D.3.2 METHOLOGY FOR EVOLUTIONARY REQUIREMENTS

Gábor Bergmann (BME), Elisa Chiarani (UNITN),Edith Felix (THA), Benjamin Fontan (THA), Charles Haley (OU), Fabio Massacci (UNITN), Zoltán Micskei (BME), 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	1.33
Status	Final
Work Package	WP 3
Deliverable Type	Report
Contractual Date of Delivery	31 January 2010
Actual Date of Delivery	29 January 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 meta-models in Section 3 and 5. Added the mapping of concepts in the Thales meta-models in Section 3 to the

				general one proposed. Fixed the references. Note the new meta-models are also uploaded as the source file for UMLet.
1.10	13 November 2009	Draft	Charles Haley (OU)	Checked with the meta-models about Security Requirements and Argumentations. Also edited the definitions.
1.11	26 November 2009	Draft	Gábor Bergmann (BME)	Revised evolution rules Meta-model, added initial example
1.12	7 December 2009	Draft	Yijun Yu (OU) Charles Haley (OU) Bashar Nuseibeh (OU)	Revised the methodology, refined the meta-models to highlight the contributions. Drafted the change management meta-model to be consistent with the discussion notes.
1.13	8 December 2009	Draft	Yijun Yu (OU) Thein Tun (OU)	Revised the meta-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 Requirement Analysis

				in Thales Context) Add subsection 3.2.2 (Application of meta-model 3.2 in Thales Requirement Workbench) Add subsection 3.3.2 (application of meta-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 meta-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	Received the reviewing comments

			(UNITN)	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

Executive summary

As a software system evolves, security concerns need to be analyzed to re-evaluate the impact of changes on the system and the assumptions on environmental properties.

Traditionally, the security requirements were handled in an ad-hoc way, while requirement models are often embedded in natural language descriptions which lead to inconsistent interpretations with respect to the meaning of the requirements. These made it difficult to analyze for requirements changes. By adopting a model-based engineering methodology, we propose to investigate such changes using a consistent conceptual model of evolving security requirements which incorporates the state-of-art requirement modeling languages such as Tropos and Problem Frames. As a unified extension to existing Security Requirements frameworks (e.g., Secure Tropos and Abuse Frames), our new meta-model is explicit in representing target specifications where vulnerability can be revealed. Essential elements such as threats are also made explicit in order to analyze attacks that are assumed to be present in a hostile operating environment. The overall goal of the model is to provide mechanisms for protecting valuable assets from damage. Using our conceptual model for security requirements, we observe how it is possible to construct arguments to examine the security of systems as they change.

To address the challenge of evolutionary security requirements, we lay out the conceptual meta-models, and the general methodology to handle changes on security requirements, including how to represent security requirements, how to model the changes of them, how to manage the changes and how to argue that the changes are fit for the purposes. As a result, we obtained a consistent meta-model representing the key concepts related to security requirements, which not only improves the elicitation of security requirements, but also enables further analysis at the design and validation stages.

Index

DOCUMENT INFORMATION 1

DOCUMENT CHANGE RECORD 2

EXECUTIVE SUMMARY 6

INDEX 7

1 INTRODUCTION 9

2 METHODOLOGY FOR EVOLUTIONARY SECURITY REQUIREMENTS 11

3 MODELING SECURITY REQUIREMENTS 14

3.1 Meta-model for requirements representation 14

3.2 Meta-model of Security Requirements Evolution 16

4 ARGUMENTATION OF SECURITY REQUIREMENTS 19

5 AN EXAMPLE OF REQUIREMENTS EVOLUTION MODELING 21

6 SPECIFYING RULES FOR EVOLUTIONARY CHANGES 24

6.1 Goals for the evolution rules 24

6.2 Using model transformations for evolution 24
6.2.1 Incremental transformations 25
6.2.2 Event – Condition – Action semantics 28
6.2.3 Change-driven transformations 29

6.3 Meta-model for evolution rules 30

6.4 Examples for evolution rules 32
6.4.1 Problem description 32
6.4.2 Solution 1: one rule per elementary change 32
6.4.3 Solution 2: single coarse-grained rule 33
6.4.4 Solution 3: automatic problem correction 34
6.4.5 Example rule application 35
6.4.6 Further discussions 36

7 SECURITY REQUIREMENT ANALYSIS IN PRACTICE 37

7.1 The security risk analysis method: Principles 37

7.2 DOORS T-REK 38

7.3 Application in Thales Requirement Workbench 40
7.3.1 Thales Security DSML 40
7.3.2 Mapping between DSML conceptual models and the security requirements meta-model 42
7.3.3 From DSML to DOORS T-REK 43

Eliminato: 42

7.4 Evolution Management 46
7.4.1 ChangeLine Meta model 46
7.4.2 ChangeRequest Meta-model 47

8 CONCLUSIONS 50

9 ACKNOWLEDGEMENT 51

REFERENCES 52

GLOSSARY 54

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 features; 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 requirements, a system may be left vulnerable to attacks [14].

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

- Ad hoc elicitation of security requirements. Most security requirements are implicit or are added after security violations have happened, which makes it difficult to prevent the problems and address the vulnerability in a proactive way;
- Imprecise modeling of requirements. Security requirements, by their very nature, demand a precise description that can be used to analyze, argue and evaluate. Vaguely expressed informal natural language descriptions, such as the requirement traceability matrix in DOORS, are difficult for analysts to give an assessment of the problem and to provide useful mitigation advices;
- Change management of security requirements are not integrated with the requirements modeling tools. It requires an explicit mapping between the changes of security requirements and the system vulnerability to be able to assess their impact on the system-to-be. Due to the large gap between the requirements and the design and implementation, mitigation is often a late response to continuous evolution of life-long software systems.
- Even when changes have happened as systematically, there is a lack of mechanism to formally argue 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? As the security requirements are often proposed by stakeholders, it is important to reach an agreement between them 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 [22], 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 hopes to observe the phenomena in the future.

These relationships between the properties establish a structure in order to facilitate the analysis of the entailment relation *W, S* ├ *R*.

In addition, to extend Jackson's framework to consider security, 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 requirements 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 requirements 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 encompasses.

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

- A problem-oriented analysis that relates the changes of security requirements 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 one, 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 requirements 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-meta-model that permits description of all changes. Throughout the document, we illustrate the models using some examples in the Arrival Management (AMAN) system from the air traffic management domain.

2 Methodology for Evolutionary Security Requirements

As mentioned earlier, the **challenges** of evolving security requirements engineering arise from multiple facets of engineering problems. ***Existing methodologies*** deal with the changes in security requirements 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 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 requirements. Such a combination could **benefit** from the strengths of individual methodology, **making clearer about the situation** of the subject system in terms of security requirements. 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 analyses, 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.

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 requirements. 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.

Putting these individual control loops of evolution rules together, our methodology will be applied and supported by adapting the existing change management approaches such as DOORS. The **introduction of new requirement attributes** will help the documentation of security requirements, paving a way for the expression of the formal requirements models.

In summary, our methodology for addressing evolutionary security requirements is based on three interleaving steps: modeling, analysis and design.

- During the modeling step, models of evolutionary security requirements are elicited and generalized according to three meta-models that capture respectively any representation of security requirements, any changes of security requirements, any managing processes of a security requirement and any argumentation about security requirements.
- During the analysis step, the models of security requirements will be used to serve the reasoning process, to discover any vulnerability early on, in order to fix them before it is too late in the design/implementation phases in software development.
- During the design step, the screened security requirement model will be used to construct a traceability mapping into security constraints on the design artifacts for the implementation purposes.

At this stage of the project, our focus is put on the modeling step, which will be detailed in the following sections including the following sub-steps:

1. Model the problem-oriented concepts $W, S \vdash R$ with richer concepts from Tropos requirements engineering methodologies (and their refinement to security)
2. Validate your design models against security requirements through argumentation
3. Define the changes you accept, and what is the action to take to create a consistent new state
4. Define how to control the changes (when you monitor the events and conditions and resolve them through actions, etc)

It is our aim to be able to define the extra attributes needed for the Thales's DSML+DOORS tool to elicit security requirements from the customers, and our objective to apply the methodology to the ATM and SmartHome case studies to cross validate the following research questions: Are all these concepts needed and useful in practice? Is there is anything missing in practice? Does the tool support improve the productivity in eliciting and analysis of the changes of security requirements? It is our hope that a series of detailed case studies will gain clear answers to these questions and ultimately justify the research results.

The meta-models for the above four steps are detailed in Sections 3-6 respectively. Section 7 illustrates the use of these steps in a change management process. Section 8 concludes the document with lessons learnt.

3 Modeling Security Requirements

In this section we present, first, the meta-model to represent requirements, and then we show how one can represent the evolution of requirements using the notions of *situations* and *evolution rules*.

3.1 Meta-model for requirements representation

In this section we introduce a new meta-model for requirements elicitation that supports the representation of dependencies between requirements, the system-to-be, and the context in which the system is going to operate, and of security related concepts necessary to reason on security requirements satisfaction. In fact, our meta-model inherits concepts from problem-frames and goal-oriented requirements elicitation approaches, and risk analysis approaches. From problem-frames we borrow the dependency between requirements, the system-to-be, and the context in which the system is going to operate. In fact, according to Zave and Jackson [22], 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*, where *S* and *R* are the *optative* domain properties. From the Secure Tropos methodology, we borrow the representation of the requirements of the system-to-be using the notion of *goals, softgoals and quality constraints;* and of how the system-to-be satisfies the requirements concerning the objects such as actors, processes and resources. From the security analysis methodology, we borrow the concepts of vulnerabilities, attackers and attacks. *Asset* in this meta-model is similar to *target* in the risk meta-model of WP5, where it is related to *risk* through *threat* and *vulnerability*.

Figure 1 represents the entities characterizing our meta-model to represent requirements and the relations between them.

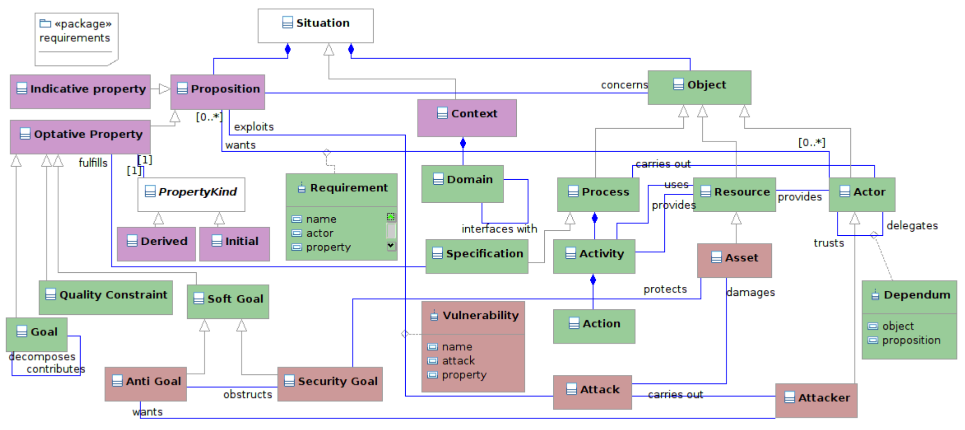


Figure 1. The meta-model for requirements elicitation

We have outlined in green the concepts inherited from the goal-oriented approaches, in magenta the concepts taken from Problem Frames approaches, and in red the concepts borrowed from risk analysis approaches.

A *situation* of our requirements model is expressed in terms of propositions and objects. *Propositions* are the sharable objects of attitudes and the primary bearers of truth and falsity. A proposition can be an optative or an indicative property concerning objects. An *object* is an actor, a process or a resource. When a stakeholder (actor) *wants* a desired or optative property, it is modeled as initial *requirements,* which can be *refined* into *derived requirements. Therefore, requirements* are desired or optative properties that the system-to-be ought to have, as wanted explicitly by stakeholders. Initial requirements and derived requirements can be captured by *goals*, the objectives that the system-to-be should achieve. A derived requirement can also be a *soft goal*, which does not have a clear-cut evaluation of the truth value. . A security goal expresses that an asset needs to be protected from harms. An anti-goal is a goal of an attacker which may obstruct the achievement of a security goal. Both security *and anti-goals are soft goals*.

Unlike requirements, a *specification* fulfils certain requirements under given indicative domain properties.. It usually captures certain dynamic behavior in order to satisfy software requirements; therefore specifications are modeled as processes.

Objects are entities used to describe a state of the world. An object can be dynamic or static*.* A static object can be an actor or a resource. An *actor* is an intentional entity such as a human, a device, a legacy software or software-to-be component that performs actions to achieve its own goals. We consider an *attacker* as a particular actor who wants an anti-goal to be satisfied. A *resource* is a physical or an informational entity which has no intention by itself. An *asset* is a resource which has a value and needs to be protected. V*ulnerability* is a weakness, a flaw or a deficiency that is exploited to carry out an attack which causes harm to or damages an asset. A dynamic object can be a *process* that consists of *activities*. An *activity* is a sequence of *actions* that can be performed by an actor to fulfill a goal.

A situation is a partial state of the world where some propositions are true and some other propositions are nor true nor false. Thus, a situation consists of objects and propositions concern these objects. Particular types of situations are *context,* the *domain*, and an *attack*. The *context* is a situation within which the system-to-be will operate. A context consists of several domains which interface with each other. An *attack* allows an attacker to fulfill an anti-goal. In particular, an attack is a situation in which vulnerability is exploited to cause damage on an asset.

For requirements analysis, these entities are related in the following seven basic types:

- *Trusts* is a relationship from one actor to another, which indicates the belief of one actor that the other will provide a resource or will perform a certain activity ;
- Delegates is a relationship from one actor to another which specifies that the fulfillment of a goal or the provisioning of an activity/resource;

Both trusts and delegates relationship are associated with a *dependum,* which specifies which object (resource/process) or which requirement (goals, softgoals) are trusted or delegated from one actor to another.

- *Provides* is a relationship either from an actor to a resource, which specifies that an actor provides a certain resource; or from an activity to resources; *Uses* is the relationship opposite to Provides.
- *Carries Out* is a relationship either from an actor *to a process*s, which specifies that an actor carries out a certain activity; *Carries Out* is a relationship either from an actor to a process, which specifies that an actor carries out a certain activity.
- *Fulfills* is a relationship from resources and activities to a goal, which specifies a goals is fulfilled by a combination of the resources and the activities;
- *Wants* is a relationship from actors to goals which associates an actor with its goals, including security and anti-goals.
- *Contributions* is a relationship among goals/security goals which indicates that a goal contribute to the satisfaction of another goal.
- *Decomposes* is a relationship from a goal to its subgoals, which indicates that a goal can be refined: AND-decomposition lists subgoals that must all be satisfied in order to satisfy the goal, whereas OR-decomposition suggests alternative ways to satisfy the goal.

For security requirement analysis, the following seven specific relationships are considered on an attack situation and a security goal:

- *Attacks* is a relationship from one situation to a vulnerable actor;
- *Damages* is a relation from an attack to the assets;
- *Exploits* is a relationship from an attack to a vulnerability, which is a (part of) specification that can be vulnerable to expose security problems;
- *Protects* is a relationship from a security goal to a set of valuable assets;
- *Obstructs* is a relation from an anti-goal to the corresponding security goal.

Such problem analysis for goal satisfaction can be done using proposition logic qualitatively, or using risk analysis quantitatively. In either way, arguments on the fulfillment of security requirements need to be acceptable after a negotiation process during which the trusted domain assumptions may not always hold. Therefore, the framework as such can support extensively evolving security requirements.

3.2 Meta-model of Security Requirements Evolution

After specifying the static view of situations about the security requirements, 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. Discrete changes have a sequence of change descriptions associated with timestamps, while continuous changes happen continuously in that the intervals can be arbitrarily further refined and the length can be arbitrarily prolonged for the security requirements in long-lived software systems.

In a nutshell, the situations that can change in the model for requirements include generally entities and the relationships between them. In particular we consider elementary types of changes, including the modification, the addition and the removal of an element (such as an entity or of a relationship). An example of possible change is the addition of a new actor (as the event that matches with the condition) that results in the addition to the model of a new entity representing the actor and new relationships such as the “wants” relationship to specify the goals the new actor wants to achieve or a “provides” relationship from the actor to the resources and activities that it offers.

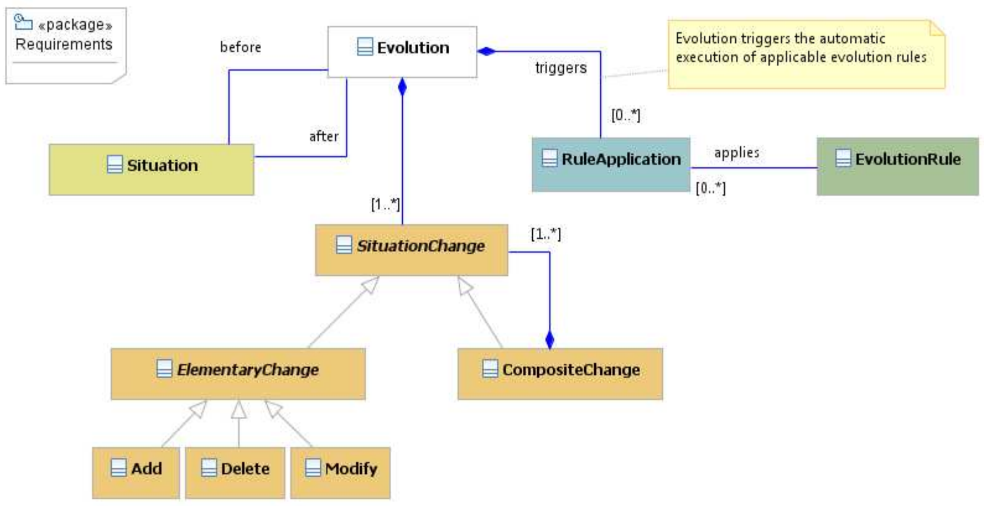


Figure 2. A meta-model for evolution of security requirements

A more complex kind of situation change can be described by a composite change, which is a transaction of elementary changes (or nested composites) that must happen together or not at all. For example, the deletion of an actor A may require the deletion of all the delegation relationships from A to another actor B, while finding for B alternative actors A’ that can provide the same activities and resources delegated to A, otherwise the incomplete change may violate the intention of B. Therefore we record such complicated changes as a transformation that preserve the satisfaction of certain high-level requirements.

A natural way is to represent the change as a transition rule between two situations, denoted respectively as *before* and *after situations* (see Figure 2). Intuitively, the before/after *situation* represents the elements in the model the change has occurred at a given time before/after an adaptation has been applied. The outcome of changes is monitored by evolution rules to decide whether an adaptation action needs to be taken. If yes, the change will trigger the application of a general evolution rule in a concrete place in the model, possibly causing additional changes.

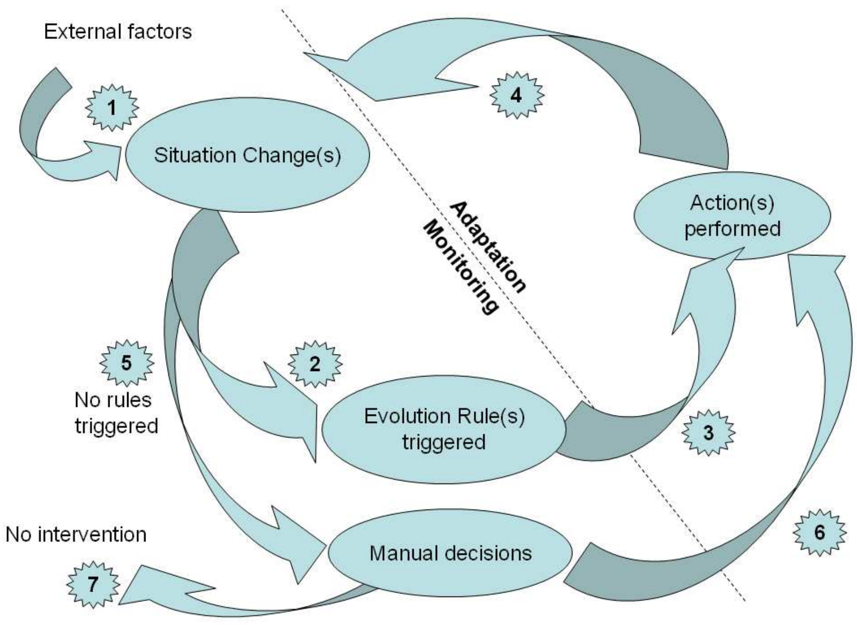


Figure 3. A continuous process to maintain the requirements

The envisioned workflow of maintaining design and requirement models through evolution requires the continuous adaptation of the models to react to changes. Changes in external factors – changed requirements, new threats, and revised design decisions – can be introduced into the model by engineers (or automatic monitoring in some cases); this model change, however, may violate constraints and requirements, cause inconsistencies. Therefore reactions are required to handle the effects of change. While most reactions will remain responsibilities of engineers, evolution rules can be defined to automatically adapt and transform the model in some cases. Failing that, rule-based automatic mechanisms are expected to be able to initiate the process of adaptation in many cases, or at least indicate the problem to the engineers.

The applicability of automatic evolution rules is greatly enhanced by machine-understandable, domain-specific refinement of the general requirement modeling concepts appearing in this deliverable. Therefore SecureChange provides a general meta-model for security-related concepts, and suggests domain-specific specialisation where applicable to facilitate tool support and automated reaction mechanisms.

Figure 3 shows how the evolution rules are used in a feedback loop to deal with the evolution of security requirements. Changes of situation are initially caused by external factors (environment context) of the system. These changes can trigger evolution rules that perform automatic adaptation, or otherwise result in a manual change process. As a result of such an adaptation action, a new situation arises that one must iteratively reevaluate for automatic rule execution or manual intervention. Or else, if the new change does not trigger any further actions, or there is no further change, the control feedback loop can exit.

4 Argumentation of security requirements

As we discussed in the introduction, the satisfaction of security requirements in the general form of the entailment $W, S \vdash R$ needs to be argued, as security requirements 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 (see Figure 4). 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 are 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.

Eliminato: Error! Reference source not found.)

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.

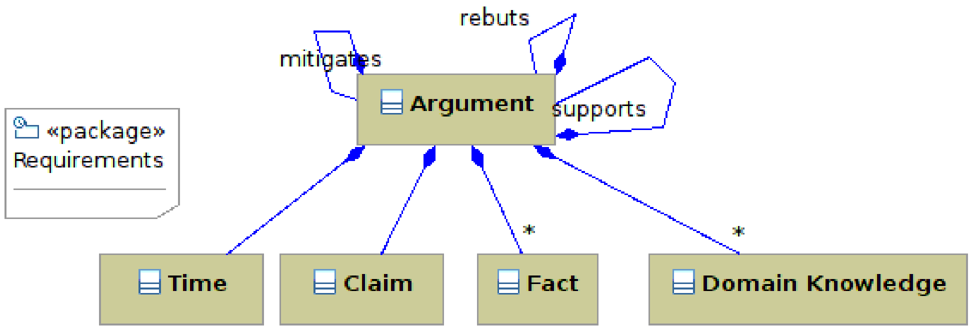


Figure 4. Meta-model for arguing the satisfaction of security requirements

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.

Here is one example of iterative development of an argument for the security in the AMAN case study. Typically such development is in the form of a dialogue. The first round of an informal argument might be:

Initial claim:

- The AMAN system is secure (C1).

Initial facts:

- The AMAN system is controlled by trustable experts (F1).
- The experts can manage the separation of distance in the runways by monitoring and active communication of the flight trajectories (F2).

Initial domain knowledge:

- When two flights close within a dangerous distance, the AMAN system will present a warning to highlight the flights on the trajectories. (DK1)

Initial Rebuttals:

- The experts can have malicious intent due to social and psychological reasons (R1 on F1).
- The ADS-B system [2] may report a wrong distance due to mechanical failures or extreme weather conditions (R2 on DK1).

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

- All experts have been through clearance to minimise the risk of being malicious F3=R1.1).
- The controlled trajectories are viewed by a group of experts who can all see what’s displayed on the screens to prevent a single person from providing wrong instructions (F4=R1.2).

Such argumentation can go on and on until all the facts and domain knowledge 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 requirements) and all the rebuttals are false. A formal treatment of argumentation using non-monotonic proposition logic can be found in [11]. As one can see, the result of such argumentations would inevitably contribute to changes in the situations of security requirements.

5 An example of requirements evolution modeling

In this section we show how we can represent the evolution of requirements that characterize the ATM case study by instantiating the meta-model presented in Sections 3 and 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 requirements.

In this example, we show how functional and security requirements of the actual ATM systems change due to the introduction of the AMAN queue management tool that supports ATCOs.

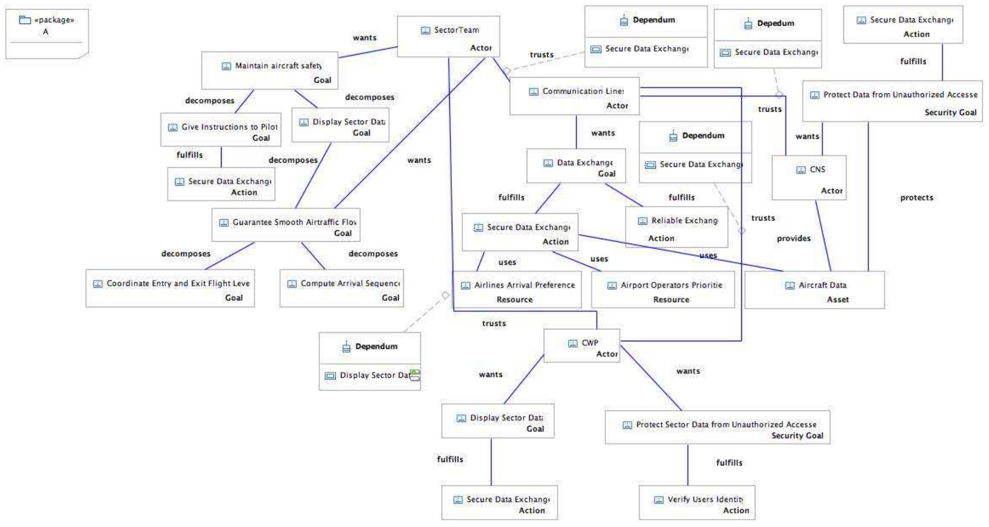

«package» A
SectorTeam Actor
wants
Maintain aircraft safety Goal
decomposes
Give Instructions to Pilot Goal
Display Sector Data Goal
fulfills
Secure Data Exchange Action
Guarantee Smooth Airtraffic Flow Goal
Coordinate Entry and Exit Flight Level Goal
Compute Arrival Sequence Goal
Dependum
Display Sector Data
trusts
Secure Data Exchange
Depedum
Secure Data Exchange Action
Protect Data from Unauthorized Accesse Security Goal
Communication Lines Actor
Data Exchange Goal
Reliable Exchange Action
CNS Actor
protects
provides
uses
Airlines Arrival Preference Resource
Airport Operators Priorities Resource
Aircraft Data Asset
CWP Actor
Protect Sector Data from Unauthorized Accesse Security Goal
Verify Users Identity Action

Figure 5. The "before" situation

Figure 5 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. The communication between the different ATM actors takes place over dedicated and secure communications lines. For example, for communication between the Sector Team and the pilots specific radio frequencies are used.

In this scenario, the security requirements are associated with the CWP and the Communication Lines :

- The CWP shall provide an authentication mechanism to verify users identity
- The Communication Lines shall provide secure and reliable communication among ATM actors

As affect 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 does not take place over secure and dedicates lines: the actors are interconnected by the SWIM, an IP based data transport network that will replace the current point to point data systems.

In this scenario we have new security requirements that need to be satisfied (see Figure 6):

- The CNS systems shall check the authenticity of aircraft tracks
- The AMAN shall provide selective access control for the different ATM actors (Sequence Manager, ATCOs,..)
- The AMAN shall disclose to another actor only the aircraft information necessary for the actor to perform its task (need to know principle)
- The AMAN shall check that the information coming from Meteo Services, Radars, Airlines and Airport Operators has not been altered
- The SWIM shall require authentication sessions for users based on digitally signed certificates
- The SWIM shall be able to detect fake stakeholders and trace them in a blacklist
- The SWIM shall ensure data integrity and confidentiality.

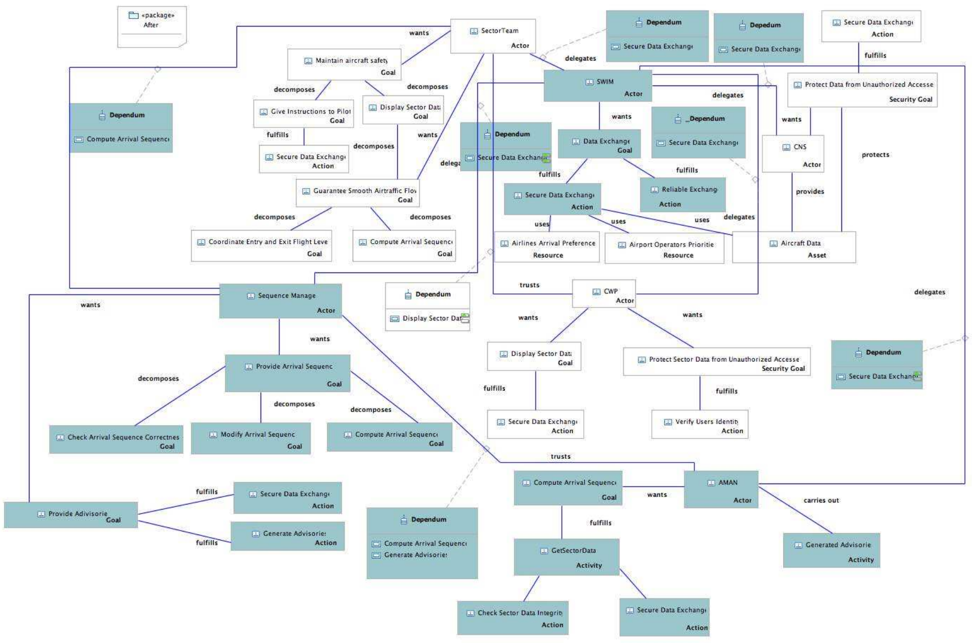


Figure 6. The “after” situation

6 Specifying rules for evolutionary changes

Having the capability to express the Before/After situations in security requirements as models, in this section we discuss the role of automated evolution rules in a workflow supporting model transformation.

6.1 Goals for the evolution rules

The framework 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;
- provide automated alerting of criteria that cease to be satisfied;
- allow flexible adaptation to security requirements of domains, e.g. ATM;
- once adapted to a domain, allow flexible refinement for a concrete application in context of actual system design artifacts;
- 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.

6.2 Using model transformations for evolution

Automated model transformations play an important role in modern model-driven system engineering in order to query, derive and manipulate large, industrial models. .

For instance, meta-modeling-based development architectures, including OMG's Model Driven Architecture (MDA), highly rely on transformations within and between different models and languages. The important role of model transformation (MT) languages and tools for the overall success of model-driven system development has been revealed in many surveys and papers during the recent years [5][6][10]. Approaches to model transformation and various solutions addressing the encountered challenges are continuously being explored.

6.2.1 Incremental transformations

Transformation of evolving models is a challenging task with high practical relevance, arising in a wide range of circumstances. 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 poses 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, trace signatures 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 [16], 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*. These aspects are called *target* and *source incrementality.*“ , respectively. It would also be beneficial if the transformation system could autonomously react to the evolution of the source model; this requires a notion of events and reactions.

The benefits of source and target incrementality can be harnessed in various transformation scenarios, most importantly tool integration. Further applications are found in the context of domain-specific modeling such as (i) model execution (simulation), where incremental transformation rules may be used to execute the dynamics semantics of a domain-specific language; (ii) constraint management, where incremental transformations are used to check and enforce the validity of a complex constraint; (iii) event-driven code generation, where the textual representation of abstract models may be incrementally maintained as the source model changes.

6.2.1.1 Source incrementality

The aim to execute transformations without re-evaluating unchanged parts of the evolving source model is called source incrementality.

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. There are two important drawbacks; one of them is the increased memory consumption due to the stored occurrence sets. Additionally, these stored result sets have to be continuously maintained whenever the model is changed, causing an overhead on model manipulation. Nevertheless, 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. This is the benefit of source incrementality: eliminating the need to continuously re-evaluate the source model.

6.2.1.2 Live transformations

To achieve target incrementality, an incremental transformation approach creates "change sets" which are merged with the existing target model instance. In order to efficiently calculate which source element may trigger changes (source incrementality), the transformation context has to be maintained which describes the execution state of the model transformation system (e.g. variable values, partial matches). Depending on whether this is possible or not, there are two main approaches to incremental transformations: re-transformation and live transformation.

Systems employing re-transformations lack the capability to maintain the transformation context over multiple execution runs, thus the entire transformation has to be re-run on the modified source models. This involves the computation of which model elements are involved in the change, and which elements should be left untouched by the transformation. Thus, the feasibility of this approach depends heavily on the trace information.

In contrast, live transformations maintain the transformation context continuously so that the changes to source models can be instantly mapped to changes in target models. Live transformations are persistent and go through phases of execution whenever a model change occurs. Similarly to re-transformations, the information contained in trace signatures is used in calculating the source elements that require re-transformation. However, as the execution state is available in the transformation context, this recomputation can be far more efficient.

Ráth et al [16] introduced an approach where a model change is captured by a change in the match set of a graph pattern. The match set is defined by the subset of model elements satisfying structural and type constraints described by the pattern. Changes in the match set can be tracked using incremental pattern matching. A model change is detected if the match set is expanded by a new match or a previously existing match is lost. Since a graph pattern may contain multiple elements, a change affecting any one of them may result in a change in the match set. Thus complex changes beyond simple atomic operations can be easily detected. The execution context of the live transformation, as required by target incrementality, is represented in the form of pattern variables, and continuously maintained by the incremental pattern matching engine after each atomic model manipulation operation. As a result, the computation required to initialize and execute the incremental transformation sequence after a change is very efficient, since pattern matching, the most cost-intensive phase of the transformation, is executed instantly.

With the help of incremental transformation rules, also called *trigger*s, a broad range of transformations can be specified in a live way. A trigger is defined in the form of a graph transformation rule: the precondition of its activation is defined in the form of a graph pattern, while the reaction is formulated by arbitrary (declarative or imperative) transformation steps.

6.2.1.3 Target incrementality

Traceability is a property of transformations, stating that some kind of mapping is available between the source and target models. Traceable transformations has various advantages: for example, target elements can be traced back to the reason of their existence (e.g. source elements) to justify or explain them; additionally, the transformation of some individual elements can depend on already established source-target mappings, which is useful for e.g. transforming containment hierarchies. Traceability is also required to achieve target incrementality: if local changes of the source model only affect corresponding parts of the target model, the transformation execution can focus on the affected part and leave other parts of the target intact, with the help of a correspondence relationship.

A straightforward way to preserve trace information is internal traceability, when target elements have a direct reference to source elements. This is prevalent in simpler frameworks that are only capable of one-to-one correspondence between source and target elements, where each target element is traced back to exactly one source element. In more complex transformation tasks, however, each rule application may have to be traced back to potentially several source and target elements, and may justify potentially several target elements simultaneously. Therefore more recent approaches have opted for external traceability, when the elements of the source and target models are interrelated via a separate mapping model. This mapping model (also known as *trace model*, reference model or correspondence model) conforms to a separate trace meta-model, that references the source and target meta-models.

In incremental transformations, separate trace models have an additional benefit: they preserve the now-obsolete mapping even after the source or target has been changed, which can serve as an important input for transformation rules. For example, a trigger can be defined to detect the deletion of a source element that has been mapped to a target element, by discovering that there is a trace element connected to a target element, but not a source element; the appropriate action would be to delete the target element and the trace relationship as well. As an illustration, Figure 7 depicts a reference element connecting a source and a target element, as well as two rules: the first one creates the reference and target elements if they do not exist; the second one deletes them if the source does not exist. See [17] for a detailed case study of traceability in incremental transformation in a DSM context.

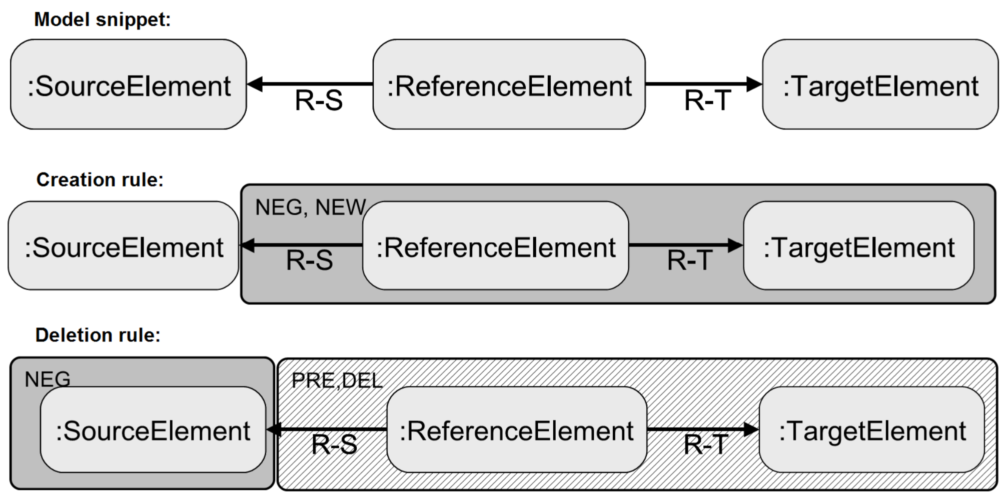


Figure 7. Example rules for target incrementality through traceability

6.2.2 Event – Condition – Action semantics

Live transformation and change-driven rules can be described using the mathematical formalism of Event-Condition-Action. The literature mentions various definitions for this concept; a relatively rich one is found in [1]. 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. A Condition commonly involves the assertion of relationships between the elements affected by the change and unaffected contextual elements, to identify the position where the change occurred in the model. The Condition may also assert entities and relationships in the unaffected part of the model so as to provide a filter for certain overall conditions of the model. Least commonly, the Condition can also include assertion of relationships between affected elements to restrict the shape of the change.

Condition, contrary to its name, does not have to be a simple decision whether the actions can be applied in reaction to the change. In a more powerful formalism, the Condition may select various sets of parameters to execute the Action with. This can be achieved e.g. if the Condition is a logic formula with free variables, the Event provides the value of some of these variables, and the Action is executed once for every possible substitution of the rest of the free variables, using the substitution values as parameters.

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.

6.2.3 Change-driven transformations

A separate approach to specify model transformations has been introduced in [18]. Depending on the setup, change-driven transformations can fulfill a number of goals. The approach can be viewed as a generalization of live transformation, a formalism to bridge the conceptual gap between batch and incremental transformation, or simply a new and intuitive way to describe reactions to changes. It is also suitable for transformation scenarios where the target model is not directly accessed or not completely materialized, but accessible only through element identifiers and a query/manipulation interface. This allows the transformation to manipulate remotely stored target models, large models that do not fit in memory, or an internal runtime representation of an application.

A key concept of the approach is capturing and explicitly representing change operations, for example as model elements. The elements that correspond to future changes are *change command*s, while the ones that record already executed changes constitute a *change history* model. The latter kind can be automatically generated on-the-fly during the execution of model manipulation. Apart from basic change operations (creation, deletion, moving, value setting, etc.), user-defined domain-specific macro change types are also allowed.

Change driven transformations are specified by a set of transformation rules that react to changes of the model by matching a single change operation and additional model elements, and create change commands to manipulate the target model. The created change command may be executed at a later time, even at a remote location. Thus rules are incremental and evaluated asynchronously to the update of the target model, and optionally asynchronously to the change of the source model that caused the change propagation.

[18] also presents an example workflow. Change history is derived on-the-fly and automatically after the source model is updated, regardless whether the model manipulation was initiated by another (not necessarily change-driven) transformation, or by user interaction. Change history is asynchronously processed by transformation rules that should depend on the change history element, and potentially an extended condition involving the source model, but not the target. Instead of directly manipulating the target model, the transformation rules only create change commands to express the required modifications, thus allowing for deferred execution, remote processing, or piping through the runtime manipulation API of an application.

The workflow is depicted on Figure 8. M_A and $M_{A'}$ are the previous and the current state of the source model A, and M_B, $M_{B'}$ are the two states of the target model B before and after the application of the change commands. CHM_A is the change history model derived by observing the change of A, and CC_B represents the change commands that affect the target model. Transformation and processing is indicated by circles.

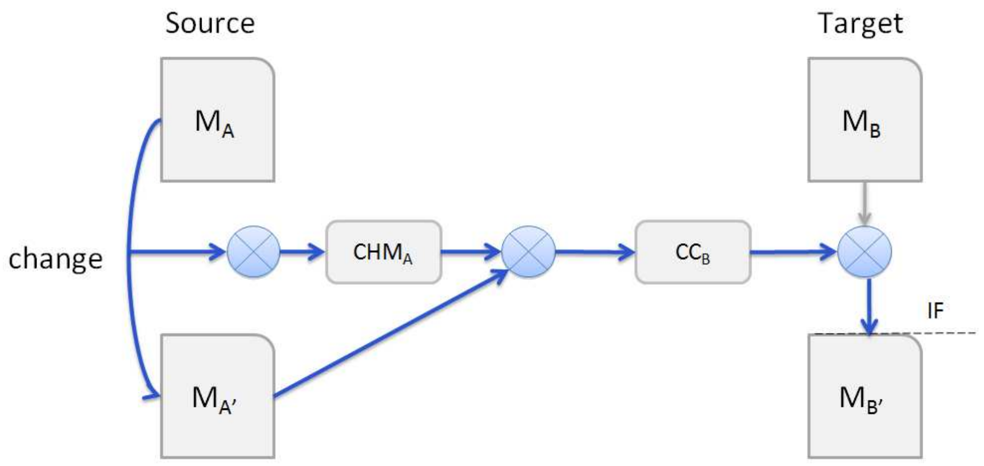


Figure 8. Change-driven workflow

6.3 Meta-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 9). Evolution rules are defined in conformance with the Event – Condition – Action semantics to specify the desired reaction to changes performed on the model. 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 parametrized 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* which 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.

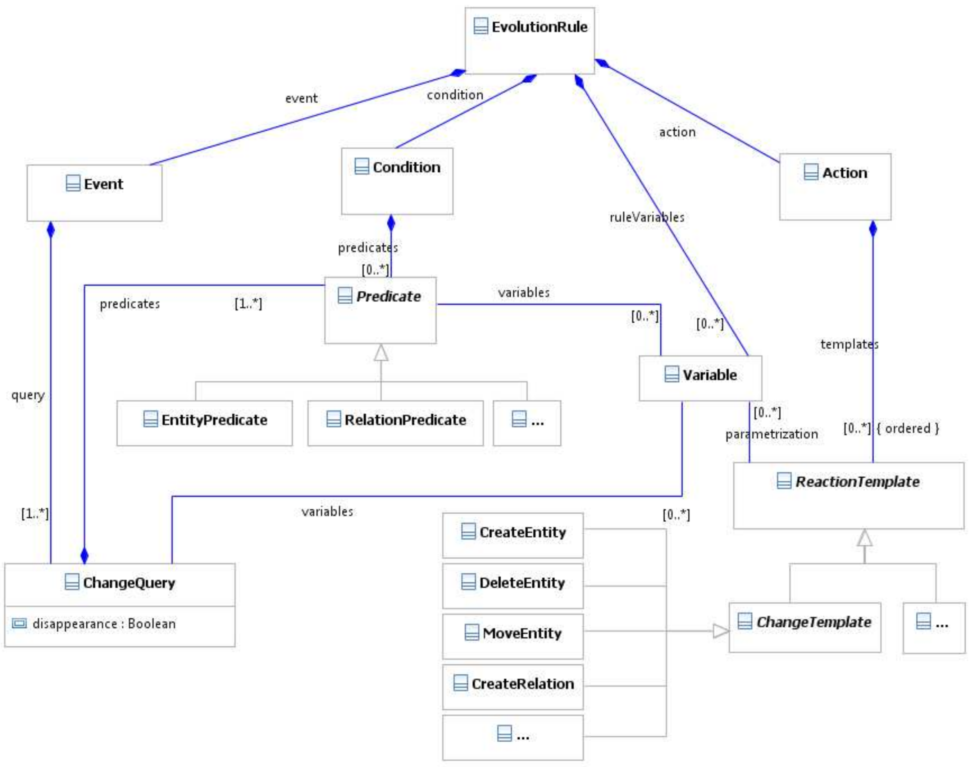


Figure 9. Meta-model for evolution rules

6.4 Examples for evolution rules

6.4.1 Problem description

In an evolving requirements model, new actors may be introduced, delegation and trust relationships may be changed, all raising security concerns. An example rule is designed to intervene in situations when an actor delegates some responsibility to another actor, but does not trust the other one with the same object. See Figure 10 for an illustration of this undesired pattern. 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. To illustrate the capabilities of the evolution rule formalism, we present three solutions to this problem.

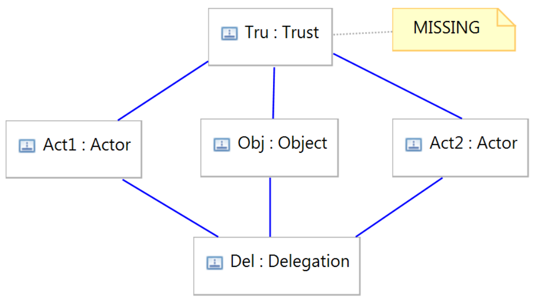


Figure 10. The undesired pattern: untrusted delegation

6.4.2 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, 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 *pseudocode* 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 Argumentation(Arg);
        create relation Argumentation.problem(Arg–AP->Del);
    }
}
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 Argumentation(Arg);
        create relation Argumentation.problem(Arg–AP->Del);
    }
}
```

6.4.3 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 (see Figure 10) 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 Argumentation(Arg);
        create relation Argumentation.problem(Arg–AP->Del);
    }
}
```

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.

6.4.4 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` introduced in Section 6.4.2 they provide three options that are 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, 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.

6.4.5 Example rule application

The given solutions can be demonstrated by applying them on the example models from Section 4 which represent the before/after situations in the ATM domain. Observing the After situation (see Figure 6) 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 CNS. This may be a security issue, as Secure Data Exchange is now delegated to SWIM, which obviously requires trust. Fortunately, the example evolution rule presented in Section 6.4 can be used to automatically detect untrusted delegations. For example, if we use the three rules introduced as Solution 2 (see Section 6.4.3) and Solution 3 (see Section 6.4.4), they will be triggered for several individual matches by this example evolution. The rule matches will map the rule

Eliminato: Error! Reference source not found.

variables to actual substitutions that experienced the Event and satisfy the Condition. Act2 will be mapped to SWIM, Obj to Secure Data Exchange, and Act1 will be mapped to CNS, AMAN, CWP, Sector Team or Sequence Manager in the various concrete matches. 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.

6.4.6 Further discussions

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) may have to be undone or compensated.

The example presented in this section shows how the goals in Section 6.1 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 argumentation or trust, removing the delegation)

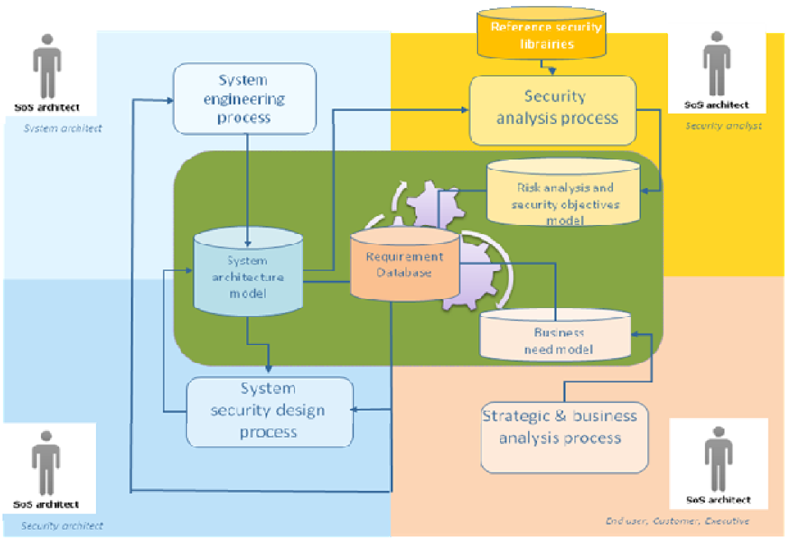
7 Security Requirement Analysis in Practice

In this section we present the Thales industrial method for security risk analysis, and we show the analogies with our methodology for security requirements 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.

7.1 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 [15] over the engineering problem:



Reference security librairies
SoS architect
System architect
System engineering process
Security analysis process
SoS architect
Security analyst
Risk analysis and security objectives model
System architecture model
Requirement Database
Business need model
System security design process
Strategic & business analysis process
SoS architect
Security architect
SoS architect
End user, Customer, Executive

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 ...). Security requirements are derived from security objectives model of dedicated DSML (see [13]). This mapping enables to add security requirements 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 [19].

7.2 DOORS T-REK

Rational DOORS [19] (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 12, 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.

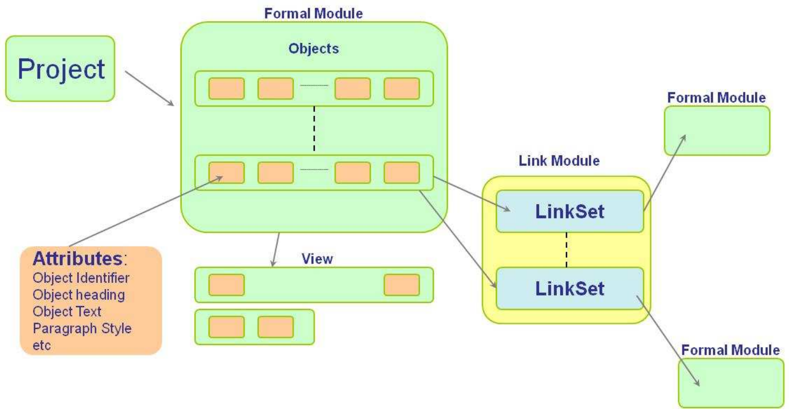


Figure 12. DOORS project structure

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 13, 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.

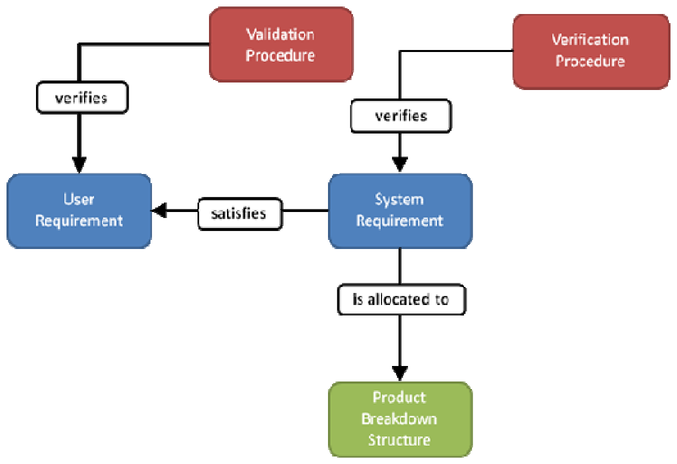


Figure 13. Simplified Datamodel in T-REK

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.

7.3 Application in Thales Requirement Workbench

7.3.1 Thales Security DSML

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 [15]. The core part of the conceptual model[1] is represented in Figure 14.

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

Essential 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. Essential 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.

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

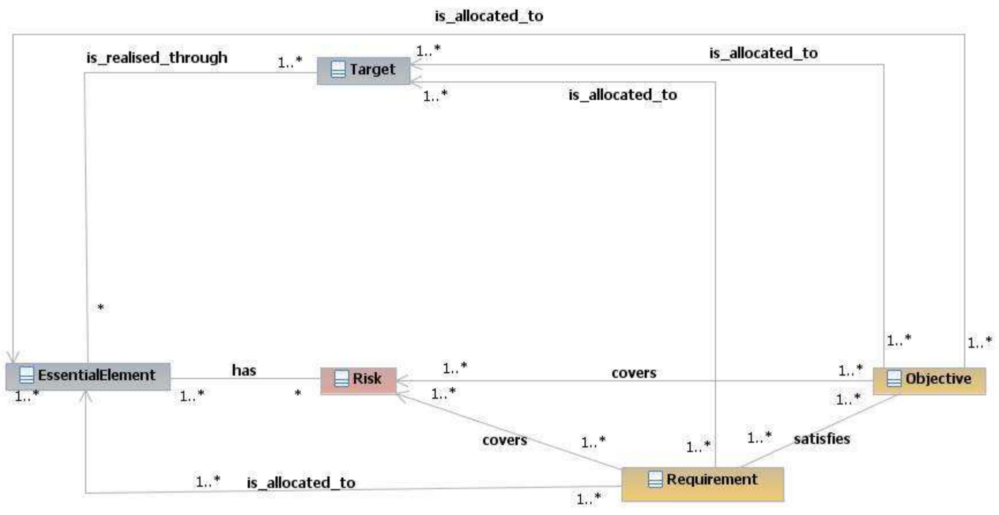


Figure 14. Conceptual model of Security Objectives and Requirements in Security DSML

In current Security DSML, we distinguish three kinds of static models[2] 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).

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

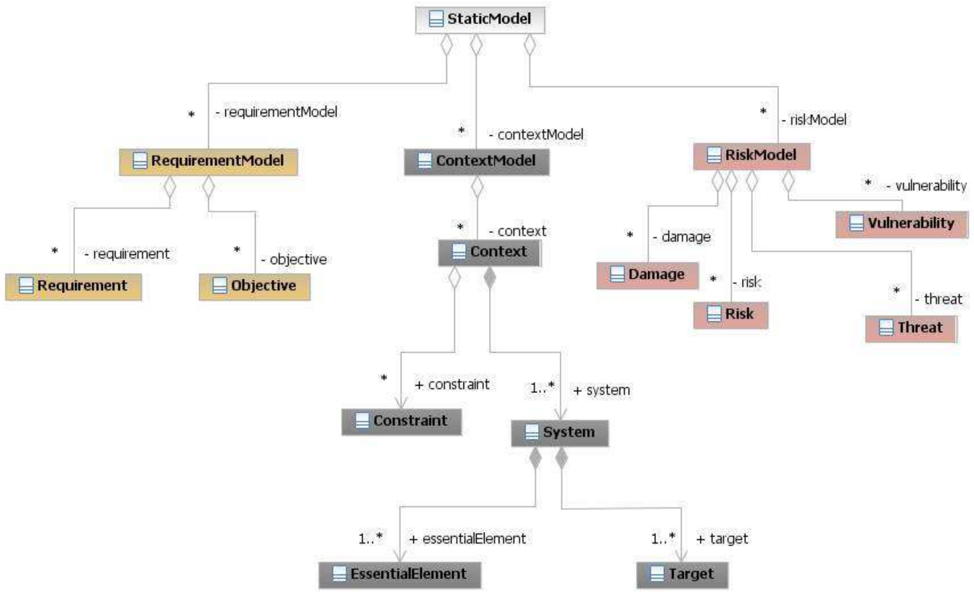


Figure 15. Security DSML Static Model description

7.3.2 Mapping between DSML conceptual models and the security requirements meta-model

Although slightly different terminology was used in Security DSML, we can find a clear mapping between our more generic security requirements meta-models in the previous sections. Using CamelCase for the DSML concepts, and lower case words for concepts in our security requirements meta-model, the mapping can be set up as follows:

- A Requirement is a requirement,
- An Objective is a security goal,
- An Essential Element is a specification in logical layer as Platform Independent Model (messages, logical component)
- A Target is a specification in physical layer as Platform Specific Model (communication channel, physical component). Target has vulnerability and is threatened that may expose a Risk to the satisfaction of a security goal.
- The StaticModel is equivalent to a situation, which contains the RequirementsModel corresponding to our security requirements in terms of security goals; the ContextModel) corresponds to our domain properties which further contain Constraints (domain assumptions) and System (logical and physical specifications).

- The RiskModel corresponds to our validation model in the quantitative perspective.
- Concepts such as Damage, Vulnerability, and Threat are the same, whereas the Boolean in predicate logic may no longer be sufficient for the quantitative risk analysis. Yet one can still reason about the security goal satisfaction using thresholds.

7.3.3 From DSML to DOORS T-REK

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 Requirement of Security DSML and DOORS Requirements.

This relation enables to connect other kind of requirement (Safety, Maintainability, Cost …) with Security Requirements 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[3]) for Traceability needs between DOORS and Security DSML.

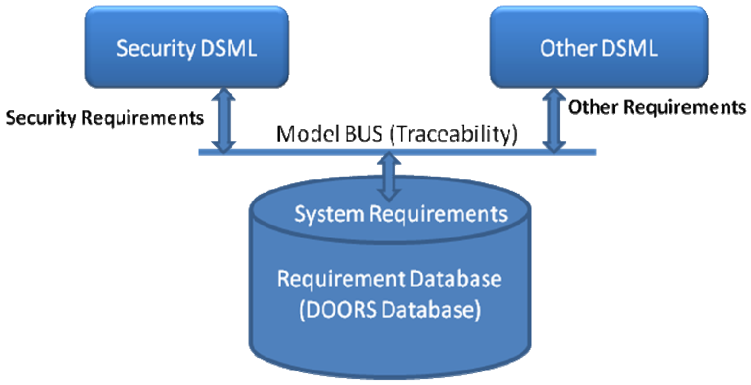


Figure 16. Mapping between DSML and DOORS

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

Figure 17 presents the extended conceptual meta-model including DOORS connections. Two kinds of entities are mapped with DOORS: Requirements and Target which are respectively represented by Requirement and Product Breakdown Structure object in DOORS. To ensure traceability between DSML and DOORS, we add an PUID (Product Unique IDentifier) attribute, PUID is the reference name of a DOORS object.

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

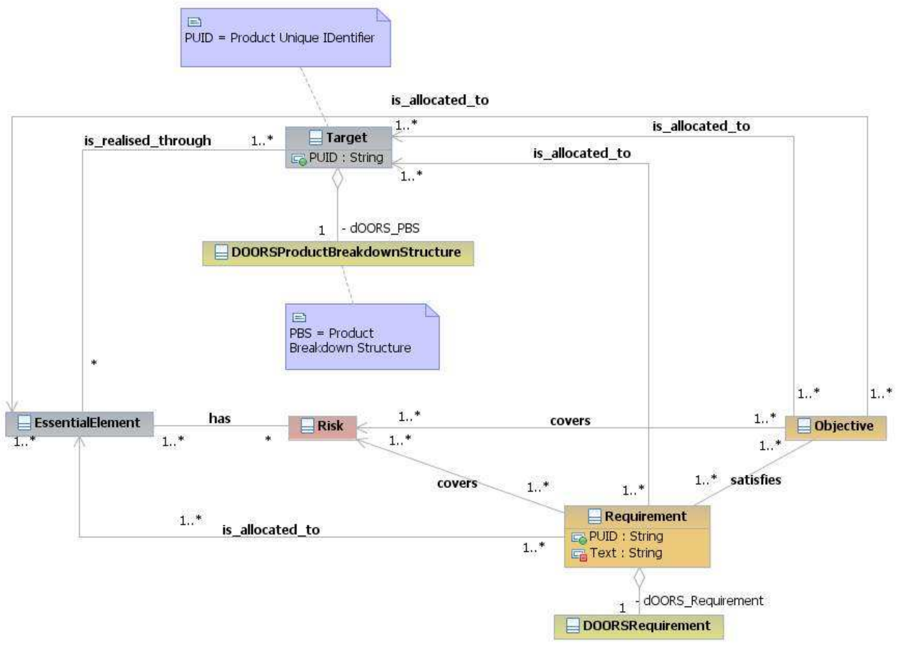


Figure 17. Extended Conceptual model including DOORS connections

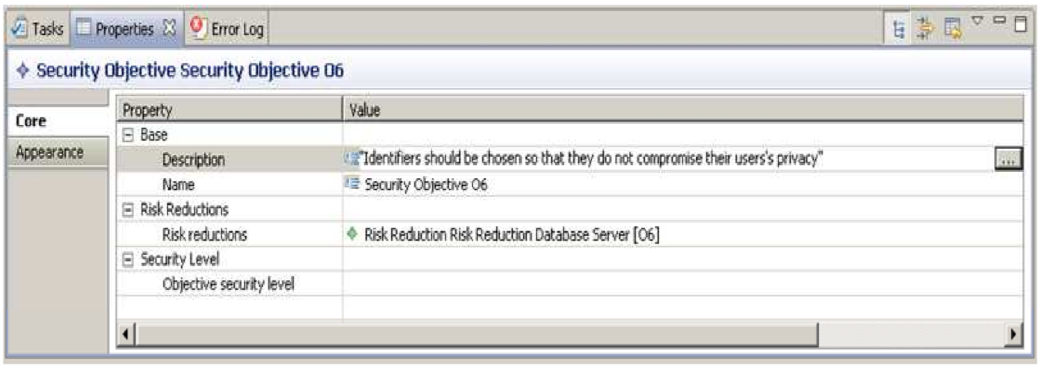
Figure 18 depicts the properties view on Security Objective O6 (Identifiers should be chosen so that they do not compromise user's privacy). Figure 19 presents the requirement derived from security objective in DOORS.

Formattato: Tipo di carattere: (Predefinito) Arial

Eliminato: Figure 18

Eliminato: Figure 19

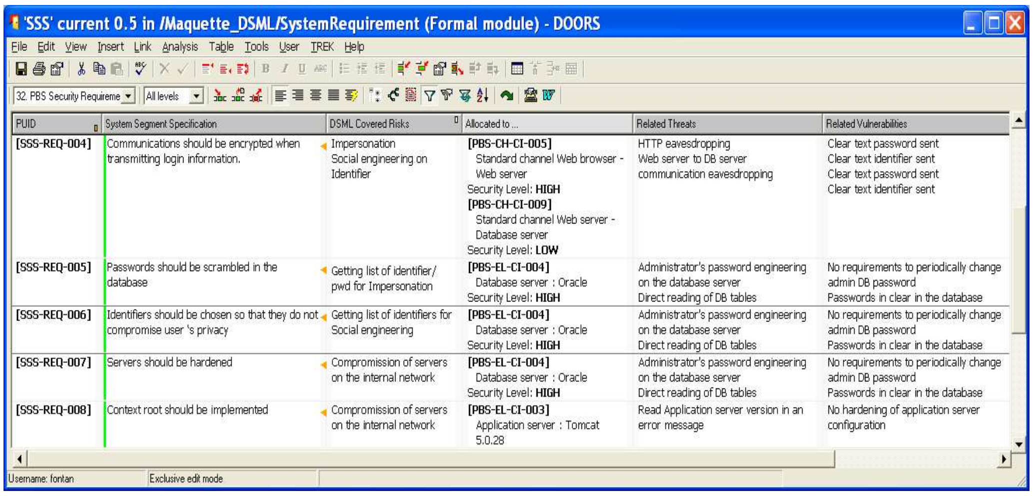
Formattato: Tipo di carattere: (Predefinito) Arial



Security Objective Security Objective O6

Property	Value
Base	
Description	"Identifiers should be chosen so that they do not compromise their users's privacy"
Name	Security Objective O6
Risk Reductions	
Risk reductions	Risk Reduction Risk Reduction Database Server [O6]
Security Level	
Objective security level	

Figure 18. Close view on the Security Objectives


'SSS' current 0.5 in /Maquette_DSML/SystemRequirement (Formal module) - DOORS

32. PBS Security Requireme | All levels

PUID	System Segment Specification	DSML Covered Risks	Allocated to ...	Related Threats	Related Vulnerabilities
[SSS-REQ-004]	Communications should be encrypted when transmitting login information.	Impersonation Social engineering on Identifier	**[PBS-CH-CI-005]** Standard channel Web browser - Web server Security Level: **HIGH** **[PBS-CH-CI-009]** Standard channel Web server - Database server Security Level: **LOW**	HTTP eavesdropping Web server to DB server communication eavesdropping	Clear text password sent Clear text identifier sent Clear text password sent Clear text identifier sent
[SSS-REQ-005]	Passwords should be scrambled in the database	Getting list of identifier/ pwd for Impersonation	**[PBS-EL-CI-004]** Database server : Oracle Security Level: **HIGH**	Administrator's password engineering on the database server Direct reading of DB tables	No requirements to periodically change admin DB password Passwords in clear in the database
[SSS-REQ-006]	Identifiers should be chosen so that they do not compromise user 's privacy	Getting list of identifiers for Social engineering	**[PBS-EL-CI-004]** Database server : Oracle Security Level: **HIGH**	Administrator's password engineering on the database server Direct reading of DB tables	No requirements to periodically change admin DB password Passwords in clear in the database
[SSS-REQ-007]	Servers should be hardened	Compromission of servers on the internal network	**[PBS-EL-CI-004]** Database server : Oracle Security Level: **HIGH**	Administrator's password engineering on the database server Direct reading of DB tables	No requirements to periodically change admin DB password Passwords in clear in the database
[SSS-REQ-008]	Context root should be implemented	Compromission of servers on the internal network	**[PBS-EL-CI-003]** Application server : Tomcat 5.0.28	Read Application server version in an error message	No hardening of application server configuration

Username: fontan | Exclusive edit mode

Figure 19. Derived Requirements expressed in DOORS

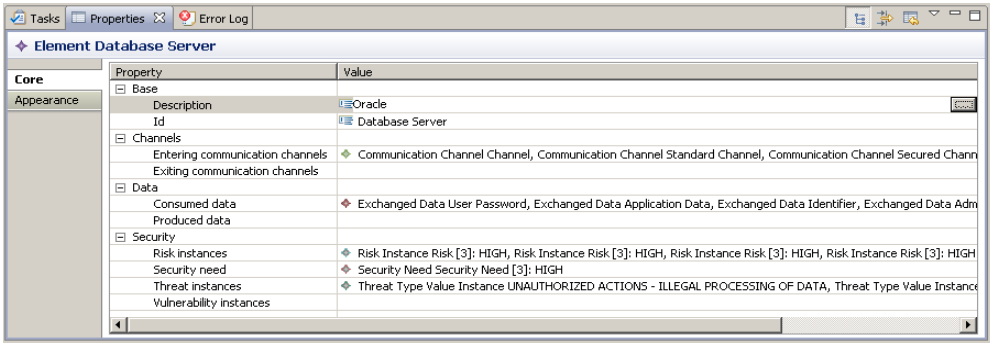
The information of target can be consulted in the Properties View (Description, constraints applied on it), as can be seen in Figure 20. This properties view of Target is also defined in DOORS as shown by Figure 21.

Formattato: Tipo di carattere: Arial

Formattato: Tipo di carattere: Arial

Eliminato: Figure 20

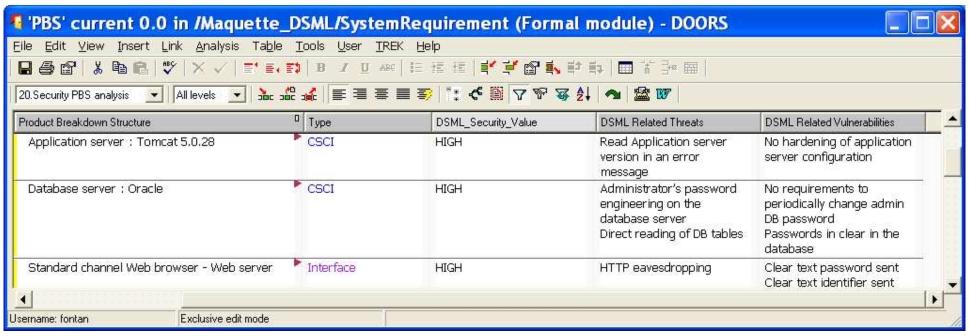
Eliminato: Figure 21



Element Database Server

Property	Value
⊟ Base	
Description	Oracle
Id	Database Server
⊟ Channels	
Entering communication channels	Communication Channel Channel, Communication Channel Standard Channel, Communication Channel Secured Chann
Exiting communication channels	
⊟ Data	
Consumed data	Exchanged Data User Password, Exchanged Data Application Data, Exchanged Data Identifier, Exchanged Data Adm
Produced data	
⊟ Security	
Risk instances	Risk Instance Risk [3]: HIGH, Risk Instance Risk [3]: HIGH, Risk Instance Risk [3]: HIGH, Risk Instance Risk [3]: HIGH
Security need	Security Need Security Need [3]: HIGH
Threat instances	Threat Type Value Instance UNAUTHORIZED ACTIONS - ILLEGAL PROCESSING OF DATA, Threat Type Value Instance
Vulnerability instances	

Figure 20. Properties of the Database Server in DSML


'PBS' current 0.0 in /Maquette_DSML/SystemRequirement (Formal module) - DOORS

20.Security PBS analysis | All levels

Product Breakdown Structure	Type	DSML_Security_Value	DSML Related Threats	DSML Related Vulnerabilities
Application server : Tomcat 5.0.28	CSCI	HIGH	Read Application server version in an error message	No hardening of application server configuration
Database server : Oracle	CSCI	HIGH	Administrator's password engineering on the database server Direct reading of DB tables	No requirements to periodically change admin DB password Passwords in clear in the database
Standard channel Web browser - Web server	Interface	HIGH	HTTP eavesdropping	Clear text password sent Clear text identifier sent

Username: fontan | Exclusive edit mode

Figure 21. Database Server description in DOORS

7.4 Evolution Management

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 requirements. 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 requirement.

7.4.1 ChangeLine Meta model

To represent traceability between changes and versioning of change, we add a further Model: ***Change Model*** which is composed by several ***Change Lines***. As shown by Figure 22, 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. These transitions enable to represent Before-after Perspective of change.

Eliminato: Error! Reference source not found.

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.

The maintenance perspective could be represented in this model by a particular combination of continuous and before after perspective.

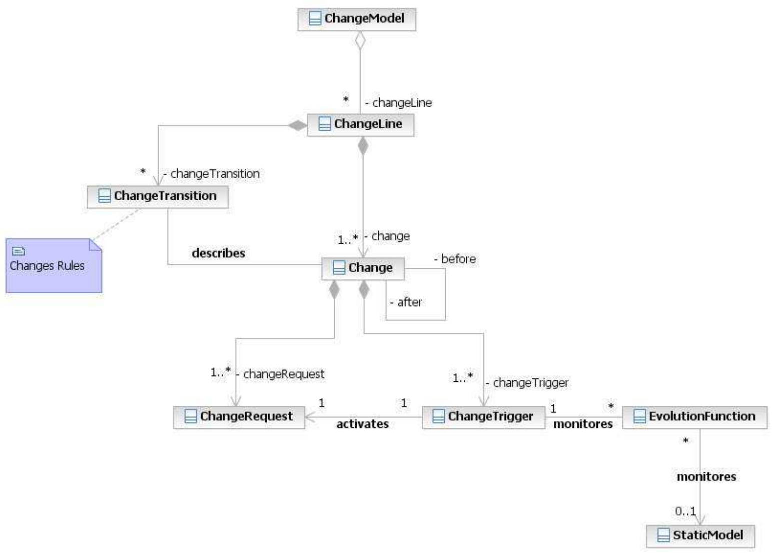


Figure 22. DSML Change Model conceptual model

7.4.2 ChangeRequest Meta-model

As shown by Figure 23, a ***Change Request*** contains a PUID to identify it and a status which represent 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.

Eliminato: Figure 23

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 23).

Eliminato: Figure 23

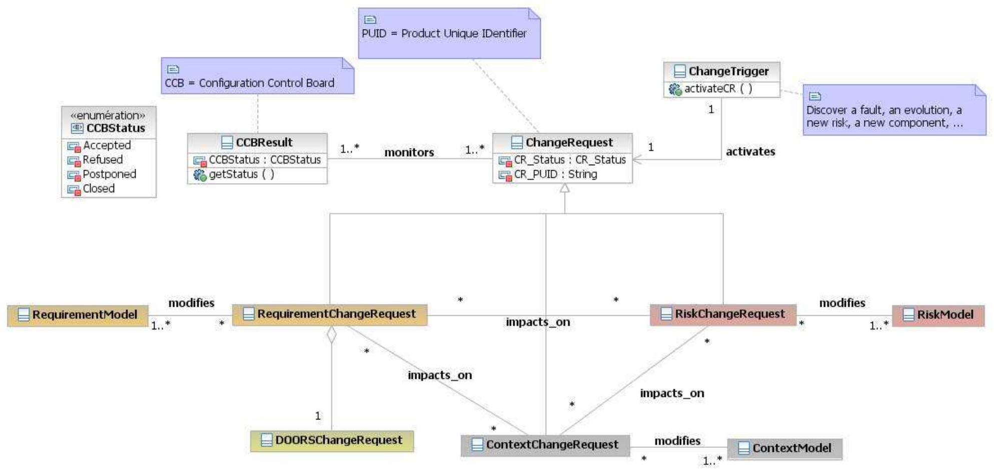


Figure 23. DSML Change Request Meta-model

7.4.2.1 Behavior of Change Request

For readability sake, Change Request Behavior is described by UML Statechart Diagram. 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.

7.4.2.2 General Behavior of Change Request

As suggested by Figure 23 and Figure 24, a ***Change Request*** (CR) starts after Change Trigger activation (e.g. discover a fault, a new requirement …). Redactor of Change Request must define the change and trace it with the impacted elements. Change Request is as default in ***Pending*** State.

Eliminato: Figure 23

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.

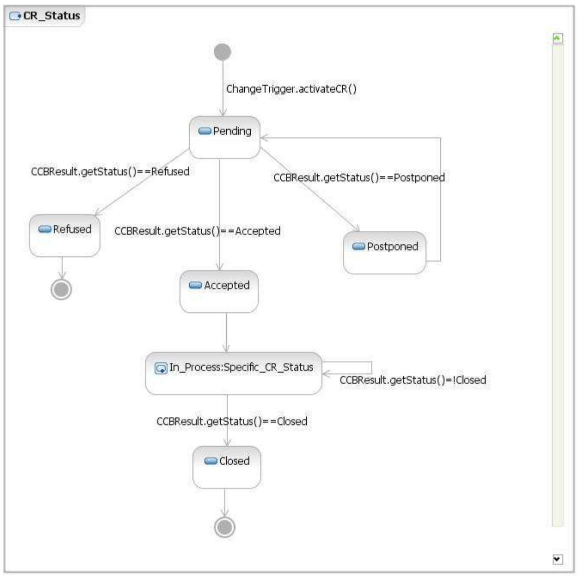

CR_Status
ChangeTrigger.activateCR()
Pending
CCBResult.getStatus()==Refused
CCBResult.getStatus()==Postponed
Refused
CCBResult.getStatus()==Accepted
Postponed
Accepted
In_Process:Specific_CR_Status
CCBResult.getStatus()=!Closed
CCBResult.getStatus()==Closed
Closed

Figure 24. Generic Change Request Status Behavior

7.4.2.3 Specific Behavior of Requirement Change Request

Specific ***Requirement Change Request*** (RCR) Behavior starts after ***Accepted*** state in generic behavior. As shown by Figure 25, 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.

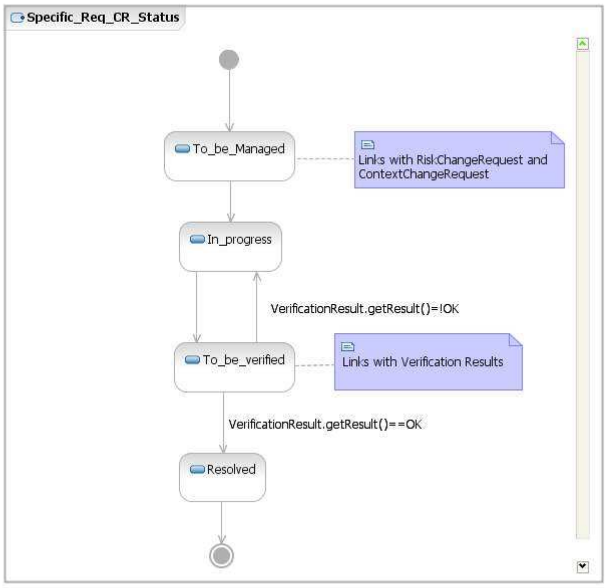

Specific_Req_CR_Status
To_be_Managed
Links with RiskChangeRequest and ContextChangeRequest
In_progress
VerificationResult.getResult()=!OK
To_be_verified
Links with Verification Results
VerificationResult.getResult()==OK
Resolved

Figure 25. Requirement Change Request Status Behavior

8 Conclusions

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

These meta-models are by no means an ultimate answer to the conceptual modeling framework for evolution of security requirements. A better bet is to consider them as an extensible framework in which new concepts and practices in the field of evolving security requirements 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 meta-models presented will be considered together to shed some lights on what is the more general representation of the meta-meta-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 requirements change patterns may be discovered, be documented and be reused from one case study to another.

9 Acknowledgement

We thank Prof. Michael Jackson for his insightful comments on the meta-models being developed earlier. We thank Prof. John Mylopoulos and colleagues at the UNITN for the discussions on the requirements language in Section 4 and 5. We would also like to thank Miss Stefanie Francois at the OU for discussions on the use of meta-model for security argumentation.

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] 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.

[11] 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.

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

[13] 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)

[14] 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.

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

[16] 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.

[17] István Ráth, András Ökrös, Dániel Varró: ”Synchronization of Abstract and Concrete Syntax in Domain-Specific Modeling Languages”. *Journal of Software and Systems Modeling*, Special Issue on Traceability, pp 1-19, *2009*.

[18] István Ráth, Gergely Varró, Dániel Varró: “Change-Driven Model Transformation”. *In Proc. of Int. Conf. on Model Driven Engineering Languages and Systems (MODELS), Denver, USA, 2009.*

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

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

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

[22] 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., 19

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., 12

A *context* consists of several domains which interface with each other., 15

A *dependum* is associated with a dependency relationship between two actors, which specifies which object (resource/process) or which requirement (goals, softgoals) are depended. Specifically, it can be defined similarly for the relationships Trusts/Delegates., 15

A *derived requirements* refines the initial requirements., 15

A *dynamic object* can be a *process* that consists of *activities*, 15

A *goal* is an objective that the system-to-be should achieve., 15

A *proposition* is the sharable objects of attitudes and the primary bearers of truth and falsity, which can be either an optative or an indicative property., 15

A *requirement* is a desired or an optative property wanted by a stakeholder., 15

A *resource* is a physical or an informational entity which has no intention by itself., 15

A *security goal* is a soft goal that an asset needs to be protected from harms., 15

A situation is a partial state of the world where some propositions are true and some other propositions are nor true nor false. Thus, a situation consists of objects and propositions concern these objects, 15

A *situation* of our requirements model is expressed in terms of propositions *and* objects., 15

A *soft goal* is an objective that does not have a clearcut evaluation of the truth value., 15

A *specification* is a process that fulfils certain requirements under given indicative domain properties., 15

A *triggering event* is a dynamic difference between two consecutive versions of a model that results in the activation of an evolution rule., 12

A *vulnerability* is a weakness, a flaw or a deficiency that is exploited to carry out a threat to cause harm to an asset., 15

An *activity* is a sequence of *actions* that can be performed by an actor to fulfill a goal, 15

An *actor* is an intentional entity such as a human, a device, a legacy software or software-to-be component that performs actions to achieve its own goals, 15

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., 12

An *anti-goal* is a soft goal of an attacker which may obstruct the achievement of a security goal., 15

An *argument* contains one and only one claim. It also contains facts and rules in domain knowledge, 19

An *asset* is a resource which has a value and needs to be protected, 15

An *attack* allows an attacker to fulfill an anti-goal. In particular, an attack is a situation in which vulnerability is exploited to cause damage on an asset., 15

An *attacker* is an actor who wants an anti-goal to be satisfied., 15

An *evolution rule* is a formal specification of automatic behavior in reaction to changes in the model, 11

An *initial requirement* is an optative property wanted by a stakeholder., 15

An *object* is an actor, a process or a resource., 15

Attacks is a relationship from one situation to a vulnerable actor, 16

Carries Out is a relationship either from an actor to a process, which specifies that an actor carries out a certain activity., 16

Contributions is a relationship among goals/security goals which indicates that a goal contribute to the satisfaction of another goal., 16

Damages is a relation from an attack to the assets, 16

Decomposes is a relationship from a goal to its subgoals, which indicates that a goal can be refined

AND-decomposition lists subgoals that must all be satisfied in order to satisfy the goal, whereas OR-decomposition suggests alternative ways to satisfy the goal, 16

Delegates is a relationship from one actor to another which specifies that the fulfillment of a goal or the provisioning of an activity/resource, 15

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., 19

DSML stands for Domain Specific Modeling Language., 2

Dynamic Oriented Object Requirement System tools dedicated on Requirement Management. For further details see [19]., 40

Essential elements are elements of logical layer specification (services, data)., 6

Exploits is a relationship from an attack to a vulnerability, which is a (part of) specification that can be vulnerable to expose security problems, 16

Facts are grounded predicates -- something that are either true or false where terms in these predicate must be constant, 19

Fulfills is a relationship from resources and activities to a goal, which specifies a goals is fulfilled by a combination of the resources and the activities, 16

Transformation is the process of deriving models from each other, 24

Incremental model transformations update existing target models based on changes in the source models, and minimize the parts of the source model that need to be reexamined by a transformation when the source model is changed. These aspects are called *target* and *source incrementality.*, 25

Mitigations are *another* special kind of arguments following the iteration of rebuttals in order to reestablish the truth value of the associated claims, 19

Obstructs is a relation from an anti-goal to the corresponding security goal, 16

Provides is a relationship either from an actor to resources, which specifies that an actor provides a certain resource and/or activity, 16

Protects is a relationship from a security goal to a set of valuable assets, 16

Rebuttals are a special kind of arguments whose purposes are to establish the falsity of their associate claims or make them indeterminable, 19

situation, 11, 16, 17, 18, 44

Targets are elements of physical layer specification (physical component, communication channel)., 42

The argument may require *sub-arguments* to establish the truth of certain facts or intermediate predicates, 19

The *context* is a situation within which the system-to-be will operate, 15

Trusts is a relationship from one actor to another, which indicates the belief of one actor that the other will provide a resource or will perform a certain activity., 15

Uses is the opposite relationship to Provides from an actor to resources., 16

Wants is a relationship from actors to goals which associates an actor with its goals, including security and anti-goals, 16