D.3.2 METHODOLOGY FOR EVOLUTIONARY REQUIREMENTS

Gábor Bergmann (BME), Elisa Chiarani (UNITN), Edith Felix (THA), Stefanie Francois(OU), Benjamin Fontan (THA), Charles Haley (OU), Fabio Massacci (UNITN), Zoltán Micskei (BME), John Mylopolous (UNITN), Bashar Nuseibeh (OU), Federica Paci (UNITN), Thein Tun (OU) Yijun Yu (OU), Dániel Varró (BME)

Document information

Document Number	D.3.2
Document Title	Methodology for Evolutionary Requirements
Version	2.19
Status	Revision
Work Package	WP 3
Deliverable Type	Report
Contractual Date of Delivery	31 January 2010
Actual Date of Delivery	18 June 2010
Responsible Unit	OU
Contributors	OU, UNITN, BME, THA
Keyword List	
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
1.1	21 September 2009	Draft	Federica Paci (UNITN)	Outline of the deliverable
1.2	6 October 2009	Draft	Zoltan Micskei (BME)	Added subtopics for chapter 4
1.3	4 November 2009	Draft	Federica Paci (UNITN)	First draft of section 3 added
1.4	5 November 2009	Draft	Yijun Yu (OU)	Section 1, 5-7 based on the submitted ESSOS paper, section 2 is newly written
1.5	6 November 2009	Draft	Gábor Bergmann (BME)	First draft of Section 4
1.6	9 November 2009	Draft	Benjamin Fontan (THA)	Add subtopic in section 3 (about DOORS and DSML) Add subtopic in section 5 (Manage Change in DOORS and DSML)
1.7	10 November	Draft	Federica Paci (UNITN)	Add Input for Section 5
1.8	12 November 2009	Draft	Gábor Bergmann (BME)	Elaborated Section 4
1.9	13 November 2009	Draft	Yijun Yu (OU)	Edited the three conceptual models in Section 3 and 5. Added the mapping of concepts in the Thales conceptual models in Section 3 to the general one proposed.

				Fixed the references. Note the new conceptual models are also uploaded as the source file for UMLet.
1.10	13 November 2009	Draft	Charles Haley (OU)	Checked with the conceptual models about Security Goals and Argumentations. Also edited the definitions.
1.11	26 November 2009	Draft	Gábor Bergmann (BME)	Revised evolution rules Conceptual model, added initial example
1.12	7 December 2009	Draft	Yijun Yu (OU) Charles Haley (OU) Bashar Nuseibeh (OU)	Revised the methodology, refined the conceptual models to highlight the contributions. Drafted the change management conceptual model to be consistent with the discussion notes.
1.13	8 December 2009	Draft	Yijun Yu (OU) Thein Tun (OU)	Revised the conceptual models, and checked and edited the executive summary, sections 2 and 3.
1.14	9 December 2009	Draft	Gábor Bergmann (BME)	Minor revisions in text and Figures 8-9.
1.15	16 December 2009	Draft	Benjamin Fontan (THA)	Add subsection 3.1.3 (Security goal Analysis in Thales Context) Add subsection 3.2.2 (Application of

				conceptual model 3.2 in Thales Requirement Workbench) Add subsection 3.3.2 (application of conceptual model 3.3 in DOORS T-REK) Rearrange and simplify section 5 Add definitions in section 9
1.18	18 December 2009	Draft	Federica Paci, Fabio Massacci (UNITN)	New conceptual model for requirements added to Section 3 Structure of section 3 changed. Example added
1.19	21 December 2009	Draft	Charles Haley, Yijun Yu (OU)	Update the text about the changed requirements meta model
1.22	25 December 2009	Draft	Federica Paci (UNITN)	Example with figures updated
	8 January 2010	Review	Ruth Breu (Innsbruck)	Review of the version 1.22 draft received
1.23	12 January 2010	Draft	Elisa Chiarani (UNITN	First Quality Check completed based on version 1.22. Minor remarks added
1.24	13 January 2010	Draft	Federica Paci (UNITN)	Received the reviewing comments from Ruth Breu. Addressed some of the comments on example and Thales

				section
1.25	13 January 2010	Draft	Benjamin Fontan (THA)	Update section 7
1.26	14 January 2010	Draft	Gábor Bergmann (BME)	Added evolution rule example with model manipulation
1.27	15 January 2010	Draft	Thein Tun, Yijun Yu (OU)	Addressed Ruth's comments concerning Sections 1, 2, 3 and 6.
1.28	20 January 2010	Draft	Federica Paci, (UNITN)	Modified Example Added
1.29	20 January 2010	Draft	Gábor Bergmann (BME)	Remade evolution rule example to fit the new concept; also expanded Section 5 to link the two examples
1.30	25 January 2010	Draft	Thein Tun (OU), Yijun Yu (OU)	Fixing the remaining issues of the first quality check
1.31	26 January 2010	Draft	Gábor Bergmann (BME)	Adjusting evolution rules chapter after the reordering.
1.32	26 January 2010	Draft	Elisa Chiarani (UNITN)	Final Quality Check
1.33	27 January 2010	Final	Federica Paci (UNITN)	Final Version with last comments about quality check addressed
2.0	25th May 2010	Revision	Thein Tun (OU)	Revised the structure of the document, based on the discussions at GA in Trento, and subsequent conversations. Added the methodology

				section (Section 2).
2.1	26th May 2010	Revision	Gábor Bergmann (BME)	Began the restructuring of the Evolution Rules section
2.2	27th May 2010	Revision	Gábor Bergmann (BME)	Moved material into the new Example section, fixed references
2.3	27th May 2010	Revision	Gábor Bergmann (BME)	Explained goals of evolution rules. Added formalization with CPs
2.4	31th May 2010	Revision	Federica Paci (UNITN)	Added section 3 on SecMER conceptual model and Section 4 on the evolution conceptual model
2.5	1th June 2010	Revision	Federica Paci (UNITN)	Revised the structure of the document
2.6	3th June 2010	Revision	Federica Paci (UNITN)	Revised the example
2.7	8th June 2010	Revision	Thein Tun (OU)	Revised Figure 1. Accommodated Thales comments/suggestions. Linked sections 2 and 3. Revised security requirement into security goal. Revised the argumentation example per revision 2.6.
2.8	9th June 2010	Revision	Federica Paci (UNITN)	Revised Section 6 by introducing the three perspectives of change and modified text describing the example.

2.9	10th June 2010	Revision	Gábor Bergmann (BME)	Compacted Section 7.1, added new Section 8.5
2.10	11th June 2010	Revision	Thein Tun (OU)	Checked and added figure numbers.
2.11	14th June 2010	Revision	Yijun Yu (OU)	Addressed formatting issues and the remaining comments in revision 2.9
2.12	15th June 2010	Revision	Federica Paci(UNITN)	Comments on the organization of the deliverable and English mistakes and bad wording
2.13	15th June 2010	Revision	Federica Paci(UNITN)	Merge the ontology metamodel with the argument metamodel
2.14	15th June 2010	Revision	Yijun Yu (OU)	Rewrite the summary, adjust the overview figure of the methodology and fix all the figure numbers in Appendix 1.
2.15	16th June 2010	Revision	Gábor Bergmann (BME)	Improved Section 8 according to advice by OU and UNITN Minor change in Sec. 3
2.16	16th June 2010	Revision	Yijun Yu (OU)	Merging 2.14 and 2.15
2.17	16th June 2010	Revision	Federica Paci (UNITN)	English corrections and typos in Executive Summary, Introduction, Section 6 and 8, Modify the appendix
2.18	16th June 2010	Revision	Yijun Yu (OU)	Minor changes to control the section and

				figure numbers
2.19	16th June 2010	Revision	Bergmann Gábor (BME)	Move part of Evolution rule example to Section 7
2.20	17th June 2010	Revision	Bashar Nuseibeh (OU)	Review and copy edit, including quality check changes by Elisa Chiarani
2.21	17th June 2010	Revision	Yijun Yu(OU)	Review and copy edit
2.22	17th June 2010	Revision	Federica Paci (UNITN)	Removal of two comments, Correction of the title of section 7.5

Executive summary

Long lived software systems evolve as their environment changes. When a change happens, security concerns need to be analyzed to re-evaluate the impact of the change on the system and on the assumptions about environmental properties.

Typically, change requests are handled in an ad-hoc way: requirements are described informally in natural language, which is prone to ambiguity and uncertain traceability to the evolving design. There is no explicit means to analyze changes with respect to the security goals underlying the evolution of the system design.

To address these problems in a repeatable and systematic way, we have developed and adopted an iterative security methodology for evolving requirements (SeCMER).

Every iteration of the SeCMER process starts with an elicitation stage that analyzes every change request into incremental changes of requirements models. These models are represented using consistent, state of the art modeling languages, such as Tropos and Problem Frames. Through a unified extension of existing Security Goals frameworks (e.g., Secure Tropos and Abuse Frames) it is then possible to represent specifications in such a way so as to reveal vulnerabilities through a systematic argumentation analysis, based on the facts and rules (propositions) about domain properties.

Using the propositions in the requirements model, the argumentation process analyzes whether the design has exploitable vulnerabilities that might expose valuable assets to malicious attacks. Any facts and domain rules that help identify a rebuttal to the security goals are mitigated by introducing induced changes of security properties from the SeCMER conceptual model.

Reflecting on the concrete results of rebuttals and mitigations in the argumentation analysis, the SeCMER process incorporates automated transformation support based on evolution rules. Every evolution rule can be specified formally by events, conditions and actions (ECA).

From a rebuttal argument, events generalize the facts and conditions generalize the domain knowledge; from a mitigation argument, actions generalize the induced changes to restore the quiescent state of security requirement models.

We illustrate the SeCMER methodology and its iterative process through a concrete example of evolution taken from the ATM domain. The example includes: the SeCMER models before and after changes of introducing the Arrival Manager tool and the SWIM communication system; the argumentation analysis for the security goal of protecting SWIM from malicious man-in-the-middle attack; and the example of evolution rules to generalize automatable monitoring and adaptation to the triggering and reactive changes to the SeCMER models.

At the end of the report, we present the state of practice in processing security requirements, which will be improved by adopting the SeCMER.

Index

1 Introduction

Long-lived software systems often undergo evolution over an extended period of time. Evolution of these systems is inevitable, as they need to continue to satisfy changing business needs, new regulations/standards and the introduction of novel technologies. Such evolution may involve changes that add, remove, or modify system behavior; or that migrate the system from one operating platform to another. These changes may result in requirements that were satisfied in a previous release of a system not being satisfied in its updated version. When evolutionary changes violate security goals, a system may be left vulnerable to attacks [16].

Dealing with changes to security goals poses several challenges, including:

- Ad hoc elicitation of security goals. Most security goals are implicit or are added after security violations have happened, which makes it difficult to prevent security problems and address vulnerabilities in a proactive way;
- Imprecise modeling of requirements. Security requirements, in order to support automation support, demand a formal description that can be used to analyze, argue and evaluate. Vaguely expressed informal natural language descriptions, are difficult for automatic functions to give an assessment of the problem and to provide useful mitigation advices;
- Change management of security requirements is not integrated with risk modeling tools. It requires an explicit mapping between the changes of security requirements and the system vulnerability in order to assess their impact on the system-to-be. Due to the large gap between requirements tools such as DOORS and risk analysis methodologies and tools, mitigation is often a late response to continuous evolution of software systems. Integration of our methodology with WP5 will address this issue.
- Even when changes have happened systematically, there are no mechanisms to argue formally about these changes with respect to the domain knowledge of the system. Will the system collapse due to a subtle change of a trust assumption, for example about the system boundary? Can the system respond to the introduction of a new fact or domain knowledge that often invalidate the existing justification of security? It is important to reach an agreement between stakeholders on the level of security of the system-to-be.

The above difficulties are intertwined in the process of requirements engineering for secure software systems. When addressing these challenges, we propose to start with a well-known engineering principle that is simple enough to deal with different requirement modeling approaches, while at the same time it allows for the high-level analysis of the changes.

According to Zave and Jackson [25], a problem-oriented system requirements analysis involves the understanding of the *indicative* domain properties in the physical world *W* and the specifications of the machine *S*, in relation to requirements *R* that are the *optative* domain properties. Descriptions of phenomena of given (existing) domains are indicative - the phenomena and resulting behaviour can be observed. Descriptions

of phenomena of designed domains (domains to be built as part of the solution) are *optative* - one aims to observe the phenomena in the future.

These relationships between properties establish a structure in order to facilitate the problem analysis. They are captured by the entailment relation: *W, S ├ R*.

In order to extend Zave and Jackson's framework to address security concerns, security-related concepts such as *assets, threats, vulnerabilities, attackers, trust assumptions, risks* and *satisfaction argumentation* [11] must be added. When a system changes, the entailment relation *W, S ├ R* may no longer hold. To be able to re-analyze the security of the system, the processes and rules of changes on the security goal models need to be represented in order to re-establish the satisfaction of *W', S' ├ R'* where *W', S', R'* are respectively the changed domain properties in the description of the problem. Since security goals tend to be hard to guarantee, effective argumentations on the satisfaction of the entailment relation needs to include both positive and negative evidence to establish to what extent the trust assumptions hold and the system boundaries encompass.

In this document, we take the position that changes of security goals can be modeled from three viewpoints, namely,

- A problem-oriented analysis that relates the changes of security goals to both the changes in the specifications and the changes in the environment contexts;
- A sequence of transactions that views changes as transitions of one valid state of the model to another, given that guard conditions, triggering events and the actions can be specified. In particular, these transactions are applied to the change management processes for security risk analysis to include the status indicating at which stage the security problems manifest; and
- An argumentation structure for the claimed satisfaction of security goals by nesting both the positive and the negative evidence in terms of facts, domain-specific knowledge, rebuttals.

Since these viewpoints are related, we identify several possible connections of them. These connections, we hope, will help one obtain a meta-conceptual model that permits description of all changes.

The remainder of this deliverable starts with Section 2, a formal description of the requirements evolution process in general that identifies key problems in dealing with evolving security properties: the assurance that the modified system can maintain all the existing security goals while new security properties need to be introduced to accommodate unexpected changes. In Section 3, the SeCMER process is introduced , to elaborate on the mechanisms for addressing the evolving security requirements problem highlighted in Section 2. An iteration of SeCMER process contains three main steps: Requirements Elicitation, Requirements Evolution, and Argumentation Analysis. Section 4 presents the detailed conceptual model used in a SeCMER process. Section 5 explains rebuttal and mitigation for analyzing the security goals, and Sections 6 and 7 present respectively the conceptual and mathematical model for the evolution rules, and their maintenance through incremental transformations. A detailed account of applying the SeCMER process is provided in Section 8.

2 Problems in Requirements Evolution

Jackson identifies five artefacts in system development -- domain knowledge (W), requirements (R), specifications (S), programs (P) and the programming platform or computer (C) [23] -- and describes their general relationships using the logical entailment operator (├) as follows.

$$W, S \vdash R$$

$$C, P \vdash S$$

The first entailment (W, S ├ R) differentiates between specifications S and requirements R by suggesting that the specifications, within a particular physical (world) context W, imply R. In other words, specifications rely on explicit domain properties in satisfying the requirements. In practice, stakeholders give descriptions of R and S. A problem, in this view of requirements engineering, is the challenge of obtaining a correct specification from the stakeholders.

Similarly, the second entailment (C, P ├ S) differentiates between programs P and specifications S by suggesting that programs, on a particular computing platform C, imply specifications. Programs, therefore, rely on properties of the programming platform in satisfying the specifications.

We view the strength of the logical entailment operator in these formulae to be non-prescriptive: it means that the artefacts (W, R, S, P and C) may be described in varying degrees of formality, from statecharts, temporal logic, etc. to natural language. Likewise, showing that an entailment relationship holds for some given artefacts also may be done to different degrees of formality, from mathematical proofs to informal arguments, depending on the description language chosen and the specific needs of the stakeholders. When formal description languages are used, the proof can be done through logical deduction.

In this sense, the two entailments provide a general framework for establishing and maintaining traceability links from requirements to program code, by factoring out properties of the world and the programming platform. Additionally, the entailments help define responsibilities of various stakeholders. In broad terms, the first entailment is the responsibility of requirements engineers, and the second entailment is that of developers.

Finally, problem structures of software to be developed from scratch have different characteristics from those of software to be developed incrementally by modifying and extending an existing system. In the latter case, appropriate representation of the existing program as a partial solution to the future problem poses an important issue.

In a typical evolutionary development project, there is an existing solution that satisfies current requirements. In particular, there is a problem R_{now} in the present state of the world W_{now}, and a specification of the current machine, S_{now}, to solve the problem such that:

$$W_{now}, S_{now} \vdash R_{now} \quad (1)$$

The current program P_{now}, implemented on a particular computer, C_{now}, satisfies the specification S_{now}:

$$C_{now} , P_{now} \vdash S_{now} \quad (2)$$

Customers of this system want a new system in future, so that:

$$W_{future} , S_{future} \vdash R_{future} \quad (3)$$

and the new system continues to satisfy requirements for the existing system:

$$W_{future} , S_{future} \vdash R_{now} \quad (4)$$

This entailment (4) captures an important property of systems in evolutionary development because its invalidation can tell us whether an existing security goal has been denied by the proposed system.

Customers need a new program, either on the same or a different computer -- we restrict ourselves to the former in this work -- which satisfies the future requirements as specified in S_{future}:

$$C_{now} , P_{future} \vdash S_{future} \quad (5)$$

Importantly, developers do not wish to develop the system from scratch -- that is to say, refine R_{future} to P_{future}. Rather, they wish to reuse P_{now}.

A key question evolutionary development needs to address is that of representing the existing solution. If we take a rather formal view of the development, we may use the following process. First, obtain the new requirements R_{new}, so that R_{now}, $R_{new} \vdash R_{future}$. Since P_{now} is already implemented on C_{now}, describing P_{now} running on C_{now} as some given properties of W_{future} means (i) P_{now} is reused as it is (ii) S_{new} (or specification for R_{new}) has to acknowledge the existence of S_{now} and takes into account potential concerns that may arise from when implementation of S_{new} is composed with P_{now}.

For example, there could be shared variables between S_{now} and S_{new}, and implementation of S_{new} must not invalidate assumptions S_{now} has on those shared variables. Taking such concerns into account, refining S_{new} to P_{new} will lead to a program that will compose with P_{now}, producing the required P_{future}.

This view assumes (i) developers do not modify P_{now} and (ii) P_{new} may be delivered in a single increment. Architecture of certain software such as product-line applications may allow these assumptions, but for other systems, these assumptions are not practical. The alternative approach suggested here recognizes that in evolutionary development projects, P_{now} is usually modified and P_{new} is rarely built in one increment.

Allowing P_{now} to change offers potential benefits. For instance, if the developers know that a complex problem can be solved using the Model-View-Controller (MVC) pattern, the problem maybe decomposed in such a way that the subproblems map to components of MVC.

It should be recognized that P_{now} may be a piece of software that has evolved over time, and its current structure may not facilitate eventual composition with P_{new}. Therefore, structural changes to P_{now} to improve its modularity often simplify composition. As well as the benefits, there are potential risks: it is often difficult to understand the full impact of a particular change.

In the next section, we present in more detail the different entities and relationships to represent the security goals and requirements and the propositions to reason in the argumentation process.

3 The SecureChange Methodology for Evolutionary Requirements (SeCMER)

As mentioned earlier, the **challenges** of addressing evolving security goals arise from multiple facets of engineering problems. ***Existing methodologies*** deal with the changes in security goals with different focuses. For example, Secure Tropos have been used to model both functional and non-functional requirements of stakeholders as ***security goals***. By modeling the delegation and trust relationship among these stakeholders, security problems of a social-technical system are elicited and reasoned about at a high level. On the other hand, Problem Frames approaches for security (e.g., ***abuse frames***) focus primarily on modeling the relationship among the specifications of a software system, the indicative domain properties, and the optative requirements. As a result, patterns relating problems with solutions become reusable for such problem-oriented analysis. Both requirements engineering approaches handle **risk assessment** by extending the basic concepts with relatively new concepts to be able to handle the risk factors of likelihood and impact, and to be able to provide guidance for the mitigation of security problems in terms of *threat*s, *assets* and *damages,* etc.

Although individually these approaches are powerful in modeling and analysis of different perspectives of the security problems, it is easy to see that none of these approaches alone could provide a comprehensive basis to reason about the changes of security goals. Such a combination could **benefit** from the strengths of individual methodology, **making clearer about the situation** of the subject system in terms of security goals. Additional benefits include **enabling** a rule-based evolution support for *transforming* and *maintaining* the unified situations, **extending** a process-oriented *change management* support for documenting the problems in terms of security, and **forming a basis** *for arguing* the security of the life-long system for these documented problems.

In fact, such a comprehensive framework requires **fewer** rather than more concepts. It would be considered a failure for us by simply adding up the existing concepts from different methodologies. Otherwise, it is still hard to combine the different modeling approaches to provide a consistent picture of the situations before or after the changes. Applying such a naïve approach invites inconsistency between these concepts, for the sake of security analysis, the situation could get worse than limiting oneself to applying each methodology separately. Therefore the first step in our methodology involves **identifying equivalent or similar concepts among different conceptual frameworks**. As a result, the combined situation framework has fewer concepts than the simple addition, and they are amenable to advanced analysis of the evolution of the life-long software systems.

The SecureChange Methodology for Evolutionary Requirements (SeCMER) aims to fill this gap.

After the first step, our methodology demonstrates the usefulness of the combined framework that can take advantage of **continuous transformation-based evolution rules** that govern the adaptation of evolving security goals. These evolution rules will

be developed into model-based transformation rules to automate the change process. The contribution of such transformation rules will help maintain the security of life-long evolving system through a **continuous control loop** that is composed of triggering events, conditions of situations and transforming or adaptation actions.

In parallel to the management of changes of requirements situations, security argumentation framework will help to provide **detailed justifications** for the documentation. The truth maintenance combines the change management systems and the argumentation framework through the control loops, implementing a full support at the requirements level for the continuous evolution of life-long software system.

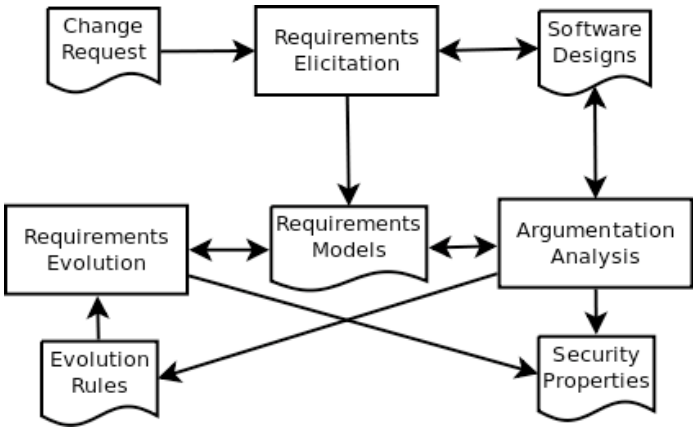

Figure 1. An Overview of SeCMER

In Figure 1, the diagram summarizes the proposed SeCMER process for handling evolutionary requirements in secure software systems.

The inputs to the process in the SeCMER methodology are:

- Change Requests: Informal requests for change made by the users and customer of the secure software system. These requests are typically managed using tools such as Issue Tracking systems.
- Existing Software Designs: Artifacts describing the main components of the systems—software, hardware, and people—their configuration, behavior and properties. They may be documented using natural language text, UML diagrams, or formal descriptions.
- Requirement Models: Statements of properties, including security goals, the existing system satisfies. When changes are implemented, it is necessary to check whether properties of the existing design are satisfied by the new design, and if not, formulate properties that need to be satisfied by the new design.

Focusing on security, the main output from the methodology is therefore either an assurance that the changes did not make the system violate the existing properties, or a formulation of new properties for the new design, namely the Security Properties to be implemented by the new design. Event-Condition-Action evolution rules discovered during the argumentation process can be used to monitor certain changes that can be handled automatically.

The proposed methodology for handling change has three main steps:

1. Requirements Elicitation: When a change is proposed through a change request, the existing design is examined to (a) identify the context of the proposed change, (b) check whether the proposed change is necessary. In terms of the framework described previously, this stage establishes W_{future} and R_{future}. A conceptual model of static requirements (see Section 4 for guidelines) supports this step.
2. Argumentation Analysis: This stage checks whether there are new security properties to be added or to be removed (Δ Security Properties) as a result of changes in the requirement model. Furthermore, a high-level and long-term feedback is possible, in order to adapt/update evolution rules in a way that more human effort can be saved by automation in the future. This stage derives the Δ Security Properties (Δ_{sp}) so that Δ_{sp} U S_{now} ├ S_{future} .This stage is supported by the conceptual model of argumentations presented in Section 5.
3. Requirements Evolution: This stage takes into account the rebuttals and mitigations of arguments and the model of the requirements to produce a model of evolution rules that will automatically establish whether the existing security properties have been broken by the change or not. This stage checks the entailment (4) in the previous section, namely that W_{future} , S_{future} ├ R_{now}. A conceptual model of transformation rules (Section 7) and a conceptual model for Change Requests (Section 6) support this step.

Section 8 presents a complete example to illustrate how the methodology works.

In practice, there are likely to be several change requests at a time, and these requests have to be stored, prioritized, scheduled, resourced, implemented and tested.

4 Security Requirements Elicitation

The first step of SeCMER methodology is the elicitation of the security goals the system-to-be should be built on.

A basic concept that comes into play when eliciting security goals is the concept of *asset*. Assets are target of *attackers* who perform malicious actions (aka *attack*) by exploiting the *vulnerabilities* of the system. Malicious actions compromise security properties of the system-to-be such as confidentiality, integrity and vulnerability. Security goals are, thus, elicited by applying a specific security mechanism to protect an asset from harms that violates a security property.

To identify the security goals of a system it is, thus, crucial to model the assets of the system, the security goals that protect the assets, the malicious intentions of an attacker that can deny the security goals, the malicious actions the attacker carries out, the vulnerabilities the attack exploits, and the negative impact on the assets of the system.

The SeCMER methodology' security goals elicitation step produces a requirements model which is an instance of the SeCMER conceptual model. The conceptual model identifies a set of core concepts that allow linking the empirical security knowledge such as information about vulnerabilities, attacks, and threats to the stakeholder's security goals. To create this link, the conceptual model amalgamates concepts from Problem Frames (PF) [12] and Goal Oriented requirements engineering methodologies (GORE) [13] with traditional security concepts such as vulnerability and attack. The combination of the two security goals engineering approaches has several advantages: with GORE analysis, malicious intentions of attackers can be identified through explicit characterization of social dependencies among actors; with PF security goals analysis, valuable assets that lie within or beyond the system boundary can be identified through explicit traceability of shared phenomena among physical domains and the machine itself.

4.1 The SeCMER conceptual model

The very top of the conceptual model (Figure 2) is adopted from DOLCE [10], a foundational ontology intended to account for basic concepts that underlie natural language and human cognition. Lower levels of the conceptual model include concepts from GORE, PF and argumentation frameworks, with security concepts occupying the lowest strata of the conceptual model. Key among the concepts that are introduced is the concept of *Proposition*, with instances such as ``Want for customers for our business" and ``Paolo is married``. The other key concept is that of *Situation*, representing a partial state of the world, e.g., ``High oil prices``, or ``Unhappy customers are many``.

The most general concept is *Thing*, which has as instances all the things that can exist in the world.

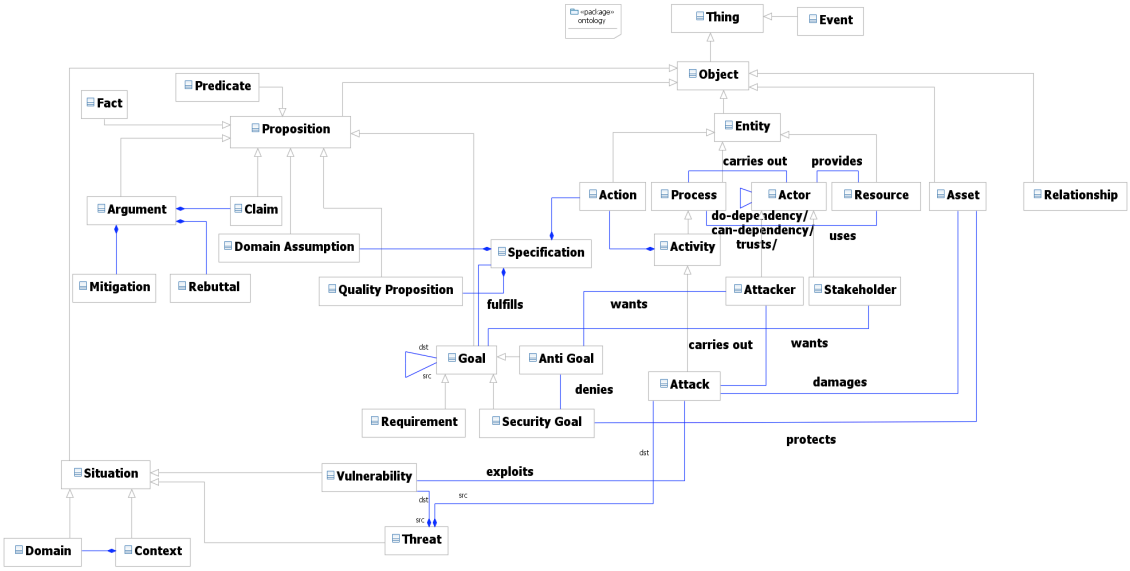

Figure 2. Security Requirements Conceptual Model

An *object* is a thing that persists (endurant in [10]). An *event*, instead, is an instantaneous happening (perdurant) that changes certain objects. Specializations of the concept Object include *Proposition*, *Situation*, *Entity*, and *Relationship*.

A *proposition* is an object representing a true/false statement. A *situation* is a partial state of the world described by a proposition (its description [10]). Arbitrary propositions are true/false/ undefined in a situation, given its partial world status. The status of the world is expressed by a predicate over the entities involved.[1]

Situations can have structure consisting of relationships and things standing in those relationships. Some entities and relationships according to the common sense always satisfy certain predicates, making them strong beliefs or trust assumptions.

Thus, the entities and relationships are modeled to reflect the predefined assumptions about the world being modeled.

An *entity* is an object that has a distinct, separate existence from all other things, though that existence need not be material. Thus, “Santa Claus”, “my cancelled trip”, “a square circle” are entities. A *relationship*, on the other hand, is an object that participates in a certain situation along with other objects (its relata); the existence of a relationship depends on that of its relata.

Entities. Entity is specialized into *Actor, Action, Process, Resource*, and *Asset*.[2] An *actor* is an entity that can act and intend to want or desire. *Stakeholder* and *Attacker* are two important specializations of Actor for the domain of security goals. A *stakeholder* is an actor who has a stake in the system- to-be, while an attacker wants to prevent the fulfillment of the requirements for the system-to-be.

[1] Note that predicates are a special form of propositions, and through reification they can be grounded into sentences of propositions.

[2] Actor, Action, Process, Resource, and Asset are concepts adopted from GORE approaches.

An *action* is an entity performed by an actor, which can generate events, and can have preconditions and post-conditions. A *process* is an entity that generates events and changes objects. *Activity* is a specialization of Process, consisting of actions. *Attack* specializes Activity, and is always carried out by an Attacker. We distinguish between Process and Activity in the conceptual model because we want to allow for processes that do not involve any actions, e.g., a fire burning, or an earthquake. A *resource* is an entity without intention or behavior. An *asset* is an entity of value that can be owned and used. For example, an asset can be an passenger (actor) whose life needs to be protected, can be an engine (process) whose behavior has a value to the protector, or can be an aircraft (resource) whose value are tangible for other actors. A *relationship* such as the organization chart of the air traffic management organization is also an asset as long as its value needs to be protected.

Relationships. Specializations of Relationship include *do-dependency*, *can-dependency* and *trust-dependency* adopted from Secure Tropos. These are all ternary relationships between two actors and an asset. In addition, there are many binary relationships that characterize other concepts in the conceptual model. For example, actors are entities that *want* goals and *carry out* actions. *Composes*, *contributes*, *uses*, and *provides* relationships are also included in the conceptual model. *AND/OR refinement* is a relationship between a goal and two or more other goals that indicates that a goal can be refined into subgoals. *Contributes* relates two goals and indicates that one goal has a positive or negative impact on the satisfaction of the other. *Provides* is the relationship from an actor to a resource, specifying that the actor provides the resource. *Uses* is the relationship from a process to a resource denoting that the process generates or consumes the resource. *Fulfills* relates an entity to a goal that the entity fulfills.

For the sake of security goal analysis, the conceptual model includes also the following specializations of Relationship: *damages*, *exploits*, *protects*, and *denies*. *Damages* is the relationship between an attack and an asset, where the attack causes harm to the asset. *Exploits* is the relationship between attack and vulnerability. *Protects* relates a security goal to an asset. Finally, *denies* relates an anti-goal to a requirement. A complete list of all the possible relationships is found in Figure 3.

Propositions. Proposition is specialized into *Fact*, *Claim*, *Argument*, *Domain Assumption*, *Quality Proposition*, and *Goal*, depending on the different types of proposition modalities. A *fact* is a true proposition. A *claim* is a proposition claimed to be true by an actor. An *argument* is a proposition consisting of a set of claims. A *domain assumption* is a proposition about the domain assumed to be true by an actor. A *quality proposition* is a proposition about the quality of the system-to-be. A goal is a concept found in GORE approaches, and represents a proposition an actor wants to make true. For security analysis purposes, Goal is specialized into *Requirement, Security Goal*, and *Anti-Goal*. A *requirement* is a goal wanted by a stakeholder. A *security goal* prevents harm to an asset through the violation of confidentiality, integrity, and availability security properties [12]. An *anti-goal* is a goal an attacker wants which denies the fulfillment of a requirement of the system-to-be.

Situations. The Context and Domain concepts coming from PF approaches are specializations of Situation. These concepts are useful to define the situation of system boundaries, to allow one place focus on analysis while hide the unnecessary details.

For the analysis of every problem or subproblem, a different situation may be selected from the physical world. Thus the *context* is a situation in which the system-to-be will operate; and a *domain* is a situation that is part of the context. In PF, domains can be classified as biddable, causal, and lexical. By biddable, a domain's behavior is not fully predicable or controllable, usually represented by human actors or natural processes. By causal, a domain's behavior is predicable or controllable, usually represented by activities. By lexical, a domain's behavior is predefined, usually by a resource. Another concept adopted from Problem Frames is Specification. A *specification* is an entity consisting of actions, quality propositions, and domain assumptions. Thus, a collection of indicative propositions is about the entities in the system-to-be.

In the security domain, *vulnerability* is a situation where some actions that are part of an attack can be carried out (i.e., their preconditions are satisfied). A *threat*, on the other hand, consists of a situation that includes an attacker and one or more vulnerabilities.

Thing :: = Event | Object | …….

Object :: = Situation | Proposition | Entity | Relationship

Situation :: = Domain | Context | Vulnerability | Threat | Specification

Context :: = {Domain}

Proposition :: = Argument | Predicate| Claim |Domain Assumption| Quality Proposition | Goal

Argument :: = {Claim | Rebuttal | Mitigation}

Goal :: = Requirement | Security Goal | Anti Goal

Entity :: = Action | Process | Actor | Resource | Asset

Activity :: = {Action}

Attack :: = Activity

Actor :: = Stakeholder | Attacker

Specification :: = {Domain Assumption} {Quality Proposition} {Action}

Relationship :: = fulfills | exploits | protects | denies | damages | wants |carries out | uses | provides | trust-dependency | do-dependency | can-dependency | composes | contributes| …….

Figure 3. Conceptual Model Representation in EBNF

Figure 3 summarizes the elements of our ontology in Extended Backus-Naur Format (EBNF). A EBNF rule of the form A::= B |C | ... indicates that concept A has concepts B and C (and possibly others) as specializations. A rule of the form A ::= {C} indicates that each instance of A consists of (has parts) zero or more instances of C. The notation [] is similar to { } but allows for zero or one instance.

5 Argumentation Analysis for Evolving Security Goals

As we discussed in the introduction, the satisfaction of security goals in the general form of the entailment *W, S* ├ *R* needs to be argued, as security goals are often a collection of claims whose satisfaction depends on the trust assumptions (facts and domain knowledge), as well as any rebuttals and mitigations.

Our argumentation is based on the informal Toulmin structures in the 1950's [2]. However, to consider it in the formal settings, we have simplified the conceptual models. The most important concepts in arguments are defined as follows: A *claim* is a (probably grounded) predicate whose truth-value will be established by an argument. An *argument* contains one and only one claim. It also contains facts and rules in domain knowledge. *Facts* are grounded predicates -- something that is either true or false where terms in these predicate must be constant. *Domain Knowledge* is a set of ungrounded predicates that can be evaluated to true or false once the values of all terms in the predicates are known.

The predicates referred by the domain knowledge do not have to be known facts. However, the predicates that appear in the domain knowledge are all relevant (necessary) to the argument for the truth-value of the claim to remove any redundancy.

Every argument also has a timestamp, which indicates the *iteration* during the argumentation process. For a given argument, an initial iteration is to establish the truth of its associated claim. The argument may require sub-arguments to establish the truth of certain facts or intermediate predicates. These sub-arguments are also arguments, but they are meant to provide supporting evidence (as sub-claims). On the other hand, *rebuttals* are a special kind of arguments whose purposes are to establish the falsity of their associate claims or make them indeterminable. Similarly, *mitigations* are *another* special kind of arguments following the iteration of rebuttals in order to reestablish the truth-value of the associated claims. Both rebuttal and mitigation arguments do not need to contain all the facts and rules. Only incremented facts or rules need to be kept in such follow-on arguments because they are always applied after previous arguments. Of course, the same reasoning mechanism should be used consistently for all arguments.

Claims can be very general. For example, "The Arrival Management (AMAN) system from the air traffic management domain is safe and secure" can easily invite different opinions. To support such claims, one need to use the facts or domain knowledge in the field; to refute the supportive evidence for the claims, one can draw on additional (often non-monotonic or negative) facts and domain knowledge to form claim rebuttals.

As a result, after argumentation analysis is done, one may turn the arguments into evolution rules as follows:

- The facts and domain knowledge rules that cause a *rebuttal* argument are generated into a pattern that match the SeCMER requirements model;
- The new facts and domain knowledge rules introduced by a *mitigation* argument (some of them are new security properties) are generalized into an

incremental transformation where the “before” state of the transformation is the SeCMER requirements model before the mitigation, and the “after” state of the transformation is the SeCMER requirements model after the mitigation.

Both the pattern and the incremental transformation may be represented explicitly as an evolution rule in the SeCMER methodology in hope that similar changes that may rebut the satisfaction of similar existing properties can be mitigated automatically.

In case it is not possible to generalize, the instance level changes will be kept as trivial evolution rules that only matches with the exact situation and does the exact mitigation. Such trivial rules can still be useful to help a regression analysis.

More detailed evolution rules as generalized mitigations can be seen in Section 6 and 7. A detailed example of the argumentation analysis is given in Section 8, along with the application of whole SeCMER methodology.

6 Security Requirements Evolution

After specifying the static view of situations about the security goals, the next step in our methodology is to deal with the dynamic view. In a reactive view of the classification, situations are observed to change over time.

We consider three different perspectives to change: the *maintenance perspective*, the *before-after perspective*, and the *continuous perspective*. The maintenance perspective corresponds to an a-posteriori unplanned evolution to which it is necessary to react upon; before-after perspective corresponds to a-priori planned, anticipated evolution, while continuous perspectives corresponds to changes happening continuously over time. Maintenance and Before-After perspectives are the most commonly used in an industrial context.

We consider *elementary* and *composite* changes in the security requirements model. Elementary types of changes include the modification, the addition and the removal of a concept or a relationship between them.

Composite changes are a transaction of elementary changes (or nested composites) that must happen together or not at all. For example, the addition of a new security goal G requires to add also a relationship “protects” between the node representing G and the node representing the asset protected by G.

6.1 ChangeLine Conceptual model

Changes are typically managed by a process, which is typically assisted by a change management system. When security-related changes are considered, the process must include the state of models with respect to validation and assessment of security goals. An orthogonal dimension is how to help human to manage the dashboard status of the security of the overall achievement, during which errors are allowed to be fixed and issues are allowed to be addressed. Resolution of such issues may lead to addressing the target of a security risk at the design level. In other words, the vulnerability of the specification can be associated with a particular risk factor in satisfying certain security goal.

To represent traceability between changes and versioning of change, we add a further conceptual model: a ***Change Model*** is composed by several ***Change Lines***. A ***Change Line*** is considered as set of ***Changes*** and ***Change Transitions*** to preserve links and grant consistency between successive changes which compose a Change Line. ***Change*** is described by a Change Trigger (e.g. discover a fault or a new threat) which activates a ***Change Request***. It’s also possible to activate a Change Trigger by a threshold defined in an ***Evolution Function*** which monitors the static model of the system. Evolution functions enable to represent Continuous Perspective of change. Change Lines enable to represent both the maintenance perspective and the before-after perspective.

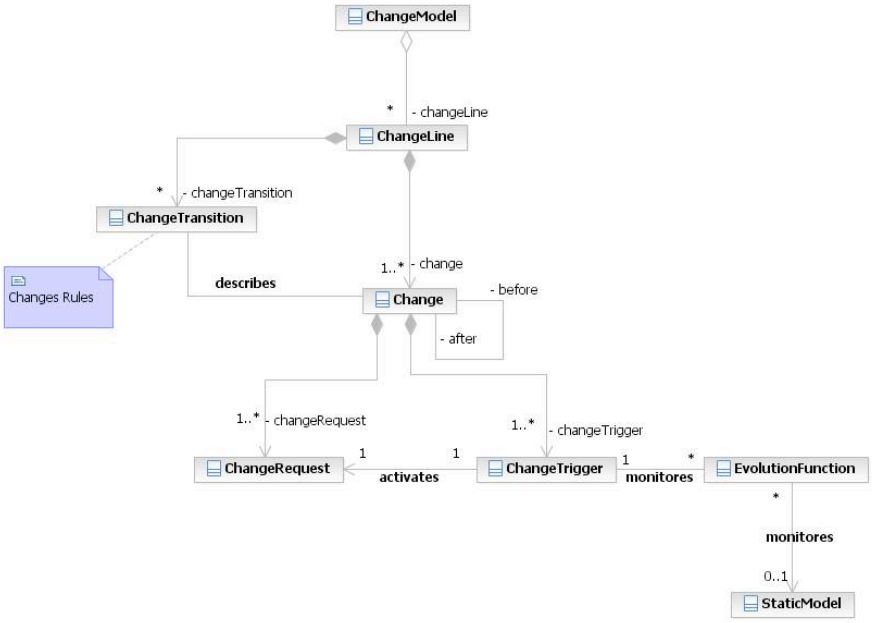

Figure 4. DSML Change Model conceptual model

6.2 ChangeRequest Conceptual model

As shown by Figure 4, a ***Change Request*** contains a PUID to identify it and a status representing the state of Change request. After the activation of Change Request by the Change Trigger, Change Request status is first defined in CCB (Configuration Control Board). The configuration (or change) control board (CCB) is a meeting between all actors of a development team (client, manager, quality, design, integration, …) to define the change request status (e.g. accepted, refused or postponed in the next version of system). The detailed behavior of Requirement Change Request is described in next section.

To instantiate a ***Change Request*** inside different models, we have specialized it in three kinds:

- ***A Requirement Change Request*** modifies the Requirement Model (Requirement, Objectives). It's possible to map this kind of Change Request with DOORS Change Request.
- ***A Context Change Request*** modifies the Context Model (e.g. system architecture).
- ***A Risk Change Request*** modifies the Risk Model (Risk, Threat, Damage, Vulnerability).

These three kinds of Change Request are dependants; a Requirement Change Request could impact on Risk Change Request and Context Change Request and vice versa. This is why we consider a traceability relation between those Change Requests. This relation is described by an association called "impacts_on" (see Figure 5).

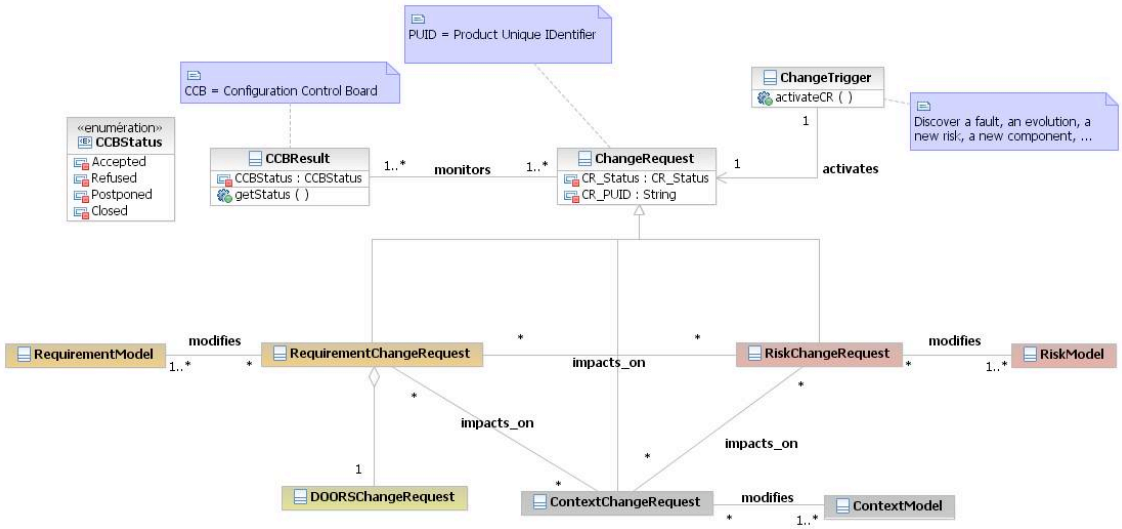

Figure 5. DSML Change Request Conceptual model

6.3 Behavior of Change Request

For the sake of readability, the generic Change Request Behavior is described by UML Statechart Diagram (see Figure 6a). We present on the one hand the generic behavior of Change Request including CCB status relations. On the second hand we describe the specific behavior of Requirement Change Request.

A ***Change Request*** (CR) starts after Change Trigger activation (e.g. discover a fault, a new requirement, etc.). Redactor of Change Request must define the change and trace it with the impacted elements. Change Request is as default in ***Pending*** State.

A CCB must be planned; it monitors the Change Request Status which could be in the following states:

- ***Refused***, CR is not relevant; it is not integrated in system. Change Request is ended in this state.
- ***Postponed***, CR is relevant but it's not possible to integrate it in the current version of the system. This CR is planned for the next version. CR returns in Pending State during this system version.
- ***Accepted***, CR is integrated in current version of system.

If CR is accepted, it will be ***In_process*** macro state. This macro state is specialized for several DSML Models (Risk, Requirement or Context).

CR is finish if and only if it's closed in CCB with client agreement.

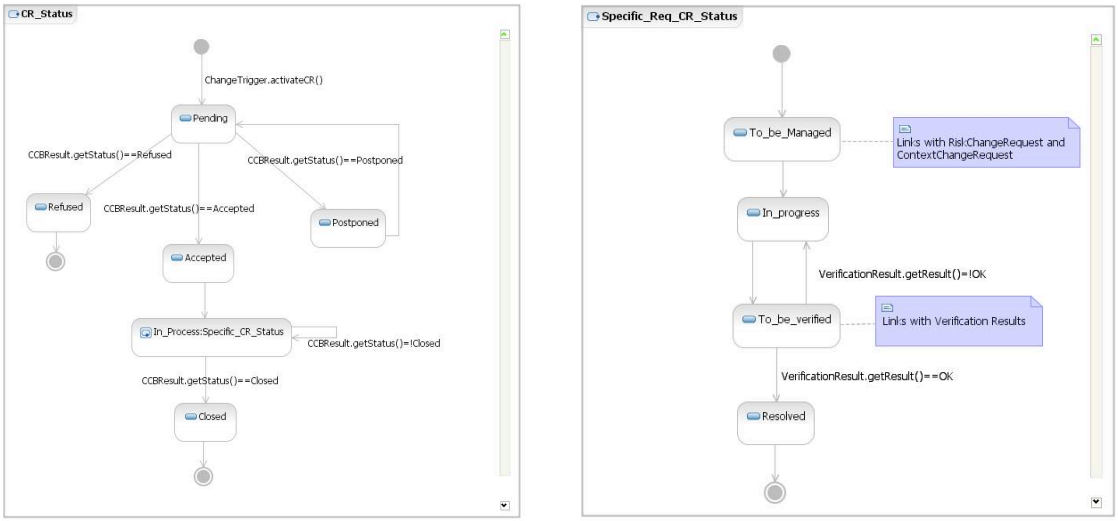

Figure 6. Change Request Status Behavior (a) generic (b) requirements-specific

Specific ***Requirement Change Request*** (RCR) Behavior starts after ***Accepted*** state in generic behavior. As shown by Figure 6b, Requirement Change Request Status is represented by the sequence of following states:

- ***To_be_Managed***, redactor of Requirement Change Request must take into account impact of this change request with the other elements (Risk and Context) and change them if necessary with new CR(s).
- ***In_progress***, redactor must define changed requirement, designer must models them, and developer must implement them.
- ***To_be_verified***, integrator must take into account these changes in test campaign (and change test scenario if necessary).
- ***Resolved***, RCR Status will reach this state if and only if changed requirement are verified in test campaign.

7 Process Automation by Evolution Rules

The SeCMER approach prominently features on an automated step. This work phase is carried out automatically by monitoring the existing requirements model (as well as other interconnected models) and reacting to applied changes. The reactions are defined by Evolution Rules. With carefully specified evolution rules, the automated rule application can save significant manual effort, e.g. in the argumentation phase.

Upon each change, reactions are performed iteratively as long as any evolution rules are still applicable. Therefore the requirement model serves both as input and output of this system component. Further inputs include the changes experienced by the requirement model, and the definition of the evolution rules themselves.

Section 7.1 elaborates why and how evolution rules can be a useful contribution to SeCMER methodology. Section 7.2 presents some background knowledge from the field of model transformation, on which our proposed concept of evolution rules is based. Section 7.3 explains the conceptual model of Evolution Rules, while Section 7.4 gives precise mathematical foundations.

7.1 Goals for the evolution rules

There are at least three ways requirements modeling environments can benefit from a mechanism for automated (rule-based) reaction to changes:

- Internal consistency checking and on-the-fly evaluation of well-formed constraints,
- Synchronization against other models (risk analysis, design, etc.) and information propagation via model transformation techniques,
- Saving human efforts by identifying the extent and influence of change to determine where manual change analysis and argumentation is needed, by preparing automatically deducible information for this manual reasoning, and possibly by complete automation of simpler, deterministic steps of the argumentation process.

The task of constraint evaluation is not specific to requirements or security engineering, only to the actual conceptual models. Therefore it can be considered out of scope for SecureChange, and will not be discussed here in detail. Results of this approach are shown in [18].

Integration with other models outside the requirements scope is a future task for SecureChange, and will be discussed in upcoming deliverables.

The current deliverable focuses on the third type of automation, which is specific to the domain of (security) requirements evolution, and closely tied to the methodology. We argue that requirement modeling environments should be equipped with an automatism that is capable of identifying the effects of the change and thereby reducing the amount of required human effort to deal with the change. We propose that **Evolution Rules** be defined to accomplish the following:

- By operating over an interconnected requirement model and argumentation model, evolution rules can identify cases when a change in the model influences an evidence in support of a previous argumentation activity, and consequently flag the argument for manual re-evaluation
- Efficient identification of security goals whose satisfaction is implied by the model. Raise alerts (e.g. towards the argumentation staff) if a previously satisfied goal becomes unsatisfied (more precisely, if the satisfaction not provable anymore) due to changes in the model. Cases where the satisfaction of a rule can be determined automatically include the following:
    - There is already a valid (not flagged) argument, constructed in a previous argumentation session that decisively supports the satisfaction of the goal.
    - The goal is decomposed (AND/OR) into subgoals, and its satisfaction is implied by the satisfaction of subgoals.
    - In some cases, model entities connected in a certain way may automatically imply the satisfaction of the goal. For example, if the goal is delegated to an actor, who carries out an action that fulfills the goal, and there is no corresponding attacker with an anti-goal, than the goal can be considered satisfied without manual argumentation. Some of these rules are expected to be domain-specific (e.g. ATM-only) and to emerge from the argumentation process by carefully scrutinized inductive optimization and rule formalization.
    - Similarly, it can be determined by given (possibly domain-specific) conditions that artifacts in other models (through traceability relations) automatically guarantee the satisfaction of the goals.
- Automatically making decisions and deterministic changes to the requirements model, or instantiating several options (i.e. draft solutions) and offering them to the requirement engineers, if and when such automation is applicable. Once again, such rules are expected to be domain-specific (e.g. for ATM) and to emerge from the argumentation process by carefully scrutinized inductive optimization and rule formalization.

The list above is not necessarily exhaustive, and while we will show a number of examples (see Section 8) some rules are expected to be specific to the application domain / case study. Therefore the focus is primarily at the proposed language and mechanism for defining and efficiently evaluating evolution rules.

The framework and language for specifying evolutions rules for the security-related aspects of the engineering model should

- support complex structural requirements that are difficult and error-prone to oversee manually;
- allow the capturing of change events in terms of similarly complex structural relations, thereby treating change as a first-class citizen;
- provide automated alerting of criteria that cease to be satisfied;
- allow flexible adaptation to domains, e.g. ATM;

- enable the flexible, scenario-specific definition of the aforementioned complex criteria;
- enable the engineer to define automated reactions to change events where applicable;
- enable the reactions for automatic reconfiguration of the design model; automatic application of security-related design decisions; and automatic reusing of design artifacts (e.g. argumentations), to be filled later by the engineers, that are required for a system evolution to be admissible from a security viewpoint.

7.2 Underlying model transformation technology

The language and efficient implementation of evolution rules relies on technology pioneered for automated model transformations. As revealed in many surveys and papers during the recent years [5][6][11], model transformation (MT) languages and tools play an important role in modern model-driven system engineering in order to query, derive and manipulate large, industrial models.

As a typical example, tool integration requires that a complex relationship be established and maintained between models conforming to different domains and tools. In the context of SecureChange, synchronization involving requirement and design models would pose a transformation problem.

Model synchronization tasks can be formulated as the obligation to keep a model of a source language and a model of a target language consistently synchronized while the underlying source model (and sometimes the target also) is evolving. Model synchronization is frequently captured by transformation rules [2]. When the transformation is executed, traceability links are also generated to establish logical correspondence between source and target models.

Traditionally, model transformation tools support the batch execution of transformation rules, which means that input is always processed “as a whole”, and output is always regenerated completely. However, in case of large, complex, and continuously evolving models, batch transformations may not be feasible. To address the issue of model evolution, incremental model transformations (i) update existing target models based on changes in the source models [18], and (ii) minimize the parts of the source model that need to be reexamined by a transformation when the source model is changed [3]. In the terminology of [6], these aspects are called *target* and *source incrementality*, respectively.

Since rules are defined in terms of patterns and actions, *pattern matching* plays a key role in the execution of model transformations. The goal of pattern matching is to find the occurrences of a pattern, which imposes structural as well as type constraints on model elements. Source incrementality can be achieved by employing *incremental pattern matching* techniques; for example, the RETE [9] incremental algorithm was used in [3].

The central idea of incremental pattern matching is that occurrences of a pattern are readily available at any time, and they are incrementally updated whenever changes are made. As pattern occurrences are stored, they can be retrieved in constant time –

excluding the linear cost induced by the size of the result set itself –, making pattern matching a very efficient process. Benchmarks [4] and practice have shown that incremental pattern matching can improve performance or scalability by up to several orders of magnitude in certain scenarios.

Based on source incrementality, it is also possible to detect the appearance and disappearance of pattern matches efficiently. Ráth et al [18] introduced a live transformation approach where a model change is captured by a change in the match set of a graph pattern, and transformation rules are triggered by such events.

7.3 Conceptual model for evolution rules

Evolution rules control how one model, or an interconnected set of models, follow the evolution of a source model in order to maintain security and other objectives (Figure 7) Evolution rules are defined in conformance with the Event – Condition – Action semantics [1] to specify the desired reaction to changes performed on the model.

Basically, an *Event* captures an elementary transition of the system to a different (not necessarily internally consistent) state, identifying the change that happened between the two states. An *Action* is a list of operations that constitute the reaction to that event. The strength of the formalism is that the reaction can depend on the context where the event happened, as defined by the *Condition* part. Event and Condition both serve as a way of monitoring the evolution of a system. The key difference is that Event captures a *dynamic* change in the system, while Condition identifies the *static* context where this change happened.

The Event part of the evolution rule is matched against every change executed on the model. The Condition may restrict the cases where the rule is applicable, and may select multiple ways to apply it. The Action part manipulates the model by issuing change commands itself; these changes will eventually be processed like any other change operation, and reacted upon by evolution rules.

Various kinds of change commands can be issued. The most basic *change kind*s are the creation of entities and relationships of a specific type, deleting them and modifying their values. This list of change kinds is extensible to incorporate a more refined notion of changes, or domain specific change macros.

An actual change command has a change kind and refers to actual entities or relationships as affected elements. The definition of an evolution rule, however, refers to *rule variable*s as affected elements instead. The Event part match changes against one or more *change queries.* Each of them captures the change in terms of the appearance or disappearance of element configurations (patterns). An attribute contains the sign of the change query. The appearing/disappearing element configuration of the change query is described by a set of predicates formed on rule variables. The Condition part describes the context of the event, likewise with predicates on variables. Some of these variables are typically used by the change queries as well. The two most common predicate types are entity predicates (constraining a variable to a given entity type) and relation predicates (constraining a variable to a given relation type, connecting a source variable and a target variable). The Action part contains a sequence of *reaction template*s that are parameterized by rule variables appearing in the Event, Condition or even preceding reaction templates,

and can be instantiated into applicable commands by substituting the parameter variables. The most important type of reaction template is the *change template* that can be instantiated into a change command of a certain change kind. The evolution rule contains all variables mentioned by the Event or the Condition, a subset of which is accessed by the Action.

Change queries are intended to match actual change events that cause the appearance or disappearance of the appropriate patterns, and substitute the variables to the affected elements. After that, the Condition is evaluated to decide whether the rule can be applied for this particular change, and to substitute remaining free variables. The Action is applied for each possible substitution; this means instantiating all reaction templates with the substituted values of variables. In case of change templates, the resulting change commands can be submitted for execution and evolution rule application.

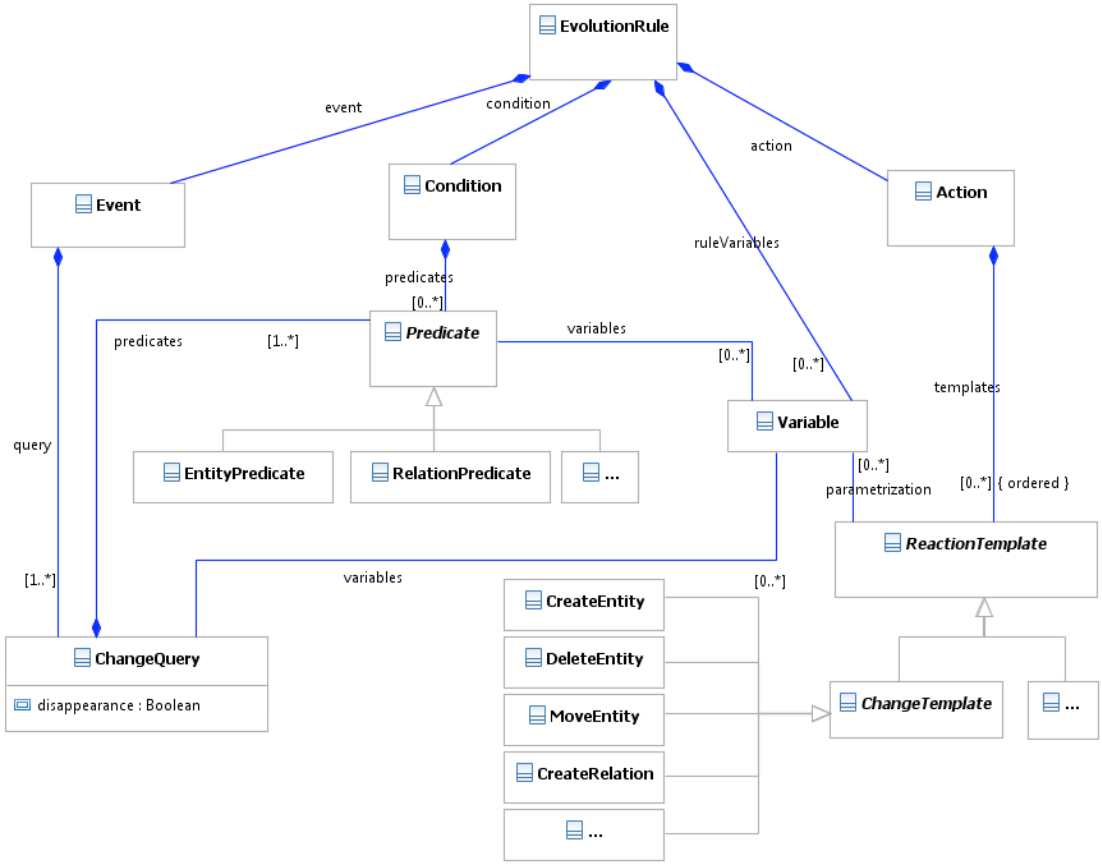

Figure 7. Conceptual model for evolution rules

7.4 Mathematical foundations

The notion of Evolution Rules has precise mathematical underpinning based on the theory of graph transformation. For purposes of formalization, we represent the requirement model and other associated models such as design as (attributed) graphs. Whether and when a rule is applicable is determined by the formalism of graph

patterns in case of static models; or the more advanced graph change patterns in case changes are taken into consideration.

7.4.1 Graph Patterns

Our Evolution Rule formalization relies on the concepts of graph model, graph pattern, pattern matching and NAC, widely known in the field of graph transformation.

Definition 1 (Graph Model) A graph model over a type system Type is a structure $G=\langle Ent,Rel,src,trg,typ\rangle$ where Ent is a set of entities (graph nodes), Rel is a set of relations (graph edges); $src,trg:Rel\rightarrow Ent$ map the relations to their source and target entities, respectively; and the typing of elements is $typ:ME\rightarrow Type$ where ME is an abbreviation for the set of model elements $Ent\cup Rel$.

Our graph model assumes that each entity and relation takes its type from a type system which is simplified here to a set of predefined types. Note that we make no assumptions on the actual types here, so that model elements from other modeling domains can be represented in connection with requirements. The notion of type compatibility is beyond the scope of this simplified formalization. Various other model features such as containment or attributes are also omitted here for brevity.

Definition 2 (Graph Pattern) A graph pattern $P=\langle V,C\rangle$ over a type system Type contains a set V of pattern variables, and a set of graph constraints $C=C_{ent}\cup C_{rel}$ attached to them. Entity constraints $C_{ent}\subseteq V\times Type$ state that a variable is a node of a certain type. Relation constraints $C_{rel}\subseteq V\times V\times V\times Type$ state that a variable is an edge of a certain type, connecting two given variables representing the source and the target of the edge. To identify the variables and constraints of a specic pattern P, we use V_P and C_P , respectively.

The pattern language also permits additional constraints such as containment, equality and inequality, attribute constraints, or pattern composition, which are not detailed here.

Definition 3 (Graph Pattern Match) A substitution $s:P\rightarrow G$ of a graph pattern $P=\langle V,C\rangle$ in a graph model $G=\langle Ent,Rel,src,trg,typ\rangle$ over a type system Type is a set of variable assignments $asgn\in V\times ME$, one for each variable $v\in V$. Let $s(v)\in ME$ denote the model element assigned by s to the variable $v\in V$.

A substitution satisfies an entity constraint $c=\langle v,t\rangle\in C_{ent}$ iff typ(s(v)) is compatible with t. A substitution satisfies a relation constraint $c=\langle v,a,b,t\rangle\in C_{rel}$ iff $src(s(v))=s(a)$ and $trg(s(v))=s(b)$ and typ(s(v)) is compatible with t.

A match $m:P\rightarrow G$ is a substitution that satisfies all constraints $c\in C$ of P, which will be denoted by $G,m\models P$. [3]

[3] Remark: from now on, we assume that a single type system Type is given, and will not include it in each further definition.

A *negative application condition* (NAC, indicated by the neg keyword) prescribes contextual conditions that, if satisfiable, invalidate a match of the pattern.

Definition 4 (Graph Pattern with Negative Application Condition) A pattern with NAC is $PN = \langle P, N^* \rangle$ where $P = \langle V, C \rangle$ is a (positive) graph pattern, and N^* is a set of negative application conditions $N_i = \langle V_i, C_i \rangle$, each being a well-formed graph pattern, such that $P \subseteq N_i$ meaning that $V \subseteq V_i$ and $C \subseteq C_i$.

Commonly, only the *subpattern* $SN_i = N_i \setminus P$ is explicitly indicated and depicted in figures and code extracts, which is defined as $SN_i = \langle SV_i; SC_i \rangle$, where $SC_i = C_i \setminus C$ and $SV_i \subseteq V_i$ is the set of variables involved in SC_i.

Definition 5 (Match of Graph Pattern with NAC) A match $m:PN \rightarrow G$ of $PN = \langle P, N^* \rangle$ in graph model G is a match of the positive pattern $G, m \models P$, where there is no $N_i \in N^*$ and match $m_i:N_i \rightarrow G$ such that $m \subseteq m_i$ (meaning that $m_i(v) = m(v)$ for all v variables of P).

Some graph pattern languages, including the one that will serve as the basis of Evolution Rules, even permit NACs to have NACs of their own. If there is no limit on the number of negations that can be nested within each other, graph patterns (without attribute constraints) become expressively equivalent to first order formulae over the predicates describing the graph model [19].

7.4.2 Graph Change Patterns

We define the advanced formalism of Graph Change Patterns (not to be confused with the change pattern concept of WP2) to capture how a graph model changes in an evolution. In addition to conventional graph patterns matched against the current snapshot, a change pattern should also contain constructs for expressing the difference between two graphs, in the form of appearance and disappearance queries.

When matching change patterns, the key idea is to simultaneously match them against a pair of graph models, called the pre-state (before state) and the post-state (after state). Appearance queries are graph patterns whose matches have appeared in the post-state, but were not present in the pre-state; and disappearance queries are patterns whose match has disappeared.

In some scenarios, the appropriate reaction to a change does not only depend on the after state, but also on the net change (or equivalently, the before state). The true strength of Graph Change Patterns is the ability to distinguish cases where the current (after) state is the same, but it was reached through different cases, from different before states. As the pattern variables are mapped to the locality of the change, a match of the Graph Change Pattern also pinpoints where the reaction should be applied.

Definition 8 (Graph Change Pattern) Graph Change Patterns (CP) can be defined as a tuple $CP = \langle PN, P_+^*, P_-^* \rangle$, where $PN = \langle P, N^* \rangle$ is the main graph pattern, while P_+^* and P_-^* are two sets of graph patterns, called appearance queries and disappearance queries (together change queries). Appearance queries $P_i = \langle V_i, C_i \rangle \in P_+^*$ and disappearance queries $P_j = \langle V_j, C_j \rangle \in P_-^*$ are allowed to share

variables with P, and the set of common variables is their interface $I_i=V_i \cap V_P$ and $I_j=V_j \cap V_P$.

CPs are matched against a pair of graphs G_{old} and G_{new}, such that G_{new} is derived from G_{old} by model manipulation, and thus Ent_{old} and Ent_{new} may intersect on elements that were preserved in the transformation, as well as Rel_{old} and Rel_{new}.

Definition 9 (Match of Graph Change Pattern) A match of the Graph Change Pattern $CP=\langle PN,P_+^*,P_-^* \rangle$ in $\langle G_{old},G_{new} \rangle$ is the structure $m=\langle m_P,m_+^*,m_-^* \rangle$: $CP \rightarrow \langle G_{old};G_{new} \rangle$, where

- m_P: $PN \rightarrow G_{new}$ is a match of PN, in the after state G_{new}.
- m_+^* consists of a match $m_i:P_i \rightarrow G_{new}$ for each P_i in P_+^* such that
- $m_i(v)=m_P(v)$ for interface variables $v \in I_i$, and
- $G_{old},m_i \not\models P_i$.
- m_-^* consists of a match $m_j:P_j \rightarrow G_{old}$ for each P_j in P_-^* such that
- $m_j(v)=m_P(v)$ for interface variables $v \in I_j$, and
- $G_{new},m_i \not\models P_j$.

Note that this definition is deliberately asymmetric for G_{old} and G_{new}, as PN is interpreted on G_{new} only.

7.4.3 Rule Formalism

Harnessing the strength of CPs, a powerful rule-based automation formalism can be defined. Without going into details of how the reactions themselves are defined, such a rule can be characterised by a *guard* that is a CP; after a change to the model, the actions associated with the rule are executed for each match of the guard. In publications by the authors in the field of model transformation, such a rule was referred to as Change-driven Rule (CDR).

Relying on technologies developed for model transformation purposes (incremental pattern matching), CP can be detected efficiently. Consequently, a rule-based system specified by CDRs can be executed in an efficient way.

In the context of Security Engineering, the Evolution Rules envisioned in Section 7.1 can be formalized as CDRs, lending both efficiency and expressivity to the approach. The Condition part of the Evolution Rule expresses constraints on the current (after) state, therefore it is formalized the PN part of the CDR. The appearance and disappearance Events are formalized as change queries in P_+^* and P_-^*, respectively. Finally, the Action is associated with the CDR (which was not formally defined in Section 7.4.2)

7.5 Examples of evolution rules

We now demonstrate the power of the language by showing how a certain issue that arises in evolving requirements models can be addressed by evolution rules.

In an evolving requirements model, new actors may be introduced, delegation and trust relationships may be changed, all raising security concerns. When an actor is taking over the responsibility (delegation) of a security goal previously achieved by a different actor, a problematic situation may arise if other actors do not have trust in the new setup. The same hold for delegating other entities (e.g. assets) instead of goals. Basically, intervention is required in situations when an actor delegates some responsibility (e.g. a security goal) to another actor, but does not trust the other one with the same object.

The appropriate reaction can range from logging the event, raising a warning or initiating an argumentation that will be finished by security engineers, to automatic intervention like creating the missing trust relationship, depending on policy. The reaction might depend on how such an undesired state of the model was produced.

To illustrate the capabilities of the evolution rule formalism, we first design a graph pattern to express the undesired configuration, and then we draft three alternative solutions with evolution rules to intervene in these situations.

7.5.1 Graph pattern for expressing the problem

Figure 8 visually depicts the graph pattern (with a negative condition) that characterizes this undesired configuration of elements.

In a match of the pattern, the (positive) pattern variables Act1, Act2, Obj, Del will be mapped to entities in the model. Act1 will be substituted for an entity of type Actor that delegates the responsibility of an entity Obj to the actor Act2 using the delegation relationship Del; where at the same time, there is no trust relationship Tru such that Act1 trusts Act2 over Obj.

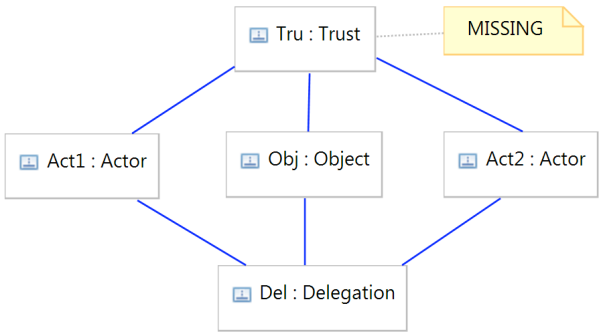

Figure 8. The undesired pattern: untrusted delegation

7.5.2 Solution 1: one rule per elementary change

The first solution would be to create several evolution rules, one for each possible elementary change that can complete the pattern and make an intervention necessary. In this case, two kinds of elementary changes can trigger the rule: the detection of a

newly added “delegation” relationship between two actors (and the dependum), or the deletion of an actor-actor trust (over a dependum).

Both changes can be captured by the Event part of a separate evolution rule (appearance event in the former case, disappearance in the latter). The condition part is required to determine whether the change actually completes the pattern: when a delegation appears, the non-existence of a trust with the same dependum will have to be checked; when a trust disappears, the existence of the delegation with the same dependum will have to be checked. The Action creates an argument prototype (i.e. a placeholder), connected to the violated security goal, to discuss the problem. Engineers will have to manually finish the argument with domain-specific knowledge, or fix the problem. Additionally, the Action contains a simple logging statement; observe how the two different cases can be handled differently. The following *pseudo code* listing describes these two evolution rules; *syntax is not final*.

```
evolution rule UntrustedDelegation1 {
    variables = (Act1, Act2, Del, DD, Tru, TD, Obj, Arg, AP);
    event = appear {
        entity Actor(Act1);
        relation Actor.delegates(Act1-Del→Act2);
        entity Actor(Act2);
        Actor.delegates.dependum(Del–DD->Obj);
        entity Object(Obj);
    }
    condition {
        no (Tru, TD) such that {
             relation Actor.trusts(Act1-Tru→Act2);
             relation Actor.trusts.dependum(Tru–TD->Obj);
        }
    }
    action {
        log “Delegation created without supporting trust: $Act1-$Obj-$Act2”;
        create entity Argument(Arg);
        create relation Argument.supports(Arg–AP->Obj);
    }
}
evolution rule UntrustedDelegation2 {
    variables = (Act1, Act2, Del, DD, Tru, TD, Obj, Arg, AP);
    event = disappear {
        entity Actor(Act1);
        relation Actor.trusts(Act1-Tru→Act2);
        entity Actor(Act2);
        relation Actor.trusts.dependum(Tru–TD->Obj);
        entity Object(Obj);
    }
```

```
    condition {
       relation Actor.delegates(Act1-Del→Act2);
       relation Actor.delegates.dependum(Del–DD->Obj);
    }
    action {
        log "Removal of trust threatens delegation: $Act1-$Obj-$Act2";
        create entity Argument(Arg);
        create relation Argument.supports(Arg–AP->Obj);
    }
}
```

7.5.3 Solution 2: single coarse-grained rule

The change query formalism introduced in this chapter allows the detection of changes that are defined by multiple predicates. This results in the capability of change queries to observe the appearance (or disappearance) of a complex pattern, regardless what the last elementary change was that completed the pattern.

In this case, the entire undesirable pattern can be captured in an appearance event of a single evolution rule; whenever the undesired pattern appears, the evolution rule will fire, independently of the order of operations that eventually resulted in the appearance of the pattern. This enables us to formulate the solution much more concisely; in this simple example, even the Condition part could be discarded.

```
evolution rule UntrustedDelegation {
    variables = (Act1, Act2, Del, DD, Tru, TD, Obj, Arg, AP);
    event = appear {
        entity Actor(Act1);
        relation Actor.delegates(Act1-Del→Act2);
        entity Actor(Act2);
        Actor.delegates.dependum(Del–DD->Obj);
        entity Object(Obj);
        no (Tru, TD) such that {
             relation Actor.trusts(Act1-Tru→Act2);
             relation Actor.trusts.dependum(Tru–TD->Obj);
        }
    }
    condition {}
    action {
        log "Untrusted delegation: $Act1-$Obj-$Act2";
        create entity Argument(Arg);
        create relation Argument.supports(Arg–AP->Obj);
    }
}
```

This kind of concise solution is much quicker to develop and understand. Development also becomes less error-prone, as the rule designer does not have to manually take care of all possible elementary changes that can result in the appearance of the complex pattern; the previous solution would have been insufficient if the rule `UntrustedDelegation2` had been accidentally omitted. The disadvantage is that the same Action part is executed regardless of the last elementary change that triggered the rule; if some cases do require special action, than more evolution rules should be used with an event granularity that is just enough to distinguish the relevant cases.

7.5.4 Solution 3: automatic problem correction

Apart from logging the detection of the pattern and reusing an argumentation, evolution rules can also correct problems present in the model. The difficulty of this approach is that often there is more than one way to remedy an issue, and the decision is hard to automate. For instance, the problem in this example can be solved by adding a missing trust relationship; or by removing the delegation (and probably implementing something else in its place). Both are valid ways to handle the issue, but engineers should select manually which one should be applied in each concrete case. To achieve this, we introduce two alternate evolution rules that implement these two reactions. Together with the rule `UntrustedDelegation` of Solution 2 introduced in Section 7.5.3, they provide three options that can be automatically offered to the engineers to choose from.

Note that the three rules can reuse each other's Event parts for more concise specification. Once again, the syntax is not final.

```
evolution rule UntrustedDelegation_AddTrust {
    variables = (Act1, Act2, Del, DD, Tru, TD, Obj);
    event = UntrustedDelegation.event
    condition {}
    action {
        log "Resolving untrusted delegation ($Act1-$Obj-$Act2) by adding
missing trust link";
        create relation Actor.trusts(Act1–Tru->Act2);
        create relation Actor.trusts.dependum(Tru–TD->Obj);
    }
}
evolution rule UntrustedDelegation_RemoveDelegation {
    variables = (Act1, Act2, Del, DD, Tru, TD, Obj);
    event = UntrustedDelegation.event
    condition {}
    action {
        log "Removing untrusted delegation: ($Act1-$Obj-$Act2)";
        delete relation DD;
        delete relation Del;
    }
}
```

Where applicable, evolution rules can directly manipulate the model to automate the solution of common problems. Some of the change patterns introduced in D2.1 can be considered as possible candidates for being automated with evolution rules.

7.5.5 Discussion

None of the above rules deal with the *disappearance* of the undesired pattern. Depending on policy, additional rules may have to be defined to react to security problems being solved, as the actions of the other evolution rule (e.g. placing a warning marker or creating an argumentation placeholder) may have to be undone or compensated.

The example presented in this section shows how the goals in Section 7.1 can be satisfied using the proposed formalism for evolution rules:

- the untrusted delegation was captured as a complex structural property
- a change event detecting the change of this complex property was defined
- the formalism is general enough to be refinable for domains or scenarios
- the rules can take appropriate domain-specific actions
- these reactions include user interaction (logging in this example) and the modification of a model (creating the argument placeholder, creating, removing the delegation)

8 Application of the Methodology

This section illustrates through an example from the ATM case study the different steps of SeCMER methodology.

8.1 Requirement model example

First we show how we can represent the evolution of requirements that characterize the ATM case study by instantiating the conceptual model presented in Section 4. Some of these arise from the change of domain properties which are not controlled by the system designers, while others arise from the change of optative properties or functional and security goals.

In this example, we show how functional and security goals of the actual ATM systems change due to the introduction of the AMAN queue management tool that supports ATCOs and of the SWIM, a IP based communication network.

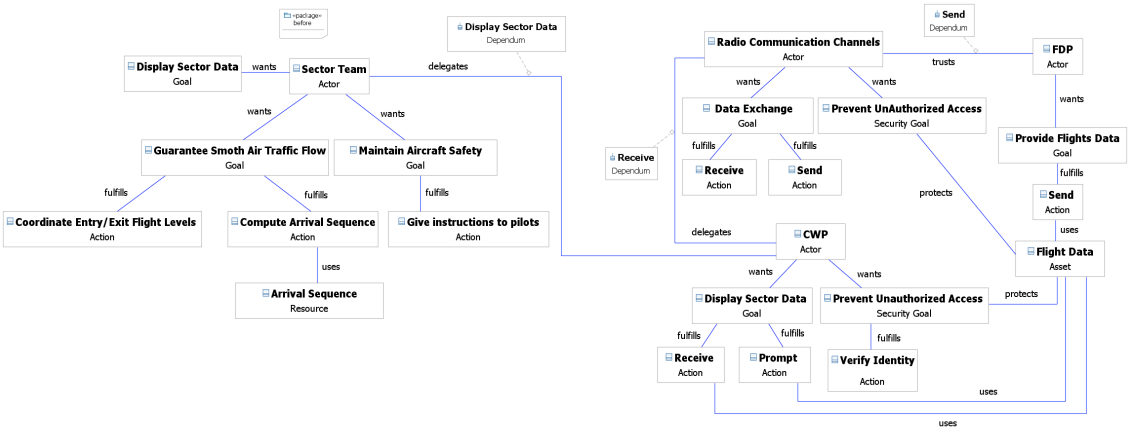

Figure 9. The "before" requirements model

Figure 9 represents the requirement model before the introduction of the AMAN. The main actors are the Sector Team at the destination airport composed by the Planning and the Tactical Controller, the CWP, and the dedicated communication lines (telephone, radio communications). The flight arrival management operations are performed by the Sector Team (Tactical and Planning Controllers) that has to compute the arrival sequence for the flights and give clearances for landing to the pilots flying in their sector on the basis of the information displayed by the CWP such air traffic, radar data, monitor displaying inbound/outbound traffic planned for the sector, telephone switchboards, airlines and airport operators preferences or priorities about arrival runways. Communications between different ATM actors take place over dedicated and secure radio communications lines.

In this scenario, the security goals are associated with the CWP and the Communication Lines that shall protect flight data info from unauthorized access.

8.2 Evolution example

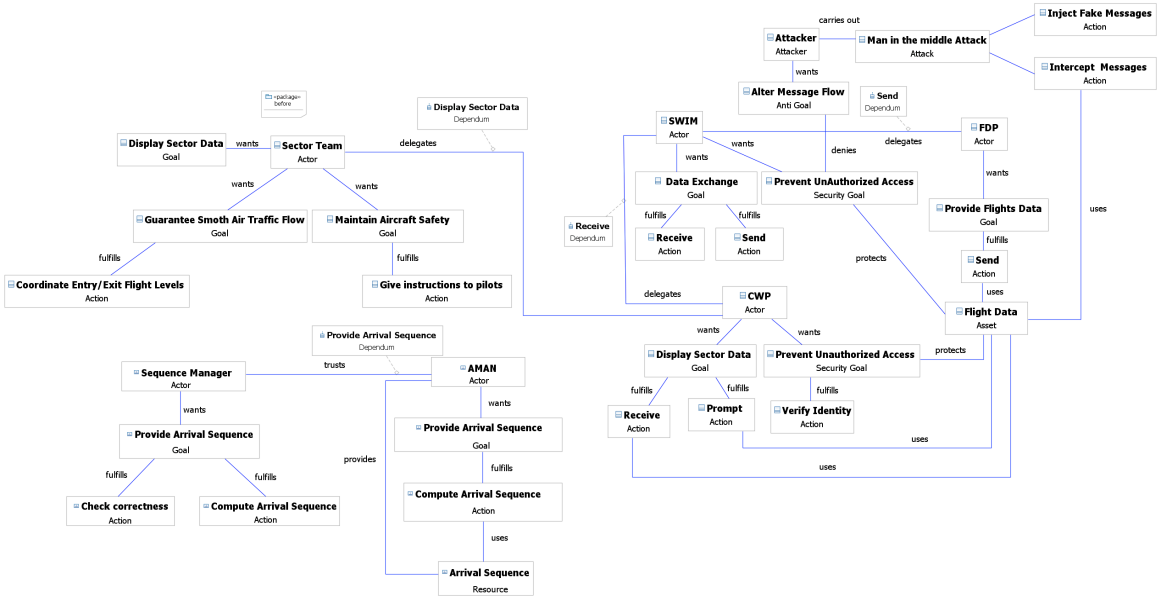

Figure 10. The "after" requirements model

As an effect of the introduction of AMAN, ATM systems go under architectural, organizational, and operational changes. At architectural level, the AMAN supports the Sector Team by providing sequencing and metering capabilities for a runway, airport or constraint point, the creation of an arrival sequence using 'ad hoc' criteria, the management and modification of the proposed sequence, the support of runway allocation at airports with multiple runway configurations, and the generation of advisories for example on the time to lose or gain, or on the aircraft speed. At the organizational level, the introduction of the AMAN requires the introduction of a new type of ATCO, called *Sequence Manager*, who will monitor and modify the sequences generated by the AMAN and will provide information and updates to the Sector Team. At the operational level, on one side the AMAN interacts with the FDP, CNS, and Meteo services to collect the Airport Operators priorities for runaways usage the Airlines priorities in terms of flight arrivals, the Meteo condition, and the aircraft position that it uses to compute an ad hoc arrival sequence or to generate advisories. On the other side, the AMAN interacts with the Sequence Manager and the Sector Team through their CWPs monitor. The Sequence Manager can check the arrival sequence and the advisories generated by the AMAN, and if necessary can modify them, while the Sector Team ATCOs can only view them. Based on the information provided by the AMAN, the Sector Team gives clearances to the pilots flying in its sector. The communication between the different ATM actors is based on the SWIM, an IP based data transport network that will replace the current point-to-point connections systems.

In this scenario, the SWIM actor has replaced the communication lines actor and it has the security goal of protecting the data flight information from unauthorized access. This security goal is denied by the fact that attacker carries out a man in the middle attack and so the *trust relation* between the SWIM and the CWP is replaced by a *delegates* relation (see Figure 10).

8.3 Argumentation for security properties

The argumentation analysis for security goals usually consists of three types of steps. Claims are to establish the satisfaction of the security goals using the facts and domain knowledge rules available in an elicited requirements model. On the other hand, while the requirements model evolves along with change in the world, additional facts and domain knowledge rules may refute the argument for the satisfaction claims. Such rebuttals must be handled properly, by revisiting the facts and domain knowledge rules in the model, or by finding additional facts and domain knowledge rules for their mitigations. These three steps can be applied to any state of the requirements model, and they can interleave with the application of evolution rules in the iterative SeCMER process.

Rebuttals. During the argumentation analysis, the “before” scenario was observed insecure by the rebuttal that the changes introduced into the system could deny the security goal. The newly acquired domain knowledge “A man-in-the-middle attack happened to the communication lines could distort the data flight information from an unauthorized access”, which violates the security goal of the SWIM actor: “the data flight information are protected from unauthorized access”. This rebuttal is confirmed by the argumentation analysis, which can be generalized into the following pattern “delegates information to an actor through a shared communication process” and “the communication process may be shared with actors not trusted”.

Mitigations. The next step during the argumentation analysis is to find mitigations to the rebuttal. One type of mitigation is to reassess the risks associated with the facts and domain knowledge raised by the rebuttals and reject a change when the risk is low. However, this is not the case in the example. The risk of exposing the data link to malicious attackers is high if no mechanisms are introduced to protect the secure transmission of data flight information. Therefore, the change to the communication line is proposed “to encrypt the data in transmission by the sender and decrypt it by the receiver end”. The domain knowledge that “it is difficult for untrusted eavesdropper to decrypt the data flight information” assures that the new system with the encryption is secure. The generalization of the mitigation step can be stated as follows: “if the before situation a delegates relation is untrusted and the communication is not encrypted, a change is needed to introduce encryption as the solution”.

Alternatives. In fact, the argumentation process can continue, with the rebuttals on the previous mitigation suggests that the data encryption with poor strength key is still easy to be decrypted by attackers armed with password dictionaries. As a mitigation step to this, the maintained could introduce the change the “untrusted *delegates* relationship” into “trusted delegates relationships”, and introduce an additional requirement on “the delegatee actor shall be trusted” by using a key to access the lock in the control room. A generalization of this mitigation is to add “obligatory actions” to the trusted delegatee actors and to avoid using the communication links through untrusted channels.

Automation. Security goals often push the system boundary to enclose emergent facts and domain knowledge, some argumentation analysis has to carried out interactively. On the other hand, conceptual model for argumentation makes it easier to turn the three types of modelled arguments into predicate logic formula that are checked using off-the-shelf reasoning tools [12].

In the next subsection, we introduce several evolution rules that formally combine the events, conditions of the rebuttals and the actions of the mitigations.

8.4 Deriving and using Evolution Rules

In an evolving requirements model, new actors may be introduced, delegation and trust relationships may be changed, all raising security concerns. The ATM evolution case study is an example of this phenomenon: the new SWIM actor is introduced, taking over the responsibility of secure communication, but other actors such as CWP not necessarily trust it. This is exactly the problem that the step described in Section 7.5 addresses; in the following, we will demonstrate how such evolution rules can be derived and applied in the concrete ATM example.

In the previous subsections, we have explained how an informal argument is constructed, rebutted and mitigated on the elicited requirements models. In case the argumentation turns out to be (partially) mechanic, we can enumerate Event-Condition-Action evolution rules where events and conditions are obtained from the rebuttals, and the actions obtained from the mitigations.

To come up with the events and conditions, we first represent a part of the complete requirements model as a graph pattern. For example, when an actor delegates some responsibility (e.g. the security goal of the CWP actor to protect the data communication line from man-in-the-middle attack) to another actor (e.g., SWIM), but does not trust the latter with the same object (e.g., the data communication link). The graph pattern that characterizes the undesired configuration of elements was previously shown in Figure 8. In context of the ATM example, Act1 can be the Actor CWP, which delegates (through a delegation relation captured in variable Del) the Goal Receive (matching the variable Obj) to Actor SWIM (which will be Act2); this variable substitution is a match of the pattern as there is no trust relationship Tru between these two actors over this goal in the model.

After assembling the graph pattern, the event and condition specifications will have to be derived from it. We can create several evolution rules, one for each possible elementary change that can complete the pattern and make an intervention necessary. This will produce an outcome similar to Solution 1 presented in Section 7.5.2. Alternatively, simpler and more concise rules can be used, similar to Solution 2 from Section 7.5.3, if the mitigation only depends on the after state, and not on the nature of the change itself.

Regardless of the chosen approach, the action part can alert the argumentation engineers, or perform automated intervention by directly manipulating the model if the mitigation is close to deterministic. See Solution 3 from 7.5.4 as an example. Some of the change patterns introduced in D2.1 can be considered as possible candidates for being automated with evolution rules.

The given solutions can be demonstrated by applying them on the example models that represent the before/after situations in the ATM domain. Observing the After situation more closely, one can notice that contrary to the old communication system, the new SWIM system is not yet trusted by actors such as CWP and FDP. This may be a security issue, as the goals Send and Receice are now delegated to SWIM, which obviously requires trust. Fortunately, the example evolution rules presented in Section 7 can be used to automatically detect untrusted delegations. For example, if we use the general evolution rules introduced earlier, they will be triggered for multiple individual matches by this example evolution. The rule matches the rule variables to actual substitutions that experienced the Event and satisfy the Condition. In one concrete match, Obj will be mapped to the goal Send, and Act1 will be mapped to FDP; in a second case, Obj will be the goal Receive and Act1 will be CWP; Act2 will be mapped to SWIM in both cases. Engineers will be able to choose from three options for each individual match: to fill in the missing trust link (this is the likely solution in our case), to abolish the delegation, or to build an argumentation explaining why there is no real problem.

8.5 Interaction of argumentation and evolution rules

As discussed before, there are several ways for the evolution rules and the argumentation process to interact. It is expected that the engineers responsible for the argumentation can define domain-specific evolution rules that automatically maintain some information related to the arguments in the model. In an ideal scenario, such automation could always identify which arguments should be manually revisited, and which are unaffected by a change in the requirements model. Of course in most cases, there is no need to revisit each argument; if the set of rules for flagging arguments is comprehensive, relying on this automated process can save manual effort.

In this ATM example, an event that can trigger an automated response in relation to an argument can be the introduction of an attacker with an anti-goal against the “Prevent Unauthorized Access” goal of SWIM. In this case, the argument in support of the security goal should be flagged for manual re-evaluation. We show how argumentation experts using the evolution rule language of SeCMER can define such a rule:

```
evolution rule AttackerInvalidates {
    variables = (Atk, AG, SecG, Arg, W1, D1, S1);
    event = appear {
        entity Attacker(Atk);
        relation Actor.wants(Atk–W1→AG);
        entity AntiGoal(AG);
        relation AntiGoal.denies(AG-D1→SecG);
        entity SecurityGoal(SecG);
    }
    condition {
        entity Argument(Arg);
        relation Argument.supports(Arg-S1→SecG);
```

```
        entity SecurityGoal(SecG);
    }
    action {
        // flag argument as potentially invalid, notify argumentation team
    }
}
```

Here is one example of iterative development of the argument triggered by the evolution rule. Typically such development is in the form of a dialogue. The first round of an informal argument might be:

Initial claim:

- The ATM system remains secure after introducing AMAN (C1).

Initial facts:

- The AMAN system is controlled by a new trustable operator called Sequence Manager (F1).
- Sequence Manager reports to Sector Team about sequences (F2).
- AMAN interacts with the FDP, CNS, and Meteo services to collect the Airport Operators priorities, the Airlines priorities, the Meteo condition, and the aircraft position (F3).
- The actors are interconnected by the SWIM (F4).

Initial domain knowledge rule:

- If the members of the Sector Team obtain important information about the aircraft, information related to the aircraft position, for instance, the information may become available to a potential attacker. (DK1)

Initial Rebuttals:

- The Sequence Manager can have malicious intent due to social and psychological reasons (R1 on F1).
- Members of the Sector Team obtain critical information not related to their tasks (R2 on F4).
- Attackers eavesdrop on the SWIM network.

Second round, one checks the R1 as a claim. Here is the supporting evidence for R1**:**

- Each Sequence Manager has been through clearance to minimize the risk of being malicious F3=R1.1).
- Role-based access control policies for Sector Team will stop members of the team accessing critical information not relevant to their tasks (F4=R1.2).

Such argumentation can go on until all the facts and domain knowledge rules are refined so that all rebuttals of the root claim are not satisfiable. In other words, a satisfaction claim is justified as long as all the facts and domain knowledge are true (e.g., trust assumptions in arguing security goals) and all the rebuttals are false. A formal treatment of argumentation using non-monotonic proposition logic can be found in [12]. As one can see, the result of such argumentations would inevitably contribute to changes in the situations of security goals.

9 Conclusions

In summary, this report describes a methodology for addressing evolutionary security goals. It is based on three interleaving steps: modeling, analysis and design. First, models of evolutionary security goals are elicited and generalized according to three conceptual models. Then, in the analysis step, the models are used to discover vulnerabilities. Finally, in the design step, requirement models are used to construct a traceability mapping into security constraints of design artifacts.

These conceptual models are by no means an ultimate answer to the conceptual modeling framework for evolution of security goals. Rather, they can be considered as an extensible framework in which new concepts and practices in the field of evolving security goals engineering can be represented.

We envisage that observations from our discussion may have important implications for research in secure software evolution. The main implication concerns approaches to secure change impact analysis. For example the observation that changing requirements may lead to changing specifications could lead to a framework for understanding the impact of changes and traceability of the changes through artifacts in both requirements and specifications.

Similarly, such a change impact analysis framework could also be useful for analyzing the impact that changes in context may have on requirements and specifications. The change impact framework can be validated by doing more research on what the interaction is between the changes in $W, S \vdash R$. Related to this, is the issue of scoping the impact of change on the system, when the system is large.

As a result, the conceptual models presented will be considered together to shed some light on what is the more general representation of the meta-conceptual model, in order to facilitate the classification of changes, the change impact analysis, the transformations of the models, and the argumentation of satisfaction. Ultimately, security goals change patterns may be discovered, be documented and be reused from one case study to another.

10 Acknowledgement

We thank Prof. Michael Jackson for his insightful comments on the conceptual models being developed earlier.

References

[1] J. Alferes, F. Banti, és A. Brogi: “An Event-Condition-Action Logic Programming Language”. *Lecture Notes in Computer Science* 4160, pp. 29-42, 2006.

[2] Becker, S.M., Haase, T., Westfechtel, B.: “Model-based a-posteriori integration of engineering tools for incremental development processes”. *Journal of Software and Systems Modeling 4(2):123-140,* 2004.

[3] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. “Incremental pattern matching in the VIATRA model transformation system”. In Gabor Karsai and Gabi Taentzer, editors, *Graph and Model Transformation* (GraMoT 2008). ACM, 2008.

[4] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró: ”A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation”. *Lecture Notes in Computer Science* 5124: pp. 396-410, 2008.

[5] Jean Bézivin: “On the unification power of models”. *Journal of Software and System Modeling* 4(2): 171-188, 2005.

[6] K. Czarnecki and S. Helsen: “Feature-based survey of model transformation approaches”. *IBM Systems Journal* 45(3): 621–645, 2006.

[7] “Deliverable 4.1: Security modeling notation for evolving systems,” SecureChange (EU ICT-FET-231101), Unpublished Draft Report ICT-FET- 231101 D4.1, 2009.

[8] “Deliverable 5.2: Documentation of forecasts of future evolvement,” SecureChange (EU ICT-FET-231101), Unpublished Draft Report ICT-FET- 231101 D5.2, 2009.

[9] C. L. Forgy. “Rete: A fast algorithm for the many pattern / many object pattern match problem”. *Artificial Intelligence*, 19(1):17–37, September 1982.

[10] Gangemi, A. and Guarino, N. and Masolo, C. and Oltramari, A. and Schneider, L. Sweetening ontologies with DOLCE. Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web, 323-333, 2002.

[11] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, Andrew Wood: ”Transformation: The Missing Link of MDA”. In: *Proc. of International Conference on Graph Transformations (ICGT)*, pp 90-105, 2002.

[12] Charles B. Haley, Robin C. Laney, Jonathan D. Moffett and Bashar Nuseibeh. “Security Requirements Engineering: A Framework for Representation and Analysis”. *IEEE Trans. Software Eng.*, 34(1): 133-153, 2008.

[13] M. McGrath. Propositions. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Phylosophy*. Fall 2008.

[14] Fabio Massacci, John Mylopolous, Nicola Zannone. Computer-aided support for Secure Tropos. Automated Software Engineering. 14(3) (2007) 341-364.

[15] John Mylopoulos, Lawrence Chung, Brian A. Nixon. ”Representing and Using Nonfunctional Requirements: A Process-Oriented Approach”. *IEEE Trans. Software Eng.* 18(6): 483-497 (1992)

[16] Armstrong Nhlabatsi, Bashar Nuseibeh and Yijun Yu. ”Security Requirements Engineering for Evolving Software Systems: a Survey”. *Journal of Secure Software Engineering 1(1):54-73*, 2009.

[17] Normand, V., Felix, E., Jitia, C. “A DSML for security analysis,” *IST MODELPLEX project restricted deliverable 3.3.g.* 2009.

[18] István Ráth, Gábor Bergmann, András Ökrös, Dániel Varró: “Live Model Transformations Driven by Incremental Pattern Matching”. In: *Lecture Notes in Computer Science* **5063**: pp. 107-121, 2008.

[19] Rensink, A. (2004) *Representing First-Order Logic Using Graphs.* In: International Conference on Graph Transformations (ICGT). pp. 319-335. Lecture Notes in Computer Science 3256. Springer Verlag. ISSN 0302-9743 ISBN 978-3-540-23207-0

[20] Ehrig, G. Engels, H. Kreowski, and G. Rozenberg, Eds. 1999 *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*. World Scientific Publishing Co., Inc.

[21] Rational, “DOORS Homepage”, http://www-01.ibm.com/software/awdtools/doors/, fetched 2010.

[22] A. van Lamsweerde. “Goal–oriented requirements engineering: A guided tour”. In *Proceedings of the International Requirements Engineering Conference (RE 01)*, pp. 1-10, 2001.

[23] Thein Than Tun, Rod Chapman, Charles B. Haley, Robin C. Laney, Bashar Nuseibeh: A Framework for Developing Feature-Rich Software Systems. ECBS 2009: 206-214

[24] Yijun Yu, Jan Jurjens, John Mylopoulos. “Traceability for the maintenance of secure software”. In: *Proc. of the 24th Int. Conf. on Software Maintenance*, pp. 297-306, IEEE, 2008.

[25] Zave, P. and Jackson M.: “Four dark corners of requirements engineering”. *ACM Transactions on Software Engineering and Methodology*, 1997. 6(1): p.1-30.

Glossary

A *claim* is a (probably grounded) predicate whose truth value will be established by an argument., 24

A *condition* of situation is evaluated to be true for situations that require a change to maintain the security requirements. It may evaluate to false if the triggering events do not lead to any change. The condition must be monitored whenever a triggering event happens., 18

A *triggering event* is a dynamic difference between two consecutive versions of a model that results in the activation of an evolution rule., 18

An *adaptation action* is the change introduced to achieve or restore the maintenance conditions, which are, in the SecureChange context, the satisfaction of security requirements., 18

An *argument* contains one and only one claim. It also contains facts and rules in domain knowledge, 24

An *evolution rule* is a formal specification of automatic behavior in reaction to changes in the model, 17

Domain Knowledge is a set of ungrounded predicates that can be evaluated to true or false once the values of all terms in the predicates are known., 24

DSML stands for Domain Specific Modeling Language., 2

Dynamic Oriented Object Requirement System tools dedicated on Requirement Management. For further details see [21]., 57

Facts are grounded predicates -- something that are either true or false where terms in these predicate must be constant, 24

Transformation is the process of deriving models from each other, 32

Mitigations are *another* special kind of arguments following the iteration of rebuttals in order to reestablish the truth value of the associated claims, 24

Rebuttals are a special kind of arguments whose purposes are to establish the falsity of their associate claims or make them indeterminable, 24

situation, 17

Targets are elements of physical layer specification (physical component, communication channel)., 59

The argument may require *sub-arguments* to establish the truth of certain facts or intermediate predicates, 24

Appendix 1. State of the Practice

In this section we present the Thales industrial method for security risk analysis, and we show the analogies with our methodology for security goals elicitation and analysis. Thales method aims at supporting the analysis and assessment of security risks for a system, and the specification of requirements for security measures to address those risks.

1. The security risk analysis method: Principles

Our prospective security risk analysis method builds upon model-based engineering methods and techniques. All activities of our method are organised around the building and usage of models, that is formalised, precisely defined, interconnected and integrated representations of the objects under study.

As represented in Figure 11 our proposed method relies on the development of a modelling framework that combines in a synchronised way a set of models that constitute separate viewpoints [17] over the engineering problem:

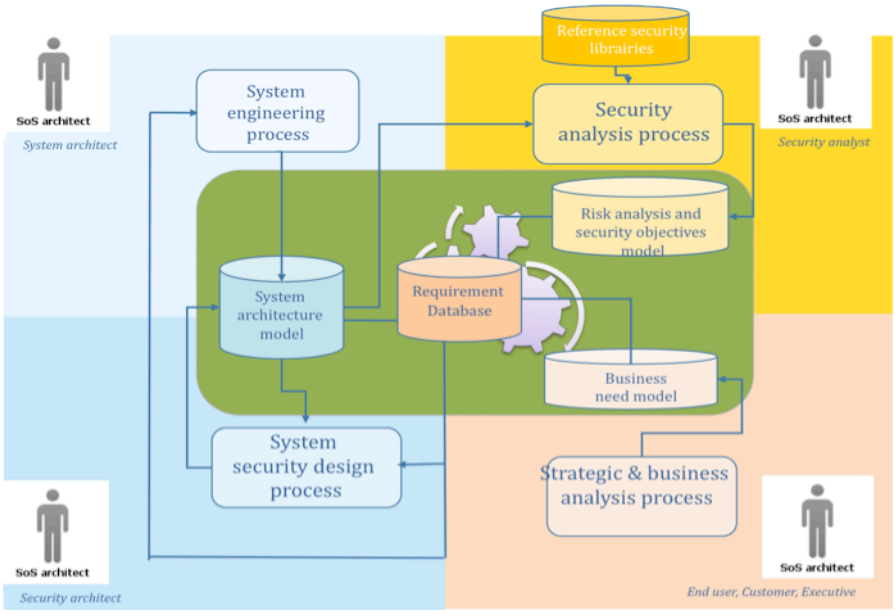

Figure 11. The security analysis method in Thales context – big picture

- The System architecture model contains the architectural design of the system; this model is developed within the mainstream engineering processes, along at least two dimensions: the functional/logical architecture of the system (functional capacities and data to be realised by the system) and the physical /implementation architecture of the system (actual hardware and software components that realise the functional capacities).

- The Business need model captures a representation of the business context for the system: business process that is supported, underlying business organisation, business objects, key performance indicators, strategic drivers, etc.
- The Risk analysis model and security objectives model capture the results of the security risk analysis method that is proposed in dedicated DSML (presented in next section). These models include a representation of the system architecture that is relevant to the needs of the security analyst, this model is called context model. This model is traced back and maintained in synchronisation with the system architecture model (see [12]). The security risk analysis information is defined as annotations or related new concepts added over the system architecture elements. The risk analysis model and security objectives model may also be traced to elements of information defined in the Business need model.
- The Requirement Database captures all kinds of systems requirements (Security, Safety, Maintainability, Cost, etc.). Security goals are derived from security objectives model of dedicated DSML (see [13]). This mapping enables to add security goals with other kind of requirement addressed for a complex system. Requirement Database is traced back and maintained in synchronisation with the system architecture model and Business need model.

The System architecture model and the Business need model are part of architecture modeling framework that we are developing to address service-oriented types of large-scale enterprise integration systems or systems of systems. In the Thales context, the official database of Requirement Management is Rational DOORS with the T-REK add-ons [21].

2. DOORS T-REK

Rational DOORS [21] (Dynamic Object Oriented Requirements System) provides:

- A requirements Database that allows all stakeholders to participate in the requirements process
- The ability to manage changing requirements with RCM Tools (Requirement Change Management)
- Powerful life cycle traceability to help teams align their efforts with the business needs and measure the impact that changes will have on everything from business goals to development
- Links requirements to design items, test plans, test cases and other requirements for easy and powerful traceability
- Automatic generation of traceability matrix.
- Automatic document generation of DOORS module into MS WORD format (.doc).

As suggested by Figure , a DOORS project is composed by two kinds of modules:

- ***Formal Modules*** gather requirements information and is used for Requirement Specification. One Requirement is considered as one object which contains a

set of attributes (standard attributes are Object Identifier, Object Heading and Object Text). It's possible to filter some attributes in views.

- ***Link Modules*** gather links information. Links module contains a set of ***Linksets*** which represent link information between two Formal Modules.

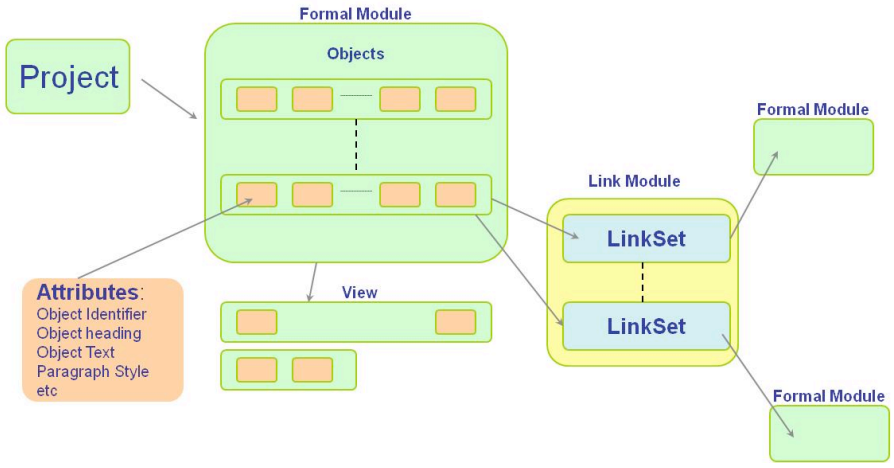

Figure 12. DOORS project structure

T-REK (Thales Requirement Engineering Kit) is an over-layer of DOORS which enables to distinguish different kinds of Formal Modules and Link Modules. ***T-REK*** offers a Relationship Manager to represent a project structure and relations between different formal modules: we call it a ***Datamodel***. In a simplified Datamodel as shown by Figure 12 we distinguish:

- Requirement Module, which represents Requirement Specification Document (it's possible to distinguish User Requirement Specification and System Requirement Specification). The link between this kind of module corresponds to "satisfies" link.
- Integration, Validation, Verification (IVV) Module, which gathers integration and tests campaign information (e.g. Test Result, Expected Test Method ...). IVV modules are linked with Requirement module by a "verifies" link.
- Product Breakdown Structure (PBS) Module, which contains all subsystems or components (depending on project granularity) and all related information (e.g kind of component software, hardware ...). Components/Subsystems are represented by a DOORS object. Requirements modules are linked with PBS modules by a "is allocated to" link.

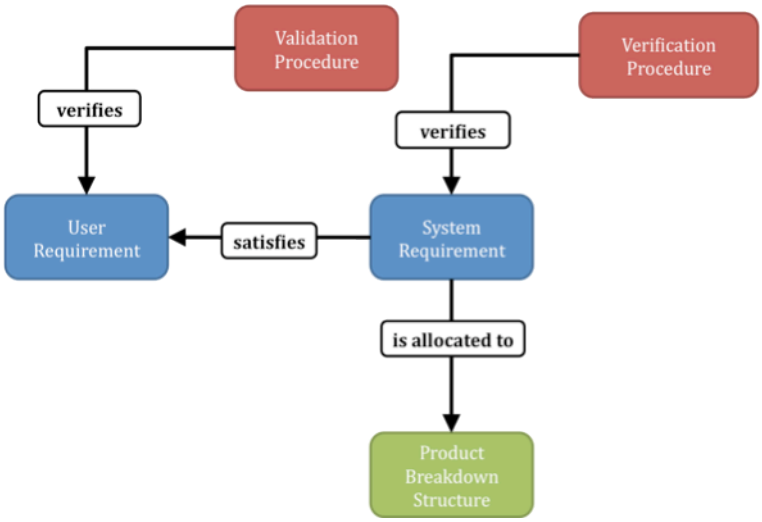

Figure 13. Simplified Datamodel in T-REK

Risk are not represented in Standard T-REK Datamodel, this is why we plan to connect our DSML based on Risk analysis with DOORS T-REK.

3. Application in Thales Requirement Workbench

This deliverable cannot be the place for a detailed presentation of the conceptual model and syntax of DSML. We are providing below representative extracts. More details are provided in [17]. The core part of the conceptual model[4] is represented in Figure 14.

The system under analysis is considered to hold targets and essential elements. Targets are physical elements subject to risk

Key elements are usually more logical, functional elements: data and functions (or services, or capabilities depending on context) that are essential to the business stakes of the company, and therefore subject to security needs. Key elements depend on targets for their implementation.

Requirements and Objectives are allocated to Essential Element and/or Target. To ensure risk traceability, Objectives and Requirements must cover Risk(s). Objective must be more general than Requirement, and to preserve traceability between those concepts, we consider a bidirectional association named “satisfies” between them.

[4] For readability sake, it is represented in the form of a conceptual model rather than a formal conceptual model.

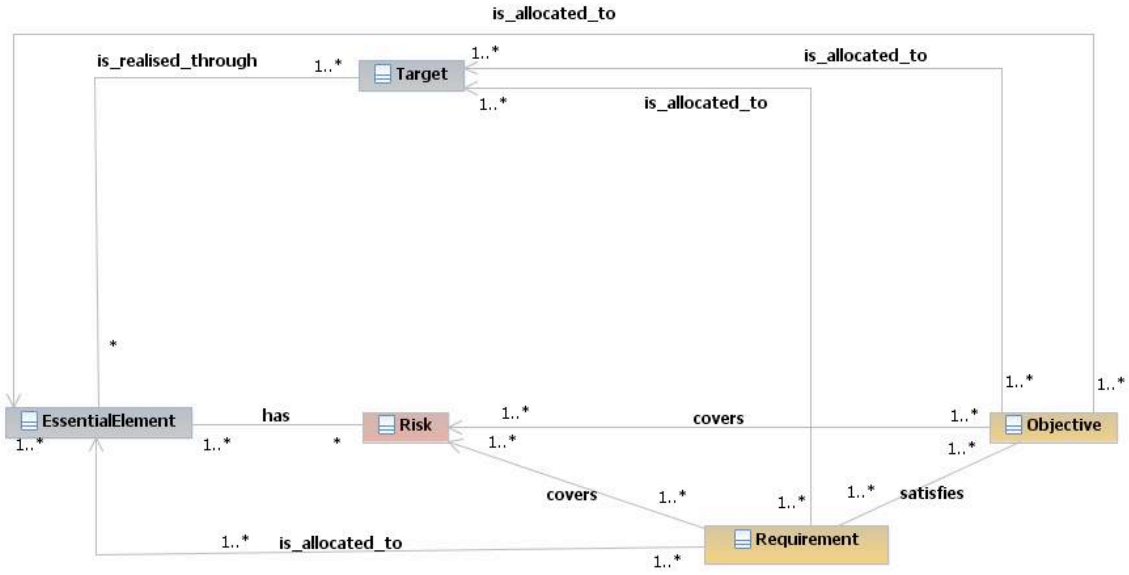

Figure 14. Conceptual model of Security Objectives and Requirements in Security DSML

In current Security DSML, we distinguish three kinds of static models[5] as shown by Figure 15:

- ***The Requirement Model*** describes the specialization of Objectives into several Requirements and links between those and the other elements of DSML (Risk, Context).
- ***The Context Model*** describes System Architecture (Essential Elements and/or Target), related constraints and links between those and the other elements of DSML (Risk, Requirement).
- ***The Risk Model*** describes the risk characterization into threats, damages and vulnerabilities and links between those and the other elements (Risk, Context).

[5] The connectors between entities are not represented here for readability sake

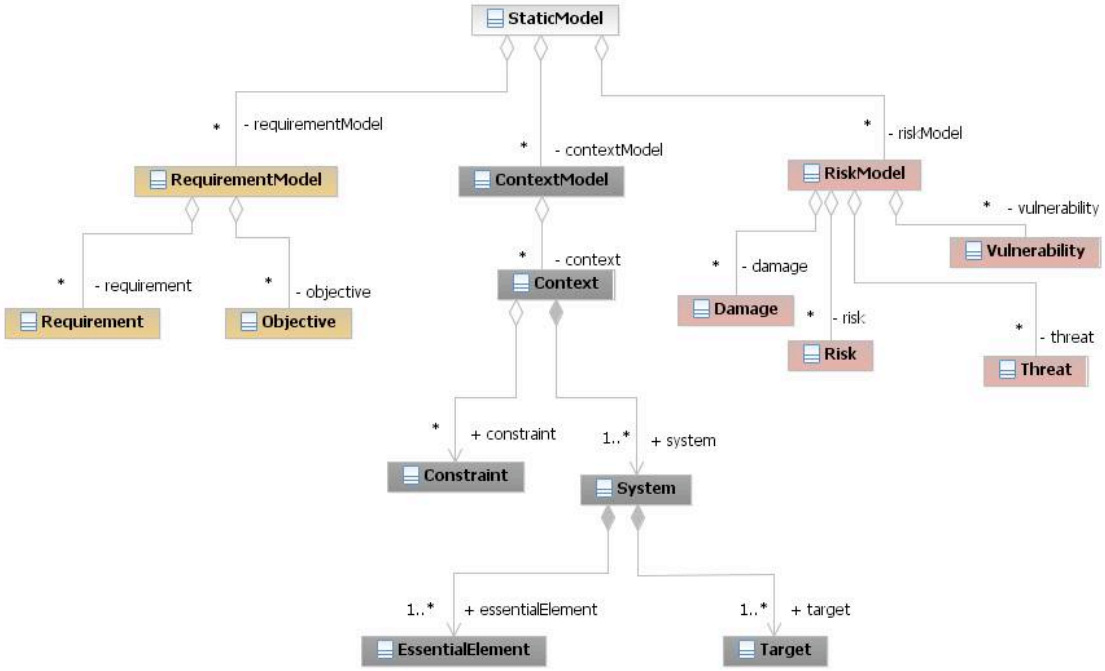

Figure 15. Security DSML Static Model description

Figure 16 shows how to realize the mapping between Thales Security DSML (or Other DSML for Need Analysis) and DOORS T-REK, to do this we must consider a **Traceability relation** between Security Goal of Security DSML and DOORS Requirements.

This relation enables to connect other kind of requirement (Safety, Maintainability, Cost, etc.) with Security Goals expressed in DSML. Requirements are stored in a common requirement Database (DOORS Database). This communication is realized via a Model Bus (Bidirectional interface XML to DXL[6]) for Traceability needs between DOORS and Security DSML.

[6] DXL (DOORS Extended Language) is the native language of DOORS

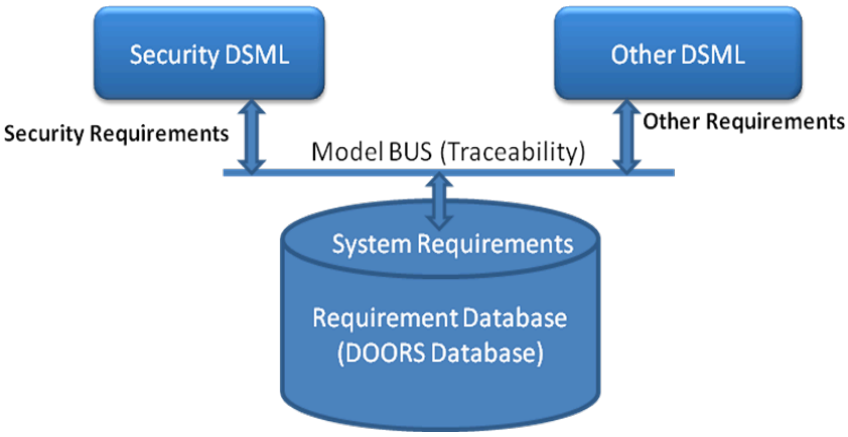

Figure 16. Mapping between DSML and DOORS

This connection enables to represent risk defined in DSML into a requirement attribute (Related Risk) and to connect Related Threat and Vulnerability into a component attribute. It's so possible to represent risk into DOORS objects.

Figure 16 presents the extended conceptual model including DOORS connections. Two kinds of entities are mapped with DOORS: Requirements and Target that are respectively represented by Requirement and Product Breakdown Structure object in DOORS. To ensure traceability between DSML and DOORS, we add a PUID (Product Unique IDentifier) attribute, PUID is the reference name of a DOORS object.

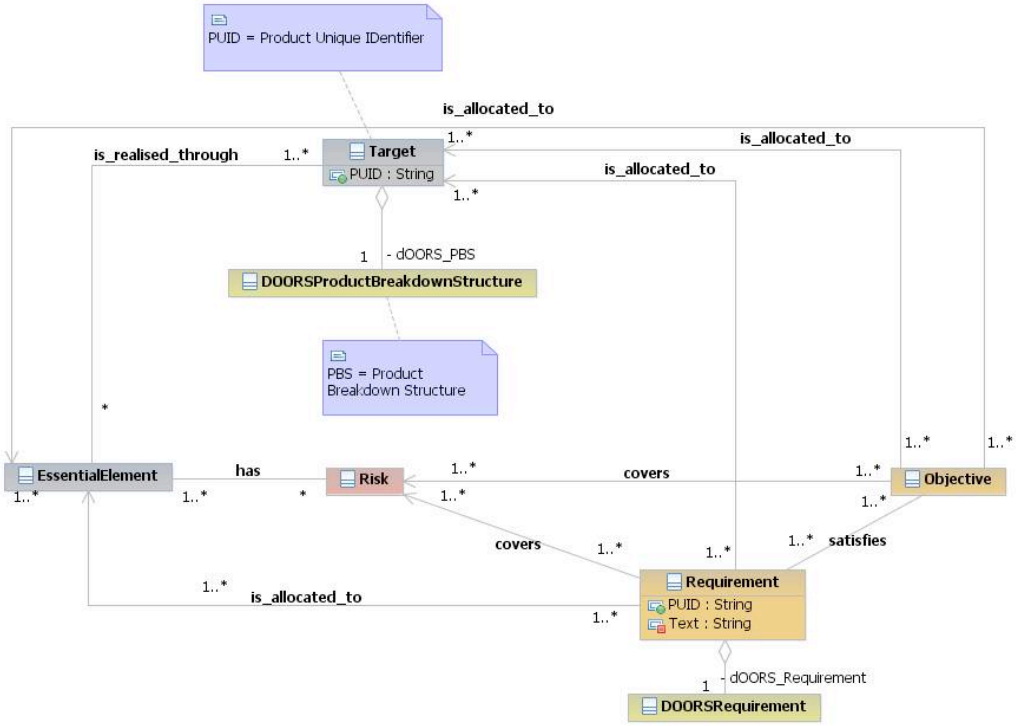

Figure 17. Extended Conceptual model including DOORS connections

Figure 18 depicts the properties view on Security Objective O6 (Identifiers should be chosen so that they do not compromise user's privacy). Figure 19 presents the requirement derived from security objective in DOORS.

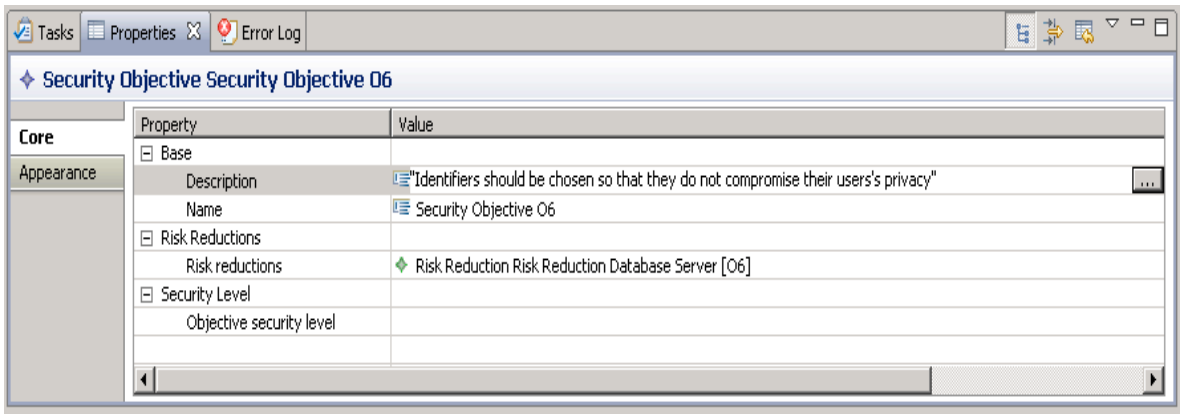

Figure 18. Close view on the Security Objectives

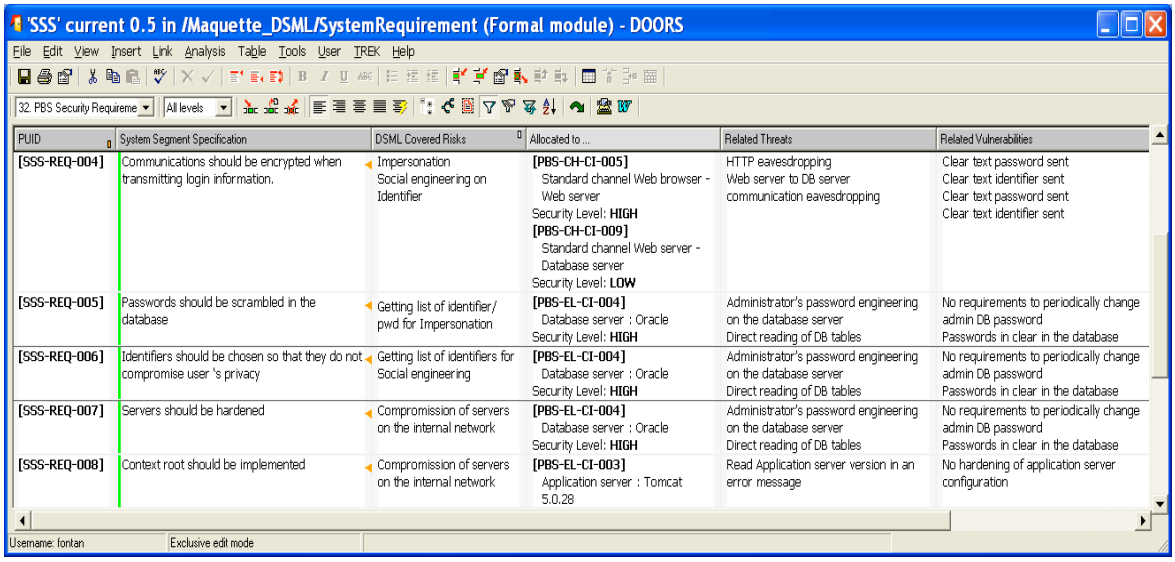

Figure 19. Derived Requirements expressed in DOORS

The information of target can be consulted in the Properties View (Description, constraints applied on it), as can be seen in Figure 20. This properties view of Target is also defined in DOORS as shown by Figure 21.

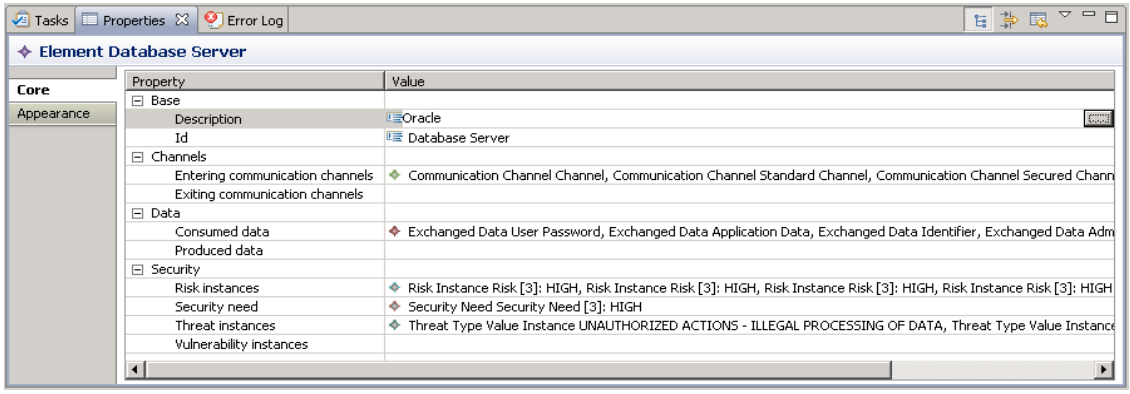

Figure 20. Properties of the Database Server in DSML

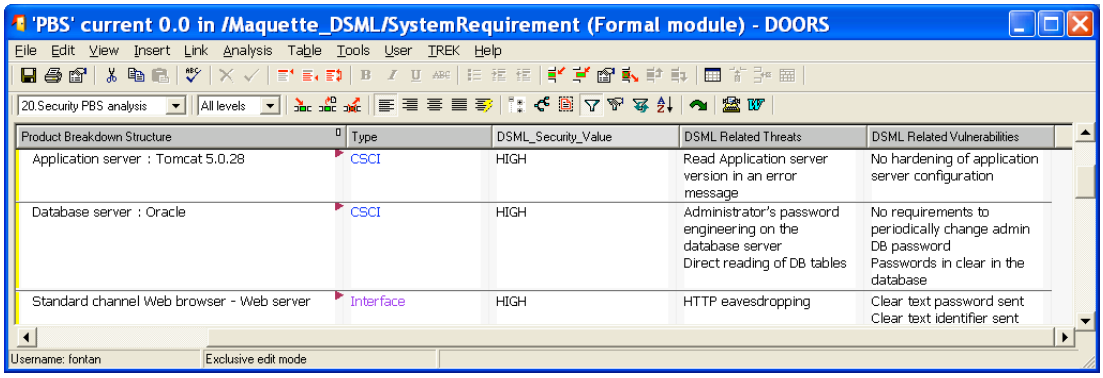

Figure 21. Database Server description in DOORS