform the user authentication. The *Brokered Authentication* pattern states that the security intention *Authentication* must be enforced for each entity on trust paths to clients. The *Authentication* intention is defined in the scope of our pattern system in addition to the intentions provided by SecureSOA and enables the application of subsidiary patterns. In particular, this intention is addressed by the patterns *STS Issuer Authentication* and *Subject Authentication* that implement the authentication schemes *issuer authentication* and *subject authentication* introduced in section 6.1.1.

There are two patterns defined that configure the provisioning of identity information (intention *Identity Provisioning*). Similar to the pattern *Brokered Authentication*, *Brokered Identity Provisioning* enforces the intention *Authentication* for each entity on the trust path. In addition, the claims provided by the *Identity Provisioning* intention are set as a requirement in the resulting constraints. The pattern *Direct Identity Provisioning* configures the requirements for a service that is connected to clients controlling and managing the identity information required by the service (e.g. a web frontend that enables users to enter required information such as name and address).

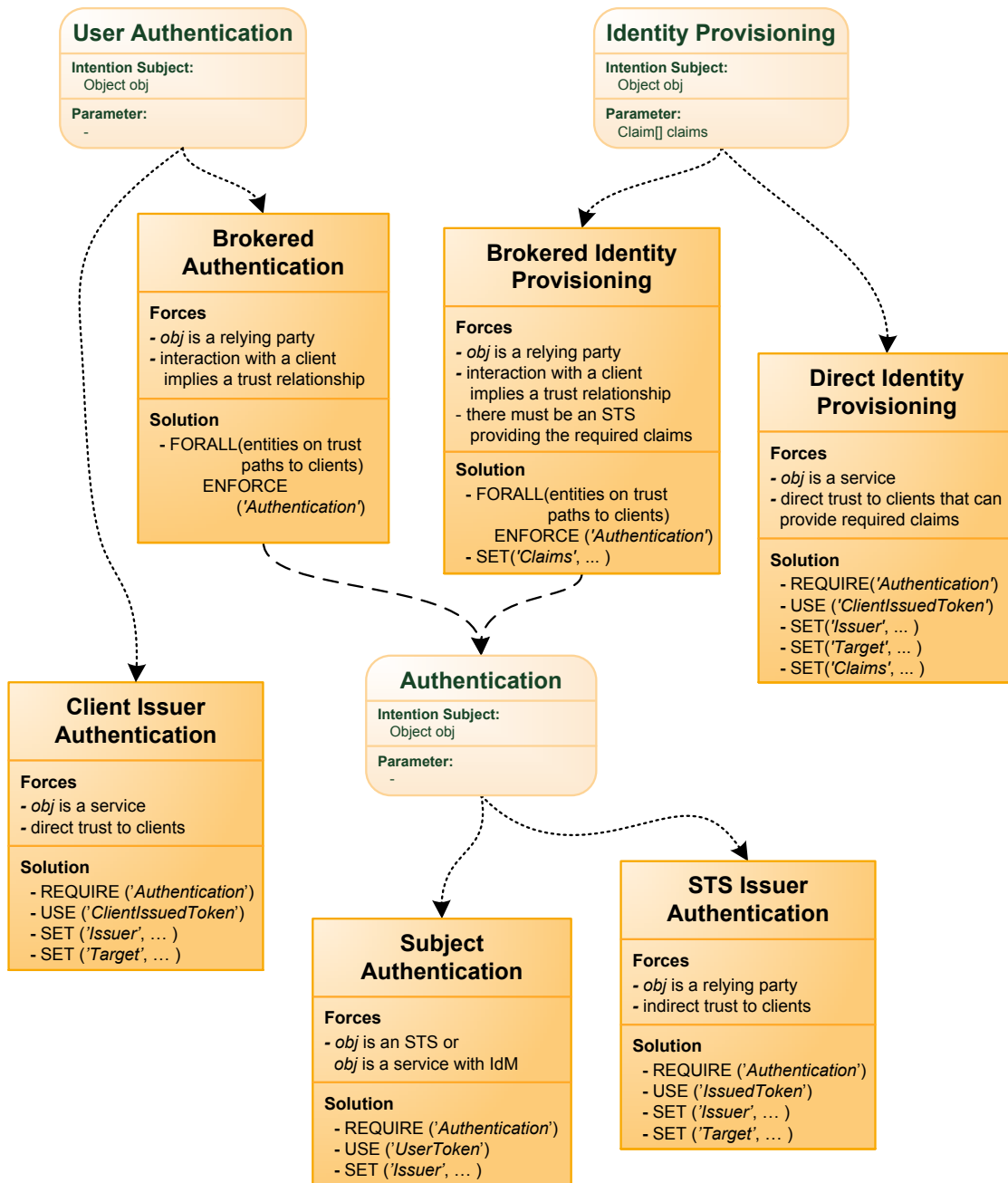


Figure 6.4: Pattern System for Identification and Authentication

The pattern that are shown in Figure 6.4 are defined as follows:

6.1.3.1 Pattern 'Brokered Authentication'

This pattern configures the authentication requirements for all participants (service and identity providers) that collaborate to perform the authentication of users.

Name:	Brokered Authentication
Problem:	User Authentication
Forces:	<ol style="list-style-type: none"> 1 ASSERT (RelyingParty CONTAINS intention.subject) 2 FORALL (Clients) { 3 ASSERT (InteractionPath(it, intention.subject) IMPLIES 4 TrustPath(intention.subject, it))
Solution:	<ol style="list-style-type: none"> 1 FORALL (TrustPath(intention.subject, Clients)) { 2 ENFORCE('Authentication')}

Table 6.1: Pattern 'Brokered Authentication'

The **forces** state in line 1 (Table 6.1, Forces) that a pattern is applicable, if the subject of the intention is a relying party (service or identity provider). In addition, trust relations must be established to all clients that invoke this service (as stated in lines 2-4 of the forces). This condition ensures that all user that invoke a service using a client are either managed by the service itself or by an trusted identity provider that is able to perform the authentication and to assert the authentication decision.

Figure 6.5 illustrates a simple examples that connects a client *C1* with a service *S1*. Since an indirect trust relationship is established over the identity providers *IP1* and *IP2*, the pattern *Brokered Authentication* can be applied at the service *S1*.

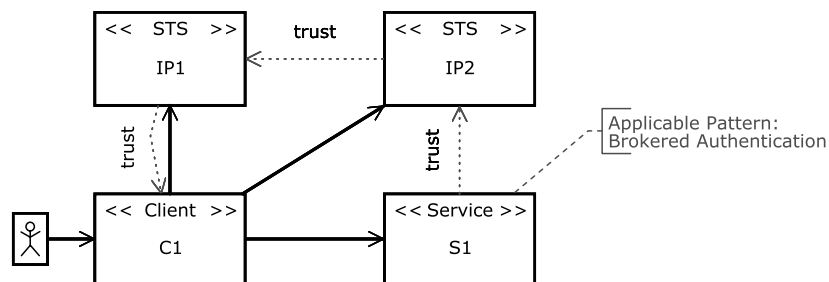


Figure 6.5: Brokered Authentication Example

The **solution** of this pattern requires the enforcement of the security intention *Authentication* for all objects on trust paths to clients. This includes the service the pattern is applied to as well as all identity providers on the trust paths from this service to clients. In Figure 6.5, the authentication intention would be required for the service *S1* and the identity providers *IP1* and *IP2*.

6.1.3.2 Pattern 'Subject Authentication'

This pattern is used to configure a direct authentication of users that are registered at a service or STS and implements the subject authentication schema introduced in section 6.1.1.

Name:	Subject Authentication
Problem:	Authentication
Forces:	<ol style="list-style-type: none"> 1 ASSERT (Trust(intention.subject, Client)) 2 ASSERT (3 (STS CONTAINS intention.subject) OR 4 (Service CONTAINS intention.subject AND 5 intention.subject.UserDirectory))
Solution:	<ol style="list-style-type: none"> 1 REQUIRE ('Authentication') 2 USE ('UserToken') 3 SET ('Issuer', Clients.where({Trust(intention.subject, it)}))

Table 6.2: Pattern 'Subject Authentication'

The **forces** state in line 1 (Table 6.2, Forces) that there must be direct trust relationship to one or more clients. A subject authentication scheme can be used to authenticate the users that are impersonated by clients, if these digital identities are managed by the subject of the intention. This condition is expressed in lines 2-5 of the forces. The subject must be an STS or a service that is attached to a user directory. An associated user directory indicates that a service is capable to perform the authentication of users.

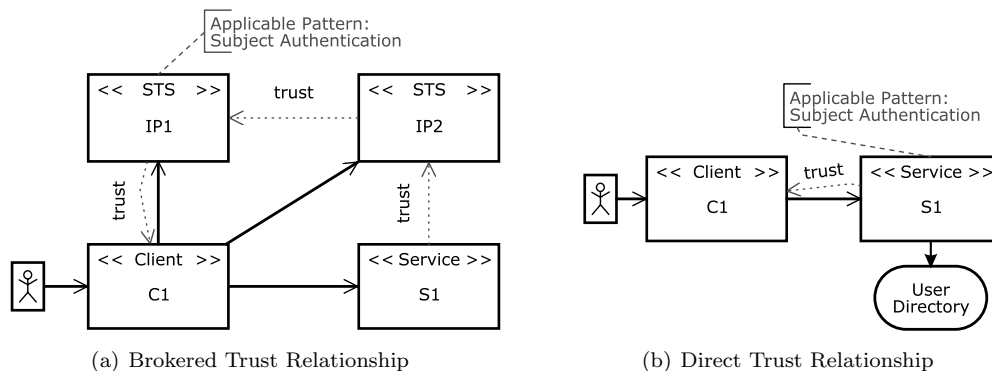


Figure 6.6: Subject Authentication Examples

The applicability of this pattern is illustrated in Figure 6.6. In use case 6.6(a), a service has established a trust path over two identity providers to the client. The pattern *Subject Authentication* is applicable to the identity provider IP1 that is connected to the client, since this identity provider manages the users' identities and is able to perform a subject authentication. Figure 6.6(b) shows a simple example with a single service that performs the identity management itself. Due to the forces described above, the *Subject Authentication* pattern can be applied in this case.

The **solution** generates an authentication constraint in line 1 (Table 6.2, Solution), requires the

usage of an *UserToken* in line 2, and assigns the set of trusted parties to the issuer property of the authentication constraint in line 3. The *UserToken* concept is resolved from the profiles using the ontology introduced in section 6.1.2.

6.1.3.3 Pattern 'STS Issuer Authentication'

This pattern implements the issuer authentication schema at a relying party. It is used to configure the authentication of identity providers that are able to assert identity information of users.

Name:	STS Issuer Authentication
Problem:	Authentication
Forces:	<ol style="list-style-type: none"> 1 ASSERT (RelyingParty CONTAINS intention.subject)) 2 ASSERT (TrustPath (intention.subject, STS))
Solution:	<ol style="list-style-type: none"> 1 REQUIRE ('Authentication') 2 USE ('IssuedToken') 3 SET ('Issuer', STS.where({Trust(intention.subject, it)})) 4 SET ('Target', RelyingParty.where({Trust(it,intention.subject)}))

Table 6.3: Pattern 'STS Issuer Authentication'

The **forces** assert in line 1 (Table 6.3, Forces) that the subject of a intention is a relying party (STS or Service). Since this pattern configures the authentication of security token services, an STS must exist that is trusted by the subject. Therefore, line 2 requires that a trust relation to an STS must exist.

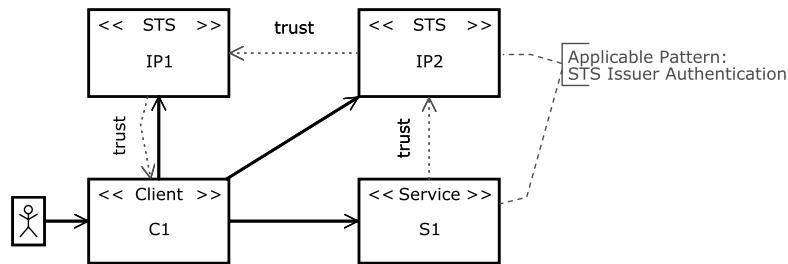


Figure 6.7: STS Issuer Authentication Example

The use case shown in Figure 6.7 defines a trust path from a service to a client across two security token services. The *STS Issuer Authentication* pattern is applicable to the service S1 and the identity provider IP2.

The **solution** generates an authentication constraint in line 1 (Table 6.3, Solution), requires the usage of an *IssuedToken* in line 2, and assigns the set of trusted parties to the issuer property of the authentication constraint in line 3 and the set of relying parties trusting the subject to the target property of this constraint in line 4. The *IssuedToken* concept is resolved from the profiles using the ontology introduced in section 6.1.2.

6.1.3.4 Pattern 'Client Issuer Authentication'

This pattern is used to configure a service to rely on an authentication that is performed by a client (e.g. a Web platform). This client acts as an identity provider and is able to assert the identity information of users. Hence, an issuer authentication schema is implemented by this pattern.

Name:	Client Issuer Authentication
Problem:	User Authentication
Forces:	<ol style="list-style-type: none"> 1 ASSERT (Service CONTAINS intention.subject) 2 ASSERT (Trust(intention.subject, Client.where ({it.UserDirectory })))
Solution:	<ol style="list-style-type: none"> 1 REQUIRE ('Authentication') 2 USE ('ClientIssuedToken') 3 SET ('Issuer', Client.where({Trust(intention.subject, it)})) 4 SET ('Target', RelyingParty.where({Trust(it,intention.subject)}))

Table 6.4: Pattern 'Client Issuer Authentication'

The **forces** assert in line 1 (Table 6.4, Forces) that the subject of an intention is a service. Line 2 requires a trust relationship to a client that is attached to a user directory.

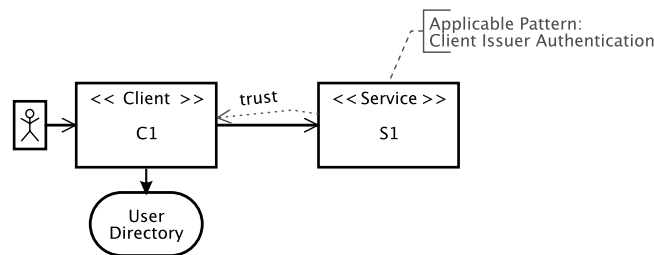


Figure 6.8: Client Issuer Authentication Example

Figure 6.8 illustrates an example that specifies a direct trust path from a service to a client, which is attached to a user directory. Therefore, the pattern *Client Issuer Authentication* is applicable to the service *S1*.

The **solution** generates an authentication constraint in line 1 (Table 6.4, Solution), requires the usage of a *ClientIssuedToken* in line 2, and assigns the set of trusted parties to the issuer property of the authentication constraint in line 3 and the set of relying parties trusting the subject to the target property of this constraint in line 4. The *ClientIssuedToken* concept is resolved from the profiles using the ontology introduced in section 6.1.2.

6.1.3.5 Pattern 'Brokered Identity Provisioning'

This pattern configures the provisioning of identity information issued by an identity provider to a service - even across multiple identity providers. The forces and the solution of this pattern are defined similarly to the *Brokered Authentication* pattern.

Name:	Brokered Identity Provisioning
Problem:	Identity Provisioning
Forces:	<pre> 1 ASSERT (RelyingParty CONTAINS intention.subject) 2 FORALL (Clients) { 3 ASSERT (4 InteractionPath(it, intention.subject) IMPLIES(5 TrustPath(intention.subject, it) AND 6 (STS.where{it.SupportedClaimTypes CONTAINS intention.Claims } INTERSECTION 7 TrustPath(intention.subject, it)) 8)} </pre>
Solution:	<pre> 1 FORALL(TrustPath(intention.subject, Clients)){ 2 ENFORCE ('Authentication')} 3 SCOPE (intention.subject){ 4 SET ('Claims', intention.Claims)} </pre>

Table 6.5: Pattern 'Brokered Identity Provisioning'

In addition to the forces defined in the *Brokered Authentication* pattern, identity providers managing the required claims must be included in all trust relations to clients that invoke this service. This condition is stated in lines 2-8 of the forces (Table 6.5) and ensures that there is an identity provider included in each trust chain that is able to assert the required identity information.

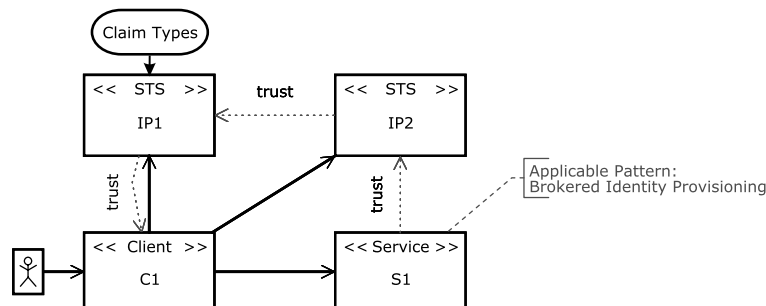


Figure 6.9: Brokered Identity Provisioning

Figure 6.9 illustrates an examples that connects a client *C1* to a service *S1* and two Identity Providers (*IP1* and *IP2*). Since an indirect trust relationship is established across the identity providers and the identity provider *IP1* is on the trust path providing the required set of claims, the pattern *Brokered Identity Provisioning* can be applied at the service *S1*.

Similar to the *Brokered Authentication* pattern, the **solution** of this pattern requires the enforcement of the security intention *Authentication* for all objects on trust paths from the subject of the *Brokered Identity Provisioning* pattern to its clients in lines 1-2 (Table 6.5, Solution). This includes the service this intention is attached to as well as all identity providers on trust paths to clients. In addition, this pattern sets the claims in the authentication constraints created for the subject of this intention in lines 3 and 4.

6.1.3.6 Pattern 'Direct Identity Provisioning'

This pattern configures the provision of identity information issued by a client to a service. For example, a web application could manage its user without using an identity provider. Composed services can be invoked using a SAML token to convey required identity information.

Name:	Direct Identity Provisioning
Problem:	Identity Provisioning
Forces:	<ol style="list-style-type: none"> 1 ASSERT (Service CONTAINS intention.subject) 2 ASSERT (Trust(intention.subject, 3 Client.where({it.SupportedClaimTypes CONTAINS (intention.Claims)})))
Solution:	<ol style="list-style-type: none"> 1 REQUIRE ('Authentication') 2 USE ('ClientIssuedToken') 3 SET ('Issuer', Client.where({Trust(intention.subject, it)})) 4 SET ('Target', RelyingParty.where({Trust(it, intention.subject)})) 5 SCOPE (intention.subject){ 6 SET ('Claims', intention.Claims)}

Table 6.6: Pattern 'Direct Identity Provisioning'

Line 1 of the **forces** in Table 6.6 requires the subject of the pattern's intention to be a service, while line 2 states that a direct trust relationship must be established between the subject and a client, which is able to provide the required claims.

In Figure 6.10, a client and a service are illustrated that are connected by a direct trust relationship. Therefore, the *Direct Identity Provisioning* pattern is applicable to the service *S1*.

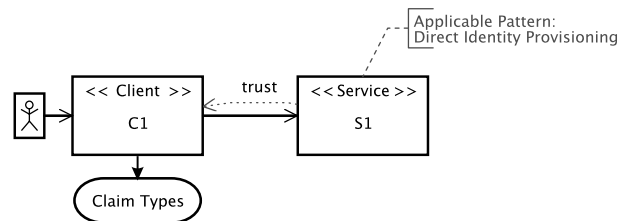


Figure 6.10: Direct Identity Provisioning

The **solution** generates an authentication constraint in line 1 (Table 6.6, Solution), requires the usage of a *Client Issued Token* in line 2, and assigns the set of trusted parties to the issuer property of the authentication constraint in line 3 and the set of relying parties trusting the subject to the target property of this constraint in line 4. Finally, this pattern sets the claims in this authentication constraints.

6.2 Patterns for Data Protection

In [SNL05] Steel et al. provide a catalogue of basic security implementation patterns for J2EE systems. These patterns enable the implementation of components and modules that are used to enforce security. Examples of security patterns are *Message Inspector*, *Assertion Builder*, and *Secure Service Proxy*. In addition, the pattern *Secure Pipe* is defined that configures the usage of these modules. This pattern describes a secure channel to prevent eavesdropping and information tampering caused by man-in-the-middle attacks. Web Service Security Patterns for WSE 3.0 have been introduced by Microsoft in [SS05]. A pattern *Data Confidentiality* is described providing a strategy for message-based encryption.

These patterns have been consolidated in the pattern catalogue published by the TeleTrusT SOA Security working group [QMT⁺10]. The basic concepts of these patterns provide the foundation to define a pattern system for data protection which is based on our security configuration pattern DSL. These implementation schemes and related security configuration patterns are presented in this section.

6.2.1 Basic Implementation Schemes

To ensure the integrity and confidentiality of exchanged information, the application of digital signatures and encryption mechanisms has to be required by a service. The application of these protection mechanisms can be performed at the transport or at the message layer. Figure 6.11 illustrates the protection of information at the transport layer. A secure channel is established between a sender and a receiver, e.g. using SSL.

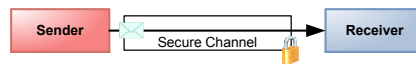


Figure 6.11: Transport-based Security

Another possibility is to apply security mechanisms at the message layer as shown in Figure 6.12. For example, WS-Security can be used to secure ingoing and outgoing messages. Since protection mechanisms are applied to the exchanged messages itself, information conveyed in these messages is protected in rest, processing and transit - even if these messages are transferred over intermediaries.

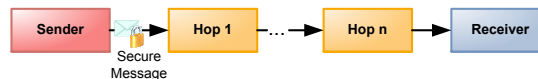


Figure 6.12: Message-based Security

6.2.2 Security Ontology

The concepts that are required to implement data protection are *SecurityProtocol*, *ProtectionToken*, and *AlgorithmType* as illustrated in Figure 6.13 and in Figure 6.14.

SecurityProtocol identifies a data protection protocol, while *ProtectionToken* denotes the type of credential used by the security protocol to secure data. These concepts are illustrated in

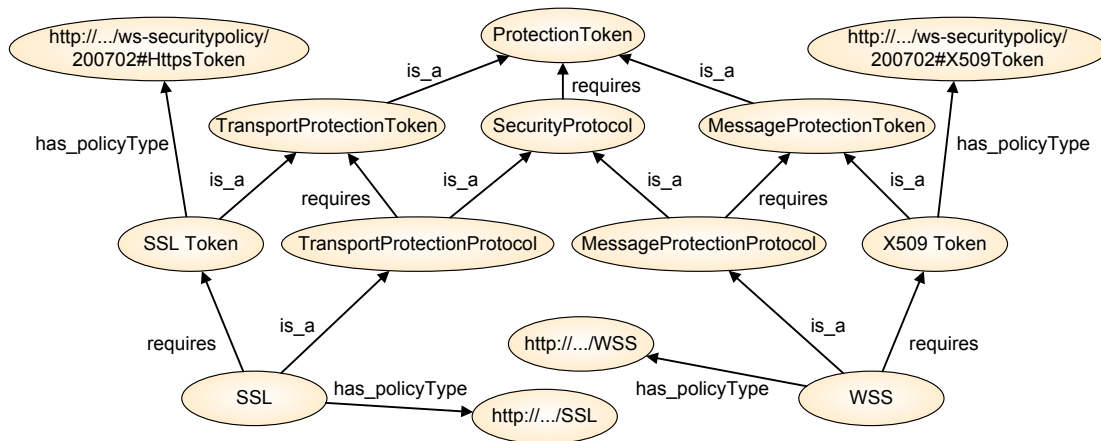


Figure 6.13: Data Protection Ontology

the ontology shown in Figure 6.13. Two subsidiary concepts are defined for *SecurityProtocol* describing message-based data protection (*MessageProtectionProtocol*) and transport-based data protection (*TransportProtectionProtocol*). These concepts are subclassed by *SSL* and *WSS* that represent specific technologies. In addition, there is a protection token concept defined for each protection protocol concept, since a specific type of security token is required by each protocol. In particular, an *SSL Token* is required by *SSL*, while *X509 Token* is required by *WSS*. Each of these concepts represents a technology that can be codified in WS-SecurityPolicy. Therefore, a policy type is assigned to each concept.

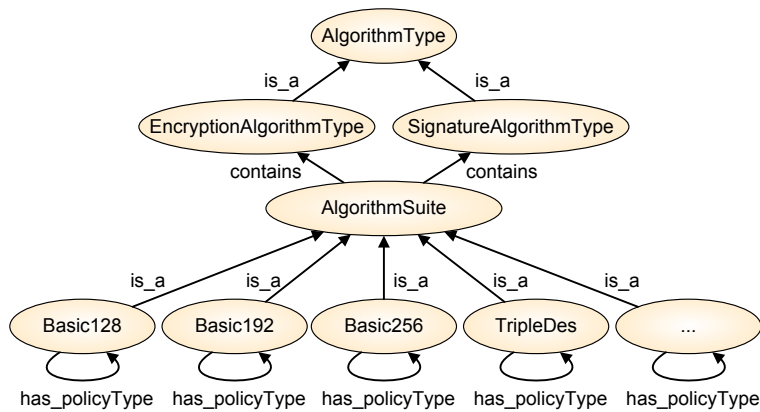


Figure 6.14: Security Mechanism Ontology

The concept *AlgorithmType* is defined in the ontology shown in Figure 6.14. There are two types of algorithms required for our approach: encryption algorithms (*EncryptionAlgorithmType*) and signature algorithms (*SignatureAlgorithmType*). WS-SecurityPolicy is based on the concept of algorithm suites that combine different types of algorithms. Therefore, an algorithm suite has to be specified in a policy instead of single algorithm types. Our ontology provides concepts for all algorithm suites that are supported by our model driven approach. Each concept is a subclass of the concept *AlgorithmSuite*. For example, if an *EncryptionAlgorithm* is required by a pattern

and the algorithm suite *Basic256* is specified in the profile, then the ontology will reveal that this requirement can be fulfilled by the *Basic256* policy type.

6.2.3 Pattern Definitions

The pattern system for data protection is shown in Figure 6.15. The security intentions '*Data Confidentiality*' and '*Data Authenticity*' are specified by SecureSOA and are addressed by the security patterns '*Secrecy*' and '*Authenticity*'. These patterns require the enforcement of the intention '*Information Protection*' to select the data protection schema introduced above. The transport-based protection schema is addressed by the pattern *Secure Pipe*, while the message-based protection schema is implemented by the *Message Protection* pattern.

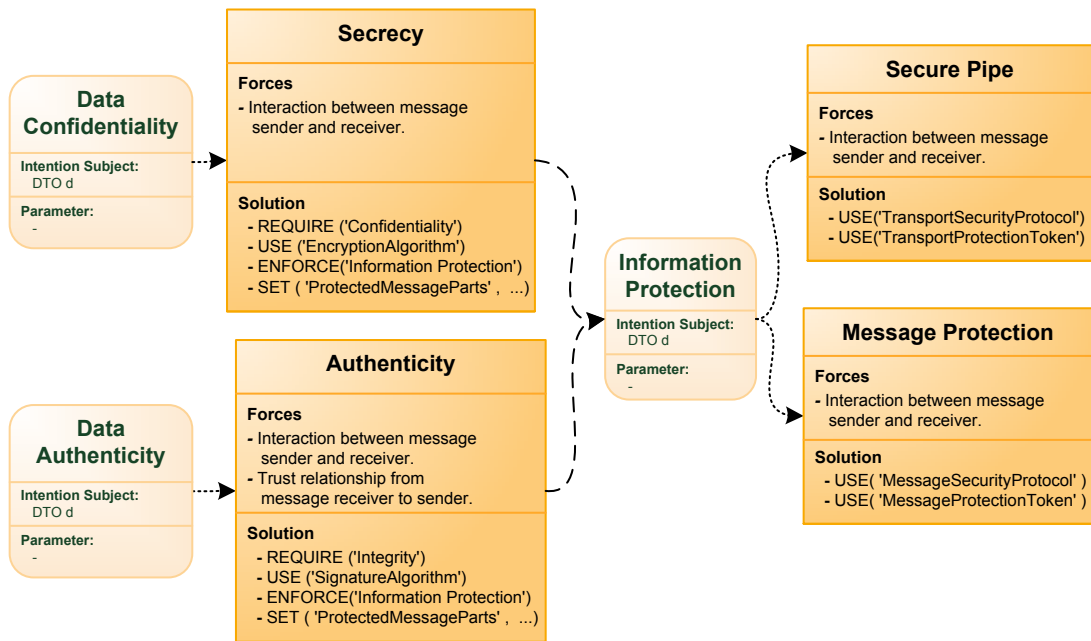


Figure 6.15: Pattern System Data Protection

The subject of the data protection intentions (*Data Confidentiality* and *Data Authenticity*) is a data transfer object as introduced in section 4.2.1.1. If a data protection intention is attached to a service, then the subject of this intention relates to the data transfer objects representing request and response messages. The patterns for *Data Protection* are defined in the following sections.

6.2.3.1 Pattern 'Secrecy'

This pattern implements the protection of exchanged information in terms of confidentiality by selecting appropriate security mechanisms and protocols.

Name:	Secrecy
Problem:	Data Confidentiality
Forces:	<ol style="list-style-type: none"> 1 ASSERT(InteractionPath(intention.subject.issuer, intention.subject.target)) 2
Solution:	<ol style="list-style-type: none"> 1 REQUIRE('Confidentiality') 2 USE('EncryptionAlgorithm') 3 ENFORCE('Information Protection') 4 SET('ProtectedMessageParts', intention.subject)

Table 6.7: Pattern 'Secrecy'

The **forces** require that an interaction path must exist between the sender and the receiver of a data transfer object. Examples are illustrated in Figure 6.16. A data transfer object representing credit card information is sent from a client C1 to a service S1. In use case 6.16(b), the message conveying this information is sent across a proxy. Since there are interaction paths in both use cases, this pattern is applicable.

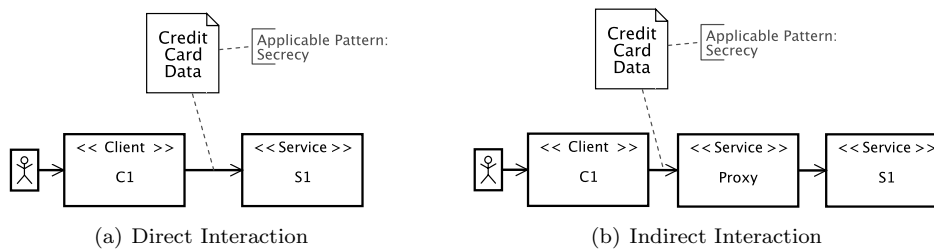


Figure 6.16: Secrecy Examples

The **solution** listed in Table 6.7 creates a confidentiality constraint in line 1 and assigns an encryption algorithm that is resolved from the profile. In line 3, the *Information Protection* intention is enforced to select the protocol according to the data protection schemes introduced in section 6.2.1. Finally, the data transfer object identified by the subject of the intention is assigned to the *ProtectedMessagePart* property of the authentication constraint. This property indicates that this data transfer object must be protected.

6.2.3.2 Pattern 'Authenticity'

This pattern ensures the authenticity of exchanged information by selecting appropriate security mechanisms and protocols.

Name:	Authenticity
Problem:	Data Authenticity
Forces:	<ol style="list-style-type: none"> 1 ASSERT (2 InteractionPath(intention.subject.issuer,intention.subject.target) AND 3 TrustPath(intention.subject.target,intention.subject.issuer))
Solution:	<ol style="list-style-type: none"> 1 REQUIRE('Integrity') 2 USE ('SignatureAlgorithm') 3 ENFORCE('Information Protection') 4 SET('ProtectedMessageParts',intention.subject)

Table 6.8: Pattern 'Authenticity'

The **forces** listed in Table 6.8 require that an interaction path must exist between the sender and the receiver of a data transfer object. In addition, a trust relationship must exist to ensure that the receiver is able to verify the identity of the sender. Examples are illustrated in Figure 6.17. These use cases are based on the secrecy use cases presented in Figure 6.16 and are enhanced with trust relationships. Due to the interactions and trust relations, the pattern *Authenticity* is applicable to the data transfer objects.

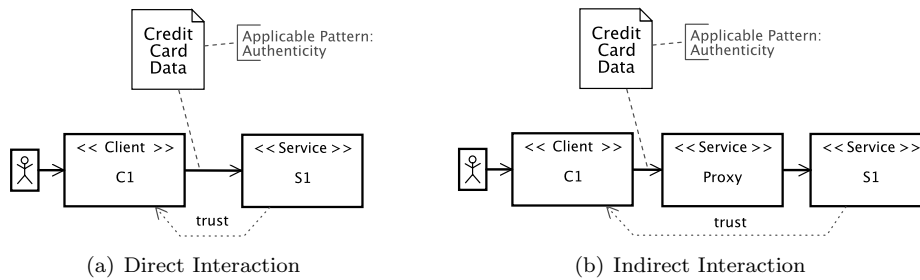


Figure 6.17: Authenticity Examples

The **solution** (specified in Table 6.8) creates an integrity constraint and sets a signature algorithm that is resolved from the profiles. In line 3, the *Information Protection* intention is enforced to select the protection protocol according to the data protection schemes introduced above. Finally, the subject of the pattern intention is assigned to the *ProtectedMessagePart* property. This property indicates that the data transfer object identified by the subject must be protected.

6.2.3.3 Pattern 'Secure Pipe'

This security configuration pattern implements a data protection at the transport layer.

Name:	Secure Pipe
Problem:	Information Protection
Forces:	1 ASSERT (Interaction(intention.subject.issuer, intention.subject.target))
Solution:	1 USE('TransportSecurityProtocol') 2 USE('TransportProtectionToken')

Table 6.9: Pattern 'SecurePipe'

The **forces** stated in Table 6.9 require a direct interaction between a message sender and a receiver. A message delivery across services acting as proxies is not allowed. An example use case is shown in Figure 6.18. The *Secure Pipe* pattern is applicable to the data transfer object, since there is a direct interaction.

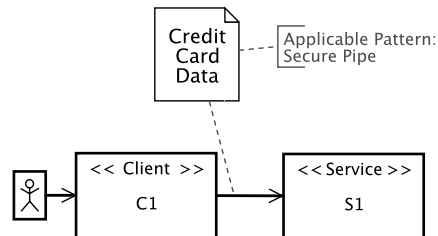


Figure 6.18: Secure Pipe Example

The **solution** of this pattern selects a transport security protocol and a transport protection token from the profiles. These properties are set in all security constraints that have been generated by the patterns that invoked the *Information Protection* intention.

6.2.3.4 Pattern 'Message Protection'

This security configuration pattern configures a data protection at the message layer.

Name:	Message Protection
Problem:	Information Protection
Forces:	1 ASSERT(InteractionPath(intention.subject.issuer,intention.subject.target))
Solution:	1 USE('MessageSecurityProtocol') 2 USE('MessageProtectionToken')

Table 6.10: Pattern 'Message Protection'

The **forces** listed in Table 6.10 require an interaction path between a message sender and a receiver. Messages can be exchanged directly or across services acting as proxies. An example use case is illustrated in Figure 6.19. Due to the forces of the *Message Protection* pattern, this pattern is applicable to the illustrated data transfer object.

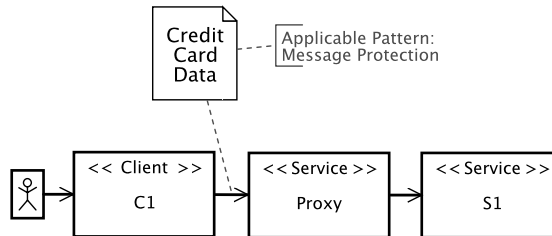


Figure 6.19: Message Protection Example

The **solution** of this pattern selects a message security protocol and a message protection token from the profiles. Similar to the pattern *Secure Pipe*, these properties are set in all security constraints that have been generated by the patterns that invoked the *Information Protection* intention.

Chapter 7

A Pattern-driven Transformation Process

Chapter 3 introduced our domain-independent model that serves as an abstraction layer to security policy languages. As outlined in 1.2, the transformation of security intentions to instances of our security policy model represents the core concept of our model-driven approach. The security configuration pattern system introduced in the previous chapter is used to facilitate the generation of policy model instances containing security constraints. This transformation process is described and discussed in this chapter. In addition, we provide a formal verification of the transformation to prove that security requirements are preserved throughout the transformation process.

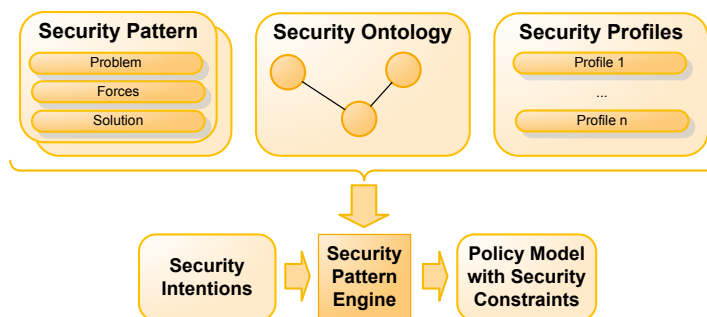


Figure 7.1: Transforming Security Intentions to Security Policies

The overall transformation process is shown in Figure 7.1. The transformation from security intentions to instances of our policy model is performed by a *security pattern engine*. A security pattern engine is a component that applies a set of security patterns to a set of security intentions. The engine has to select patterns that match to the required security intentions and whose forces evaluate as true. In accordance to the solutions stated in these patterns, an instance of our security policy model is generated containing alternative sets of security constraints. In addition to the security patterns, a security ontology is used by the pattern engine to map concepts referenced in the patterns' solutions to specific technologies that are listed in the profiles. For example, the solution of a security pattern can require a message-based protection. Using the

ontology, the pattern engine can map this requirement to a specific version of WS-Security that is configured in the profiles.

7.1 Security Intention Transformation

A security pattern engine applies a security pattern system to the security intentions stated in a system design model and returns an instance of our security policy model (see section 1.2.2). An important feature of a pattern engine is the capability to interpret and enforce the operations of our domain-specific language that has been specified in section 5.3.

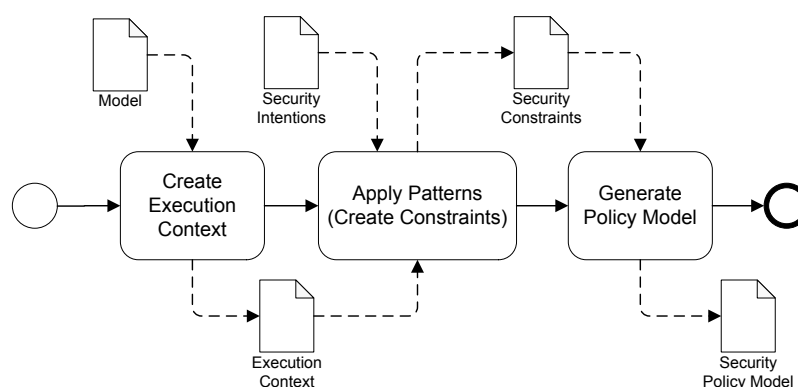


Figure 7.2: Pattern Engine Transformation Process

The transformation process performed by an engine is illustrated in Figure 7.2. A SecureSOA *system model* is translated to a relational model denoted as *execution context*. This model provides the foundation to apply security configuration patterns to a set of *security intentions* that results in the generation of *security constraints*. Since multiple security patterns might be applicable to a security intention and a pattern might require the enforcement of additional security intentions, the application of security patterns can be represented as a tree. Security constraints that have been created in the pattern application process are associated to a specific node in this pattern application tree (see section 5.3.3.4). Finally, the pattern application tree and related security constraints are transformed to security alternatives that are combined in a *security policy model*. In the following, we will discuss these steps in more detail.

7.1.1 Execution Context Creation

The execution context is a data structure that stores a relational SecureSOA model. The pattern engine executes the forces and the solution of a security configuration pattern on the basis of the information in the execution context. This relational model is generated from SecureSOA models that have been created by the architect of the system in a modelling tool. The generation of the relational model is based on a mapping from the data exchange format supported by this modelling tool. For example, the web based modelling tool Oryx [DOW08] exports instances of SecureSOA models using the data exchange formats RDF and JSON. Therefore, the implementation of our model-driven approach provides a mapping from JSON to the SecureSOA relational model.

A simple example is shown in Figure 7.3 that will be used throughout this chapter. This example contains three participants (1: Web Frontend, 2: Web Service, 3: STS). The security intentions reference a set of security profiles that is specified as

$$profile = \{(UsernamePasswordToken, default), (SSL, default), (WSS, default)\}$$

The creation of the relational model based on the elements defined in this model results in the instantiation of the following sets: $Object = \{1, 2, 3\}$, $Client = \{1\}$, $Service = \{2\}$, and $STS = \{3\}$. In addition, the following relations represent the interactions and trust relationships in the example: $OO_{Interaction} = \{(1, 2), (1, 3)\}$ and $OO_{Trust} = \{(2, 3), (3, 1)\}$.

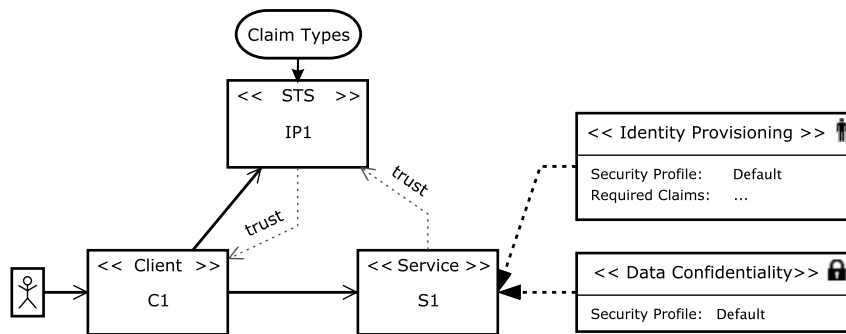


Figure 7.3: SecureSOA Modelling Example

7.1.2 Applying Security Patterns

In the next step, security patterns are applied to the security intentions specified in the execution context (see Figure 7.2). We represent the application of security patterns to a security intention as a function *TransformIntention* that operates on the execution context, expects a security intention as input parameter and returns a sets of security constraints. This function is invoked for each security intention in the execution context independently. The intention transformation process that is executed by this function is illustrated in Figure 7.4. The first task in this flow selects security patterns that refer to the required security intention. Then, the forces of each pattern are evaluated. A pattern will be applied, if the forces evaluate as true. As shown in 7.4, the task *Apply Solution* executes the operations specified in a pattern's solution that result in the generation of security constraints.

Since multiple patterns might be applied for a security intention, multiple sets of security constraints might be generated. As aforementioned, operations in a pattern's solution might require the enforcement of additional security intentions. The execution of this operation results in a recursive invocation of the function *TransformIntention* to execute the intention transformation process for the security intentions required additionally. The sets of security constraints, which are returned by each recursion step, are labelled with a number that refers to a node in the tree of applied pattern. This number facilitates the creation of security policies later on. In the following sections, the steps *Evaluate Forces* and *Apply Solution* of the intention transformation process shown in Figure 7.4 are described in detail.

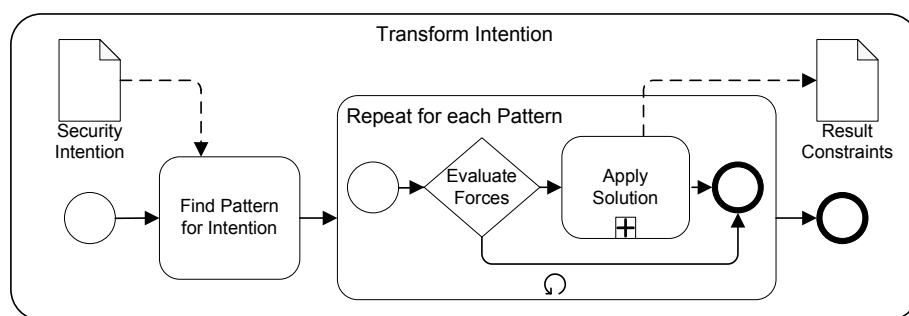


Figure 7.4: Pattern Engine: Intention Transformation

7.1.2.1 Forces Evaluation

The forces of a pattern are a sequence of operations that are defined by our security configuration pattern DSL. A pattern is applicable to a security intention, if all *ASSERT* operations stated in the forces return true. As specified in section 5.3, an *ASSERT* operation returns true, if the value passed to this operation is not equal to zero, is not equal to false, and is not an empty set. This value is the result of the functions and operations of our DSL that evaluate sets and relations in the execution context. The pattern engine must interpret these functions and operations in accordance with the formal semantics of these functions.

For example, consider the evaluation of the forces for the intention *'IdentityProvisioning'* as shown in Figure 7.3. The first line of the pattern's forces state that the security intention the pattern is applied to must be a relying party: *ASSERT(RelyingParty CONTAINS intention.subject)*

The execution context contains the set $RelyingParty = \{2, 3\}$. Since the intention is attached to the service, we know that $(intention.subject = 2)$. Due to $intention.subject \in RelyingParty$, the pattern engine can evaluate *RelyingParty CONTAINS intention.subject* as true. Therefore, the first *ASSERT* operation evaluates as true.

7.1.2.2 Solution Application: Generation of Security Constraints

The transformation process of security intentions that is illustrated in Figure 7.4 applies the solution of a security configuration pattern by executing the sub process *'Apply Solution'*. This process is illustrated in Figure 7.5. The application of a pattern's solution results in the execution of the operations in this solution that instantiate and modify security constraints. The DSL to describe a solution has been introduced in section 5.2.

An operation in the domain-specific language is selected and interpreted to generate and manipulate security constraints. As shown in Figure 7.5, there are three basic types of operations that create or modify security constraints: *REQUIRE* creates a specific constraint identified by a security goal, while *SET/USE* operations modify properties in the constraints. *ENFORCE* triggers the transformation of an additional, subsidiary security intention. In particular, the *ENFORCE* operation requires a recursive execution of the intention transformation process illustrated in Figure 7.4 that result in the creation of additional security constraints and the modification of existing constraints. Therefore, the constraints created in prior intention transformations have to be passed to the subsidiary transformations invoked recursively. This enables *SET/USE* op-

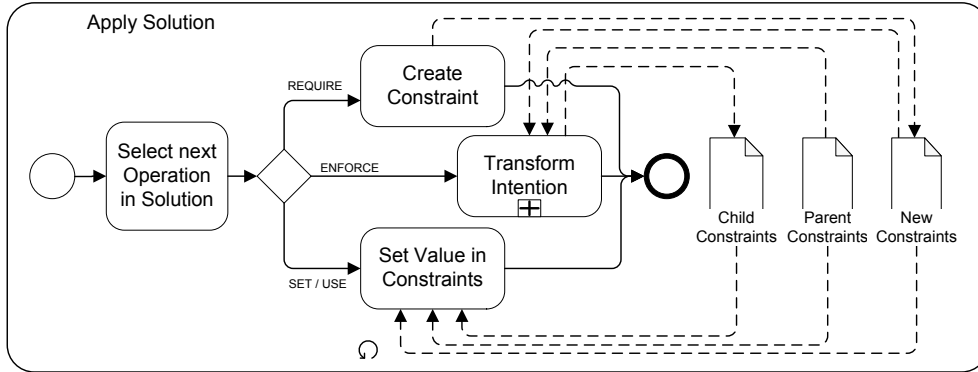


Figure 7.5: Pattern Engine: Solution Application

erations to modify properties in the constraints, which were created in the current intention transformation process and in prior transformations steps. To organise the security constraints created in different recursive transformation steps, the intention transformation process illustrated in Figure 7.4 is based on three sets: *Parent Constraints*, *New Constraints*, and *Child Constraints*. These sets will be introduced later on.

As aforementioned, the recursive transformation of security intentions triggered by the ENFORCE operation results in the application of multiple security patterns that can require the enforcement of further intentions. A simple example based on three patterns is shown in Figure 7.6. A pattern 'A' is applied that results in the execution of operations as specified by the solution of 'A'. The REQUIRE operation results in the creation of security constraint *a*, while SET and USE operations modify this constraint. Moreover, the solution of 'A' contains an ENFORCE operation that requires the transformation of an additional security intention. In Figure 7.6, the security configuration patterns 'B' and 'C' address this intention and are applied as well. Pattern 'B' creates security constraint *b* in constraint set 1.1, while 'C' results in the creation of the security constraint *c* added to constraint set 1.2. The SET and USE operations modify the properties in the constraints created by these patterns and in constraint *a* that has been created by pattern 'A'. To avoid that the modifications performed by patterns 'B' and 'C' on constraint *a* conflict with each other, constraint *a* is duplicated for each pattern and added to the constraint sets 1.1 and 1.2. These constraints are marked as a copy of constraint *a*. SET/USE operations performed by pattern 'A' after applying patterns 'B' and 'C' will also effect the constraints in the sets 1.1 and 1.2 created by these patterns. The result of the overall intention transformation process are three constraint sets labelled with 1, 1.1, and 1.2. These labels relate the sets to the pattern application tree.

7.1.2.3 Security Constraint Sets

The example introduced above revealed that patterns operate on constraints that have been created by superior patterns, subsidiary patterns, or by themselves. As shown in Figure 7.5 the transformation flow is based on three sets that are used to handle these types of constraints:

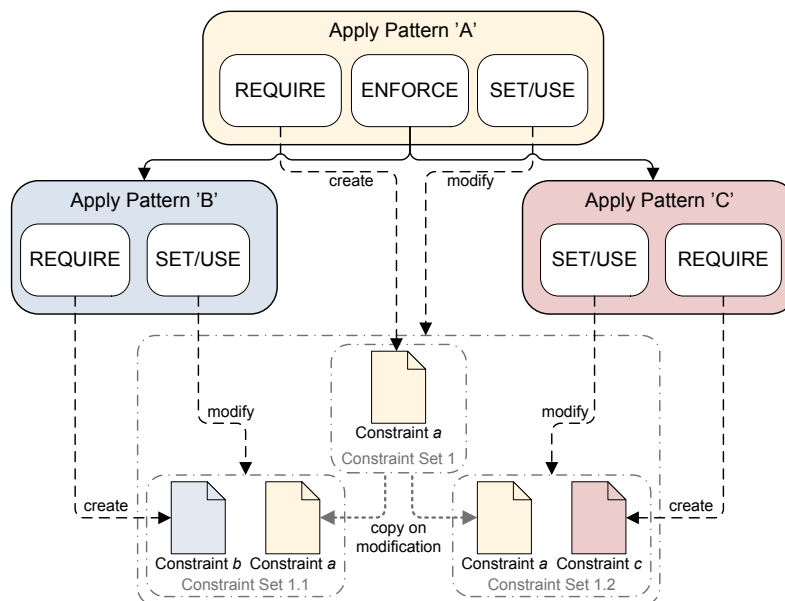


Figure 7.6: Application Example of Security Patterns

1. **New Constraints (NC)** - The *New Constraints* set contains all constraints that have been generated in the scope of the current transformation process by executing the REQUIRE operation. This set will be passed to a subsidiary transformation in conjunction with the *Parent Constraint* set, if an ENFORCE operation is executed. Combined with the *Child Constraints*, the constraints in this set are returned as the result of the transformation process.
2. **Parent Constraints (PC)** - The constraints in this set have been generated by prior intention transformations and have been passed initially to the current transformation process as a parameter. If SET or USE operations are executed that refer to constraints in the *Parent Constraints* set, then modifications will be applied to copies of these constraints to avoid conflicts with other patterns that are applied to the same security intention (cf. Figure 7.6, copy on modification). These copies are enhanced with references to their parent constraints and stored in the *New Constraints* set. Moreover, the *Parent Constraints* set will be passed to a subsidiary transformation in combination with the *New Constraints* set, if an ENFORCE operation is executed.
3. **Child Constraints (CC)** - These constraints have been generated by recursive calls of subsidiary intention transformations. An intention transformation process returns the *Child Constraints* in combination with the *New Constraints* set as a result.

7.1.2.4 Executing Constraint Operations

As aforementioned, four operations are used to create and modify constraints. These operations work on the constraint sets as follows:

- **REQUIRE (String SecurityGoal)** - This operation maps a security goal to a security constraint type, instantiates a security constraint of this type and adds it to the *New Constraints* set. As described in section 3, our SOA Security meta-model provides constraints for the security goals *Authentication*, *Confidentiality*, and *Integrity*.
- **ENFORCE (String SecurityIntention)** - This operation requires the enforcement of an additional security intention and results in a recursive invocation of the intention transformation process as described above. The constraints that result from this invocation are added to the *Child Constraints* set.
- **SET (String Key, AnyType Value)** - The workflow executed by the SET operation is illustrated in Figure 7.7. This operation is used to assign a value to a specific security constraint property. The property is identified by the key that is passed to this operation. Therefore, the ontology is used in the first step to resolve the name of this property. Each property belongs to a specific type of security constraint (data protection constraint or authentication constraint). In the next step, all security constraints of this type are selected. Security constraints that are selected in the *Parent Constraints* set are duplicated and placed in the *New Constraints* set. In addition, references to the parent constraints are added to each copy. This step ensures that the execution of alternative patterns does not result in conflicting modifications on constraints in the *Parent Constraints* set. Finally, the property identified by the key is modified in the constraints selected in the *New Constraints* set and the *Child Constraints* set.

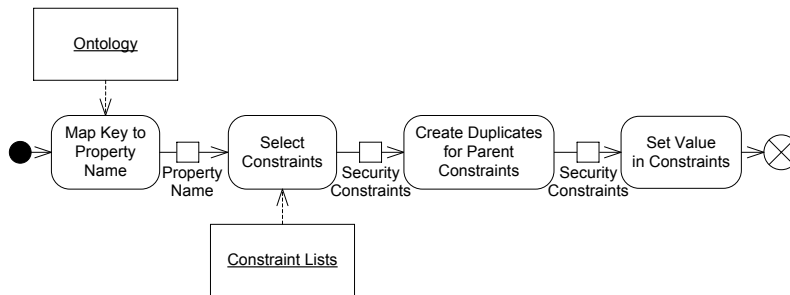


Figure 7.7: SET Operation Flow

- **USE (String Concept)** - While the SET operation modifies a key with a specific value, the USE operation fetches the value for a security concept from the security profile as shown in Figure 7.8. The SET operation is used in a second step to assign this value.

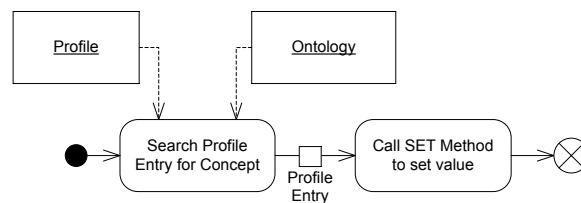


Figure 7.8: USE Operation Flow

In addition to these operations, there are operations defined in section 5.3 that can be used in the solution as well, but that do not result in a direct effect on the security constraints (e.g. FORALL-operation).

7.1.2.5 Transformation Example

Figure 7.9 illustrates the transformation of the security intentions *'Identity Provisioning'* and *'Data Confidentiality'* that are stated in the model shown in Figure 7.3. To transform the intention *'Identity Provisioning'*, the pattern *'Brokered Identity Provisioning'* is applied. This pattern is applicable, since the subject of the intention is a service, there is a trust path to the client that interacts with the service, and the STS on the trust path is capable to provide the required claims. The solution of this pattern requires the enforcement of the intention *'Authentication'* for all entities on the trust path (S1 and IP1). The forces of the security patterns *'Subject Authentication'* and *'STS Issuer Authentication'* evaluate as true, since the entity IP1 is an STS and the service S1 has a indirect trust relationship to the client.

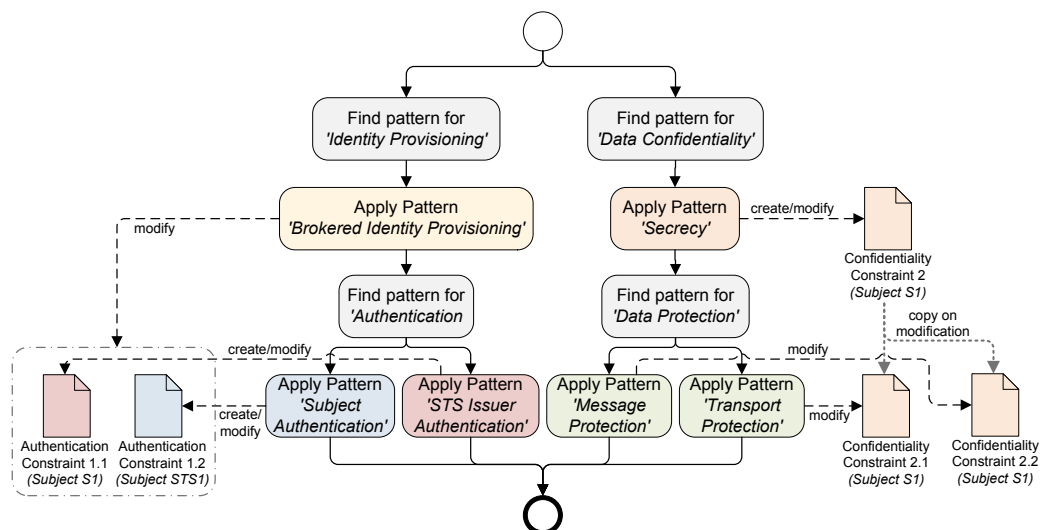


Figure 7.9: Transformation Process Example

The pattern *'STS Issuer Authentication'* results in the creation of an authentication constraint (added to constraint set 1.1) for intention subject S1, sets the client as issuer and the service as target of this constraint, and uses a *'UserToken'*. Since *UsernamePasswordToken* is configured as a credential type in the profiles and the security ontology for identity management reveals that *UsernamePasswordToken* is a *UserToken*, this credential type is set in the *Credential* property of the authentication constraint. The relationship between *UserToken* and the *Credential* property is described by the ontology as well. Similarly, the pattern *'Subject Authentication'* generates an authentication constraint (added to constraint set 1.2) for the intention subject STS1, sets the issuer and the target and resolves an *STSIssuedToken* using the ontology and the profiles. After applying the security patterns for the intention *'Authentication'*, the solution of the *'Brokered Identity Provisioning'* pattern sets the required claims provided by the *'Identity Provisioning'* intention in the authentication constraint for subject S1.

The pattern *'Secrecy'* is applied to satisfy the security intention *'Data Confidentiality'*. The

solution of this pattern creates a confidentiality constraint and requires the enforcement of the intention 'Data Protection'. The patterns 'Message Protection' and 'Transport Protection' are applied to satisfy this intention, since the client interacts with the service without intermediaries. The patterns use the profile and the ontology to set the properties *SecurityProtocol* and *ProtectionToken* in the constraints. Finally, the solution of the 'Secrecy' pattern resolves an *EncryptionAlgorithm* from the profile. The ontology reveals that the algorithm suite *Basic256* stated in the profile is an *EncryptionAlgorithm* that can be set as an algorithm type in the constraint. Moreover, the protected message parts are set by the solution.

7.1.3 Security Policy Model Generation

The application of security configuration patterns results in security constraint sets returned by the intention transformation process. This process implements the *Apply Patterns* step illustrated in Figure 7.2. In a final step, an instance of our policy model is created on the basis of the security constraint sets. However, not all constraints that are provided in these sets must be required at the same time. Multiple patterns might have been applied for a security intention. The sets of security constraints created by the application of these solutions represent policy alternatives. For instance, the enforcement of the security intention *Data Protection* might result in the application of two patterns and, therefore, the creation of two constraints: one constraint that requires security at the transport layer and one constraint that requires security at the message layer. Therefore, these constraints have to be assigned to alternatives that are combined in the policy model.

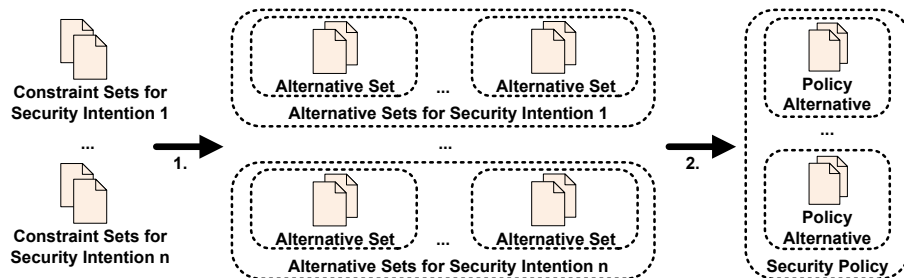


Figure 7.10: Policy Model Generation Steps

The generation of security policies is performed in two steps and is shown in Figure 7.10. As aforementioned, the intentions transformation process has been executed for each security intention that is defined in the SecureSOA model. Therefore, multiple constraint sets have been generated for each security intention. In a first step, these constraints are recombined in sets representing alternatives concerning a specific security intention. Finally, the alternative sets referring to different security intentions are combined to policy alternatives that are added to an instance of our security policy model.

As shown in the example in Figure 7.9, the enforcement of a security intention can be represented as a tree, since multiple patterns might be applied for a security intention and each solution might require the enforcement of additional security intentions. The application of these patterns results in the creation of sets with new security constraints that can be organised hierarchically according to the pattern application tree. Each constraint set is labelled with the position of its constraint set in the constraint hierarchy.

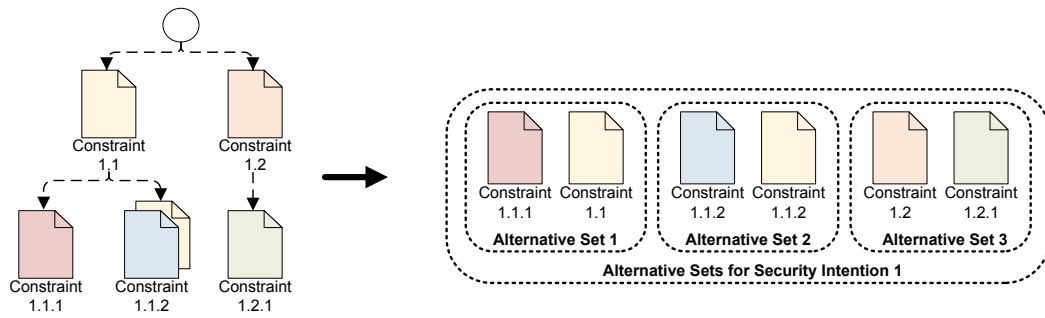


Figure 7.11: Generation of Policy Alternatives

The sets in the pattern application tree have to be recombined to sets that represent alternatives concerning a specific security intention. The constraints with the same distance to the root node in the hierarchy are alternatives, since these constraints have been generated by alternative patterns that are applicable for the same security intention. Therefore, the generation of alternative sets works as follows: All constraints that are on the path from the root node to a leaf in this tree are combined in an alternative set that is added to the set of alternatives for a specific security intention.

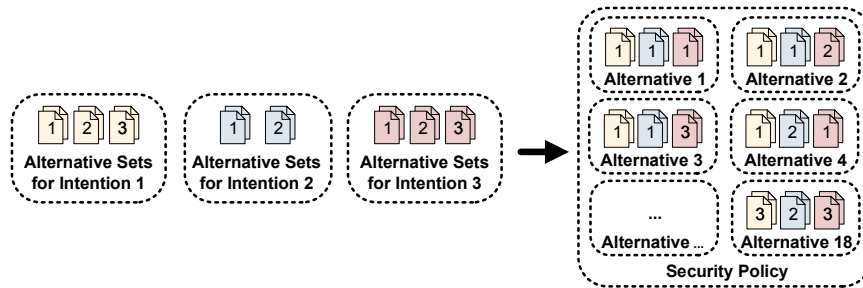


Figure 7.12: Merging Alternative Sets for Multiple Security Intentions

Figure 7.11 shows a simple example to illustrate this step. Two patterns have been applied for the the intention '1' resulting in the creation of two security constraints. Therefore, these constraints are labelled with 1.1 and 1.2. For each pattern, subsidiary patterns were applied that created constraints in the sets 1.1.1, 1.1.2 and 1.2.1. In particular, the pattern 1.1.2 duplicated the constraint 1.1, since a modification of this constraint was required by its solution. Based on this tree, a security policy is generated that consists of three alternatives containing the constraints on the paths to the leafs of the tree. Please note that the second alternative does not contain constraint 1.1, because it has been replaced by a duplication in the 1.1.2 constraint set.

The example in Figure 7.11 illustrates the generation of a alternative sets for a single security intentions. If constraint sets have been created for multiple security intentions, then the recombination will be performed for each security intention independently. As shown in 7.10, these sets have to be merged to create a security policy. A policy alternative is created by selecting a constraint set for each intention. These sets are combined to form a policy alternative. Therefore, each alternative set is combined with alternative sets that refer to another security intention. This step is illustrated in Figure 7.12.

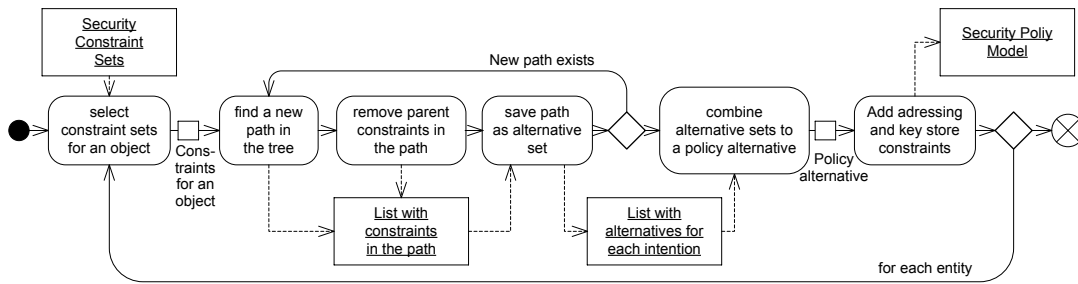


Figure 7.13: Policy Generation Flow

The overall policy generation flow is shown in Figure 7.13. This flow operates on security constraint sets that have been generated in a previous step. A policy will be generated for each object in the SecureSOA model. In the first step, alternative sets are created for each intention attached to this object. Therefore, the paths to the leaves in the constraint sets are determined. In each path, parent constraints are removed that have been duplicated and modified by a subsidiary pattern. The resulting constraints are added to an alternative set. After calculating the alternative sets for all intentions of an object, each alternative for an intention is combined with the alternatives for all other intentions. The combined sets of constraints are added to a security policy as policy alternatives.

Figure 7.14 illustrates the policy generation for the example introduced in Figure 7.3. Several constraints have been generated for the security intentions 'Identity Provisioning' and 'Data Confidentiality'. Two constraints have been created for the intention 'Identity Provisioning', while one constraint has been created by the pattern 'Secrecy' that has been modified by subsidiary patterns. There is a single constraint for STS1 and three constraints for the service S1. Therefore, a security policy with a single alternative is generated for STS1. The security policy for S1 contains two policy alternatives, since the alternative for the 'Identity Provisioning' intention is merged with the two alternatives for 'Data Confidentiality'.

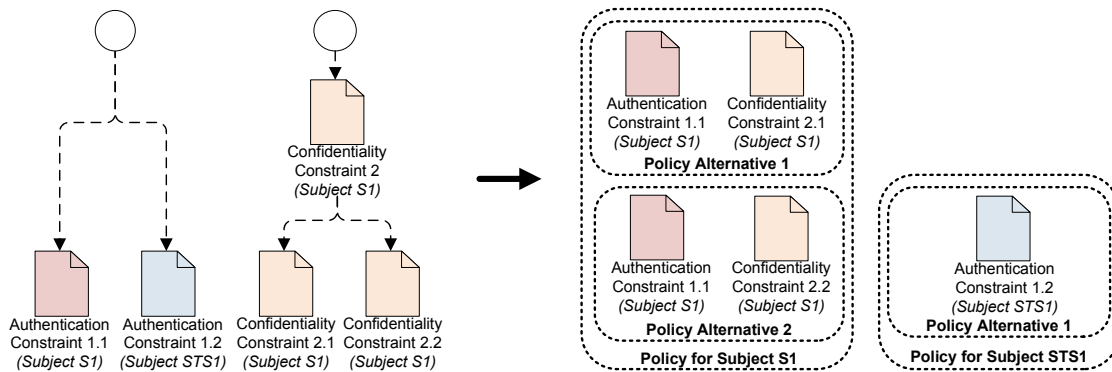


Figure 7.14: Policy Generation Example

7.2 Formal Verification of the Transformation Process

The transformation process introduced in the previous section translates a SecureSOA enhanced system design model to our domain-independent model. To proof the correctness of this transformation step we have to verify that the protection characteristics of data transfer objects (evaluated by functions such as $enc_{SecureSOA}(d)$ and $enc_{dim}(d)$) are preserved during the transformation. In this section, we will prove the correctness of the transformation concerning the encryption of data transfer objects.

Theorem: Let $m_{SecureSOA}$ be a relational SecureSOA model and let m_{dim} be a domain-independent relational model that has been created by applying the security pattern system introduced in chapter 6 to $m_{SecureSOA}$. Since the application of these security patterns results in the generation of an instance of our security policy model and do not alter components of the SecureSOA model, we assume that both models share a common set of components representing our SOA interaction model (e.g. Objects, Interactions, or Data Transfer Objects). The functions $enc_{SecureSOA}(d)$ and $enc_{dim}(d)$ evaluate encryption properties of data transfer objects as introduced in section 4.2.3 and in section 3.3.2. Let $DataTransferObject$ be a component of $m_{SecureSOA}$, then

$$\forall dto \in DataTransferObject : enc_{SecureSOA}(dto) \Rightarrow enc_{dim}(dto) \quad (7.1)$$

Proof: So given an arbitrary $dto \in DataTransferObject$ we assume $enc_{SecureSOA}(dto) \equiv true$. From definition 4.3 it follows that a security intention exists that refers to the data transfer object dto :

$$enc_{SecureSOA}(dto) \Rightarrow \exists i \in DataConfidentiality : (i \ SD \ dto) \quad (7.2)$$

Next, we have to show that a security pattern exists that is applicable to this security intention. We use an initial variable assignment function σ_1 that provides the values for all components of m_{Secure} and m_{dim} . Then we know

$$dto \in \sigma_1(DataTransferObject) \quad (7.3)$$

$$i \in \sigma_1(DataConfidentiality) \quad (7.4)$$

$$i \in \sigma_1(intention) \quad (7.5)$$

$$true \equiv \sigma_1(ForcesResult) \quad (7.6)$$

Let *Secrecy* denote the index of the pattern Secrecy in our security configuration pattern system. We will show that the forces of this pattern evaluate as true ($\llbracket Forces_{Secrecy} \rrbracket (\sigma_1) \equiv true$).

As specified in the definition of the pattern Secrecy in section 6.2.3, $Forces_{Secrecy}$ denotes a single operation (o_1) whereas

$$o_1 \equiv ASSERT(InteractionPath(intention.subject.issuer, intention.subject.target)) \quad (7.7)$$

In the first step, we will show that *InteractionPath* returns a non-empty set and then we will apply the semantics of the *ASSERT* operation. From Formula 3.4 follows that the data transfer object dto must have an issuer:

$$\exists o_{Issuer} \in \sigma_1(Objects) : (dto, o_{Issuer}) \in \sigma_1(DO_{Issuer}) \quad (7.8)$$

Let $o_{Issuer} \in \sigma_1(\text{Objects})$ whereas $dto \sigma_1(DO_{Issuer}) o_{Issuer}$. With 7.2 and 7.5 we obtain

$$\sigma_1(\text{intention}) \sigma_1(SD) dto \wedge dto \sigma_1(DO_{Issuer}) o_{Issuer} \quad (7.9)$$

By definition of $\llbracket \text{intention.subject.issuer} \rrbracket$ (cf. Definition 5.22), this implies

$$\llbracket \text{intention.subject.issuer} \rrbracket (\sigma_1) = o_{Issuer} \quad (7.10)$$

Correspondingly, it follows by definition of $\llbracket \text{intention.subject.target} \rrbracket$ (cf. Definition 5.23)

$$\exists o_{Target} : \llbracket \text{intention.subject.target} \rrbracket (\sigma_1) = o_{Target} \quad (7.11)$$

Formula 3.4 states that there is a path between the issuer o_{Issuer} and the target o_{Target} of the data transfer object dto and we obtain:

$$\begin{aligned} & o_{Issuer} \sigma_1(OO_{Interaction})^* o_{Target} \\ \Rightarrow & \text{there is a sequence } p_0, \dots, p_n \text{ with } p_i \in \sigma_1(\text{Object}) \text{ for all } 0 \leq i \leq n \\ & \text{where } (p_i, p_{i+1}) \in \sigma_1(OO_{Interaction}) \text{ for all } 0 \leq i < n \\ & \text{and } p_0 = o_{Issuer}, p_n = o_{Target} \end{aligned} \quad (7.12)$$

This implies that there is a graph $Path_{OO_{Interaction}}(o_{Issuer}, o_{Target}, \sigma)$ as defined by Formula 5.14 that returns $(V_{OO_{Interaction}}, E_{OO_{Interaction}})$ where $\{p_0, \dots, p_n\} \subseteq V_{OO_{Interaction}}$. By definition of $InteractionPath$ (cf. 5.16), it follows that

$$\begin{aligned} & \llbracket InteractionPath(\text{intention.subject.issuer}, \text{intention.subject.target}) \rrbracket (\sigma_1) \supseteq \{p_0, \dots, p_n\} \\ \Rightarrow & \llbracket InteractionPath(\text{intention.subject.issuer}, \text{intention.subject.target}) \rrbracket (\sigma_1) \neq \{\} \end{aligned} \quad (7.13)$$

From the definition of τ (cf. Definition 5.5) and 7.13 we obtain

$$\tau(\llbracket InteractionPath(\text{intention.subject.issuer}, \text{intention.subject.target}) \rrbracket (\sigma_1)) \equiv true \quad (7.14)$$

The semantics of the *ASSERT* operation have been specified in Formula 5.31 that provides the variable assignment function σ_2 by defining the changes to the function σ_1 . From this definition and Formulae 7.14 and 7.6 we obtain

$$\begin{aligned} & \sigma_2 = \sigma_1 \oplus \{ForcesResult \rightarrow true\} \\ \Rightarrow & \sigma_2 \supseteq \{ForcesResult \rightarrow true\} \end{aligned} \quad (7.15)$$

σ_2 represents the variable assignment function after the termination of the forces $(o_1 | \sigma_1 \rightarrow \epsilon | \sigma_2)$. According to the semantics of the forces defined in Formula 5.32 we obtain

$$\llbracket Forces_{Secrecy} \rrbracket (\sigma_1) \equiv \sigma_2(\text{ForcesResult}) \equiv true \quad (7.16)$$

Therefore, it is proven that the security configuration pattern *Secrecy* is applicable to the *Data Confidentiality* intention i . In the next step, we will show that the application of this pattern results in the generation of a model m_{dim} that contains a confidentiality constraint related to the data transfer object dto .

The initial variable assignment function σ_1 introduced above is used to execute the solution. We assume that this function initialises variables representing the components of m_{dim} . These sets and relations are initialised as empty sets. In addition, the pattern application tree is initialised with a root node that represents the application of the pattern *Secrecy*. The relation *PatO* associates this node with the policy subject $\llbracket intention.PolicySubject \rrbracket$. From the semantics of the policy subject specified in Formula 5.21 we obtain $\llbracket intention.PolicySubject \rrbracket = o_{Target}$ where $dto DO_{Target} o_{Target}$. The function σ_1 initialises these sets and relations as follows:

$$\begin{aligned}
 \sigma_1(ConfidentialityConstraint) &= \{\} & (7.17) \\
 \sigma_1(SecurityConstraint) &= \{\} \\
 \sigma_1(CPAT) &= \{\} \\
 \sigma_1(PatO) &= \{(1, o_{Target})\} \\
 \sigma_1(CD) &= \{\} \\
 \sigma_1(PID) &= 1 \\
 \sigma_1(PAT) &= (V_{PAT}, E_{PAT}) \text{ with } V_{PAT} = \{1\} \text{ and } E_{PAT} = \{\}
 \end{aligned}$$

The solution of the pattern *Secrecy* has been specified in section 6.2.3. $Solution_{Secrecy}$ denotes the a sequence of operations (o_1, o_2, o_3, o_4) with

$$o_1 \equiv REQUIRE(Confidentiality) \quad (7.18)$$

Definition 5.39 provides the semantics of the REQUIRE operation and enables the calculation of σ_2 . With $n_{SC} := \# \sigma_1(SecurityConstraint) = 0$ we obtain

$$\sigma_2 = \sigma_1 \oplus v_{CC}(\sigma_1) \cup v_{SC}(\sigma_1) \cup \{CPAT \rightarrow \sigma_1(CPAT) \cup (1, \sigma_1(PID))\} \quad (7.19)$$

Using the Definitions 5.36 and 5.38 we can calculate $v_{CC}(\sigma_1)$ and $v_{SC}(\sigma_1)$ as

$$v_{CC}(\sigma_1) = \{ConfidentialityConstraint \rightarrow \{1\}\} \quad (7.20)$$

$$v_{SC}(\sigma_1) = \{SecurityConstraint \rightarrow \{1\}, CD \rightarrow \{1, dto\}\} \quad (7.21)$$

By replacing $v_{CC}(\sigma_1)$ and $v_{SC}(\sigma_1)$ in Formula 7.19 we obtain

$$\sigma_2 \supseteq \left\{ \begin{array}{ll} ConfidentialityConstraint & \rightarrow \{1\}, \\ SecurityConstraint & \rightarrow \{1\}, \\ PatO & \rightarrow \{(1, o_{Target})\}, \\ CPAT & \rightarrow \{(1, 1)\}, \\ PAT & \rightarrow (\{1\}, \{\}), \\ CD & \rightarrow \{(1, dto)\} \end{array} \right\} \quad (7.22)$$

The execution of the solution is a transformation $o_1 | \sigma_1 \rightarrow^* \epsilon | \sigma_5$. Each execution step might cause an enhancement of the constraint sets and the pattern application tree. Since elements can be added to sets by executing the operations, but not removed, we can derive

$$\begin{aligned}
\sigma_2(\text{ConfidentialityConstraint}) &\subseteq \sigma_5(\text{ConfidentialityConstraint}) & (7.23) \\
\sigma_2(\text{SecurityConstraint}) &\subseteq \sigma_5(\text{SecurityConstraint}) \\
\sigma_2(\text{CPAT}) &\subseteq \sigma_5(\text{CPAT}) \\
\sigma_2(\text{PatO}) &\subseteq \sigma_5(\text{PatO}) \\
\sigma_2(\text{PAT}) &\subseteq \sigma_5(\text{PAT}) \\
\sigma_2(\text{CD}) &\subseteq \sigma_5(\text{CD})
\end{aligned}$$

The variable assignment function σ_5 provides the values for the components of the model m_{dim} that is returned by the pattern solution. In addition, the model m_{dim} contains components to represent policies and policy alternatives. To create these components, we use the sets *Policy*, *PolicyAlternative*, *PO*, *AP*, *CA* as specified by the formulae 5.41, 5.42, 5.44, 5.45, and 5.46. The definition of these sets is based on the set $alternativePaths = \{path_1, \dots, path_n\}$ specified in 5.43 that contains all paths in the pattern application tree representing policy alternatives.

In the next step, we have to show that policy alternatives have been generated for the target of the data transfer object. From the definition of the initial variable assignment function σ_1 in Formula 7.17 and the statement on σ_5 (cf. Formula 7.23) we know $\sigma_5(\text{PAT}) = (V_{PAT}, E_{PAT})$ with $V_{PAT} = \{1\}$ and $1 \sigma_5(\text{PatO}) o_{Target}$. Since there is at least one node in V_{PAT} , it follows that there is at least one path in $alternativePaths$. Therefore, the $alternativePaths$ definition 5.43 implies that

$$\begin{aligned}
\exists path_a \in alternativePaths \text{ with } path_a = (p_1, \dots, p_k), k \geq 1 & & (7.24) \\
\text{and } p_1 = 1 & \\
\text{and } p_k \sigma_5(\text{PatO}) o_{Target} &
\end{aligned}$$

According to the definition of the set *SecurityAlternative* in Formula 5.44 we obtain $a \in SecurityAlternative$. Formulae 5.41 and 5.42 state that each object is assigned to a policy that has the same id as the object itself. Therefore, we know $\exists p \in Policy : p PO o_{Target}$. From the definition of *AP* in Formula 5.46 and from Formula 7.24 we can derive the following implication

$$\begin{aligned}
(p_k, o_{Target}) \in \sigma_5(\text{PatO}) \wedge (p, o_{Target}) \in PO & & (7.25) \\
\Rightarrow a AP p &
\end{aligned}$$

This proves that a policy alternative has been generated for the target of the data transfer object:

$$\exists a \in SecurityAlternative : dto \sigma_5(DO_{Target}) \circ PO^{-1} \circ AP^{-1} a \quad (7.26)$$

Finally, it remains to prove that a confidentiality constraint exists for all security alternatives that refers to the data transfer object. So given arbitrary $a \in SecurityAlternative$ we assume

$$dto \sigma_5(DO_{Target}) \circ PO^{-1} \circ AP^{-1} a \quad (7.27)$$

The definition of the set *SecurityAlternative* is based on the set $alternativePaths$ (cf. Formula 5.44). Let $path_a = (p_1, \dots, p_k)$ denote the path in $alternativePaths$ that corresponds to alternative a . The definition of this set in Formula 5.43 implies that $p_1 = 1$. Since the node 1 is the root element of the pattern application tree, it is an element of each path in $alternativePaths$

We know from 7.27 that o_{Target} relates to the policy of a :

$$\exists p : a \ AP \ p \wedge p \ PO \ o_{Target} \quad (7.28)$$

The definition of AP in Formula 5.46 implies $p_k \ \sigma_5(PatO) \ o_{Target}$. Formulae 7.22 and 7.23 entail $1 \ \sigma_5(PatO) \ o_{Target}$ and $1 \ \sigma_5(CPAT) \ 1$. With $p_1 = 1$ we obtain $p_1 \ \sigma_5(PatO) \ o_{Target}$ and $1 \ \sigma_5(CPAT) \ p_1$. By definition of CA in Formula 5.45, it follows $1 \ CA \ a$.

With $1 \in \sigma_5(ConfidentialityConstraint)$, $1 \ \sigma_5(CD) \ dto$ (cf. Formulae 7.22 and 7.23) we obtain

$$\exists c \in \sigma_5(ConfidentialityConstraint) : c \ CA \ a \wedge c \ \sigma_5(CD) \ dto \quad (7.29)$$

According to the semantics of the confidentiality constraint specified in Definition 3.6 Formulae 7.26 and 7.29 imply

$$enc_{dim}(dto) \equiv true \quad q.e.d$$

We have proven the correctness of the transformation from a model $m_{SecureSOA}$ to a model m_{dim} concerning the encryption property of data transfer objects. The transformation of a security intention *Data Confidentiality* will always result in the creation of confidentiality constraints that refer to the data transfer objects that are annotated by the security intention. The correctness of the transformation regarding the security goals integrity and authentication can be shown in a similar way.

Chapter 8

Proof of Concept: The SOA Security LAB

In this chapter, our SOA Security LAB is presented that supports the on-demand creation and orchestration of composed applications and services. Our platform enables the testing, monitoring and analysis of Web Services regarding different security configurations, concepts and infrastructure components. Since security policies are hard to understand and even harder to codify, our model-driven approach is used to simplify the creation of security configurations.

8.1 SOA Security LAB Overview

The Web Service specifications introduced in section 2.1 provide a flexible messaging framework to enable an interoperable communication between loosely coupled services. SOAP and WSDL are core specifications in this framework that specify the structure of exchanged messages and facilitate the definition of service contracts to describe exposed service interfaces. Web Service frameworks such as Axis2, Oracle Metro, or Microsoft Web Service Communication Foundation facilitate the usage of these specifications by providing tools and APIs to expose and invoke class methods as Web Services. These frameworks hide the complexity of the Web Service specification stack and support developers to focus on the business logic implemented in these methods. For instance, a code-first approach is supported by these frameworks to enable an automated creation of services and service contracts on the basis of programming language classes and implemented methods. In addition, tools are provided to generate a proxy class that offers a programming language interface to invoke a specific Web Service. This proxy class can be generated automatically on the basis of a WSDL file and handles the creation and exchange of SOAP messages. Altogether, Web Service frameworks enable developers to build service-based systems without expert knowledge in Web Service technology.

However, Web Service frameworks are not capable to simplify the configuration of security requirements. Security policies must be deployed with each service to configure the enforcement of these requirements. In addition, the handling of required credentials such as certificates or security tokens must be configured or implemented. As outlined in the introduction of this work, the provision of security policies is an error-prone task due to the complexity of WS-SecurityPolicy.

To simplify the generation of security policies, tool support is offered by most Web Service frameworks. Policies editors are provided to offer a user interface that enables the selection of security options and the generation of WS-SecurityPolicy documents. However, strong security knowledge is still required to understand the underlying security concepts and to select a set of security options that matches to the user's requirements. In addition, the enforcement of these policies requires the provision and handling of additional security-related information: The access and validation of certificates must be enabled, access to security Security Token Services might be required, and the validation and request of user credentials must be implemented.

The high level of configuration and implementation effort results in a complex and time-consuming development process for secure Web Services. Even the instantiation and integration of a small set of secure Web Services requires the provision of complex security policies and the generation and provision of digital certificates. The implementation of advanced identity management and federation approaches requires the setup of additional services such as identity providers based on WS-Trust. However, this complexity impede the provision of secure Web Service-based applications for educational and research purposes that enable the testing of security mechanisms, the execution of web service attacks, and the validation of research ideas.

To provide a simplified way to setup, test, and experience Web Service security mechanisms in practise, we have designed a web-based test platform for Web Service security. Our SOA Security LAB facilitates the configuration and instantiation of web applications orchestrating Web Services in virtual machines. These composed applications can be executed to analyse the exchange of messages and the invocation of security services. The model-driven approach presented in this thesis is used to generate the security configurations for Web Services used by the composed applications.

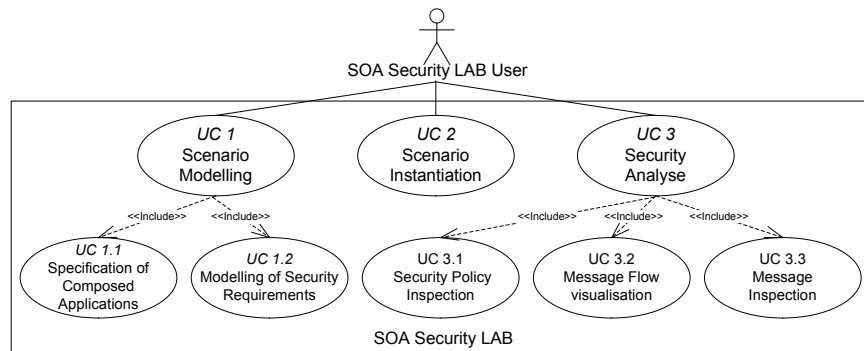


Figure 8.1: SOA Security LAB Usage

As illustrated in Figure 8.1, there are three basic use cases that describe the functionality of the SOA Security LAB:

UC 1: Scenario Modelling - The SOA Security LAB supports the visual design of scenarios that can be instantiated and executed in virtual machines. An SOA Security LAB scenario is an FMC-based SecureSOA model (cf. chapter 4) that describes the usage of composed web applications, the orchestration of Web Services, the integration of security services (e.g. an Security Token Service), and the statement of security requirements. The composed web applications provide frontends that can be used to execute the scenario. The scenario modelling use case includes two basic use cases:

UC 1.1: Specification of Composed Applications - To create a scenario, the basic actors and their relationships have to be defined. Therefore, the SOA Security LAB facilitates the selection of a composed web application and a set of services that implement required interfaces. In addition to services that provide business functionality for the composed web applications, security services (e.g. STS) can be connected to frontends and services to model advanced scenarios. In the current version of the SOA Security LAB, a predefined set of web applications and services can be used. In future versions, an upload and orchestration of custom services might be supported.

UC 1.2: Modelling of Security Requirements - Next to the selection and orchestration of frontends and services, the SOA Security LAB enables the visual configuration of security requirements and settings. In particular, security intentions, security annotations, trust domains, and trust relationships can be specified as described in chapter 4. The model-driven process introduced in this thesis is used to implement the generation of security policies to configure deployed services and service clients.

UC 2: Scenario Instantiation - To enable the execution of modelled scenarios, the SOA Security LAB supports the instantiation of scenarios in a virtual machine. Web applications and Web Services are deployed in an application server along with required configuration files.

UC 3: Security Analysis - The execution of deployed scenarios results in the invocation of deployed services and the exchange of SOAP messages. Messages sent to these services adhere to the services' security policies generated on the basis of the SecureSOA model. The SOA Security LAB enables users to investigate how security policies have been enforced to comply to the modelled security requirements. Altogether, the SOA Security LAB facilitates the analysis of generated policies and the examination of exchanged messages. Therefore, the security analysis use case includes the following basic use cases:

UC 3.1: Security Policy Inspection - Security policies that have been generated by our model-driven approach can be visualised by the SOA Security LAB. This view enables users to inspect policies and to analyse how modelled security requirements are represented in WS-SecurityPolicy.

UC 3.2: Message Flow Visualisation - The execution of scenarios results in the exchange of SOAP messages between web applications and services. To reveal how security services have been integrated and invoked, our LAB supports the visualisation of the message flow in a scenario. In addition, a message processing view is provided to illustrate the message handling. SOAP Messages that have been sent or received by services are processed by a chain of message handlers, which are provided by the Web Service framework that has been used to build these services. For example, security handlers are used to perform the encryption and decryption of messages. The SOA Security LAB enables the visualisation of incoming and outgoing handler chains and the flow of messages through these chains.

UC 3.3: Message Inspection - Messages that have been exchanged and processed by Web Services and Web Service clients can be inspected by the user to investigate how security policies affect the message exchange. Our LAB supports a syntax highlighting of the XML message structure and identifies message parts that are specified by specific Web Service protocols such as WS-Security.

8.2 SOA Security LAB Architecture

Figure 8.2 illustrates the overall architecture of the SOA Security LAB. Our platform is composed of three main components that realise the SOA Security LAB use cases introduced in the previous section. The *Scenario Modelling* component enables the creation of SecureSOA models and the generation of security policies (Use Case 1), the *Deployment Management* component stores deployment artefacts and supports the dynamical instantiation of composed applications (Use Case 2), and the *Security Analysis* component supports the monitoring and analysis of applied security mechanisms (Use Case 3). Each component provides GUI components that are highlighted in Figure 8.2.

As illustrated in Figure 8.2, the SOA Security LAB uses the Tele-Lab Virtual Machine Service to instantiate and control virtual machines. This service has been developed in scope of the Tele-Lab Internet Security project at the Hasso Plattner Institute. The Tele-Lab project provides a novel eLearning platform for Internet Security and enables students to perform practical exercises in virtual machines [WM08]. Each virtual machine that is instantiated by the Tele-Lab VM Service for the SOA Security LAB provides a Tomcat servlet container and a management service. The management service enables the deployment of web applications and Web Services.

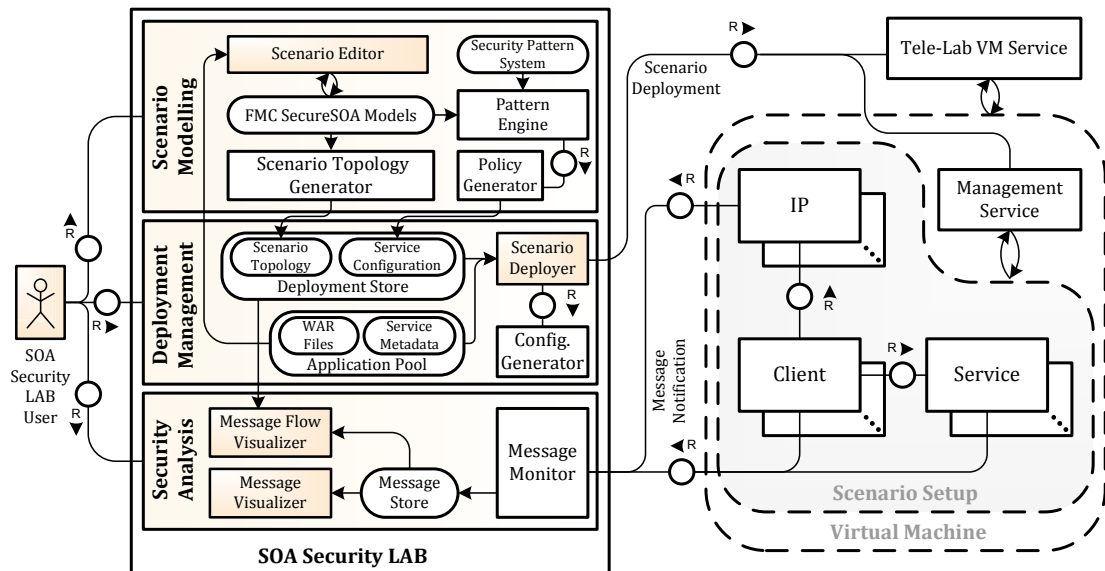


Figure 8.2: Service Security LAB Architecture

The main components of our SOA Security LAB are structured as follows:

1. **Scenario Modelling** – The *scenario modelling* component provides a *scenario editor* to create and alter SOA Security LAB scenarios. A scenario is described by a FMC-based SecureSOA model and specifies the functional aspects (web applications, services and interactions) as well as security aspects (security intentions, security annotations, trust domains, and trust relations). The scenario editor is based on the web-based modelling tool Oryx [DOW08] and has been enhanced with a custom stencil set for our FMC-based SecureSOA dialect. This stencil set is enhanced by our SOA Security LAB with modelling

elements for each web application and Web Service managed in the application pool. This pool is provided by the *Scenario Management* component.

After creating or updating SecureSOA models, security policies are generated for these scenarios using the transformation process described in section 1.2.3. The *pattern engine* component shown in Figure 8.2 performs the first transformation step and resolves platform-independent policy models on the basis of our security configuration pattern system. These platform-independent models are passed to the *policy generator* component that converts these models to Web Service security policies. The WS-SecurityPolicy documents and configuration files are provided to the *deployment management* component.

2. **Scenario Deployment** – The *scenario deployment* component enables the execution of scenarios and manages deployment files and metadata that are required to instantiate scenarios. This component provides an application pool that stores Web Services and composed web applications as WAR files. WAR archives wrap java server pages, servlets, java classes and metadata (e.g. WSDLs) that are required to execute Web Services and web applications in a servlet container such as Tomcat. In addition to WAR archives, the application pool stores dependencies between web applications and Web Services, since each composed application or service might require the provision of additional services that implement a specific interface. As mentioned above, the scenario modelling component uses this information to generate a stencil set for Oryx.

For each SOA Security LAB scenario, deployment information is needed that lists Web Services and web applications, which should be instantiated in a virtual machine. This information is generated by the *Scenario Modelling* component and is stored in the *Scenario Topology Store (Deployment Store)* of the *Deployment Management* component. In addition, configuration documents are stored in the *Deployment Store* that must be deployed with Web Services or web applications. For example, security policies generated for each scenario are added to the *Service Configuration* store.

The *Scenario Deployer* component handles the process that instantiates and configures a scenario in a virtual machine. In a first step, missing configuration documents – e.g. key and trust stores – are created by the *Configuration Generator* component and are added to the *Service Configuration* store. In the next step, the Tele-Lab VM service is invoked to instantiate a new virtual machine. As soon as the virtual machine is provided, the *Scenario Deployer* enhances the archives used in the current scenario with the configuration documents in the *Service Configuration* store and leverages the virtual machine's management Web Service to deploy applications, services and configuration files to the application server as shown in Figure 8.2. Finally, the URL of the composed web application running in the virtual machine is returned to the user to enable the execution of the deployed applications and services.

3. **Security Analysis** – The *Security Analysis* component enables users to inspect the flow of SOAP messages that were exchanged during the execution of a scenario in a virtual machine. This functionality is based on the usage of monitoring agents that have been deployed in the virtual machine. These agents intercept messages and forward them to the *Message Monitor* of the *Security Analysis* component. The *Message Flow Visualiser* aligns monitored messages with the modelled entities and enables users to track the communication and the behaviour of security modules. The content of exchanged messages is visualised by the *Message Visualiser* component that supports a highlighting of the message structure.

8.3 Security Pattern Engine Implementation

As illustrated in 8.2, the security pattern engine and the policy generator are provided by the *Scenario Modelling* component. These components implement our model-driven approach to enable the transformation of abstract security intentions to technical security policies for service-based systems. The security pattern engine performs the pattern-based transformation step described in chapter 7 and leverages the security pattern system introduced in chapter 6 to transform SecureSOA models to an instance of our platform-independent policy model described in chapter 3. This policy model instance is provided to the policy generator that performs the transformation to WS-SecurityPolicies documents. These policies can be deployed with Web Services in virtual machines to configure security requirements. The implementation of the policy generation process has been provided by R. Warschofsky and is described in [WMM10, War10].

In this section, the groovy-based implementation of the security pattern engine is introduced, which performs the pattern-based transformation step to generate an instance of our policy model. The pattern engine's groovy classes compile to java bytecode and are organised in packages. These packages are described in this section.

8.3.1 Security Pattern Engine Package Structure

The implementation of our security pattern engine provides classes that realise the pattern-driven transformation process described chapter 7. These classes are organised in a package structure that is shown in Figure 8.3.

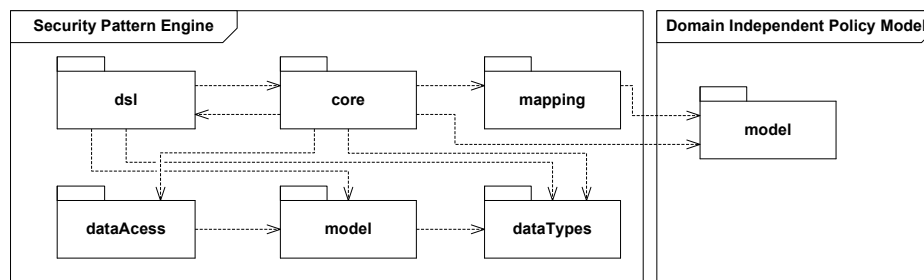


Figure 8.3: Security Pattern Engine Package Overview

The pattern-driven transformation process is applied to a SecureSOA model in three steps (cf. Figure 7.2): The execution context is created to store SecureSOA entities and relations, security constraint sets are generated by applying our security pattern system to security intentions in the SecureSOA model, and the security policy model is generated on the basis of the security constraint sets. To perform this process, the classes in the package *core* implement these steps and required data structures. The execution context and the security pattern system are defined in the packages *dataTypes*, *model*, and *dataAccess*.

The package *dataTypes* provides classes to define and handle basic data types (Long, Boolean, and Set) that are used to implement security pattern DSL operations. In addition, these data types are utilised in the package *model* that provides classes to define security patterns and the execution context. Instances of these classes are created using factory classes provided by the package *dataAccess*.

To enable the application of security patterns, the package *dsl* provides classes that implement our security pattern domain-specific language. These classes are used in the *core* package to evaluate a pattern's forces and to apply a pattern's solution. This application step results in the generation of security constraints and the manipulation of constraint properties. The instantiation of security constraints and the mapping of values to constraint properties is performed by classes in the package *mapping*.

In the following sections, the classes are introduced that are provided by the security pattern engine packages shown in Figure 8.3. This overview starts with packages that provide basic functionality. Finally, the classes in the *core* package are introduced that use the classes in the basic packages to implement the security pattern-based transformation process.

8.3.2 Package 'dataTypes'

The classes in the package *dataTypes* define basic data types that are used to implement the operations specified by our security pattern language. As introduced in chapter 5.3, variables of type set, long, and boolean are required to implement these operations.

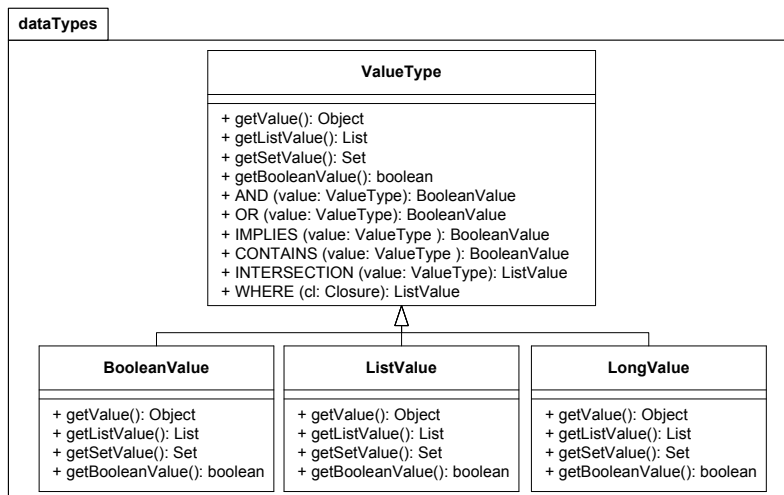
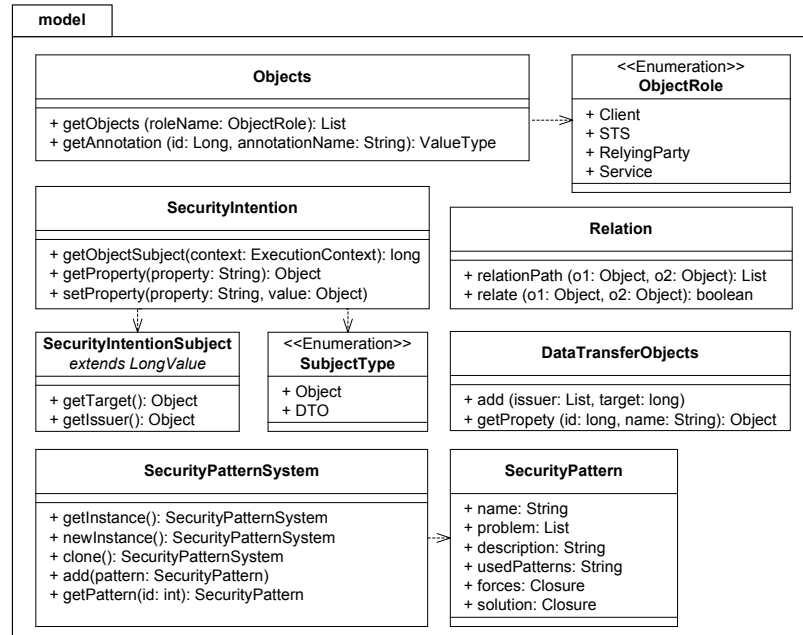


Figure 8.4: Package *dataTypes*

The classes *BooleanValue*, *LongValue*, and *ListValue* provide implementations for these enhanced data types. Each data type class implements the boolean operators (AND, OR, IMPLIES) and set functions (CONTAINS, INTERSECTION, WHERE) that are specified by our security pattern DSL.

To provide a common interface for data type classes, the abstract class *ValueType* is used as a parent class that defines these operations. In addition, this class provides an interface with multiple methods to access a value of a data type instance. For example, the method *getBooleanValue()* is implemented by all child classes. An instance of the class *BooleanValue* returns its current value, while a *LongValue* instance will evaluate if its value is equal to zero. The usage of this interface guarantees that functions and sets defined by our domain-specific language can be used seamlessly to state security configuration patterns.

8.3.3 Package 'model'

Figure 8.5: Package *model*

The package *model* provides classes to represent components of the relational SecureSOA model and the security pattern system. Instances of these classes are stored in the execution context to facilitate the transformation of security intentions.

The *Objects* class is provided to store information concerning the objects used in a model, their roles and related object annotations. To query a set of objects that share a specific role (enumeration *ObjectRole*), the function *getObjects(roleName: ObjectRole)* is provided. In addition, the function *getAnnotation* is offered by the *Objects* class to return a set of annotations that are associated with a specific object. Trust and interaction relations are defined in SecureSOA to represent security dependencies between objects. The class *Relation* is provided in the package *model* as a generic class to represent these dependencies. This class offers a set of methods to evaluate the relations between object. Interaction relations involve the exchange of information that is represented as data transfer objects in SecureSOA. The class *DataTransferObjects* is used to store information regarding these modelling elements. This class provides a method to access the properties of data transfer objects such as the issuer and target.

The *model* classes introduced so far provide context information for security configuration patterns. These classes are used to evaluate the patterns' forces and to apply the solution. Moreover, the class *SecurityIntention* is used to represent a security intention, which identifies the problem that is addressed by a security pattern. Instances of this class are generated from the SecureSOA model and are passed to the security pattern engine (cf. Figure 7.2). To provide a data structure for security patterns and the security pattern system, the package *model* contains the classes *SecurityPattern* and *SecurityPatternSystem*.

8.3.4 Package 'dataAccess'

To enable the instantiation of *model* classes representing the execution context, the package *dataAccess* defines interfaces for factory classes. The *FactoryManager* class is provided to get an instance for a required factory interface. This class uses the system properties to resolve a factory implementation. If no factory implementation is specified in the system properties, a default implementation providing test data will be instantiated. The unit tests use these default factory classes, while the SOA Security LAB provides its own factory class implementations to facilitate the generation of the execution context.

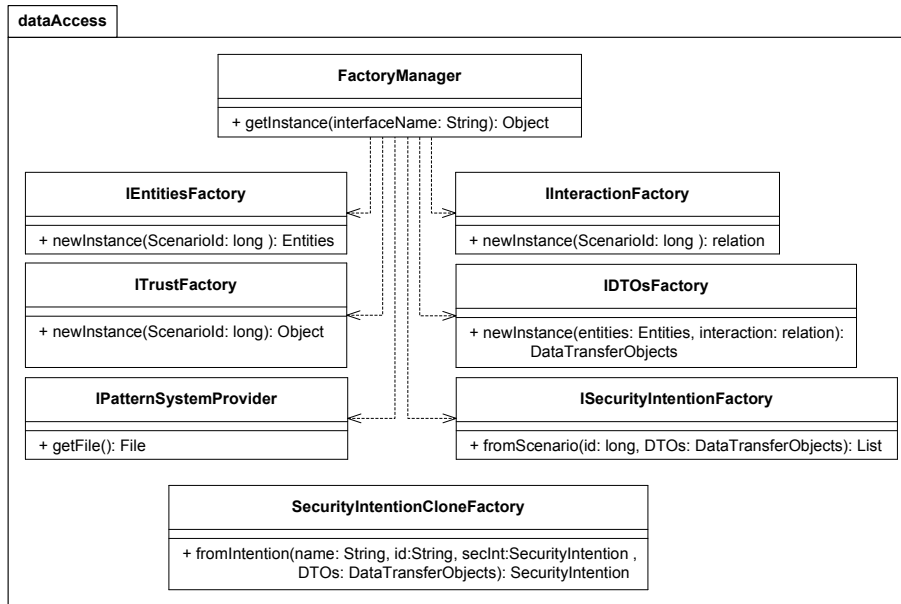


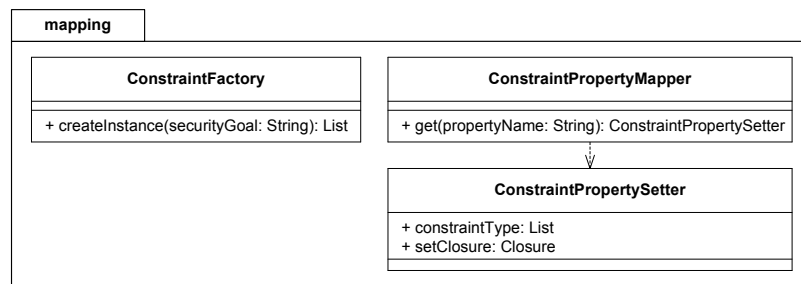
Figure 8.6: Package *dataAccess*

8.3.5 Package 'mapping'

The application of security patterns results in the creation of security constraints that are specified by our domain-independent policy model. The implementation of this model [WMM10, War10] provides a package that contains classes for each policy model element introduced in chapter 3.2. Since security configuration patterns are based on the statement of high level security requirements (e.g. *REQUIRE('confidentiality')*), mappings to specific security constraint types and properties are required. Classes that facilitate these mappings are provided in the package *mapping*.

The class *ConstraintFactory* provides the method *createInstance* that enables the instantiation of security constraints. A string that denotes a security goal is passed to this method to identify the required constraint type. This factory class is used to implement the operation *REQUIRE* specified by our security pattern DSL.

In addition, our security pattern DSL offers operations to update properties in constraints (*SET/USE* operations). To set a constraint property, instances of additional classes might be

Figure 8.7: Package *mapping*

required that are defined by our security policy model. For example, an instance of the class *credential* is required to set a specific security token type in an authentication constraint. These operations are encapsulated in closures that are managed by the class *ConstraintPropertyMapper*. For each property that is used in a constraint, a closure is stored in combination with the constraint type the closure is applicable to. The class *ConstraintPropertyMapper* provides the function `get(propertyName: String)` to resolve a closure that is capable to update the specified constraint property.

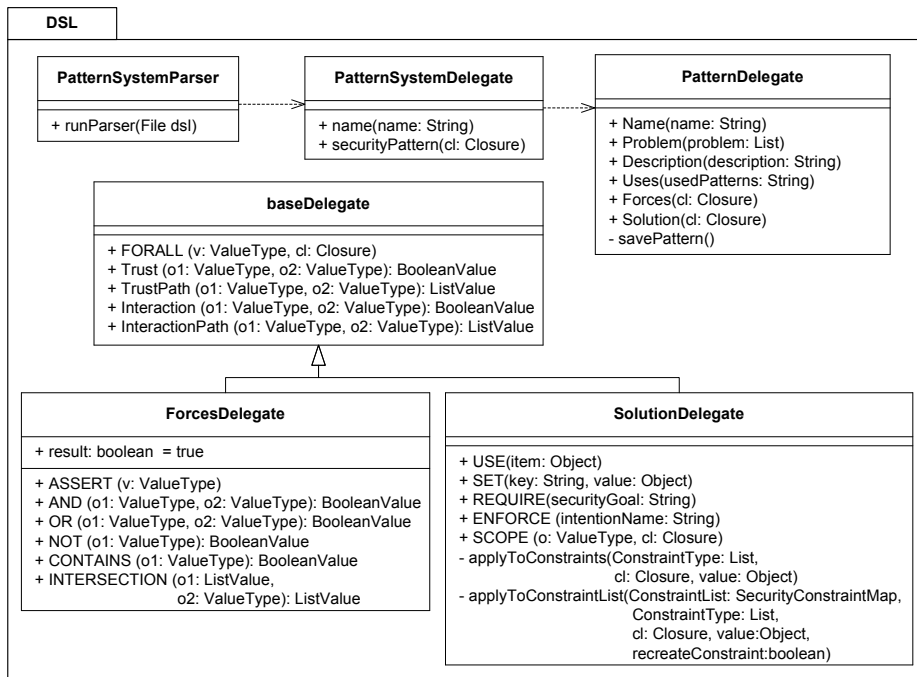
8.3.6 Package 'dsl'

Each Groovy object is assigned to a meta-class object that handles method calls executed on that object. To enable the customisation of method calls, groovy supports the assignment of a custom meta-class object that support the enhancement of objects with additional methods. This mechanisms facilitates the definition of domain-specific languages. For each closure that is used in a DSL script, an instances of a delegate class can be assigned that provides an implementation for invoked methods. The delegate classes that implement our security pattern DSL are defined in the package *dsl* that is illustrated in Figure 8.8.

The class *patternSystemParser* provides a parser to import a security pattern system document, which contains a script that specifies a pattern system. This script contains one or more *patternSystem* operations that accept a closure as input parameter providing operations for security pattern definitions. To execute the pattern system script, the *patternSystemParser* adds an instance of the class *PatternSystemDelegate* to the *patternSystem* operation as a delegate class. This delegate instance is used to execute the closures provided to *patternSystem* operations and contains a method `securityPattern` that is invoked for each *securityPattern* operation used in a *patternSystem* closure.

An instance of the class *PatternDelegate* is added as a delegate to closures passed to the `securityPattern` operation. Each of these closures invokes the operations *Name*, *Problem*, *Description*, *Uses*, *Forces*, and *Solution* to specify a security pattern. The information provided by these operations is stored in an instance of the class *SecurityPattern* that is added to the singleton *SecurityPatternSystem* (package *model*).

The class *SecurityPatternSystem* serves as a storage for security pattern definitions used by the security pattern engine. Each security pattern specifies forces and a solution to enable the verification of the pattern's applicability and to facilitate the execution of this pattern. A pattern's forces and solution are specified by closures that contain a set of operations as

Figure 8.8: Package *dsl*

specified by our security pattern DSL. To enable the execution of these scripts, the delegate classes *ForcesDelegate* and *SolutionDelegate* are provided in the package *dsl*. These classes are based on the class *BaseDelegate* that provides common operations and functionality.

The *baseDelegate* class offers implementations for the DSL functions *Trust*, *TrustPath*, *Interaction*, and *InteractionPath* that are used to evaluate trust and interaction relations between objects. In addition, the DSL operations *FORALL* and *SCOPE* are specified that can be used in the patterns' forces and solutions. The class *ForcesDelegate* is a child class of *baseDelegate* and implements the DSL methods *ASSERT*, *AND*, *OR*, *NOT*, *CONTAINS*, and *INTERSECTION* that are used to evaluate a patterns' forces. The forces will evaluate as true, if all *ASSERT* operations return true. The *ForcesDelegate* class tracks the results of these operations and provides the property *result* that determines the pattern's applicability.

Moreover, the class *SolutionDelegate* is a child class of *baseDelegate* that provides implementations for the DSL operations *USE*, *SET*, *REQUIRE*, and *ENFORCE*. These operations are implemented in accordance with the concepts described in section 7.1.2.4 and are used to create and modify security constraints that are organised in three sets: the *New Constraint* set, the *Parent Constraint* set, and the *Child Constraint* set (cf. section 7.1.2.3). To represent these sets, three instances of the class *SecurityConstraintMap* are used that is provided by the *core* package. The implementation of the *REQUIRE* operation is based on the class *ConstraintFactory* (package *mapping*). This operation is used to instantiate security constraints that are added to the *New Constraint* set. Moreover, implementations are provided for the *SET/USE* operations that modify properties of constraints contained in all constraint sets. The private functions *applyToConstraints* and *applyToConstraintList* facilitate this task and leverage the class *ConstraintPropertyMapper* (package *mapping*).

8.3.7 Package 'core'

The package *core* provides classes that implement our pattern-driven transformation process. This process is executed by the *PatternEngine* class that represents the main class of our implementation. The *PatternEngine* class offers the method *generatePolicyModel* that translates security intentions to our security policy model. This method creates an execution context (class *ExecutionContext*) using the *ExecutionContextFactory*, invokes a new instance of the *PatternApplication* class to transform the security intentions to security constraints, and generates an instance of the security policy model that contains alternative sets of security constraints.

The *ExecutionContextFactory* leverages the package *dataAccess* to create a new instance of the class *ExecutionContext*. The execution context contains instances of the *model* classes that represent components of the SecureSOA model and the security configuration pattern system. In addition, a reference to an instance of the *SecurityProfile* class is provided that stores the security profiles (cf. chapter 4.2.1.1), which has been passed to the pattern engine. This class provides the method *getEntries* to query a list of security mechanisms for a specific security concept. The *SecurityOntology* class is used to identify security mechanisms in a profile that relate to this concept (cf. chapter 5.4). The *SecurityOntology* instance is initialised with concepts and predicates (relations) that represent our security ontologies for identity management and data security introduced in chapter 6.1.2 and in chapter 6.2.2.

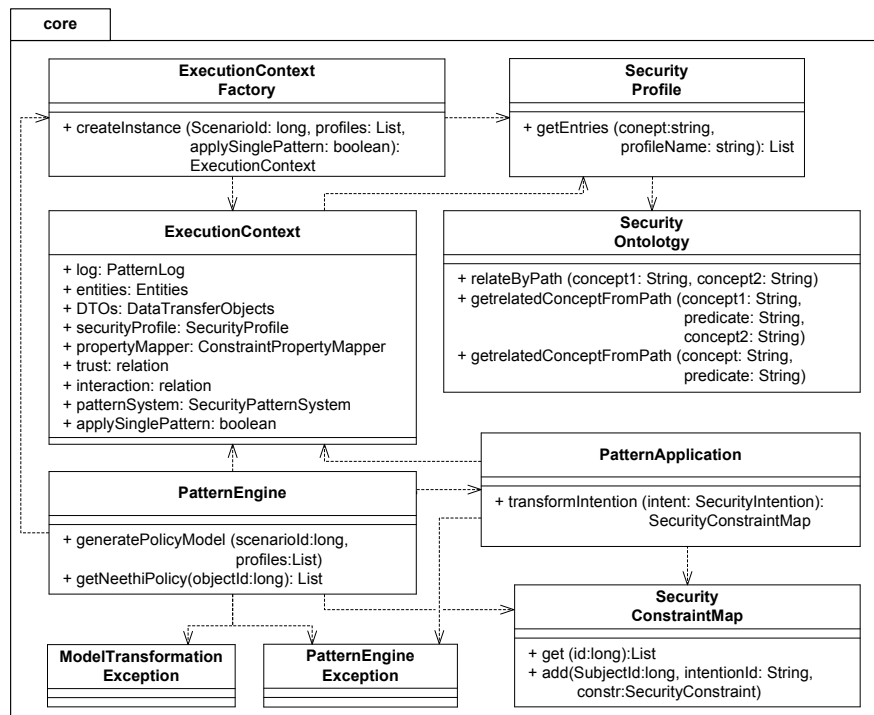


Figure 8.9: Package *core*

The *PatternApplication* class operates on the execution context and generates security constraints that are stored in an instance of the class *SecurityConstraintMap*. This class stores security constraints in combination with an identifier that represents the constraint's position in the

pattern application tree (c.f. chapter 7.1.2.2). The identifiers facilitates the assignment of security constraints to security alternatives that are combined in an instance of the security model as described in chapter 7.1.3.

The *transformIntention* method provided by the *PatternApplication* class has been introduced in chapter 7.1.2. This method retrieves all patterns from the *SecurityPatternSystem* instance that address the required security intentions. For each closure that is defined as these patterns' forces, an instance of the delegate class *ForcesDelegate* is configured with the current execution context. Then, the forces of each pattern are executed to determine the patterns' applicability.

Finally, the solution of each applicable pattern is executed. A delegate class is configured for each solution closure that provides the implementation for the operations specified in a solution. The security constraints created so far are passed to the solution delegate and are stored in the solution delegate's *parent constraints* set. The execution of a solution closure results in the invocation of the DSL operations implemented by the solution delegate. To implement the operation ENFORCE that requires the enforcement of an additional security intention, the solution delegate creates a new *PatternApplication* instance to invoke the intention transformation process recursively. The constraint set of the new *PatternApplication* instance is initialised with the *parent constraints* set and *new constraints* set of the delegate. Properties of constraints in these sets might be modified by enforcing subsidiary security intentions. Security constraints created by the new *PatternApplication* instance are returned by the ENFORCE operation and are added to the *child constraints* set of the solution delegate. In chapter 7.1.2.3 the security constraint sets and their usage have been illustrated and described in detail. After executing the solution closure, the delegate returns the constraints of the *new constraints* set and the *child constraints* set as the result of the solution execution.

8.4 SOA Security LAB Usage

The SOA Security LAB implements the use cases introduced in this chapter to model, execute, and analyse Web Service scenarios. To enhance the usability of our LAB, a wizard is integrated that guides users through this process. Figure 8.10 illustrates the activities in the overall process.

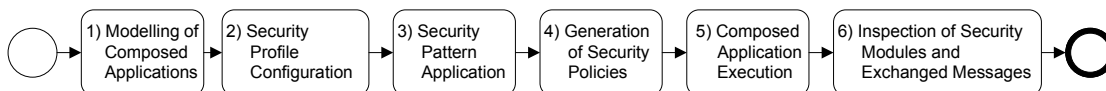


Figure 8.10: Process to Model, Execute, and Analyse Web Service Scenarios

Each activity is supported by a wizard step that provides a view for a use cases introduced in section 8.1. These wizard steps are used as follows:

1. **Modelling of Composed Applications** – The first wizard step enables users to create a composed application by modelling the structure of the desired system (use case *UC 1.1*, cf. Figure 8.1). Moreover, security related aspects such as security intentions and trust relationships can be modelled as well. This wizard step integrates the web-based modelling tool Oryx [DOW08] to model the scenario structure as described in section 8.2. The security design language SecureSOA (see section 1.2.1) is added as a stencil set to this tool.

Figure 8.11 illustrates the modelling of a SecureSOA scenario in the first wizard step. This scenario defines a web shop that is composed of three Web Services. The *Classical Music*



Figure 8.11: Scenario Modelling

Shop Service manages the items that are offered by the *Shop Application*. To handle the payment and delivery of purchased items, the services *Order Processing Service* and *Money Transfer Service* are orchestrated as well. These services are annotated with security intentions to require the authentication of users and the confidentiality of conveyed information. An identity provider is used to manage user accounts. The trust relations indicate that the *Money Transfer Service* relies on authentication statements issued by the identity provider.

To guarantee the instantiation of models in virtual machines, the modelling editor performs a verification of SecureSOA scenarios. In case of modelling errors, the editor highlights related modelling elements. For example, a composed web application must be connected to services that implement required interfaces. Moreover, the modelled security intentions, annotations and trust relationships are verified to ensure that security patterns can be applied to secure a scenario. For instance, the security intention 'user authentication' requires a trust relationship between the service and its clients to enable the authentication.

2. **Security Profile Configuration** – In addition to the system model, the user has to specify security profiles that are referenced by security intentions. As introduced in section 1.2.1, each profile describes security mechanisms that can be used to secure the composite application. The SOA Security LAB provides a high level security profile and a default profile for each scenario. These profiles can be altered in the second wizard step shown in Figure 8.11. For example, a certificate-based authentication can be configured in a profile instead of a password authentication. The SOA Security LAB verifies that a required set of security mechanisms is configured.

The screenshot shows the SOA Security LAB web interface. At the top, there is a header with the SOA Security LAB logo, the text 'Test and visualise service security in the cloud', and a badge for '1st Place at the IEEE Services Cup 2010'. The user is logged in as 'michael.menzel' and has access to 'Account', 'Administration', 'Logout', 'Contact', and 'Imprint'. The main content area is titled 'Composed Application: SuperShop' and 'Step 2: Configure your Security Profile'. A large yellow number '2' is displayed on the right. Below the title, there are 'Previous' and 'Next' buttons. The interface shows a 'Security Configuration Pattern: default' with a table of security mechanisms. The table has columns for 'Use', 'Security mechanism', 'Version', and 'remove'. The 'Use' column contains green checkmarks. The 'remove' column contains the word 'remove'. A description box for 'Basic 192 RSA Algorithm Suite (Basic192)' is visible. Below the table, there is an 'Add mechanism:' dropdown menu with 'User Certificate (Credential)' selected, and an 'Add' button.

Use	Security mechanism	Version	remove	Description
✓	Web Services security protocol (WSS)	1.0	remove	Basic 192 RSA Algorithm Suite (Basic192) RSA algorithm suite that uses 192-bit keys for encryption
✓	Security Assertion Markup Language (SAML)	2.1	remove	
✓	Username/password token (UsernamePasswordToken)	-	remove	
✓	Basic 192 RSA Algorithm Suite (Basic192)	-	remove	

Add mechanism: Add

Figure 8.12: Configuration of Security Profiles

3. **Security Pattern Application** – After specifying the SecureSOA model and related profiles, the pattern engine is used to transform security intentions to an instance of our domain-independent policy model. The result of this transformation is illustrated in the third wizard step illustrated in Figure 8.13. For each SecureSOA object, the security patterns are listed that have been applied.



Figure 8.13: Security Pattern Application

For example, the security configuration pattern *Secrecy* has been applied to the *Order Processing Service* that has been annotated with the security intention *Data Confidentiality*. Since this pattern requires the enforcement of the security intention *Data Protection*, the pattern engine has selected and applied the pattern *Message Protection*. As illustrated in Figure 8.13, these patterns are shown in the list of applied patterns.

4. **Generation of Security Policies** – In the next step, our domain-independent model is transformed to WS-SecurityPolicy documents that can be deployed and enforced at the services and frontends in the virtual machine. These policies are used to enhance the service interface descriptions (WSDLs files) with security requirements. The fourth wizard step implements use case *UC 3.1* (cf. Figure 8.1) and enables users to inspect these interface descriptions and embedded security policies. As illustrated in Figure 8.14, our SOA Security LAB supports the highlighting of metadata parts such as WS-SecurityPolicy assertions.

The screenshot shows the SOA Security LAB web application. At the top, there are logos for SOA Security LAB, IEEE Services Cup 2010, and HPI. The user is logged in as michael.menzel. The main content area is titled 'Composed Application: SuperShop' and shows 'Step 4: Deploy Security Configurations'. A large yellow number '4' is displayed on the right. Below this, there are 'Previous' and 'Next' buttons. A section titled 'Available Services and Frontend:' lists several services: Money Transfer Service, Order Processing Service, Classical Music Shop Service, Identity Provider, and Shop Application. The main part of the interface shows a 'wsPolicy' XML document with syntax highlighting. The XML content includes definitions for a MoneyTransferService and a wsPolicy with various assertions like SymmetricBinding, ProtectionToken, and X509Token. On the right side, there is a tree view showing the structure of the XML document, with 'WS-SecurityPolicy' selected. The interface also includes a 'Show Start' button and a 'Logged in as michael.menzel' status bar.

Figure 8.14: Inspection of Generated Security Policies

5. **Composed Web Application Execution** – The deployment management component shown in Figure 8.2 handles the instantiation and setup of virtual machines (use case UC 2, cf. Figure 8.1). The web frontend and the orchestrated Web Services used in a scenario are deployed at the application server running in a virtual machine. In addition, the security policies created in the previous step are used to configure these components.

The web frontend of the composed application is shown in the fifth step to enable the execution of the scenario. As shown in Figure 8.15, the shop application provides an interface to search for pieces of music. The list of items offered by the store has been provided by the *Classical Music Shop Service*. Moreover, the shopping cart of the store offers the possibility to purchase items by invoking the orchestrated services *Order Processing Service* and *Money Transfer service*.



Figure 8.15: Execution of a Composed Application

6. **Inspection of Security Modules and Exchanged Messages** – The execution of SecureSOA scenarios in virtual machines results in the exchange of Web Service messages. In a final step, the flow of messages is visualised (use case UC 3.2, cf. Figure 8.1) as shown in Figure 8.16. Sequence numbers are assigned to each interaction dependency in the SecureSOA model that refer to the exchanged messages listed below.

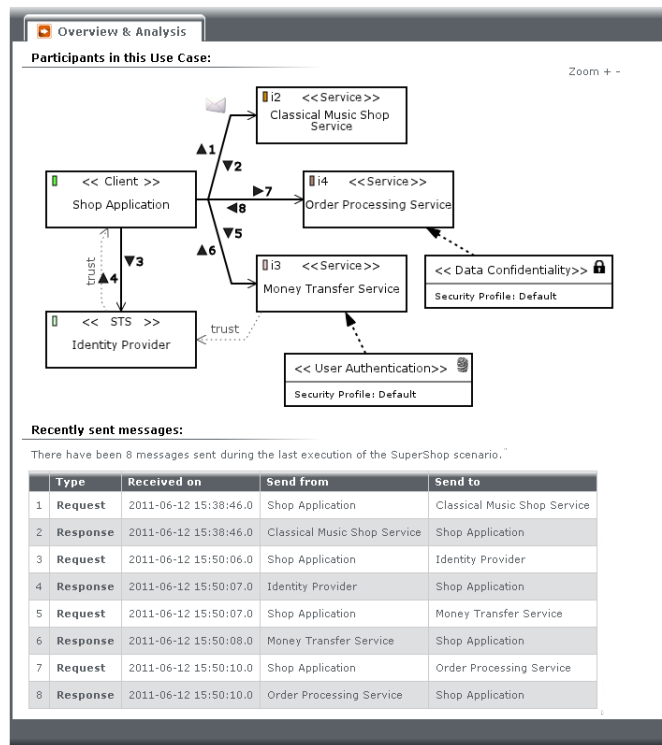


Figure 8.16: Visualisation of the Message Flow

In addition, users can select the Web Services and web application in the model to inspect the security modules that are used to enforce security policies. Figure 8.17 visualises the chain of security modules of the *Order Processing Service* that processes incoming and outgoing messages. An intercepted message is shown that is received from the network and passed to a security handler. In compliance with the security intention *Data Confidentiality*, WS-Security and XML-Encryption are used to secure this message.

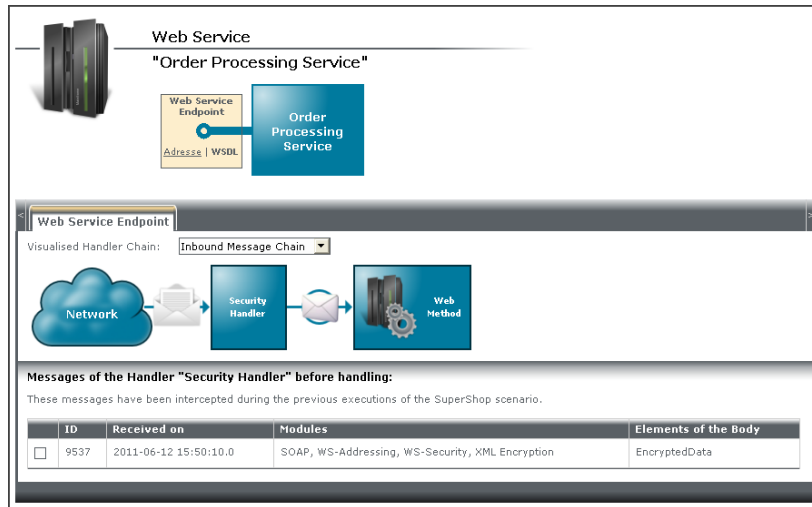


Figure 8.17: Message Handler Chain Illustration

The messages that passed through these modules can be inspected as well (use case UC 3.3, cf. Figure 8.1). Figure 8.18 shows the visualisation of the incoming message in our platform. Our SOA Security LAB analyses visualised messages and highlights identified security protocols and mechanisms.

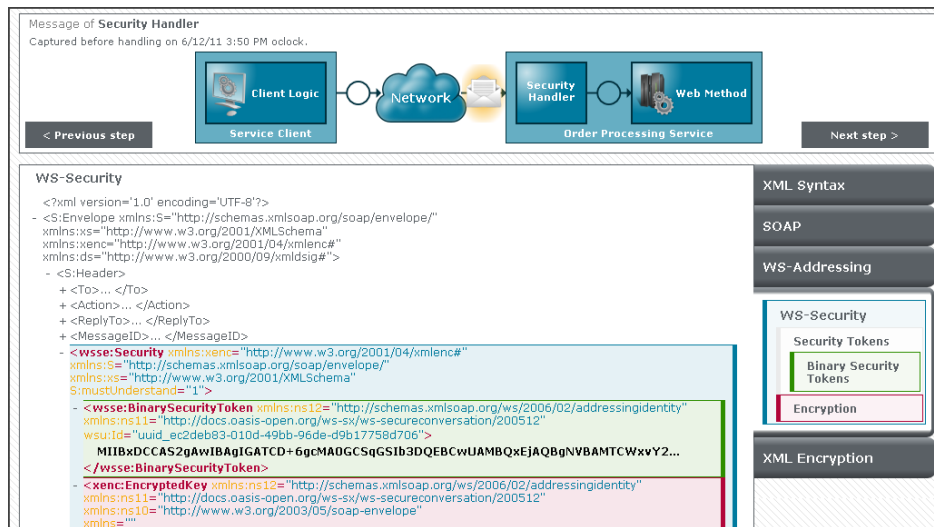


Figure 8.18: SOAP Message Visualisation

Chapter 9

Conclusion

In this thesis, a model-driven methodology has been presented to simplify the creation and handling of security policies for service-based systems. High level security requirements annotated in visual system design models are translated to specific security policies using the security pattern-driven transformation introduced in this work.

Security policies for service-based systems state technical security requirements and capabilities to enable interactions between service providers and service consumers in a secure manner. In particular, these policies facilitate a declarative configuration of services and enable a negotiation of security requirements with service consumers. Consumers can retrieve a service's policies to select a policy alternative to invoke a service securely. In the scope of the Web Service specifications, WS-Policy and WS-SecurityPolicy provide a language to organise security requirements in policies concerning the usage of security mechanisms, methods, and standards in Web Service-based systems. Since a multitude of security mechanisms and options must be required and combined in policies to implement specific security patterns for service-based systems, the creation of security policies is an error-prone task. Strong security expert knowledge is needed due to the complexity of security policy languages.

Since model-driven security promises a solution to these issues by enabling a simplified generation and management of security policies, this topic has been in the focus of security research in recent years. Multiple extensions of modelling languages have been proposed to state and review security requirements on a non-technical layer. These extensions provide the foundation for model-driven approaches that enable a formal verification of security requirements or facilitate the transformation of these requirements to specific security configurations (e.g. access control policies). However, the approaches proposed so far are not capable to address complex security use cases in service-based systems, since they do not provide a generic modelling and transformation framework on the basis of SOA security best practices.

To overcome these limitations, this thesis presented a model-driven approach that introduced a security modelling language to enhance system design models with security modelling elements for service-based systems. Moreover, our approach provides a transformation process that utilises security patterns for SOA to guide the generation of security policies for service-based systems.

The model-driven approach introduced in this work is based on a meta-model for security in Service-oriented Architectures. This model has been presented in this thesis and provides a basic SOA interaction model to facilitate the description of participants and their relations in

SOA. In addition, this model enables the representation of security policies as well as specific security requirements concerning confidentiality, integrity, and authentication. Our SOA security meta-model provides the foundation of our security design language SecureSOA that facilitates the enhancement of any system design modelling language with security artefacts. The system design language FMC block diagrams has been enhanced with SecureSOA to illustrate the modelling of security requirements. Moreover, the SOA security meta-model has been used to specify a domain-independent model that supports our model-driven transformation process by providing an abstraction layer to security policy languages. The model-driven process translates security requirements expressed in SecureSOA models to our domain-independent policy model. A mapping to security policy languages such as WS-Policy and WS-SecurityPolicy enables the generation of specific security configurations.

However, additional knowledge is required to perform the transformation from simple security intentions defined in SecureSOA models to technical requirements stated in our domain-independent model. To drive this transformation, we foster the usage of security patterns that convey the required security expert knowledge in a reusable manner. As common security patterns are stated in an informal way, a formalised pattern structure has been introduced in this work to enable an automated application of security patterns. This pattern structure has been denoted as security configuration patterns. Each pattern provides a solution for a problem in a specific context. A problem is identified by a security intention, while the context of a security configuration pattern is specified by a relational SecureSOA model. The forces of a pattern state conditions on this model to define the applicability of this pattern. The solution provided by a pattern results in the creation and modification of security constraints in the domain-independent model.

Moreover, a domain-specific language has been introduced in this thesis to enable the specification of a pattern's forces and solution in an accessible way. On the basis of our security configuration pattern DSL, a system of security configuration patterns has been presented that addresses the security intentions User Authentication, Identity Provisioning, Data Confidentiality, and Data Authenticity. The formal semantics of the security configuration pattern DSL and SecureSOA enable a formal verification of the transformation process. To illustrate the formal verification of security intention transformations, we have proven that data transfer objects preserve their encryption properties during the transformation process.

Altogether, the model-driven approach introduced in this thesis constitutes a suitable foundation to describe and implement a model-driven transformation of abstract security intentions to enforceable security configurations for service-based systems. To illustrate the applicability of our approach, the SOA Security LAB has been introduced as a model-driven platform to configure, instantiate, execute, and analyse Web Service-based scenarios.

The approach presented in this thesis has been designed to provide a generic framework and methodology to implement model-driven security for service-based systems. So far, security intentions, constraints, ontologies and patterns have been specified to express requirements concerning the confidentiality and integrity of exchanged information and the identification and authentication of users. In future work, SecureSOA can be customized with additional security intentions such as a non-repudiation intention. The enhancement of SecureSOA with additional security intentions requires the provision of additional security patterns, which can be declared using our security pattern language. For example, security patterns for non-repudiation have been listed in [QMT⁺10]. Moreover, the provision of additional security configuration patterns might require the enhancement of our policy model with additional constraints. For instance, a logging constraint might be required to realise non-repudiation.

To prove that the statement of security intentions results in the generation of corresponding security constraints, this thesis illustrated the formal verification of this transformation step. A formal verification of the entire transformation process including the policy generation step was not in the scope of this thesis, but would be a comprehensive topic for future research. Such a verification would require the formalisation of supported policy languages such as WS-SecurityPolicy and the specification of a formal semantic for security constraint attributes.

Another important aspect in the scope of SOA refers to the seamless composition of independent services to complex service orchestrations. The model-driven approach presented in this thesis can be used to generate security policies for composed services as well. Nevertheless, it must be considered that composed services orchestrate multiple other services, which have their own security requirements. For example, specific identity information of service users might be required by orchestrated services that must be conveyed across the service composition to these services. Therefore, composed services must broker the security requirements of orchestrated services to its consumers by including them in the service composition's security policy. However, the aggregation of security requirements retrieved from orchestrated services might result in conflicts due to conflicting requirements. A classification of security requirement dependencies and an approach to determine an aggregation of security requirements are described in [MWM08]. The application of this approach to the model-driven transformation presented in this thesis would facilitate the generation of security policies for composed applications that comply to the required level of security.

Appendix A

Glossary

Data Transfer Object - Data Transfer Objects represent transferred information as introduced by Fowler in [Fow03] and are used to model Web Service messaging. A data transfer object is '*[...] little more than a bunch of fields and the getters and setters for them. [...] it allows you to move several pieces of information over a network in a single call. [...] the data transfer object is responsible for serializing itself into some format that will go over the wire.*'

Domain-independent Security Policy Model - The domain-independent security policy model has been introduced in this thesis to provide an abstraction layer for security policy languages such as WS-SecurityPolicy. It is used to express technical security requirements concerning the exchange of messages between clients and services. A domain-independent security policy consists of a set of policy alternatives. Each policy alternative requires a set of security constraints that describe technical requirements for a specific security goal.

Object - An object represents an actor (e.g. service or client) participating in a service-based communication. Objects consist of a set of attributes and can participate in an interaction with other objects to exchange data transfer objects.

Pattern-driven Transformation Process - The pattern-driven transformation process that has been introduced in this work leverages security configuration patterns to transform high-level security intentions (e.g. User Authentication) to an instance of the domain-independent security policy model.

Policy Assertion - Policy assertions are used in WS-Policy documents [BBea05] to specify a set of domain-specific requirements that must be fulfilled by service consumers.

SAML - SAML (Security Assertion Markup Language) [RHPM06] is an XML-based framework that has been standardised by the OASIS Security Services Technical Committee. It enables the description, issuing, and exchange of identity information.

SecureSOA - SecureSOA is a security modelling language for service-based systems that enables the enhancement of system design modelling languages with high-level security intentions and security annotations. The syntax, notion, and formal semantics of this language have been introduced in this thesis. The enhancement of system design languages is based on the integration schema introduced by SecureUML.

Security Annotation - A Security Annotation is a modelling element of SecureSOA that associates a set of security attributes with objects and data transfer objects to represent security-related capabilities. In the scope of this thesis, the security annotations 'User Directory' and 'Supported Claim Types' are provided to express security capabilities related to identity management.

Security Configuration Pattern - Security Patterns provide reusable security expert knowledge to solve security-related problems. A special class of formalized security patterns (denoted as Security Configuration Patterns) has been introduced in this thesis. These patterns provide rules and operations to create security constraints on the basis of high-level security intentions. A security configuration pattern addresses a specific problem that is described by a security intention and specifies operations to perform the generation and modification of security constraints. The applicability of a pattern depends on the forces of this pattern that specify conditions in the scope of a SecureSOA model.

Security Configuration Pattern DSL - The security configuration pattern domain-specific language (DSL) provides the syntax and semantics of operations that enable the specification of security configuration patterns' forces and solutions.

Security Configuration Pattern System - A security configuration pattern system is a set of security configuration patterns that address the security intentions provided by SecureSOA.

Security Constraint - A security constraint is an element of our domain-independent policy model and is used to specify technical security requirements for a specific security goal. In the scope of this thesis, security constraints are defined for authentication, confidentiality, and integrity.

Security Design Model - A security design model enables the visual modelling of service-based systems with security requirements. It is composed of a security design language that has been enhanced with SecureSOA.

Security Intention - A security intention is a modelling element defined by SecureSOA and is used to annotate objects (e.g. services) and data transfer objects (e.g. service requests) with security requirements. These requirements demand the enforcement of specific security goals or the provision of security related information. In the scope of this thesis, a basic set of four security intentions has been defined: User Authentication, Identity Provisioning, Data Authenticity, and Data Confidentiality.

Security Interaction Model - The SOA interaction model describes basic entities and relations of service-based systems. It provides the foundation of the security modelling language SecureSOA and the domain-independent security policy model that have been introduced in this work.

Security Ontology - A security ontology defines high-level security concepts that provide the vocabulary for the security configuration patterns. In addition, an ontology associates these high-level security concepts with concepts that represent the security mechanisms listed in the security profiles. For example, a security configuration pattern might contain the Operation '*USE(IssuedToken)*', while a security profile might list '*SAML-2.0-Token*' as a supported security mechanism. To enable the execution of the security pattern operation, a security ontology would provide the knowledge that a *SAML-2.0-Token* is an *IssuedToken*.

Security Policy - Security Policies are used in the scope of service-based systems to specify technical security requirements for specific subjects (e.g. services). The syntax of a security policies is specified by a security policy language. Various languages exists (e.g. WS-Security or Rampart Configuration Language) to support different platforms and application domains. A domain-independent security policy model has been introduced in this work that serves as an abstraction layer to security policy languages.

Security Profile - A security profile provides a list of technical security mechanisms (e.g. SAML 2.0) that can be used by the pattern-driven transformation process to translate high-level security intentions (such as 'User Authentication') to technical security policies. Profiles are labelled by a name and are referenced by security intentions, which refer to services that are configured with security policies generated by the pattern-driven transformation process. The security mechanisms listed in the profiles denote algorithms, protocols and standards that are supported by the run-time environment of the configured services. Multiple security profiles can be defined for a run-time environment to group security mechanisms in different security levels. For example, a profile 'low security' might list username/password as a valid security mechanism, while a profile 'high security' would require a certificate-based authentication method.

Security Token - Security Tokens convey user-related information (e.g. username/password or certificates) and are serialised in an XML structure. A basic set of security tokens is specified by WS-Security to enable a secure messaging.

Security Token Service (STS) - A service that implements a WS-Trust interface is denoted as a security token service (STS).

SOA Security LAB - The SOA Security LAB provides a test environment for secure Web Services and supports the on-demand creation and orchestration of composed applications and services. Our platform enables the testing, monitoring and analysis of Web Services regarding different security configurations, concepts and infrastructure components. The model-driven approach introduced in this thesis is used to model security requirements and to simplify the creation of security configurations.

SOAP - SOAP [GHM⁺07] is an XML-based message format to exchange information using HTTP or other protocols. Initially, it has been designed to implement an XML-based remote procedure call. With the advent of additional Web Service specifications, SOAP evolved into a generic messaging framework.

System Design Modelling Language - A system design modelling language provides modelling elements to describe the structure of a service-based system. In the scope of this thesis, FMC block diagrams are used as a system design modelling language to enable the modelling of clients, services and their communication channels.

Web Service - A Web Service provides remotely accessible application components (referred to as Web Methods), listens for certain text-based requests (usually made over HTTP), and reacts to them. Although Web Services are protocol-independent, most Web Service implementations expect their Web Methods to be invoked using HTTP-requests conveying SOAP messages based on XML. Due to the usage of these standards, Web Services are independent of operating systems and programming languages.

WSDL - The Web Service Definition Language (WSDL) specification [CMRW07] provides an XML-based language to describe service interfaces in a platform and protocol independent way. WSDL represents services as a set of endpoints, which offer multiple operations.

WS-Policy - The WS-Policy W3C standard [BBea05] describes an extensible, XML-based grammar to express general characteristics, capabilities and requirements of actors in an XML Web Service-based system. A WS-Policy document provides a series of policy alternatives, while each policy alternative describes a collection of policy assertions. Policy assertions are used to specify requirements that must be fulfilled by service consumers. Additional specifications such as WS-SecurityPolicy provide policy assertions for specific application domains.

WS-Security - WS-Security [NKMHB06] defines enhancements for SOAP to enable a secure messaging in terms of integrity, confidentiality, and authentication. This specification provides a framework to integrate digital signature and encryption methods and is based on XML-Signature and XML-Encryption to enable an end-to-end protection of messages. In addition, it enables the exchange of key and authentication information. WS-Security has been proposed as a standard by Microsoft and IBM in 2002 and was established as an OASIS standard in 2004.

WS-SecurityPolicy - WS-SecurityPolicy [NGG⁺07a] is an OASIS Standard that provides a set of security-related policy assertions for WS-Policy. These assertions express requirements and capabilities concerning the usage of WS-Security, WS-Trust, and WS-SecureConversation.

WS-Trust - WS-Trust [NGG⁺07b] defines a Web Service interface to issue, renew, verify, and cancel security tokens. Security tokens are used to convey user information to services to enable the authentication of users. A service that provides this interface is denoted as Security Token Service (STS).

Bibliography

- [AIS⁺77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobsen, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Lanuage: Towns - Buildings - Construction*. Oxford University Press, New York, 1977.
- [BBea05] Siddharth Bajaj, Don Box, and et. al. Web services policy 1.2 - framework (ws-policy). Online: <http://www.w3.org/Submission/WS-Policy/>, April 2005. [Last accessed: 14. August 2011].
- [BC87] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical Report CR-87-43, Tektronix, Inc., September 1987.
- [BDL06] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: from uml models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, January 2006.
- [Cha06] David Chappel. Understanding windows cardspace. Online: <http://msdn2.microsoft.com/en-gb/library/aa480189.aspx>, April 2006. [Last accessed: 14. August 2011].
- [CJ06] Kim Cameron and Michael B. Jones. Design rationale behind the identity metasystem architecture. Online: http://research.microsoft.com/en-us/um/people/mbj/papers/Identity_Metasytem_Design_Rationale.pdf, 2006. [Last accessed: 14. August 2011].
- [CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0. Online: <http://www.w3.org/TR/wsd120/>, June 2007. [Last accessed: 14. August 2011].
- [De108] Nelly A. Delessy. *A Pattern-driven Process for secure Service-oriented Applications*. PhD thesis, Florida Atlantic University, Boca Raton, Florida, May 2008.
- [DFLP07] Nelly Delessy, Eduardo B. Fernandez, and Maria M. Larrondo-Petrie. A pattern language for identity management. In *Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI '07)*, page 31, Washington, DC, USA, 2007. IEEE Computer Society.
- [DOW08] Gero Decker, Hagen Overdick, and Mathias Weske. Oryx - an open modeling platform for the bpm community. In *Proceedings of the 6th International Conference on Business Process Management (BPM)*, volume LNCS 5240, pages 382–385, Milan, Italy, September 2008. Springer Verlag.

-
- [Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
- [ESP07] Roland Erber, Christian Schläger, and Günther Pernul. Patterns for authentication and authorisation infrastructures. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA 2007)*, pages 755–759, Regensburg, Germany, 2007. Springer Verlag.
- [FDLP06] Eduardo B. Fernandez, Nelly A. Delessy, and Maria M. Larrondo-Petrie. Patterns for web services security. In *Proceedings of the Workshop on Service-Oriented Architecture and Web Services (OOPSLA 2006)*, Portland, Oregon, USA, 2006. ACM Press.
- [FMUW03] Damien Foggon, Daniel Maharry, Chris Ullman, and Karli Watson. *Programming Microsoft .Net XML Web Services*. Microsoft Press, 2003.
- [Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [GHM⁺07] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2 part 1: Messaging framework (second edition). Online: <http://www.w3.org/TR/soap12-part1/>, April 2007. [Last accessed: 14. August 2011].
- [GHR06] Martin Gudgin, Marc Hadley, and Tony Rogers. Web services addressing 1.0. Online: <http://www.w3.org/TR/ws-addr-core/>, May 2006. [Last accessed: 14. August 2011].
- [GJLS09] Sebastian Gajek, Meiko Jensen, Lijun Liao, and Jorg Schwenk. Analysis of signature wrapping attacks and countermeasures. In *Proceedings of the 7th IEEE International Conference on Web Services*, pages 575–582, Los Angeles, CA, USA, 2009. IEEE Computer Society.
- [Gro08] Object Management Group. Meta object facility (mof) 2.0 query/view/transformation specification. Online: <http://www.omg.org/spec/QVT/1.0/>, April 2008. [Last accessed: 14. August 2011].
- [gro09a] ATLAS group. Atlas transformation language 3.0. Online: <http://www.eclipse.org/m2m/at1>, June 2009. [Last accessed: 14. August 2011].
- [Gro09b] Object Management Group. Omg unified modeling language version 2.2. Online: <http://www.uml.org/>, Februar 2009. [Last accessed: 14. August 2011].
- [HB08] Michael Hafner and Ruth Breu. *Security Engineering for Service-oriented Architectures*. Springer Verlag, October 2008.
- [JF09] Meiko Jensen and Sven Feja. A security modeling approach for web-service-based business processes. In *Proceedings of the 7th IEEE Workshop on Model-Based Development for Computer-Based Systems (MBD)*, pages 340–347, San Francisco, CA, USA, 2009. IEEE Computer Society.
- [JFH⁺05] Audun Josang, John Fabre, Brian Hay, James Dalziel, and Simon Pope. Trust requirements in identity management. In *Proceedings of the 2005 Australasian workshop on Grid computing and e-research (ACSW Frontiers 2005)*, pages 99–108, Darlinghurst, Australia, 2005. Australian Computer Society, Inc.
-

-
- [Jon06] Michael B. Jones. The identity metasystem: A user-centric, inclusive web authentication solution. Technical report, Microsoft Corporation, Redmond, WA 98052, USA, March 2006.
- [Jue02] Jan Juerjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on The Unified Modeling Language (UML 2002)*, pages 412–425, Dresden, Germany, 2002. ACM Press.
- [Jue03] Jan Juerjens. *Secure Systems Development with UML*. Springer Verlag, Heidelberg, Germany, 2003.
- [KMW⁺09] Heiko Klarl, Florian Marmé, Christian Wolff, Christian Emig, and Sebastian Abeck. An mda-based environment for generating access control policies. In *Proceedings of the 6th International Conference on Trust, Privacy & Security in Digital Business (TrustBus 2009)*, pages 115–126, Linz, Austria, 2009. Springer Verlag.
- [Lod04] Torsten Lodderstedt. *Model driven security: from UML models to access control architectures*. PhD thesis, Albert-Ludwig University of Freiberg, March 2004.
- [MA05] Michael McIntosh and Paula Austel. Xml signature element wrapping attacks and countermeasures. In *Proceedings of the 2005 workshop on Secure web services (SWS 2005)*, pages 20–27, Fairfax, VA, USA, 2005. ACM Press.
- [MD96] Gerard Meszaros and Jim Doble. A pattern language for pattern writing. Online: <http://hillside.net/patterns/writing/patternwritingpaper.htm>, 1996. [Last accessed: 14. August 2011].
- [MLM⁺06] Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. Online: <http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html>, February 2006. [Last accessed: 14. August 2011].
- [MWM08] Michael Menzel, Christian Wolter, and Christoph Meinel. Towards the aggregation of security requirements in cross-organisational service compositions. In *Proceedings of the 11th International Conference on Business Information Systems (BIS 2008)*, Innsbruck, Austria, May 2008. Springer Verlag.
- [NGG⁺07a] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. Web services security policy language 1.2 (ws-securitypolicy). Online: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html>, Juli 2007. [Last accessed: 14. August 2011].
- [NGG⁺07b] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. Ws-trust 1.3. Online: <http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.doc>, March 2007. [Last accessed: 14. August 2011].
- [NKMHB06] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web services security: Soap message security 1.1. Online: <http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>, February 2006. [Last accessed: 14. August 2011].
-

-
- [QMT⁺10] Dr. Bruno Quint, Michael Menzel, Ivonne Thomas, Dr. Thomas Strtkuhl, and Eva Saar. *SOA Security in der Praxis - Entwurfsmuster für eine sichere Umsetzung*. TeleTrusT Deutschland e.V., Berlin, Germany, September 2010.
- [RFMP06] Alfonso Rodríguez, Eduardo Fernández-Medina, and Mario Piattini. Towards a uml 2.0 extension for the modeling of security requirements in business processes. In *Proceedings of the 3rd International Conference on Trust, Privacy & Security in Digital Business (TrustBus 2006)*, pages 51–61, Krakow, Poland, 2006. Springer Verlag.
- [RFMP07] Alfonso Rodríguez, Eduardo Fernández-Medina, and Mario Piattini. A bpmn extension for the modeling of security requirements in business processes. *IEICE Transactions*, 90-D(4):745–752, 2007.
- [RHMP06] Nick Ragouzis, John Hughes, Rob Philpott, and Eve Maler. Security assertion markup language v2.0 technical overview. Online: <http://www.oasis-open.org/committees/download.php/20645/sstc-saml-tech-overview-2\%200-draft-10.pdf>, 2006. [Last accessed: 14. August 2011].
- [Sch03] Markus Schumacher. *Security Engineering with Patterns - Origins, Theoretical Model, and New Applications*. Springer Verlag, Berlin, 2003.
- [SFBH⁺06] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns - Integrating Security and System Engineering*. John Wiley & Sons, 2006.
- [SNL05] Christopher Steel, Ramesh Nagappan, and Ray Lai. *Core Security Patterns - Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall International, 2005.
- [SS05] Alex Stamos and Scott Stender. *Web Service Security*. Microsoft Press, 2005.
- [TAW05] Peter Tabeling, Rmy Apfelbacher, and Stefan Wappler. Fmc metamodel - the fundamental modeling concepts metamodel explained. Online: <http://www.fmc-modeling.org/>, September 2005. [Last accessed: 14. August 2011].
- [TIN04] Michiaki Tatsubori, Takeshi Imamura, and Yuhichi Nakamura. Best-practice patterns and tool support for configuring secure web services messaging. In *Proceedings of the International Conference on Web Services (ICWS 2004)*, pages 244–251, San Diego, CA, USA, 2004. IEEE Computer Society.
- [TM11] Ivonne Thomas and Christoph Meinel. Conceptual evolution of identity management - from domain-based identity management systems to open identity management models. In *In Digital Identity and Access Management: Technologies and Frameworks*. IGI Global, 2011. (forthcoming).
- [VOH⁺07] Asir S. Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Ümit Yalcinalp. Web services policy 1.5 - framework. Online: <http://www.w3.org/TR/ws-policy/>, September 2007. [Last accessed: 14. August 2011].
- [War10] Robert Warschofsky. Transformation and aggregation of web service security requirements. Master's thesis, Hasso-Plattner-Institut for IT-Systems Engineering, 2010.
-

- [WM08] Christian Willems and Christoph Meinel. Tele-lab it-security: an architecture for an online virtual it security lab. *International Journal of Online Engineering (iJOE)*, 4:31–37, 2008.
- [WMM10] Robert Warschofsky, Michael Menzel, and Christoph Meinel. Transformation and aggregation of web service security requirements. In *Proceedings of the 8th European Conference on Web Services (ECOWS 2010)*, pages 43–50, Ayia Napa, Cyprus, December 2010. IEEE Computer Society.
- [WS07] Christian Wolter and Andreas Schaad. Modeling of task-based authorization constraints in bpmn. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, pages 64–79, Brisbane, Australia, 2007. Springer Verlag.
- [YB97] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *Proceedings of the 4th Pattern Languages of Programming Conference (PLoP 1997)*, Washington, USA, 1997. ACM Press.
- [YWM08] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. A survey on security patterns. *Progress in Informatics*, 5:35–47, 2008.
- [Zim95] Walter Zimmer. Relationships between design patterns. *Pattern languages of program design*, pages 345–364, 1995.

