Network of Excellence

Deliverable D6.2

First Version of the Security Requirements Modelling Language and an Initial Solution for the Scenarios
Abstract

Security aspects are becoming increasingly important starting from the early phases of system development (including requirement analysis). In this report, we illustrate two most prominent paradigms in modeling and analyzing security aspects of a system – i.e., goal-oriented and problem-oriented. Essentially, the security requirements expressed in a goal-oriented model are analyzed from the stakeholders’ intents and trust relations among actors in the system. In problem-oriented models, requirement analysis is focused on deriving possible security threats that might occur. Both approaches can be applied hierarchically. Once security requirements are identified, we can define the system architecture using a series of well-defined security patterns or Service-Oriented Architectures (SOA). We also acknowledge that requirements might change, therefore we support this evolution at requirement, architecture, and system level. We present two cases of change which are known known –existing changes– or unknown unknown –possible changes identified in the future. This report also presents two different approaches to validate a requirement framework empirically: user-centered evaluation and functionality tests. Each approach is complementary one to another. User-centered evaluation is based on a rigorous methodology to obtain outcomes and feedbacks. Functionality tests evaluate the system/prototype as a whole by end-to-end testing.

Keyword List

goal-oriented model; problem-oriented model; security adaptation contracts; security patterns; software architecture; SOA.
## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Type of Change</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1, rev290</td>
<td>Initial ToC of D6.2</td>
<td>Yudistira Asnar &amp; Fabio Massacci (UTN)</td>
</tr>
<tr>
<td>rev332</td>
<td>SW Archi Evolution added</td>
<td>Riccardo Scandariato (KUL)</td>
</tr>
<tr>
<td>rev340</td>
<td>Relation with WP7 added</td>
<td>Riccardo Scandariato (KUL)</td>
</tr>
<tr>
<td>rev380</td>
<td>Initial descriptions on problem-oriented RE</td>
<td>Holger Schmidt (UDE)</td>
</tr>
<tr>
<td>rev502</td>
<td>Reworked descriptions on problem-oriented RE (Chapters 2.1 &amp; 5.2.1)</td>
<td>Holger Schmidt (UDE)</td>
</tr>
<tr>
<td>rev526</td>
<td>Reworked description on problem-oriented RE (Chapter 3.1)</td>
<td>Holger Schmidt (UDE)</td>
</tr>
<tr>
<td>rev529</td>
<td>Reworked description on problem-oriented RE (Chapter 5.2.1)</td>
<td>Holger Schmidt (UDE)</td>
</tr>
<tr>
<td>-</td>
<td>Committed UNITN and ATOS contributions</td>
<td>Fabio Massacci (UTN)</td>
</tr>
<tr>
<td>rev726</td>
<td>Restored the section on requirement evolution with service contracts and reworked Chapter 3.2</td>
<td>J. A. Martín (UMA)</td>
</tr>
<tr>
<td>1.0, rev756</td>
<td>Restored up-to-date version of UDE’s texts</td>
<td>Holger Schmidt (UDE)</td>
</tr>
<tr>
<td>rev910</td>
<td>Inserted updated industrial evaluation and restored links with other WPs</td>
<td>J. A. Martín (UMA)</td>
</tr>
<tr>
<td>2.0, rev1114</td>
<td>Prepared the document for internal review</td>
<td>J. A. Martín (UMA)</td>
</tr>
<tr>
<td>rev1195</td>
<td>Addressed some comments from the review</td>
<td>J. A. Martín (UMA)</td>
</tr>
<tr>
<td>rev1258</td>
<td>Addressed some comments from Marianne Busch’s review</td>
<td>Holger Schmidt (UDE)</td>
</tr>
<tr>
<td>rev1271</td>
<td>Addressed all comments from Marianne Busch’s and Marinella Petrocchi’s reviews</td>
<td>Holger Schmidt (UDE)</td>
</tr>
<tr>
<td>rev1289</td>
<td>Completed the remaining comments throughout the document</td>
<td>J. A. Martín (UMA)</td>
</tr>
<tr>
<td>3.0</td>
<td>Accepted by the internal reviewers</td>
<td>Marianne Busch (LMU) &amp; Marinella Petrocchi (CNR)</td>
</tr>
</tbody>
</table>
Table of Contents

LIST OF FIGURES ................................................................. 9

LIST OF TABLES ..................................................................... 11

1 INTRODUCTION ................................................................. 13

2 REQUIREMENT LANGUAGES .................................................. 15
  2.1 Goal-Oriented .................................................................. 15
  2.2 Problem-Oriented Modeling Languages ............................ 18

3 FROM REQUIREMENTS TO ARCHITECTURES ....................... 21
  3.1 From Requirements to Architectures via Patterns ............... 21
  3.2 From Requirements to SOA via Security Adaptation Contracts .. 22

4 MANAGING REQUIREMENT EVOLUTION ............................... 25
  4.1 General Approach .......................................................... 25
  4.2 Evolutionary and Problem-Oriented Requirements Engineering .. 26
  4.3 Managing Evolution at Goal-Oriented Requirement Framework .... 27
  4.4 Managing Evolution at Architectural Design ....................... 28
  4.5 Managing Evolution at Service Adaptation ....................... 29

5 EMPIRICAL VALIDATION ...................................................... 31
  5.1 User Centered Evaluation Approach .................................. 31
  5.2 Functionality Tests (End-to-End Tests) Approach ................ 33

6 LINKS WITH OTHER WPS .................................................... 37

7 FINAL REMARKS ............................................................... 39

BIBLIOGRAPHY ..................................................................... 41
List of Acronyms

**GRC** Governance, Risk and Compliance process

**OCL** Object Constraint Language

**SAC** Security Adaptation Contract

**SOA** Service-Oriented Architecture

**UML** Unified Modeling Language

**UML4PF** UML for Problem Frames
List of Figures

Figure 2.1: The SI* Plan-Do-Check-Act GCR Lifecycle ............................................. 15
Figure 2.2: The SI* Conceptual Modeling ................................................................. 17
Figure 2.3: The Final Engineering Workflow ............................................................ 18
Figure 2.4: Problem Diagram Annotated with Confidentiality and Performance Requirements using the UML4PF Notation ................................................................. 19
Figure 2.5: Attacker in UML4PF Profile ................................................................. 20
Figure 2.6: Integrity Statement in UML4PF Profile ..................................................... 20
Figure 2.7: Availability Statement in UML4PF Profile .................................................. 20
Figure 3.1: Method for Quality-Based Derivation of Software Architectures .................. 21
Figure 3.2: An example of Initial and refined SI* specification using security adaptors ......... 22
Figure 4.1: Problem Diagram with Evolution Operators in UML4PF Notation ................. 26
Figure 4.2: Example of controllable rule (a), and observable rule (b). ............................... 28
Figure 4.3: Example of combining two observable evolution rules ................................. 28
Figure 4.4: Overview of the synthesis of secure adaptors ............................................. 30
Figure 5.1: Living Labs approach .............................................................................. 32
Figure 5.2: TRAC tool ............................................................................................... 35
List of Tables

Table 2.1: The SI* Method Checklist................................................................. 16
1 Introduction

It is now widely accepted that security should be taken into account since the early phase of the software engineering process [42]. Understanding, modelling, and evolving security requirements is therefore a key step in the process. Among the various approaches, many broad classifications divide them into goal-based, model-based, problem-based, and process-oriented approaches [42, 47]. Among those approaches, the goal-based methods and the problem based methods are the ones closer to early phases. Model-based approaches (such as UMLSec or SecureUML) and process-based approaches are closer to system development and thus fall within the scope of WP7. The goal-based approach focuses on identifying threats to the satisfaction of stakeholders’ goals and trust relations as the base for identifying system vulnerabilities. The problem-based approach presents development as the representation and step-wise transformation of software problems and, in case of security, of trust assumptions. Both approaches are orthogonal to one another as they address different concerns and therefore must be integrated in order to provide a comprehensive solution. We discuss both of them more in details in Chapter 2.

Once the requirements are expressed, a natural next step is to analyze and develop the system according to those requirements. For this purpose, we present two complementary approaches in Chapter 3. On the design phase, we can characterize an architecture conforming with the requirements using a series of well-defined security patterns (Section 3.1). When the system is already partially deployed and some incompatibilities arise due to changes in the components or the environment, then we use a service adaptation approach (Section 3.2).

The next challenge of Future Internet is evolution. Supporting evolution in a principled way is becoming of uttermost importance for larger and larger classes of systems. For instance, long-lived systems (e.g., including smart cards) are characterized by an operational time that is much longer than the development time. Due to their longevity, such systems inevitably face changes in their environment, their (security) requirements, as well as their design and implementation. The Internet of the Future will surely be of such kind.

As a second example, services in the “Internet of Future” will be the result of very complex compositions spanning across multiple administrative domains, and will be operated by a heterogeneous consortium of stakeholders (service providers, platform providers, content providers, consumers) that will dynamically evolve over time, often in opportunistic ways due to business considerations. In this scenario, change is deemed as a normal business condition. For instance, trust relationships among the stakeholders of such services are very likely to change over time, along with other (security) requirements.

It has been observed by Ghezzi [14] that “modern software applications are often embedded in dynamic contexts, where requirements, environment assumptions, and usage profiles continuously change. Changes are difficult to predict and anticipate, and are out of control of the application. In many cases, changes cannot be handled off-line, but they require the software to self-react by adapting its behavior dynamically, in order to continue to ensure the required quality of service. The big challenge in front of us is how to achieve the necessary degrees of flexibility and dynamism required in this setting without compromising dependability.”

In the case of security requirements engineering evolution refers to a process of continually updating the security requirements of systems in responding to changes in the operating environment, the underlying technology, the business needs, or the applicable regulations. Such evolution may involve changes that add, remove, or modify features, redesign the system for migration to a new platform, or that integrate with other application. We discuss how to instantiate the approach in Chapter 4.

In order to evaluate the different modelling languages for security requirements and the approaches to analyze and reason about them, we introduce in Chapter 5 different evaluation techniques. Besides the formal evaluation included in each of the presented approaches to model security requirements, this final chapter focuses in empirical evaluation. This chapter puts special emphasis on a user-centered evaluation approach (in order to assist during the development process and assess user satisfaction) and an automated functionality evaluation based on testbeds.
2 Requirement Languages

In this chapter we illustrate two most prominent paradigms in modeling and analyzing security aspects of a system: goal-oriented and problem-oriented. Essentially, goal-oriented models for security requirements are analyzed from the stakeholders’ intents and trust relations among actors in the system. Goal-oriented models are introduced in Section 2.1. In problem-oriented models (Section 2.2), requirement analysis is focussed on deriving possible security threats that might occur.

2.1 Goal-Oriented

The presented goal-oriented approach is based on the SI* approach [16] as refined by a number of later research. The methodology considers as a “system”, a more complex construction and namely a socio-technical system [44] where both social and technical aspects, and its relations are considered as an integrated whole. The method as a whole [3] assumes that a security initiative is a long-life program that an organization needs to carry on (i.e., not a project-based initiative) based on the Deming management cycle of plan-do-check-act [12] which we show in Figure 2.1.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Define Target System</td>
<td>• Develop Control Mechanisms</td>
</tr>
<tr>
<td>• Analyze Security and Risk Concerns</td>
<td>• Test and Deploy Control Mechanisms</td>
</tr>
<tr>
<td>• Analyze Required Controls</td>
<td>• Execute Control Mechanisms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SI-GRC Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
</tr>
<tr>
<td>• Review Performance &amp; Effectiveness Controls</td>
</tr>
<tr>
<td>using Indicators</td>
</tr>
<tr>
<td>• Review Current Business Settings</td>
</tr>
<tr>
<td>and Security &amp; Compliance Req(s)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Act</th>
<th>actor is an autonomous entity that has its own intentions (human and software), capabilities, and entitlements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Improve Existing Controls</td>
<td>Goal is a state-of-affair that an actor intends to achieve;</td>
</tr>
<tr>
<td>• Reorganize Existing and New Controls</td>
<td>Process is a means to fulfill a goal, to furnish a resource;</td>
</tr>
<tr>
<td>• Introduce New Controls</td>
<td>Resource is an artifact that is consumed/produced by a process;</td>
</tr>
</tbody>
</table>

Figure 2.1: The SI* Plan-Do-Check-Act GCR Lifecycle

In the management jargon this is often termed Governance, Risk and Compliance process (GRC). In other words, one needs to consider how to monitor and continuously improve the implemented controls in the system. Table 2.1 shows the basic checklist for the method.

Here we put the emphasis on the requirements phase where analysts model the target system and analyze the security GRC issues and make a first security analysis. WP7 and WP10 take care of the risk and detailed design issues.

The SI* modeling framework [16] is a modeling framework extending the i* framework [51] to support security requirement analysis. In Figure 2.2, the conceptual model of this modeling framework is based on elementary concepts representing requirement analysis. Additional details can be found in the tutorial paper [4].

Actor is an autonomous entity that has its own intentions (human and software), capabilities, and entitlements.

Goal is a state-of-affair that an actor intends to achieve;

Process is a means to fulfill a goal, to furnish a resource;

Resource is an artifact that is consumed/produced by a process;
Table 2.1: The SI* Method Checklist.

- The **Role Identification and Organizational Set-up** specifies the roles and responsibilities for the security team, the business process owners and the IT management.

- The **Specification of the Target of Analysis** captures at high level the socio-technical system with the key relevant information. The analyst should quickly identify the following aspects:
  - **Actor Models** capture relevant actors (i.e., social and technical ones) their organizational structure.
  - **Goal Models** capture the objectives of the organizations by spelling out the intentions and capability of identified actors and the functional interdependency between actors and goals.
  - **Process and Services Maps** identify the processes which achieve the organization’s goals. In this model, one can see the relation between how a goal is being carried out by a business process, and how the business process is supported by series of business services and/or resources. An organization typically has formalized its business process in some formalization (e.g., BPMN, Flow-chart, UML Activity Diagram). This formalization should be re-used as much as possible.

- The **Security and Risk Analysis** captures which aspects of the system are not protected:
  - **Business Continuity Critical Points Identification** specifies the business processes and actors whose commitment is essential to achieve the high-level goals of the organization.
  - **Unauthorized Processing Identification** identifies existing permissions (if known) and how they are delegated to other actors. Provide as output potential unlawful processing and over-entitlement scenarios.
  - **Trusted Computing Base Identification** (TCB) identifies the trust relationships between actors in possibly making references to specific goals. Provided as output the boundaries of the TCB that can be sources of potential failures of reliability and misuses of permission in terms of scenarios for failures of the organizational goals.
  - **Unwanted Scenario Identification** supports the analysts in identifying threat, events and gives guidance how to structure them at resource, process or strategic level.
  - **Risk Assessment** estimates the risk level of identified risks and specifies which risks will be treated or accepted.

- **Control Analysis** identifies the control mechanisms put in place to address the risks.
  - **Control Goal Model** specifies and elaborate the control goals in order to cover most (if not all) risks with appropriate and precise measures.
  - The **Control Processes and Services Map** identifies the control processes that achieve the control goals.

**Event** is an uncertain circumstance that affects a goal satisfaction (in/directly);

**Trust** captures a believe of the capability/honesty of one actor (trustor) to another actor (trustee) in fulfilling/using a business object (trustum);

**Delegation** depicts a transfer of the responsibility/right from one actor (delegator) to another actor (delegatee) in fulfilling/using a business object (delegatum).

The **Target System Modeling** step covers three modeling activities. Firstly, the **Actor Modeling** captures...
actors (i.e., human or technical) involve in the system including their structure in the organization. Analysts then capture and analyze the strategic interests in the Goal Modeling.

During the Security Analysis steps the following activities are carried on:

1. The Unauthorized Processing Identification models actors’ entitlements and the delegation permission to another actor,

2. Trusted Computing Base Identification where capture the trust relationship between actors involved in the system,

3. Unwanted Scenario Identification which identifies potential events that might affect the system as a consequence of the previous two steps.

The first analysis step can then be used to identify more precisely two possible threats which are very important for compliance:

**Lack of Authorization** (or unlawful processing) is present when an actor has been assigned a process or managing a resource without a proper authorization path stemming from the owner. This aspect is particularly critical when demonstrating compliance with privacy legislation;

**Over-entitlement** when an actor has been delegated the permission to access a resource or to execute a process but the latter is not required to achieve the goals assigned to the actor. At minimum this might be a violation of the minimal disclosure principle for the compliance with privacy legislation or might be the source of more serious troubles if the actor can potentially misbehave (e.g., fraud, internal-trading).

Once the model has been specified it can be used for a precise analysis of boundary of the (un)trusted computing base at organizational level:

**Potential Unreliability** might happen when an actor has been assigned a goal but is not trusted to achieve it. This might generate a potential cascading failure for some of the key goals of the actor who delegated this responsibility.

**Potential Misuse** when an actor has been delegated an authorization but is not trusted since it might misuse it.

Analytical techniques are available with the SI* tool vulnerabilities and threats of a SI* model formalized in Answer-Set Programming (ASP) [16, 31].

The objective of the second step is to analyze the trusted actors to either confirm that they are really so (and thus to exclude them for further risk analysis) or to subject them to a deeper risk analysis using
the methodologies developed in WP10 or mapped to the architectures and processes identified in WP7 as we discuss further in the next chapter.

2.2 Problem-Oriented Modeling Languages

Both goal- and problem-oriented requirements engineering approaches cope with the complexity of real-world problems by reducing problems to smaller ones. Goal-oriented approaches hierarchically refine goals to subgoals, whereas problem-oriented approaches decompose a complex problem into a number of parallel subproblems that should be simple and should belong to known problem classes.

In Jackson’s problem frame approach [24], software development problems are represented by problem diagrams. Such diagrams pay special attention to the environment in which the software (called machine) will operate. That environment is represented by means of a context diagram that shows how the environment is structured in terms of problem domains and how the machine can interact with its environment. Interaction between domains is modeled by considering shared phenomena, which are controlled by only one domain and can be observed by other domains. Requirements are optative statements that refer to one or more problem domains and constrain at least one problem domain. Annotating (parts of) context diagrams with requirements yields problem diagrams. Problem frames are abstracted versions of problem diagrams. A simple subproblem of a more complex software development problem can be fitted to a problem frame by instantiating the frame diagram accordingly. Problem frames substantially support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described and reasoned about when analyzing a problem in depth.

We have carried over the problem frame approach to UML\(^1\) and equipped it with tool support. The tool UML4PF\(^2\) [9, 20] is based on a UML profile that supports requirements analysis according to the problem frame approach. The UML4PF profile replaces Jackson’s original notation, and it is supported by an Eclipse\(^3\) plugin that allows software engineers to work with the defined profile. The UML4PF profile is complemented by a large number of validation conditions expressed in OCL\(^4\). These validation conditions express semantic integrity conditions concerning the developed UML models, and they can be checked automatically.

Furthermore, we have enhanced the problem frame approach by taking into account quality requirements, in particular security requirements. Figure 2.4 shows a problem diagram annotated with confidentiality and performance requirements using the UML4PF notation. The diagram describes the problem to send messages in a chat application. Users can send text messages, which should be sent to the other users in that chat room over a network, and which should be shown in the current chat session. This functional requirement is complemented by a performance requirement (stereotype \(<\text{GaCommStep}\>) taken

---

\(^1\)http://www.uml.org/
\(^2\)http://www.uml4pf.org/
\(^3\)http://www.eclipse.org/
\(^4\)http://www.omg.org/spec/OCL/2.0/
from the MARTE profile [45]) and a confidentiality requirement defined using the UML4PF profile [21]. The confidentiality of the text messages has to be preserved for the users against network attackers.

Based on the Security Engineering Process using Patterns (SEPP) [38], we have defined a UML profile for dependable software that provides special stereotypes to express security requirements [21]. For example, Figure 2.5 shows the $\texttt{attacker}$ stereotype to represent potential attackers, Figure 2.6 shows the $\texttt{integrity}$ stereotype to represent integrity requirements, and Figure 2.7 shows the $\texttt{availability}$ stereotype to represent availability requirements. These stereotypes (and further stereotypes for, e.g., confidentiality and authenticity requirements) can be applied to problem diagrams in UML4PF notation.
Figure 2.5: Attacker in UML4PF Profile

Figure 2.6: Integrity Statement in UML4PF Profile

Figure 2.7: Availability Statement in UML4PF Profile
3 From Requirements to Architectures

Once the security requirements are identified, we can define the system architecture using a series of well-defined security patterns or SOA. The former (presented in Section 3.1) allows to implement the architecture from scratch starting from the problem-oriented model whereas the latter (Section 3.2) is able to adapt existing parts of the architecture in order to comply with new requirements.

3.1 From Requirements to Architectures via Patterns

In the literature there are several proposals to link goal-based requirements to architectures by means of security patterns (investigated in WP7). The key idea is that by linking security objectives and security goals to patterns, we can simplify the selection of the right pattern for the problem at hand. For instance, Mouratidis et al. [33] use the Tropos requirements notation to model a security pattern language, allowing them to document the rationale of each pattern. Weiss [48] links security patterns to the soft-goals introduced by the NFR requirements notation [8].

We are now working towards an extension of the approach for Problem Frames. We developed a problem-oriented and model- and pattern-based method that allows software engineers to take quality requirements into account right from the beginning of the software development process [1, 22, 39]. The method comprises requirements analysis as well as the derivation of a software architecture from requirements documents, in which quality requirements are reflected explicitly. Figure 3.1 illustrates the method.

We first decompose the overall problem into subproblems (Problem Diagrams), each of which is related to one or more functional requirements. We annotate each subproblem by complementing functional requirements with related quality (especially security) requirements (Quality Problem Diagrams) using UML4PF as presented in Section 2.2. In the next step, we take a design decision concerning the kind of distribution of the software architecture (Choose Design Alternative). Afterwards, we go back to the requirements descriptions and regarding the design decision split the problem diagrams accordingly (Split Problem Diagrams). Each subproblem must belong to only one of the distributed components. Analogously to splitting the problem diagrams and so splitting the functional requirements, we also have to split the corresponding quality requirements (Split Quality Requirements). Then, we set up an initial architecture by mapping each machine domain in a problem diagram to a component (Initial Architecture). After that, we elaborate the problem diagrams annotated with quality requirements by introducing domains reflecting specific solution approaches (Concretized Quality Problem Diagrams). In the next step, we derive an architecture, which is implementable and achieves the required level of performance and security (Implementable Architecture). We make use of problem diagrams annotated with quality requirements and concretized quality problem diagrams. To obtain the implementable architecture, we refine the architecture through applying design patterns and applying identified solution mechanisms to satisfy quality requirements.
architecture, we first merge related components (Merge Components). Next, we apply appropriate design patterns (Apply Design Patterns). Finally we make use of mechanisms and patterns and the concretized quality problem diagrams (Apply Quality Mechanisms/Patterns).

The UML4PF tool presented in Section 2.2 also supports the transition from requirements engineering to architectural design [1, 39]. More specifically, it allows developers to check semantic integrity conditions in the different models.

3.2 From Requirements to SOA via Security Adaptation Contracts

In highly dynamic environments where part of the system is out of our control, several (possibly incompatible) services are already deployed and changes are expected to occur, requirements need to be adapted to the new situations which arise from those changes and the system must be analyzed and updated accordingly. In this scenario of changing services, both the system evolution and the requirements are usually driven by the desired goals.

In order to pass from goal-oriented requirements to services, we build on our previous work on Security Adaptation Contracts [29] inspired on software adaptation techniques [49, 6]. Software adaptation is a discipline which studies how to better adapt the system to overcome incompatibilities between the services and unexpected changes in the behavior of the services, their performance or the environment. The basic intuition is that, based on an initial SI* specification, we apply a Security Adaptation Contract (SAC) to synthesize a compliant agent, then verify the system (finding possible attacks) and finally including the resulting secure orchestrator (called adaptor) as a new element of the SI* specification.

SACs specify in an unified and concise manner how to overcome incompatibilities in interface, behavior, and security QoS between services. Stateful services (such as Windows Workflows [40] or WS-BPEL [2] processes) are particularly prone to deadlock situations due to these incompatibilities. For instance, an operation with different signature than expected or a sequence of operations which is offered in the wrong order make the orchestration to lock. In addition, SACs are used to automatically synthesize secure orchestrators able to preserve secrecy properties on the system [28].

For the requirements to service transformation we aim at verifying in detail the interactions represented by functional dependencies and security goals in order to find possible attack scenarios and automatically generate new agents, called security adaptors [29]. Every adaptor is described in a declarative-manner by a SAC which specifies the goal of the orchestration, a mapping between the operations (and arguments) of the services, and the restrictions that must be obeyed to fulfil the functional and security requirements. Security adaptors are synthesized based on the SAC corresponding to the SI* specification in a way that the adaptors are able to orchestrate the elements of the system to avoid possible security attacks.

Figure 3.2: An example of Initial and refined SI* specification using security adaptors

Figure 3.2(a) shows an example initial SI* specification. This example is based on a system consisting of three services, the agents represented by circles. One service plays the role of the client which requests
certain data from the second service, the server. Therefore, the client has a functional dependency (the lines labelled with $D_F$) with the server on the data resource (represented by a rectangle). That data must be signed by a third service, the signer. This is a security goal represented by a rectangle with rounded corners. In addition, there is another security goal which states that the request must be only known between the client and the server. Incompatibilities in these services (e.g., some information which is disclosed when it is considered confidential by another party or mismatches in the signature or the protocol) violate the specification as they are not able to fulfil the security requirements. However, the initial specification can be encoded into a SAC in order to synthesize an adaptor which solves these violations and is finally included in the refined SI* specification depicted in Figure 3.2(b). During this process, a detailed description of the attacks found in the initial specification and the final behavior of the adaptor are also obtained.

In order to analyze incompatibilities and possible security flaws we must provide the Crypto-CCS processes [30] of the agents/services. Crypto-CCS processes model the secure conversations between the different agents of the system. This information can be obtained from the documentation or the actual code of implemented services. Otherwise, if the services are being developed at the same time as the system expressed in the SI* specification, then the adaptor synthesis can work based on an initial draft of the interface and behavior of the services and then iterate on this procedure while completing the detailed behavior of the service. Based on the service processes and the SAC which represents the SI* specification, we can automatically verify that the security properties in the SI* specification are preserved and incompatibilities are solved by means of a new agent, the adaptor.

Further details of the approach to synthesize SAC can be found in [28].
4 Managing Requirement Evolution

We assume that requirements might change during the life-cycle of the system. In this chapter we present our approach to requirement evolution (Section 4.1) which is integrated in all the stages previously presented in this paper. Therefore, we support requirement evolution at goal-oriented (Section 4.3) and problem-oriented models (Section 4.2) and in their possible resulting architectures developed using security patterns and Web services (Section 4.4 and Section 4.5, respectively).

4.1 General Approach

A majority of approaches to software evolution has focused on the evolution of architecture and source code level. However, in recent years, changes at the requirement level have been identified as one of the drivers of software evolution [11, 19, 52]. As a way to understand how requirements evolve, research in PROTEUS [36] classifies changing requirements (that of Harker et al [18]) into five types, which are related to the development environment, stakeholder, development processes, requirement understanding and requirement relation. Later, Lam and Loomes [25] discusses the problem of evolving requirements and presented the EVE framework for characterizing changes.

Several approaches have been proposed for supporting requirements evolution. Madhavji [27] proposed a process model for change management (PRISM) to handle the versions of the changed artifacts, to collect change-related data. Han [17] presented an approach to impact analysis and change propagation to software change. Impact analysis and change propagation are performed on software artifacts and their dependences. Software artifacts are represented using augmented EBNF. The impact analysis concerns the introduction, modification, and deletion of software artifacts and dependencies. It is comprised of automatic applying change patterns and interactive confirmation of potential impacts. The change propagation process is a combination of automatic propagation based on codified rules and interactive user guidance, also based on impact analysis results. Zowgi and Offen [52] work at meta level and view requirement models as theory (or a belief set) in some nonmonotonic logic. Requirements are considered as set of belief about the theory (“machine”) going to be built. Requirement evolution are the changes occur when mapping a theory to another theory through a process of rational belief revision. The process begins from an incomplete requirement model, at each step, a new set of requirement changes is brought to bear. After a series of revisions, the requirement model is refined and completed in each step. The limitation of this approach is the overhead in encoding requirement model into logic.

Russo et al. [37] propose an analysis and revision approach to restructure requirements to detect inconsistency and manage changes. The main idea is to allow evolutionary changes to occur first and then verify their impact on requirement satisfaction in the next step. Also based on this idea, Garcez et al. [11] focus on evolving requirement specifications rather than evolving requirement. This work supports modification while preserving particular requirement goals and properties. It proposed the use of a cycle comprised of two phases: analysis and revision. During the analysis phase, techniques of abductive reasoning are used to check specifications if a number of desirable properties of the system is satisfied. If not, diagnosis information is generated. The revision phase uses the techniques of inductive learning to modify the specifications according to diagnosis information generated. Similar to Garcez et al., Ghose's [15] framework is based on formal default reasoning and belief revision, aiming to address the problem of inconsistencies due to requirement evolution.

Other notable approaches include Brier et al.'s [5] to capturing, analyzing, and understanding how software systems adapt to changing requirements in an organizational context; Felici et al. [13] deals with the nature of requirements evolving in the early phase of the system; Stark et al. [41] study the information on how change occurs in the software system and attempts to produce a prediction model of changes; Lormans et al. [26] use a formal requirement management system to motivate a more structural approach to requirement evolution.

Our objective is to model the evolution of requirements in the two main practical cases:

1. when it is known to have happened and we need to re-align the system with the evolution of requirements – the known known
2. when we know it to be possible, but maybe unknown whether it would happen and therefore must understand the best choice for the requirements – the *known unknown*

In order to solve the first problem the problem-based description of requirements is more appropriate: problems have changed of importance (or new problems arised) and we need to adapt the solutions and the architectures. The second problem is better addressed by goal-based approaches: we have different ways to achieve the same goal and we better choose the one that might work best also in the future.

We do not deal with the *unknown unknown* case and namely when the system has to adapt to the changing circumstances. This field belongs more appropriately to the development of autonomic software.

### 4.2 Evolutionary and Problem-Oriented Requirements Engineering

The UML4PF approach presented in Section 2.2 allows one to perform the first steps of software evolution, namely evolutionary requirements engineering and evolutionary design. In evolutionary requirements engineering, new requirements have to be analyzed in the context of a set of already given requirements [10]. The basic idea is to adjust an existing requirements engineering process so that evolution is supported. In a problem-oriented requirements engineering process as outlined in Section 2.2, the original software development problem is decomposed into a number of subproblems that are analyzed according to the problem frame approach [24]. Evolution is performed by defining operators for each process step and each document that is generated in the respective step to incorporate the new evolution requirements into the existing requirements documents or to create additional documents, when necessary. In fact, the evolution task benefits from the chosen problem decomposition.

The described software evolution method is tool-supported: the UML4PF profile has been extended by a set of stereotypes supporting software evolution as well as corresponding OCL constraints. Furthermore, evolution operators that help the engineer in performing the necessary changes in the different steps have been defined. Basically, the evolution operators describe (local) transformations of model elements. These transformations can be described graphically by showing the relevant diagrams before and after the transformation, and formally by stating their pre- and postconditions in OCL. These conditions can then be used as specifications for implementing the operators.

Such a local transformation is shown in Figure 4.1: the evolution requirements introduce a new problem diagram. In this diagram, we can see a new machine domain stereotyped by `≪new_domain≫` and the relevant evolution requirement stereotyped by `≪evolution_requirement≫`. All the domains relevant to solve this sub-problem already exist and are therefore simply added to the problem diagram. The associations between different domains are stereotyped by `≪new_association≫`. The evolution stereotypes

![Figure 4.1: Problem Diagram with Evolution Operators in UML4PF Notation](image-url)
enable the engineer to keep track of the changes made throughout the evolution. After the evolution is finished, the evolution stereotypes are deleted.

4.3 Managing Evolution at Goal-Oriented Requirement Framework

Our approach is based on the explicit representation of different kinds of evolution. Based on the actor who can decide which evolution would happen, we categorize requirement evolutions into two classes:

**Controllable evolution** is under control of designer to meet high level requirements from stakeholder.

**Observable evolution** is not under control of designer, but its occurrence can be estimated with a certain level of confidence.

Controllable evolutions, in other words, are designers’ moves to identify different design alternatives to implement a system. The designer then can choose the most “optimal” one based on her experience and some analyses on these alternatives. In this sense, controllable evolution is also known as design choice.

Observable ones, in contrast, correspond to potential evolutions of which realization is outside the control of the designer. They are moves of reality to decide how a requirement model looks like in the future. Therefore, the stakeholder and designer have to forecast the reality’s choice with a level of uncertainty. The responses are then incorporated into the model.

We capture the evolution in terms of evolution rule. We have defined **controllable rule** and **observable rule** corresponding to controllable and observable evolution.

**Definition 4.1** A controllable rule $r_c$ is a set of tuples $\langle RM, RM_i \rangle$ that consists of an original model $RM$ and its possible design alternative $RM_i$.

$$r_c = \bigcup_{i=1}^{n} \{ RM \xrightarrow{*} RM_i \}$$

**Definition 4.2** An observable rule $r_o$ is a set of triples $\langle RM, p_i, RM_i \rangle$ that consists of an original model $RM$ and its potential evolution $RM_i$. The probability that $RM$ evolves to $RM_i$ is $p_i$. All these probabilities should sum up to one.

$$r_o = \bigcup_{i=1}^{n} \{ RM \xrightarrow{p_i} RM_i \}$$

Figure 4.2 is a graphical representation of evolution rules taken from a case study on the security of service middleware for air traffic management. Left, Figure 4.2(a) describes a controllable rule where a requirement model containing IKMI (RE1) has four design choices: A, B1, B2, and B4. Right, Figure 4.2(b) shows that the initial model ISS-ENT -1 (including RE1 and RE4) can evolve to ISS-ENT -2 (including RE1 to RE4), or remain unchanged with probabilities of $\alpha$ and $1 - \alpha$. These rules are as follows:

$r_c = \{ RE1 \xrightarrow{*} A, RE1 \xrightarrow{*} B1, RE1 \xrightarrow{*} B2, RE1 \xrightarrow{*} B3 \}$

$r_o = \{ ISS-ENT-1 \xrightarrow{\alpha_1} ISS-ENT-2, ISS-ENT-1 \xrightarrow{1 - \alpha_1} ISS-ENT-1 \}$

After all local evolutions at subparts are identified, we then combine these rules into a global evolution rule that applies to the whole model. In the following we discuss how to combine two independent observable evolution rules.

Given two observable rules:

$$r_{o_1} = \bigcup_{i=1}^{n} \{ RM1 \xrightarrow{p_{1i}} RM_{1i} \} \text{ and } r_{o_2} = \bigcup_{j=1}^{m} \{ RM2 \xrightarrow{p_{2j}} RM_{2j} \}$$

Let $r_o$ is combined rule from $r_{o_1}$ and $r_{o_2}$, we have:

$$r_o = \bigcup_{1 \leq i \leq n, 1 \leq j \leq m} \{ RM1 \cup RM2 \xrightarrow{p_{1i} + p_{2j}} RM_{1i} \cup RM_{2j} \}$$
4.4 Managing Evolution at Architectural Design

Changes have a multi-level impact reaching from the requirements analysis down to the run-time configuration of systems. As illustrated by Mens et al. [32], achieving co-evolution between different types of software artifacts is a challenging task that is still open to seminal research. In this context, we are interested in two major challenges: (i) systematically reflect requirements changes in the system’s architecture; (ii) automatically reconfigure the system according to architectural changes, without stopping it.

The proposed solution relies on two key elements:

1. the introduction of change patterns to reflect security requirements changes in component-based models
2. models@runtime to dynamically reconfigure the running system according to changes in the architecture.

Change patterns provide advice to the architect about how to handle a change. A change at the requirements level is captured in a change scenario, which consists of a pair of requirements templates that describe, in a generic way, the situations before and after the anticipated change. To interpret a scenario in the context of a concrete system, a binding needs to be defined in order to link a requirements template to that system’s requirements model. Moreover, the pattern provides a collection of architecture-level solutions that enable the system to respond to the change, while minimizing the effort required to evolve the architecture. Additionally, the principled solutions suggested by a change pattern aim at...
reducing the impact (in terms of disruptive change) of the evolution. This is important when the system that evolves has already been deployed, and recalling the system to carry on major changes to the architecture is prohibitive. The solution contains a generic architectural template, and a transformation based on that template, called the guidance. The architectural template provides a reference point to start applying the change. This way, the guidance can be expressed in generic terms. Again, to apply the guidance in the context of a concrete system, a binding needs to be defined between the architectural template and the concrete system’s architectural model. Finally, the solution describes the feedback, which is a transformation that captures the influence of the architectural changes on the requirements model.

The change patterns approach has been validated by means of an empirical study involving 12 subjects. In the experiment, the required effort for co-evolving the requirements and architecture of a system was reduced by over 40%.

A key step in the run-time reconfiguration process consists of comparing the old and the new architecture. This comparison allows to understand which reconfiguration operations (i.e., change a binding, or add/remove a component) must be performed on the system. On the basis of this comparison, a sequence of reconfiguration commands is generated. The exact command language available for dynamic reconfiguration depends on the possibilities provided by the reflective dynamic middleware platform (e.g., OSGi, OpenCOM or Fractal). For the OSGi component platform, for instance, Kermeta can be used [34].

During the model comparison, the comparator uses an abstract factory to instantiate atomic reconfiguration commands. These commands are not directly executed, and so the running system is not adapted during the model comparison. The commands are temporarily stored and sorted, before the whole sequence of commands is actually executed. That allows to use planning algorithms to sort them.

The key ideas of our solution are illustrated in a position paper by Yskout et al. [50]. That paper focuses on the evolution of trust relationships. The directions for future work include: (a) the instantiation of the approach for a new class of security requirements and (2) integrating the two techniques in a prototype providing the end-to-end management of evolution. Concerning the former, we are exploring the evolution of attribute-based access control policies. In this context, a change to the policy rules might require that more context information is gathered and more application state is maintained, in order to be used in the decision process. This clearly impacts the software architecture and the runtime configuration.

### 4.5 Managing Evolution at Service Adaptation

As regards the evolution of services and their security requirements, we exploit the application of security adaptation contracts to both adapt the system to an updated $SI^*$ specification of the security requirements (Figure 3.2(a)) and refine the specification accordingly.

The overall process is depicted in Figure 4.4. First, we need a Crypto-CCS process defining each of the existing and outdated services. These Crypto-CCS processes allow us to know the detailed interaction between the agents in the specification. With such processes, we can automatically verify that the secrecy properties of the new specification are preserved using partial model-checking techniques developed in [30]. If, because of the evolution, the services are incompatible in signature or behavior, or the secrecy requirements are not met, then the goals and functional dependencies of the new $SI^*$ specification are encoded in a security adaptation contract. Then, this contract drives the synthesis of an orchestrator (called security adaptor) which is able to adapt the current services to the new security requirements. Based on such a contract, the Crypto-CCS processes of the services, and the secrecy property to preserve, a security adaptor is automatically synthesized. The resulting adaptor solves the incompatibilities, meets the secrecy requirements, and it is finally included in a refined $SI^*$ specification (Figure 3.2(b)).

We can iterate over the process above so that we achieve more and more refined versions of the specification and the system is able to adapt to new requirements and changes in the environment.

As regards the evaluation of the result of the adaptation, test beds (Section 5.2) are better suited to assess the effectiveness of the updated system. The reason for this is that the evolution of service orchestrations is usually too frequent and usually transparent to the user. Tests beds, based both on the $SI^*$ specification and the goal of the system, can be executed automatically after each iteration of the adaptation process and they keep up with the fast evolution of these systems.
The most general attacker for the given adaptor, services and secrecy property

Functional adaptor robust against attacks to the given secrecy property

Figure 4.4: Overview of the synthesis of secure adaptors
5 Empirical Validation

This chapter describes two main systems/platforms evaluation methodologies which are widely used in industry:

- User-centered evaluation approach.
- Functionality tests (End-to-End Tests).

5.1 User Centered Evaluation Approach

The user-centered evaluation approach (see ISO 9241-210:2010) defines a general process for including human-centered activities throughout a development life cycle by focusing on users through the planning, design and development of a product. One (iterative) step in this life cycle is concerned with the evaluation of design solutions against the previously specified user requirements.

In this section we describe several user centered evaluation approaches:

- **Usability Testing**: we see one of the best ways to receive feedback on new products and services is to conduct tests with real users, and to try to elicit feedback from them about their reaction to the product, what they like and dislike about it, how they would change or improve it, and what parts they cannot use. This is considered to be usability testing.

  We see usability testing as a branch of Human Computer Interaction (HCI) concerned with a target audience’s ability to come to terms with the intricacies of a product or service. The usability of an interface can be defined as a measure of the effectiveness, efficiency and satisfaction with which concrete users can achieve specified goals in a particular environment with that interface.

- **Low-Fi Prototype**: One of the main advantages of low-fi prototyping is to enable the gathering of feedback in very early stages of the user interface development to ensure that no effort is put into development of interface concepts users cannot deal with. This allows the design to be tested before (extensive) coding and therefore it is easier to change. It is also a well-known fact that testing initially with a rough design which is more focused in functionality aspects than more “polished-looking” prototypes induces more feedback. This enables to detect potential functional and usability problems at an early stage before the development is completed and to get a deeper understanding of the users’ expectations and impressions.

  There are several tools that help developers to create the GUI. One of these tools is Axure RP Pro, which enables rapid visual development and instant generation of interactive HTML prototypes (viewable in most browsers) without coding. The tool enables to make annotations and page notes for clarifying functionality as well.

- **Living Labs**: There are different definitions to explain the concept of Living Labs. One of the most prominent is provided by (European Network of Living Labs1).

  A Living Lab is both a methodology for User Driven Innovation (UDI) and the organizations that primarily use it.

  A Living Lab is about experimentation and co-creation with real users in real life environments, where users together with researchers, firms and public institutions look together for new solutions, new products, new services or new business models. But also Living Labs are about societal involvement, about promoting innovation in a societal basis, involving academia, SMEs, public institutions and large companies in an Open Innovation process that because happens in real environments has an immediate impact. This is how Living Labs aim to contribute to a new Innovation System where users and citizens become active actors and not only passive receivers.

  Living Labs are supported by the European policy, the i2010 Policy Framework promote and support user-driven open innovation methodology in the Directorate General Information Society and Media cutting across the different challenges under the ICT priority:

1[http://www.openlivinglabs.eu/](http://www.openlivinglabs.eu/)
– the different challenges under the ICT priority of the Co-operation programme of the EU 7th Research Framework Programme (FP7);
– the Competitiveness and Innovation Programme (CIP);
– other actions of the European Commission.

The following picture shows the main steps in Living Labs methodology

**Figure 5.1: Living Labs approach**

– **Step 1** “User-centric approach”: The basic idea of Living Labs is to involve the user in the innovation process. The user can contribute with ideas, experiences and knowledge from their daily life and the interaction with the products, services or applications. This approach contrasts with the technology-centric approach. There are two aspects important to consider when interacting with a larger population: the ability to capture the ideas and inputs from a large population and the ability to evaluate and understand technology-use in a specific context.

– **Step 2** “User involvement”: The idea is to involve the users or beneficiaries of the new service/product/idea in all the innovation processes. For instance the end-users can help development teams to define and validate software collaboration tools and they also could contribute to the initial set of requirements focusing on detailed description of use cases.

– **Step 3** “Building local communities”: The innovation groups are composed of local stakeholders including companies, policy makers, public authorities, end-users, with different roles and position (for instance employee and manager).

If we would apply Living Labs evaluation methodology to one specific Future Internet application, as can be Smart Home, then the stakeholders to take in account could be:
- End-user: residential, industrial or commercial end-users.
- Metering appliances providers.
- Energy supplier companies.
- Smart applications providers.
- Data management applications providers.
- Home area networks providers.
- others...

– **Step 4** “Focus on value creation”: The LL methodology is present in all the innovation cycle. The value creation is basic to reduce the pre-commercial gap, but is important to differentiate and try to mix what is valuable for the people and what is valuable for the companies.

– **Step 5** “Pursue openness strategies”: Information should be freely shared, easy to access, not only to members of the living labs, but also to other communities that can be active participants, bringing in new ideas and good practices. Living Labs are open innovation environments.

– **Step 6** “Phasing, cycling and spiral development”: Once the LLs are established (communities created, user involvement ensured, communication and monitoring tools in place), innovation activities can start iteratively, appropriately guided by the organizers’ interventions. The usual evolving phases are:
- User scenarios definition.
- Limited experimentation on simple use cases, in order to achieve quick results and be able to learn jointly and continue the process further.
- Monitoring and evaluation of Living Labs.
Interventions consist in the identification of user needs, problems to be solved, formulation of hypotheses for solution, planning further developments. The outcome is a joint experience on the learning and evaluation of solutions. This is what makes the LL evolve.

- **Usability Questionnaire**: since questionnaires and interviews investigate how users perceive situations of use, it is an effective method to investigate the platform strengths and weaknesses from the users perspective and provide feedback to the developers. Usability enquiries imply standardized questionnaires (e.g., System Usability Scale, PSSUQ, IsoMetrics), interviews and surveys related to usability.

- **Interviews**: Interviews are a suitable method to investigate peoples’ opinions, feelings, emotions and experiences. There is a multitude of interview types available, e.g., expert interviews, semi-structured interviews, group interviews, focus groups, to name just a few.

### 5.2 Functionality Tests (End-to-End Tests) Approach

If we decide to use this approach for making our system or platform evaluation, a Test Case should be created per each case study defined. Validation is the process of evaluating software during, or at the end of the development process, to determine whether it satisfies specified requirements. The Validation Test Suite or Test Case should contain the full set of functional, usability, reliability, performance and supportability test cases that are required to validate and verify the corresponding case studies and scenarios.

In order to create this Validation Test Suite the following activities would be required:

- **Test Case Definition**: The definition of validation test case requirements must consider all the components and modules arising from the scenario under test.

One example of Test Case Template for Future Internet Smart Grid application (for instance Smart Home), could include the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Test Case Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Id</td>
<td>NESSoSSTestCase01</td>
</tr>
<tr>
<td>Test Case Summary</td>
<td>End-User acts as a seller of electricity into the smart grid</td>
</tr>
<tr>
<td>Pre-Conditions</td>
<td>End-User can sell energy to the smart grid</td>
</tr>
<tr>
<td>Modules under test</td>
<td>Consumer home energy display</td>
</tr>
<tr>
<td>Architectural nodes</td>
<td>Smart metering device, ICT Gateway... others</td>
</tr>
<tr>
<td>Steps</td>
<td></td>
</tr>
<tr>
<td>Expected results</td>
<td>User upload energy to the smart grid and receive an incentive</td>
</tr>
<tr>
<td>Overall test case status</td>
<td>Passed, failed, inconclusive</td>
</tr>
<tr>
<td>Priority</td>
<td>Passed, failed, inconclusive</td>
</tr>
<tr>
<td>Severe defect</td>
<td>Yes, no</td>
</tr>
<tr>
<td>Test conducted by</td>
<td>Mr. Mrs</td>
</tr>
<tr>
<td>Test site</td>
<td>Multifamily residence</td>
</tr>
</tbody>
</table>

- **TestBed Definition**: The Testbed is the place (laboratory with the hardware and software infrastructure) where tests are performed in order to check the performance of the platform/system.

Keeping on with the same example than before “Smart Home” application, the TestBed could be located in a Multifamily residence with the following hardware and software requirements:

- Hardware requirements: Metering appliances, smart appliances, ICT Gateway, Home Network, consumers energy display, etc...
- Software requirements: smart applications, data management applications, etc...

One option which could be useful for building the testbed of Future Internet applications is PlanetLab\(^2\) approach. One of PlanetLab’s main purposes is to serve as a testbed for overlay networks.

\(^2\)[http://www.planet-lab.org/about]
Researchers can evaluate and deploy end-user services on top of PlanetLab. Research groups are able to request a PlanetLab slice in which they can experiment with a variety of planetary-scale services, including file sharing and network-embedded storage, content distribution networks, routing and multicast overlays, QoS overlays, scalable object location, scalable event propagation, anomaly detection mechanisms, and network measurement tools. There are currently over 600 active research projects running on PlanetLab.

PlanetLab is a collection of machines (nodes) distributed over the globe and all of them connected to the internet.

**Execution:** This section addresses the issue of assigning a verdict to the execution of a validation case. Once a test case has been performed, the easiest exit criterion to evaluate is the fact that the test has been successfully executed and the verdict is that it has passed. However, when the verdict is that a test case has been executed, but unsuccessfully, a number of factors have to be considered in order to fully evaluate the test.

Three types of verdict will be used to qualify the results of a test case execution: pass, fail and inconclusive.

- **Passed:** means that the test results prove that the observed test outcome gives evidence of conformance to the functionality under testing.
- **Failed:** means that the observed test outcome either demonstrate non-conformance to the functionality under testing or contains at least one invalid validation event.
- **Inconclusive:** means that an event, not related to the component under testing, took place or led to a wrong result. In this case, the validation case result has no significance and can be discarded.

In addition to the verdict described above, the test cases could also have assigned a priority on the level of attention a particular validation case result will require. This section of the test case would only be completed if the validation case status has been set to **failed**. We can set four priorities:

- **Critical/Major:** this test case needs to have immediate attention as it is a vital part of a module's function and may affect the testbed or individual modules.
- **High:** means the validation case that failed will need priority over other cases that are Medium and Low. These are important flaws that will need attention and will be treated after than Critical.
- **Medium:** it refers to test cases with higher priority than Low (since they imply more serious errors) and less than High. These will be treated after than High.
- **Low:** is when it may not affect the testbed or modules but may need to be fixed due to cosmetic or non-critical reasons. Low priorities will always be treated last if there are a number of failed validation cases.

**Reporting:** when the test case fails then one manner of reporting this error to software developers is using some bugs tracking tool such as Trac. Trac is a web-based software project management tool. It provides a number of tools for managing bugs and development. At its core is an interface for a version control system, such as Subversion. This provides visualisation of commits and the differences between them. A Wiki is provided for documentation. Finally an issue database built around the concept of tickets provides a simple means of tracking issues and bugs within a project. Tickets provide a method for assigning tasks, bugs or milestones to particular developers, in order to ensure clear ownership is maintained.

This figure shows the general format of a ticket. Some of the more important elements of the form include the reporter of the issue, its priority, the owner it has been assigned to, the current status, and a full description that can be extended as new information is established.

3[http://trac.edgewall.org/](http://trac.edgewall.org/)
Figure 5.2: TRAC tool
6 Links with other WPs

From problems to patterns. A key connector with WP7 is represented by the joint work by UDE and KUL on the Security Twin Peaks [23]. The requirements specification (describing the problem) and the architectural design (shaping a solution) are carried on concurrently and iteratively, while still maintaining the separation between the problem and solution space. This process of co-developing the requirements and the software architecture is referred to as the Twin Peaks model [35]. Our work presented an elaboration of the original Twin Peaks model in the context of security, called the Security Twin Peaks. By leveraging architectural security patterns, the model provides constructive insights in the process of specifying and designing a security-aware system, by pinpointing interaction points between the software architect's and the requirements engineer's perspective. Originally, we focused on goal-oriented requirements methodologies, like KAOS [46]. We are now working towards an extension of the Security Twin Peaks for Problem Frames. In order to achieve such objective, it is key to unearth the relationships among security problem frames and architectural security patterns. In particular, we are investigating whether a symmetry exists among the two that could be leveraged to create traceability links (a key asset for the Security Twin Peaks). This is the main focus of the collaboration in the context of WP7.

From objectives to patterns. Concerning security patterns (investigated in WP7), there are several existing proposals that suggest to link security objectives and security goals to patterns, with the aim of simplifying the selection of the right pattern for the problem at hand. For instance, Mouratidis et al. [33] use the Tropos requirements notation to model a security pattern language, allowing them to document the rationale of each pattern. Weiss [48] links security patterns to the soft-goals introduced by the NFR requirements notation [8]. In WP7, the usefulness (for the software architect) of adding this links and annotations to the patterns description is quantitatively investigated via an empirical study involving about 100 participants.

From requirements to services. Security adaptation contracts are used both for i) secure orchestration of services with incompatible signature, behavior, and security QoS [7]; and for ii) analyzing possible privacy flaws and automatically proposing possible countermeasures in the form of security adapters [29]. In WP7 we proposed secure adaptation by contract at design time whereas, in WP6, we applied similar technology to verify and refine security requirements in the analysis stage. By means of this verification, new threats are found and possible countermeasures are proposed as new agents of the requirement specification. These agents represent security adapters which are later implemented as security orchestrators or wrappers, hence relating this work once more with D7.2 of WP7.

Industrial evaluation. Since it would be useful to incorporate user needs feedback into the design process, the “industrial evaluation” chapter is related with other NESSoS work packages. A particularly related work package is WP11, where the user cases are defined (eHealth and smart Home) which will be applied throughout the evaluation.
7 Final Remarks

Traditionally, security requirement analysis is often forgotten during a security engineering process. Most practitioners aim at improving the security of their system once the system has been developed or during its architecture design. However, this practice is slowly moving by considering security aspects since the early phase of a system development (i.e., requirement analysis).

In this report, we have illustrated two most prominent paradigms in analyzing security aspects of a system, i.e., goal-oriented and problem-oriented. Both paradigms have their own particularities in terms of their basic concepts, modeling language, process, and possible analyses. Essentially, a goal-oriented approach concentrated on how to analyze security requirements from the stakeholders’ intents and trust relations among actors in the system. In the problem-oriented approach, the analysis starts on analyzing security requirements by deriving possible security problems that might occur when a machine (software) operates in fulfilling users’ requirements. Both approaches can structure their analyses into a hierarchy, e.g., decompose a complex goal/problem into more tangible/simpler subgoals/subproblems.

Once security requirements are identified, we can define the system architecture considering those requirements. One can define the architecture using a series of well-defined patterns; or if the designers decide to use a SOA then they can derive service adaptation contract to be adapted by the service developers. We also acknowledge the fact that requirements are often changing throughout the life of a system (i.e., at the development or operation phase). In this report, we explained how to deal with changes at a requirement, architecture, or system level. Currently, we only support two cases of change which are known known existing changes or unknown known possible identified changes in the future. The report also presents two different approaches to validate a requirement framework empirically: user-centered evaluation and functionality test. Each approach is complementary one to another. The first approach results in very rigour outcomes and feedbacks (due to a rigour methodology) but it is, sometimes, too limited and simplistic (i.e., lab-scale and depend on the evaluation context), while the second approach is less rigour, in term of the methodology, but it evaluates the system/prototype as a whole by end-to-end testing. Note that both approaches might be closely depended on the complexity of the case study.
Bibliography


