D.3.2 METHODOLOGY FOR EVOLUTIONARY REQUIREMENTS

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

Document information

Document Number	D.3.2
Document Title	Methodology for Evolutionary Requirements
Version	3.18
Status	Y2 Draft
Work Package	WP 3
Deliverable Type	Report
Contractual Date of Delivery	31 January 2010
Actual Date of Delivery	31 January 2011
Responsible Unit	OU
Contributors	OU, UNITN, BME, THA
Keyword List	Requirements, Elicitation, Argumentation, Evolution
Dissemination level	PU

Document change record

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

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

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

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

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

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

	2010			figure numbers
2.19	16th June 2010	Revision	Bergmann Gábor (BME)	Move part of Evolution rule example to Section 7
2.20	17th June 2010	Revision	Bashar Nuseibeh (OU)	Review and copy edit, including quality check changes by Elisa Chiarani
2.21	17th June 2010	Revision	Yijun Yu(OU)	Review and copy edit
2.22	17th June 2010	Revision	Federica Paci (UNITN)	Removal of two comments, Correction of the title of section 7.5
3.0	29th Sep 2010	Y2 Draft	Thein Tun (OU)	Initial formalization of argumentation in section 5. Minor changes in sections 3 and 4.
3.1	05 Oct 2010	Y2 Draft	Thein Tun (OU)	Defining link with WP2 in section 3.1. Possible tool support for informal argumentation in OpenPF (section 5).
3.2	20 Oct 2010	Y2 Draft	Thein Tun (OU)	Revised Section 5, and added argumentation meta-model, example using the tool support for informal argumentation, Event Calculus and formalization of argument, and overview of formal reasoning in OpenPF.

3.3	27 Oct 2010	Y2 Draft	Gábor Bergmann (BME)	Addressed some of the reviewer concerns: added discussion of time complexity and missing citations. Also updated the formal definitions to be more easily comprehensible.
3.4	31 Oct 2010	Y2 Draft	Gábor Bergmann (BME)	Added subsection to explain "big picture" i.e. role of Evolution Rules in a process
3.5	1 Oct 2010	Y2 Draft	Thein Tun (OU)	Simplified the conceptual model according Paris discussions. Revised the argumentation syntax and example.
3.6	22 November 2010	Y2 Draft	Federica Paci, John Mylopolous (UNITN)	Cleaned section about conceptual model and added section about integration with risk assessment.
3.9	27 November 2010	Y2 Draft	Gábor Bergmann (BME)	Added generic change model to Section 6. Fixed numbering in Appendix.
3.10	1 December 2010	Y2 Draft	Thein Tun (OU)	Added integration with WP2, and revised the argumentation example.
3.11	05 December 2010	Y2 Draft	Gábor Bergmann (BME)	Incorporated Tun's changes from the other file. Improved wording in Section 6, moved the Thales part into the Appendix.

3.12	10 December 2010	Y2 Draft	Thein Tun (OU)	Overall editing and polishing.
3.13	5 January 2011	Y2 Draft	Thein Tun (OU)	Overall editing and polishing. Addressing internal reviewers comments
3.14	7 January 2011	Y2 Draft	Yijun Yu (OU)	Section 2/3 update the description of argumentation analysis
3.15	13 January 2011	Y2 Draft	Thein Tun (OU)	Overall editing and polishing. Extended Executive Summary.
3.16	14 January 2011	Y2 Draft	Karmel Bekoutou (UNITN)	quality check completed-minor remarks
3.17	26 January 2011	Y2 Draft	Thein Tun (OU)	Addressed quality check remarks and update some figures.
3.17	28 January 2011	Final	Federica Paci (UNITN)	Minor Corrections to Section 5 and 6

Executive summary

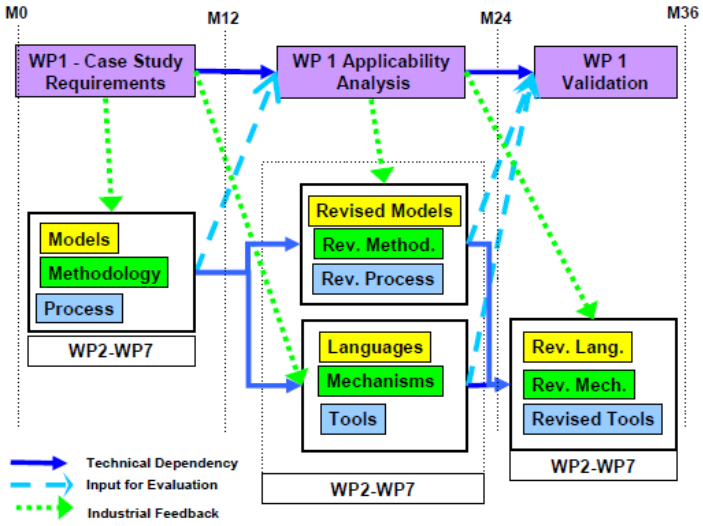
Long lived software systems evolve as their environment changes. When a change happens, security concerns need to be analyzed to re-evaluate the impact of the change on the system and on the assumptions about environmental properties. Typically, change requests are handled in an ad-hoc way: requirements are described informally in natural language, which is prone to ambiguity and uncertain traceability to the evolving design. There is no explicit means to analyze changes with respect to the security goals underlying the evolution of the system design.

To address these problems in a repeatable and systematic way, we have developed and adopted an iterative security methodology for evolving requirements (SeCMER). Every iteration of the SeCMER process starts with an elicitation stage that analyzes every change request or risk assessment into incremental changes of requirements models. These models are represented using consistent, state of the art modeling languages, such as Tropos and Problem Frames. Through a unified extension of existing Security Goals frameworks (e.g., Secure Tropos and Abuse Frames) it is then possible to represent specifications in such a way so as to reveal vulnerabilities through a systematic argumentation analysis, based on the facts and rules about domain properties. Using the propositions in the requirements model, the argumentation process analyzes whether the design has exploitable vulnerabilities that might expose valuable assets to malicious attacks. Facts and domain rules that help identify a rebuttal to the security goals are mitigated by introducing induced changes of security properties from the SeCMER conceptual model. In addition to the structured approach to handling change, the SeCMER approach incorporates incremental transformation of requirement models based on evolution rules. Every evolution rule can be specified formally by events, conditions and actions (ECA). Whenever a change to the requirement model matching some evolution rule(s) is detected, the transformation engine applies specified actions on the requirement model and check whether the existing security goals are still satisfied after the change. If not, the change the passed onto the argumentation process, in order to consider whether the security goal can be restored. When even that is not possible, the security goal will be passed back to the elicitation process where the goal will have to be renegotiated and reformulated. We illustrate the SeCMER methodology and its iterative process through a concrete example of evolution taken from the ATM domain. The example includes: the SeCMER models before and after changes of introducing the Arrival Manager tool and the SWIM communication system; the argumentation analysis for the security goal of protecting SWIM information from unauthorized access; and the example of evolution rules to generalize automatable monitoring and adaptation to the triggering and reactive changes to the SeCMER models.

The SeCMER methodology is not stand-alone: it is integrated at the conceptual level, process level and the tool level with other methods and techniques developed by the SecureChange project. This report discusses how the SeCMER methodology integrates with other SecureChange approaches dealing with the process, architecture and risk. The integration with design and testing are described in respective work packages. At the end of the report, we present the state of practice in processing security requirements, which will be improved by adopting the SeCMER.

Position of the deliverable in the project timeline

The main artifacts of WP3 are the SeCMER conceptual model, the SeCMER methodology for changing requirements, and a CASE tool prototype that supports the different steps of SeCMER methodology. Considering the SecureChange project timeline depicted in the following figure, the SecMER conceptual model and the SeCMER methodology have been conceived during the M0-M24 timeframe, while the CASE tool is going to be developed during the M24-M36 timeframe. The SeCMER methodology presented in this report belongs to the timeframe M0-M24.



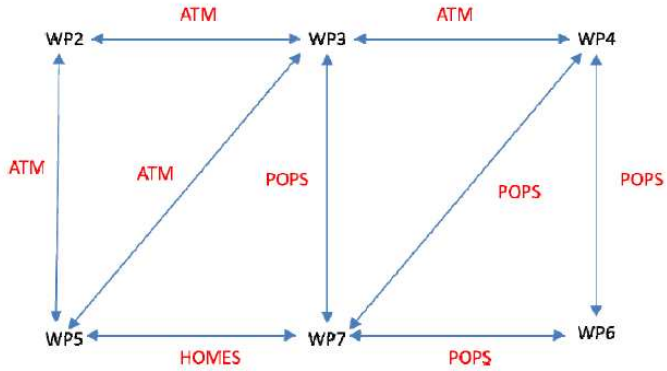
Validation

The WP3 artifacts are SeCMER conceptual model, the SeCMER methodology for changing requirements, and a CASE tool prototype. Each of these artifacts is subject to the validation activities in SecureChange.

The validation activities have not started yet and will be carried out during the third year of the project by organizing a dedicated workshop with ATM experts. For the purpose of the validation, we will use the *process level change* and the *organizational level change* and the security properties *information protection* and *information access*. WP3 uses also the POPS case study, but to a lesser extent to illustrate the integration with testing. The change requirement that is addressed is *specification evolution*, and the security property is *life-cycle consistency*.

Integration

The strategic position of WP3 in terms of case studies and integration with technical artifacts of the other work packages is shown in the figure below. The ATM case study serves as the example for demonstrating the integration with artifacts of WP2, WP4, and WP5. The POPs case study is used for exemplifying the integration with artifacts of WP7.



WP3-WP2. The integration link between WP3 and WP2 is described in this report and in D2.2. The integration is both at artefacts and at process level. The SeCMER conceptual model of the evolving requirements is a specialisation of the Requirement Model package of the Integrated Meta Model presented in D2.2, while the SeCMER methodology steps are an instantiation of the Overall Process. The integration is demonstrated based on the ATM case study, addressing the organization level change and the security properties of information protection.

WP3-WP4. The integration link between WP3 and WP4 is reported described in this report and in D4.2. The integration shows how UMLseCh can be used to help with verifying that requirements are actually met by a system and that they are complete with respect to high-level security objectives. The integration is demonstrated with the ATM case study, addressing the organization level change and the security properties of information protection and information provision.

WP3-WP5. The integration link between WP3 and WP5 is described in this report. The integration is both at conceptual level and at process level. At the conceptual level, an integration of concepts is presented and it is explained how requirement model artifacts should be mapped to risk model artifacts and vice versa. The process level integration leverages on the conceptual level integration for the integration of the requirements elicitation and risk assessment methodologies. The integration is demonstrated in the ATM case study, addressing the organization level change and the security properties of information protection and information provision.

WP3-WP7. The integration link between WP3 and WP7 is described in this report. The integration is both at conceptual level and at process level. At the conceptual level, an integration of concepts is presented and it is explained how requirements artifacts should be mapped to test artifacts and vice versa. At the process level, the integration of requirements methodology and testing methodology is described.The integration is demonstrated based on the specification evolution change requirement of the POPS case study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

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

1.1 Challenges of Evolution

When evolutionary changes violate security goals, a system may be left vulnerable to attacks [22]. Dealing with changes to security goals poses several challenges, including:

- **Ad hoc elicitation of security goals**. Most security goals are implicit or are added after security violations have happened, which makes it difficult to prevent security problems and address vulnerabilities in a proactive way. The SecureChange Methodology for Evolutionary Requirements (SeCMER) described in this report provides a conceptual model and a process for making the elicitation of security goals systematic.
- **Imprecise modeling of requirements**. Security requirements, in order to support automation support, demand a formal description that can be used to analyze, argue and evaluate. Vaguely expressed informal natural language descriptions are difficult for automatic functions to give an assessment of the problem and to provide useful mitigation advices. SeCMER provides a light-weighted approach to formalizing and reasoning about changing security goals.
- **Limited analysis of the impact of change**. Even when changes have happened systematically, there are no mechanisms to argue formally about these changes with respect to the domain knowledge of the system. Will the system collapse due to a subtle change of a trust assumption, for example about the system boundary? Can the system respond to the introduction of a new fact or domain knowledge that often invalidate the existing justification of security? SeCMER offers an approach based on argumentation and model transformation to reason about the impact of change.
- **Lack of an integrated approach**. Change management of security requirements is often not integrated with risk modeling tools. Addressing this limitation requires an explicit mapping between the changes of security requirements and the system vulnerability in order to assess their impact on the system-to-be. When requirements tools such as DOORS and risk analysis methodologies and tools are not integrated, mitigation is often a late response to continuous evolution of software systems. Integration of our methodology with other work packages such as WP5 (Risk Assessment) addresses this issue.

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.

1.2 A Framework for Requirements Evolution

According to Zave and Jackson [27], requirements engineering involves the understanding of the given, or *indicative*, domain properties in the physical world *W* and the specifications of the machine *S*, in relation to requirements *R*, which described the required, or *optative*, properties. These relationships between properties establish a structure in order to facilitate the problem analysis. They are captured by the entailment: $W, S \vdash R$. Based on the work of Zave and Jackson [27], Gunter et al [13] propose a reference model for requirements engineering. This report extends the work of Gunter et al [13] in two ways: first, it show describes the relationships between evolving artefacts, and second, it introduces the security-related notions into the framework.

1.2.1 Relationships between evolving artefacts

This section gives a description of the requirements evolution through relationships between evolving artefacts. These relationships highlight several properties, including the assurance that the modified system can maintain all the existing security goals while new security properties need to be introduced to accommodate changes[1].

Gunter et al [13] identifies five artefacts in system development -- domain knowledge (W), requirements (R), specifications (S), programs (P) and the programming platform or computer (C) -- and describes their general relationships using the logical entailment operator ($\vdash$) as follows.

$$W, S \vdash R$$

$$C, P \vdash S$$

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

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

We view the strength of the logical entailment operator in these formulae to be non-prescriptive: it means that the artefacts (W, R, S, P and C) may be described in varying

[1] This subsection is based on our paper [25].

degrees of formality, from statecharts, temporal logic, etc. to natural language. Likewise, showing that an entailment relationship holds for some given artefacts also may be done to different degrees of formality, from mathematical proofs to informal arguments, depending on the description language chosen and the specific needs of the stakeholders. When formal description languages are used, the proof can be done through logical deduction.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.2 Security-related notions

The challenges of addressing evolving security goals arise from multiple facets of engineering problems. Existing methodologies deal with the changes in security goals with different focuses. For example, Secure Tropos have been used to model both functional and non-functional requirements of stakeholders as *security goals* [20]. 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 [15] approaches for security (e.g., *abuse frames* [18, 19]) focus primarily on modeling the relationship between the attacker behavior and system properties. Although individually these approaches are powerful in modeling and analysis of different perspectives of the security problems, it is not clear how the synergy between them can be exploited.

We extend the framework of Gunter et al [13] to address security concerns by considering the security-related concepts such as *assets, threats, vulnerabilities,*

attackers, trust assumptions, risks and *satisfaction argumentation* [12], as well as risk-related concepts such as *threats*, *assets* and *damages*.

1.3 Overview of SeCMER

Based on the framework presented in the previous section, this report proposes the SecureChange Methodology for Evolutionary Requirements (SeCMER).

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

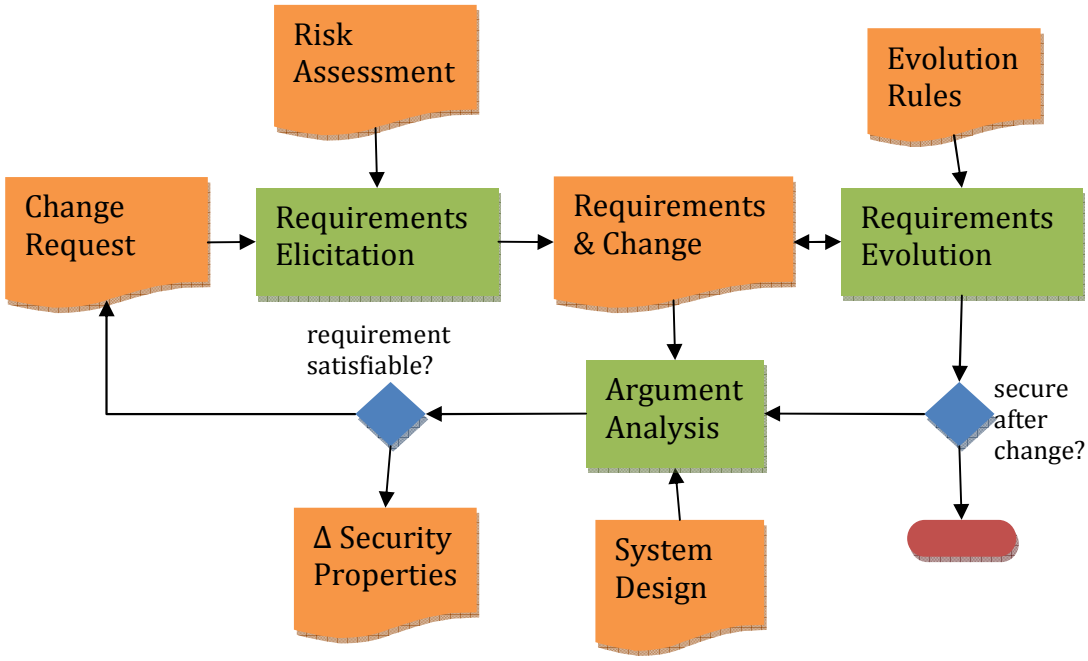


Figure 1 An Overview of SeCMER

The inputs to the process in the SeCMER methodology are:

- **Change Request**: Informal requests for change made by users and customers of the secure software system. These requests are typically managed using tools such as Issue Tracking systems.
- **Risk Assessment**: Risks analysis can produce risk treatments that are candidates for requirements and change.
- **Existing System Designs**: Artifacts describing the main components of the systems—software, hardware, and people—their configuration, behavior and properties. They may be documented using natural language text, UML diagrams, or formal descriptions.
- **Requirement & Change**: Statements of properties, including security goals, the existing system satisfies, together with the changes that need to be made to the existing requirements model. When changes are implemented, it is necessary to

check whether properties of the existing design are satisfied by the new design, and if not, formulate properties that need to be satisfied by the new design.

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

The proposed methodology for handling change has three main steps:

1. **Requirements Elicitation**: When a change is proposed through a change request or risk assessment, the existing design is examined to (a) identify the context of the proposed change, (b) check whether the proposed change is necessary. In terms of the framework described previously, this stage establishes W_{future} and R_{future}. A conceptual model of static requirements (see Section 2 for guidelines) supports this step.
2. **Argument Analysis**: This stage checks whether there are new security properties to be added or to be removed (Δ Security Properties) as a result of changes in the requirement model. Furthermore, a high-level and long-term feedback is possible, in order to adapt/update evolution rules in a way that more human effort can be saved by automation in the future. This stage derives the Δ Security Properties (Δ_{sp}) so that $\Delta_{sp} \cup S_{now} \vdash S_{future}$. This stage is supported by the conceptual model of argumentations presented in Section 3. This stage may be carried out either before or after the evolution stage.
3. **Requirements Evolution**: This stage monitors any changes made to the requirement model, and when changes that match the patterns of evolution are detected, predefined transformation is applied to the requirements model. The transformation will automatically establish whether the existing security properties have been broken by the change or not. This stage checks the entailment (4) in the previous section, namely that $W_{future}, S_{future} \vdash R_{now}$. A conceptual model of evolution (Section 7), and automated transformation (Section 4) support this step.

In practice, there are likely to be several change requests at a time, and these requests have to be stored, prioritized, scheduled, resourced, implemented and tested. However, these issues are outside the scope of SeCMER.

1.4 The ATM Example

Since this report focuses the on process level change requirement and the information access and information protection properties, the scenario fragment we are going to consider is transmission of FDD (Flight Data Domain) data to the AMAN (Arrival Management system) via the new communication network. We want to focus on how to enforce access control policies on FDD transmission and how to ensure confidentiality of FDD. In terms of security means, we are going to apply the SeCMER methodology for requirement change management to the ATM case study. We will produce SeCMER models before and after changes of introducing the Arrival Manager

tool and the communication network and the argumentation analysis for the security goal of protecting FDD from malicious attack.

1.5 Structure of the report

The structure of the remainder of the report follows an iteration of the SeCMER methodology, which includes three main steps: Requirements Elicitation, Requirements Evolution, and Argumentation Analysis. Section 2 presents the detailed conceptual model and the process used in SeCMER. Section 3 explains the argumentation framework for analyzing the security goals and their changes. Sections 7 defines the change model, and Section 4 discusses how the process of change can be automated using the incremental model transformation technique. Application of SeCMER to the ATM example is provided in Section 5. Section 6 presents how SeCMER is integrated with the SecureChange approaches to process, architecture, risk, design and testing. Various concepts associated with the notion of change are discussed in Section 7. Conclusions can be found in Section 8.

2 Requirements Elicitation

The first step of SeCMER methodology is the elicitation of the security goals the system-to-be should be built on, as highlighted in the following figure.

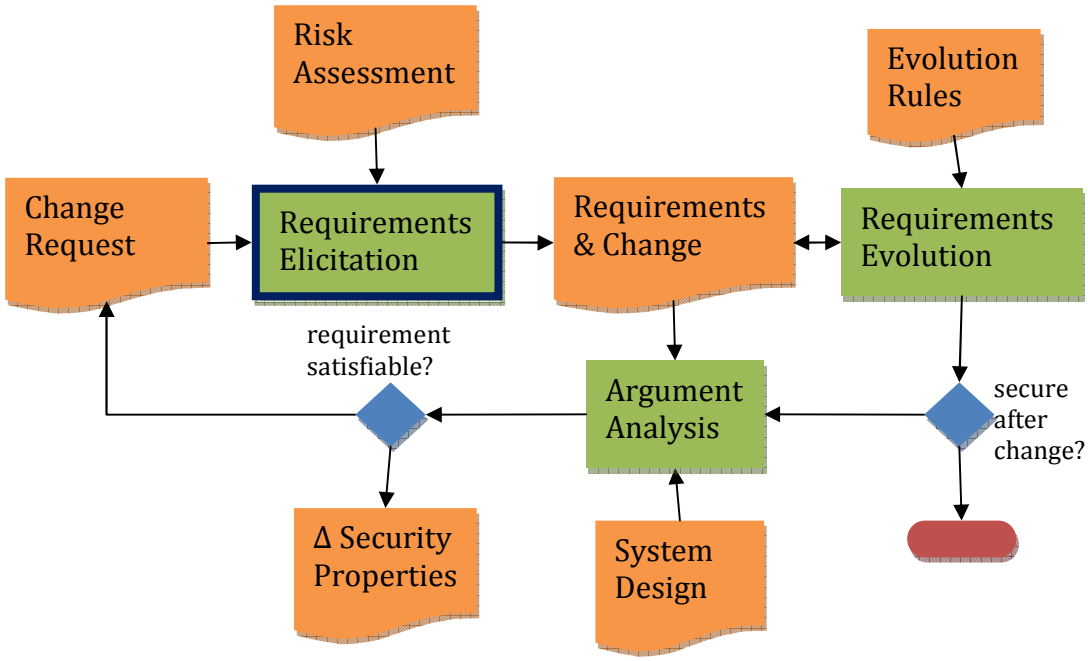


Figure 2 An Overview of SeCMER (Requirements Elicitation)

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

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

The SeCMER methodology' security goals elicitation step produces a requirements model which is an instance of the SeCMER conceptual model. The conceptual model identifies a set of core concepts that allow linking the empirical security knowledge such as information about vulnerabilities, attacks, and threats to the stakeholder's security goals. To create this link, the conceptual model amalgamates concepts from Problem Frames (PF) [15] and Goal Oriented requirements engineering methodologies (GORE) [17, 26] with traditional security concepts such as vulnerability and attack. The combination of the two security goals engineering approaches has several advantages:

with GORE analysis, malicious intentions of attackers can be identified through explicit characterization of social dependencies among actors; with PF security goals analysis, valuable assets that lie within or beyond the system boundary can be identified through explicit traceability of shared phenomena among physical domains and the machine itself.

2.1 The SeCMER conceptual model

The most general concept is *World*, which has as instances all the things that can exist in the world. Lower levels of the conceptual model include concepts from GORE, PF and argumentation frameworks, with security concepts occupying the lowest strata of the conceptual model. Key among the concepts that are introduced is the concept of *Proposition*, with instances such as ``Want for customers for our business" and ``Paolo is married``. The other key concept is that of *Situation*, representing a partial state of the world, e.g., ``High oil prices``, or ``Unhappy customers are many``.

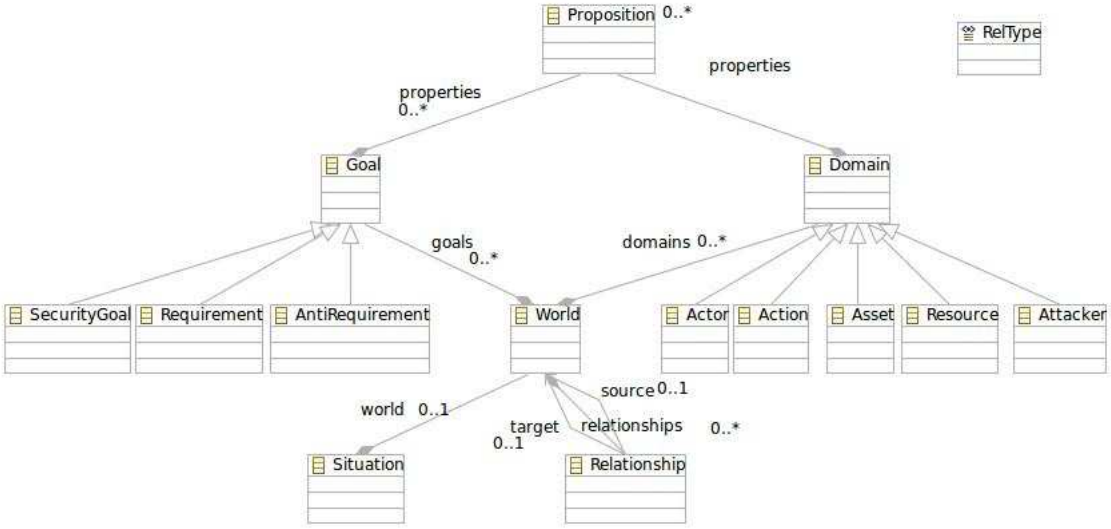


Figure 3 Security Requirements Conceptual Model

A *proposition* is an object representing a true/false statement. A *situation* is a partial state of the world described by a proposition (its description in [11]). Arbitrary propositions are *true/false/undefined* in a situation, given its partial world status. The status of the world is expressed by a predicate over the entities involved.[2]

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

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

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

Domain. Domain is specialized into *Actor, Action, Asset,* and *Resource*.[3] An *actor* is an entity that can act and intend to want or desire.

An *action* is an entity performed by an actor, which can generate events, and can have preconditions and post-conditions. *Attack* (not shown in diagram) is an action that may be carried out by an Attacker. A *resource* is an entity without intention or behavior. An *asset* is an entity of value that can be owned and used. For example, an asset can be an passenger (actor) whose life needs to be protected, can be an engine (process) whose behavior has a value to the protector, or can be an aircraft (resource) whose value are tangible for other actors. A *relationship* such as the organization chart of the air traffic management organization is also an asset as long as its value needs to be protected.

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

For the sake of security goal analysis, the conceptual model includes also the following types of Relationship: *argues*, and *interfaces*. *Argues* is necessary to show whether certain requirements can be met or not. *Interfaces* are links between domains. A complete list of all the possible relationships is found in Figure 3.

Propositions. A goal is a concept found in GORE approaches, and represents a proposition an actor wants to make true. For security analysis purposes, Goal is specialized into *Requirement,* and *Security Goal*. A *requirement* is a goal wanted by a stakeholder. A *security goal* prevents harm to an asset through the violation of confidentiality, integrity, and availability security properties [14].

Situations. The Domain concept coming from PF approaches is a specialization of Situation. This concept is useful to define the situation of system boundaries, to allow one place focus on analysis and to hide the unnecessary details. For the analysis of every problem or sub-problem, a different situation may be selected from the physical world. Thus the context is a situation in which the system-to-be will operate; and a domain is a situation that is part of the context. In PF, domains can be classified as biddable, causal, and lexical. By biddable, a domain's behavior is not fully predicable or controllable, usually represented by human actors or natural processes. By causal, a domain's behavior is predicable or controllable, usually represented by activities. By lexical, a domain's behavior is predefined, usually by a resource.

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

```
1  grammar eu.securechange.ontology.Ontology with org.eclipse.xtext.common.Terminals
2  generate ontology 'http://securechange.eu/ontology'
3
4  Situation: ('model' time=ID ':')? world=World;
5  World: {World} (entities+=Entity | relationships+=Relationship)*;
6  Entity: Goal | Domain;
7  Goal: name=ID type='goal' (asset=Asset)? ((',')? properties+=Proposition)* | SecurityGoal | Requirement;
8  SecurityGoal: name=ID type='sec' (asset=Asset)? ((',')? properties+=Proposition)*;
9  Requirement: name=ID type='req' (asset=Asset)? ((',')? properties+=Proposition)*;
10 Domain: name=ID type='dom' (asset=Asset)? ((',')? properties+=Proposition)* | Actor | Action | Resource;
11 Asset: name=ID '$' (value=STRING)?;
12 Actor:  name=ID type='actor' (asset=Asset)? ((',')? properties+=Proposition)*;
13 Action: name=ID type='action' (asset=Asset)? ((',')? properties+=Proposition)*;
14 Resource: name=ID type='resource' (asset=Asset)? ((',')? properties+=Proposition)*;
15 Proposition: name=STRING;
16 Relationship: type=RelType '(' (','? entities+=[Entity])* ')'  ((',')? properties+=Proposition)*;
17 enum RelType: CARRIESOUT='carries out' | FULFILS='fulfils'  // refinement
18     | AND_DECOMPOSES='and decomposes' | OR_DECOMPOSES='or decomposes' // decompositions
19     | HELPS='+' | HURTS='-' | BREAKS='--' | MAKES='++' // contributions
20     | WANTS='wants' | DEPENDS='depends' | DELEGATES='delegates' | TRUSTS='trusts' // i*
21     | PROVIDES='provides' | CONSUMEs='consumes' | INTERFACES='interfaces' // PF
22     | DAMAGES ='damages' | ATTACKS='attacks' | PROTECTS='protects' // security
23     | ARGUES ='argues'; // argumentation
24
25 terminal ID:
26  ('#' (!('#'))+ '#') | ('^'? ('a'..'z' | 'A'..'Z' | '_' | '.') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '.')*);
```

Figure 4 Conceptual Model Representation in EBNF of xtext

Figure 3 summarizes the elements of our ontology in Extended Backus-Naur Format (EBNF), supported by the Eclipse/TMF (xtext). Lines 1-2 introduce the default terminals and the URI identifier of the grammar. Lines 4-23 are EBNF rules, among which the first rule 'Situation' defines the root element of the model. An EBNF rule of the form 'A: B | C' indicates that the concept A has sub-concepts B or C as specializations. A rule of the form 'A: B*' indicates that each instance of A consists of (has parts) zero or more instances of B. The notation 'B+' is similar to 'B*', but allows for one ore more instances. Similarly the notation 'B?' indicates an optional element (zero or one), whilst '[B]' denotes a reference to B instead of an optional B. Furthermore, the string constants used in these rules are treated as preserved keywords in the concrete syntax, such as 'actor', 'goal', etc.

The terminal ID (Lines 25-26) is an extension to the default ID in xtext. It identifies the domains and propositions using a space-separated free-formed phrase quoted by '#', such as '#A security goal#', instead of 'a_Security_Goal'.

2.2 The SeCMER requirements elicitation process

The requirements elicitation process is an iterative process that consists of the following steps:

1. *Domain modeling*
2. *Goal and Action modeling*
3. *Trust modeling*
4. *Delegation modeling*

The process starts with the *domain modeling activity* in which the relevant actors stakeholders and existing software (subsystems) are elicited and modeled with their goals.

Goal and action modeling focuses on the goals/requirements associated with each actor in the actor diagram and analyzes them using various forms of analysis. In

particular, Means-Ends analysis aims at identifying actions, resources and goals that provide means for achieving a high-level goal. The Contribution analysis identifies goals that can contribute positively or negatively in the fulfillment of the goal under analysis. The AND/OR Decomposition analysis refines a high-level goal into AND/OR composition of sub-goals, resulting in a finer goal structure. During these analyses, new dependencies can be discovered so as to revise and enrich the model produced.

Trust modeling consists of identifying actors who trust other actors for fulfillment of certain goal, actions, and resources, and identifying actors which own goal, plans, and resources.

Delegation modeling consists of identifying actors which delegate to other actors the permission and task of execution on goals, plans, and resources.

2.3 The ATM Example

We now describe the ATM example of the scenario fragment relating to the transmission of FDD (Flight Data Domain) data to the AMAN (Arrival Management system) via the new communication network. First we produce SeCMER model before the introduction of the Arrival Manager tool and the communication network.

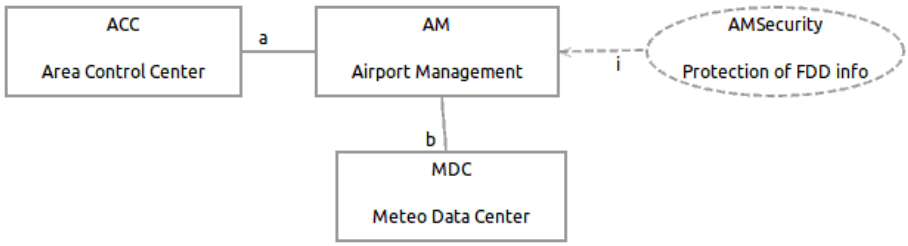


Figure 5 A SeCMER requirement model capturing the relevant domains before the change

Figure 5 shows a SecMER requirement model fragment capturing the relevant domains before the changes. The model captures the given domains and their connections as they currently are in ATM domains. The diagram shows that the Airport Management system is connected to the Meteo Data Center and the Area Control Centre through interfaces ‘a’ and ‘b’ respectively. ‘a’ and ‘b’ are point-to-point communication systems before SWIM is introduced. The the Airport Management system has several components, including the Arrival Management (AMAN) system.

Figure 6 shows another SeCMER requirement model showing how the introduction of the SWIM Network, an IP based data transport network, changes the structure of the ATM components, together with a simple description of the interfaces between the components. In the after change diagram, the two specific legacy systems, namely, Airport Management and Meteo Data Center are connected through the SWIM network. This change, in a sense, replaces the interface ‘b’ with the SWIM network, SWIM boxes and adapters. The diagram also makes explicit the security goal that needs to be maintained after the change has been introduced.

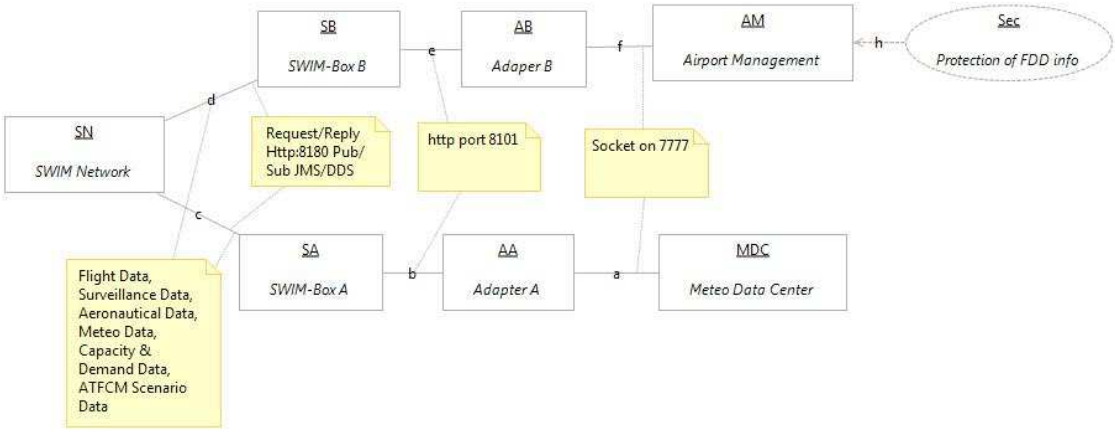


Figure 6 A SeCMER requirement model after the change

3 Argumentation Analysis

As discussed in the introduction, the satisfaction of security goals in the general form of the entailment *W, S* ├ *R* needs to be argued either semi-formally or formally [14]. Security goals are often a collection of claims whose satisfaction depends on a combination of facts, trust assumptions and domain knowledge. Arguments may rebut and mitigate one another. This section describes the argumentation analysis step of the SeCMER methodology as highlighted in the following figure.

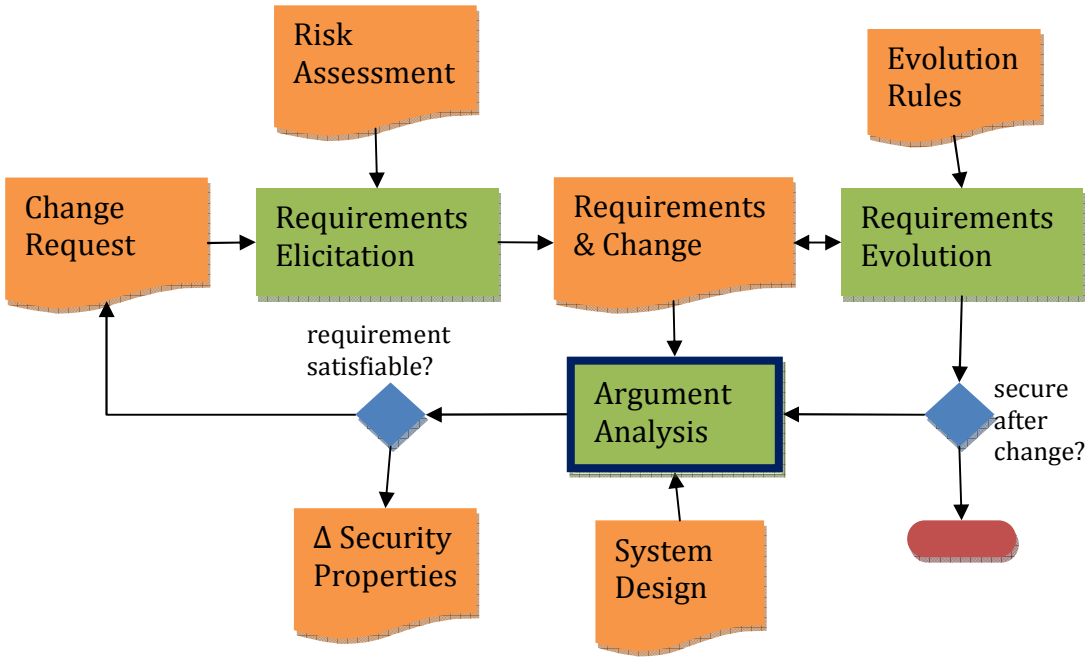


Figure 7 An Overview of SeCMER (Argument Analysis)

3.1 Overview

Our argumentation is based on the informal Toulmin structures first published in the 1950's [3]. To consider it in the formal settings, however, we have simplified the conceptual models. The most important concepts in argumentation are defined as follows:

- A *claim* is a predicate whose truth-value will be established by an argument.
- An *argument* contains one and only one claim. It also contains facts and warrants.
- A *fact* is a grounded predicate -- something that is either true or false where terms in the predicate must be constant.

- A w*arrant* may be either facts or a *trust assumption*, an ungrounded predicate that can be evaluated to true or false once the values of all terms in the predicate are known. A warrant link facts in an argument to the claim.

We now present a formalized meta-model of the arguments.

3.2 Meta-model of arguments

The meta-model of SeCMER arguments we implemented is described in Figure 8.

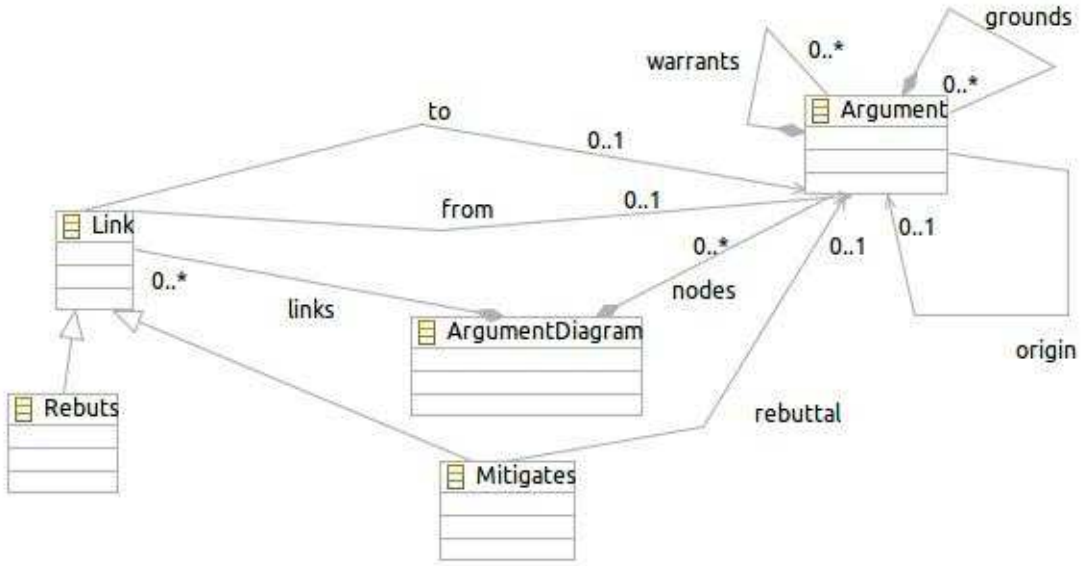


Figure 8 Meta-Model of the Argumentation

- An *argument diagram* may have several arguments linked to each other.
- Every argument has an optional timestamp, which indicates the time (or the round) during the argumentation process at which the argument is introduced. For a given argument, an initial iteration is to establish the truth of its associated claim. The initial claim needs to be supported by some facts, and warranted by either further fact(s) or sub-argument(s). Since a warrant in an argument can be an argument, arguments can be nested. This allows high-level arguments to rely upon predicates the truth values of which are established by later sub-arguments. Therefore, these sub-arguments are also arguments, but they are meant to provide supporting evidence (as sub-claims).
- As well as internal nesting of arguments, arguments may be related to each other through *rebuttal* and *mitigation/restore* relationships. A rebuttal argument is a kind of argument whose purposes are to establish the falsity of their associated argument or make them inconsistent. Similarly, *mitigations* are *another* special kind of arguments following the iteration of rebuttals in order to reestablish the truth-value of the associated *original* claims. Mitigations may or may not negate the claims of the rebuttals: sometimes they add further facts overlooked by the rebuttals. In cases of arguments with several levels of

rebuttals and mitigations, it is desirable to show explicitly the original argument whose claim a mitigating argument is targeting at. For this purpose, the *restore* relationships are used between the mitigation relationship and the original argument. The arguments that introduce the rebuttal and mitigation relationships 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, throughout the argumentation, the same reasoning mechanism should be used consistently for all arguments.

3.3 Visualizing SeCMER Argumentation

We have extended the OpenPF tool[4] to support the informal argumentation described by the meta-model in Figure 8. One aspect of the tool is to allow security engineers to document and visualize the argument structures before and after introducing a change.

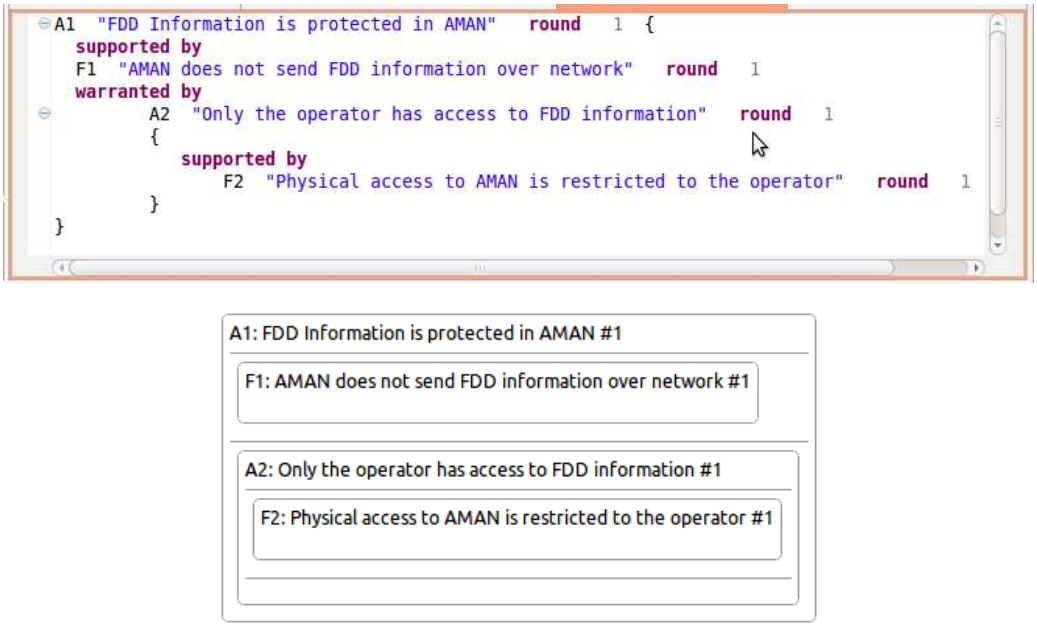


Figure 9 Textual Input Syntax and Visual Syntax of SeCMER Arguments

The diagram in Figure 9 shows the input textual syntax and visual syntax of an argument in the OpenPF. An argument is represented as a node with three compartments. At the top of a node is a label indicating the ID and description of the claim, together with the timestamp (round number). The grounds are written as nodes inside the middle compartment and the warrants are written as nodes inside the bottom compartments. Warrant may be either a fact or an argument, thus allowing nesting of sub-arguments.

[4] The tool and examples here can be downloaded from http://sead1.open.ac.uk/openpf/ and http://computing-research.open.ac.uk/trac/openre/wiki/Examples/argument/SecureChange

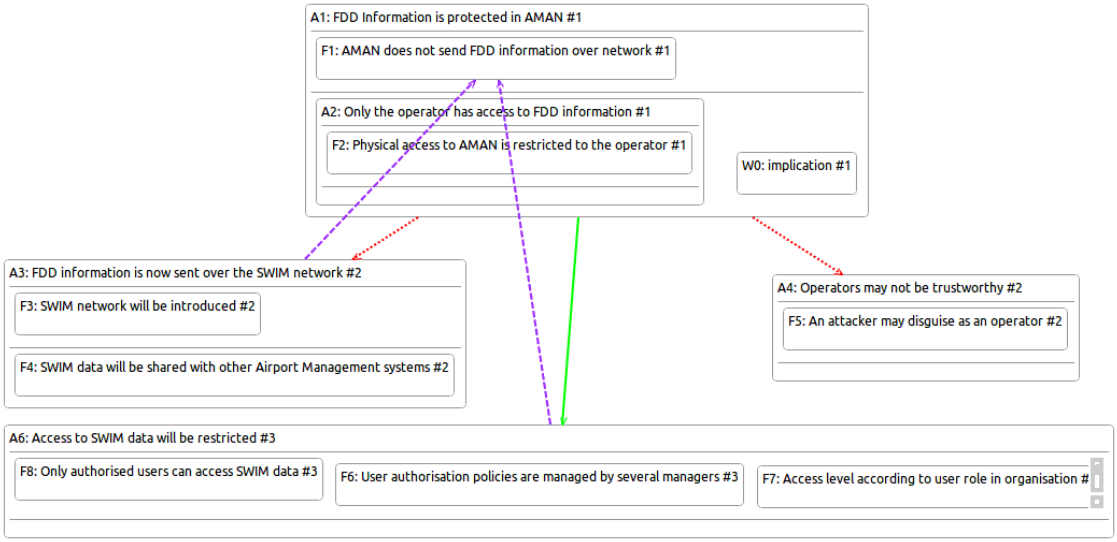


Figure 10 Visual Syntax of Links between Arguments

Furthermore, in an argument diagram, there may be more than one argument, which are related to each other through rebuttal and mitigation links. Figure 10 illustrates these four links: rebuttal link is represented by a red dotted line, whilst the mitigation link is represented by a solid green arrow. Purple dashed arrows represent relationship between facts (or trust assumption) whose value has been negated in a rebuttal or restored by a mitigation. The optional solid pink arrow shows the rebuttal the mitigation addresses in restoring the initial argument, which is useful when there are several rebuttals to an argument and several mitigations to those rebuttals.

3.4 Formally Checking Links between Arguments

Once visualized in this way, arguments during the change process can be evaluated formally. To illustrate this point, in this section we use proposition logic as the underlying formalism, whilst Section 3.3 will extend the formalism to temporal logic to reason about the software behaviors. We have implemented the algorithm described in Figure 11, which uses the rule that for each argument the validity can be established by checking that Ground /\ Warrant → Claim assuming that Ground and Warrant are true. By adding and removing grounds and warrants in later rounds of arguments, the validity of the initial argument is rechecked. For each path in the argument structure, we first check the rebuttal arguments (lines 2-14) before checking the mitigation arguments (lines 14-24).

Input: 1) $A =< (C_n, \Delta G_n, \Delta W_n) \mid (n \in \mathcal{N}) \wedge (\Delta G_n, \Delta W_n \vdash C_n) >$: a sequence of incremental arguments where C_n is the proposition claimed by the increment, grounds ΔG_n and warrants ΔW_n are respectively incremental sets of ground facts and domain knowledge rules;
2) $B = \{(p, q) \mid (p \vdash \neg q)\}$: a set of mappings between two facts or domain knowledge rules p and q, where q is called a *trust assumption* of an argument in a round earlier than that of p from a sequence $< a_0, \ldots, a_n >$. Note that if both $(p, q) \in B$ and $(q, r) \in B$, then (q, r) is removed from B when the incremental argument that concerns p is considered.

Output: For the argument at the i-th round with a claim C_i where $0 \leq i \leq n$:
1) $R = \{(a_j, a_{j+1}) \mid (\mathcal{KB}_j \vdash C_i) \wedge (\mathcal{KB}_{j+1} \nvdash C_i)\}$: a set of rebuttal relationships, where $\mathcal{KB}_i$ is the knowledge base consists of all the facts and the domain knowledge rules in A, except for those trust assumptions denied by B in the latest incremental arguments a_i for $0 < i \leq n$;
2) $M = \{(a_j, a_{j+1}, a_{j+2}) \mid (\mathcal{KB}_j \vdash C_i) \wedge (\mathcal{KB}_{j+1} \nvdash C_i) \wedge (\mathcal{KB}_{j+2} \vdash C_i)\}$: a set of mitigation relationships.

```
1  begin
2      KB_0 := {}
3      for i = 0, |A|, +1 do
4          for each q ∈ (ΔG_i ∪ ΔW_i) do
5              for j = |A|, i + 1, −1 do
6                  if ∄p ∈ (ΔG_j ∪ ΔW_j) | (p, q) ∈ B then
7                      KB_i := KB_i ∧ q
8                  end
9              end
10         end
11         for i_0 = 0, i, +1 do
12             V_{i,i_0} := eval (KB_i ⊢ C_k)
13         end
14     end
15     for i = 0, |A| − 1, +1 do
16         for j = i, |A|, +1 do
17             if V_{j,i} ∧ ¬V_{j+1,i} then
18                 R := R ∪ {(A_j, A_{j+1})}
19                 if V_{j+2,i} then
20                     M := M ∪ {(A_j, A_{j+1}, A_{j+2})}
21                 end
22             end
23         end
24     end
25 end
```

Figure 11 The algorithm for checking rebuttal and mitigation arguments

We will now illustrate the idea of the algorithm using the example introduced in Figure 9.

1. Starting from the example root node A1, in Round #1, we have:

   W0 ⇔ (F1, A2 → A1)…….//implicit in the A1 structure and W0 is a shorthand

   F2 → A2……..//implicit in the A2 structure

   F1, F2,W0 ……..//grounds and warrant of A1 and A2

2. Rebuttals negate the associated facts or the claims. By introducing the change in A3 at the round #2, for example, we have

3. W0 ⇔ (F1, A2 → A1)……//implicit in the A1 structure

   F2 → A2……..//implicit in the A2 structure

   F2, W0 ……//removing F1 by A3

   F3, F4 ……// grounds of A3

   F3, F4 → A3 ……..……//implicit in A3 structure

   A3 ⇔ ¬ F1……..// given in A3 structure

It is thus possible to establish that ¬A1 can be satisfied, which denies the original claim in A1. Thus A1 is rebutted by A3. Similarly, the changes introduced by A4 at the round #2 also rebut the claim of A1:

1. A4 ⇔ (F1, A2, !F5 → A1)……//W0 replaced by A4

   F2 → A2……//implicit in the A2 structure

   F5 → A4……//implicit in the A4 structure

   F5 ……..……//ground of A4

   F1, F2 ……..……//ground of A1 A2

From these examples one can see that the claim of an argument may be rebutted in more than one way.

A mitigation argument may negate the effect of a rebuttal by restoring the truth of the original claim. For example, after round #3 in A6, in relation to A3 we have

1. W0 ⇔ (F1, A2 → A1) …… ……//implicit in the A1 structure

   F2 → A2……..//implicit in the A2 structure

   F2, W0 ……..//removing F1 by A3

   F3, F4 ……..……..……// grounds of A3

   F3, F4 → A3 ……..……//implicit in A3 structure

   A3 ⇔ ¬ F1……..……..// given in A3 structure

   A6 ⇔ (¬F5 & F2 & F8 & F6 & F7 → A1) ……//given in A6 structure

   F6, F7, F8 → A6 ……..…… //implicit in A6 structure

   F6, F7, F8 ……..……..// grounds of A6

   ¬F5……..……..//not a given fact in A1, A3 and A5

This partial argument shows that there is an outstanding rebuttal to argument about the system security after change. In particular, the rebuttal argument A4 needs to be mitigated in order to make the system secure after change.

3.5 Arguments and the conceptual model

Arguments provide a way to structure the system artifacts involving the concepts in the conceptual model (Figure 3). Indeed, the conceptual model introduces the concepts in argumentation, but their relationships to other concepts are further explored now.

Goals and requirements are generally regarded as claims, warrants are the context, and propositions, including the relationships, are the facts. Other domain concepts such as action, resource and actor are orthogonal to the propositions of an argument, in the sense that they can be used in the description of any part of an argument. Moreover, one can describe the behavioral semantics using the temporal predicates supported by the Event Calculus formalism, which is explored in the next section.

3.6 Formalization of arguments using the Event Calculus

First introduced by Kowalski and Sergot [16], the Event Calculus (EC) is a system of logical formalism, which draws from first-order predicate calculus. It can be used to represent actions, their deterministic and non-deterministic effects, concurrent actions and continuous change. We chose the EC formalism, because it is suitable for describing and reasoning about event-based temporal systems such as the Air Traffic Management systems. Several variations of EC have been proposed, and the version we adopted here is based on the discussions in [24]. The calculus relates events and event sequences to fluents, or time-varying properties, which denote states of a system. Table 1, based on [21], gives the meanings of the elementary predicates of the calculus we use in this paper.

Table 1 Elementary Predicates of the Event Calculus

Predicate	Meaning
Happens(a, t)	Action a occurs at time t
Initiates(a, f, t)	Fluent f starts to hold after action a at time t
Terminates(a, f, t)	Fluent f ceases to hold after action a at time t
HoldsAt(f, t)	Fluent f holds at time t
t1 < t2	Time point t1 is before time point t2

The domain independent rules in Table 2, taken from [21], state that: Clipped(t1, f, t2) is a notational shorthand to say that the fluent f is terminated between times t1 and t2 (EC1), Declipped(t1, f, t2) is another notational shorthand to say that the fluent f is initiated between times t1 and t2 (EC2), fluents that have been initiated by occurrence of an event continue to hold until occurrence of a terminating event (EC3), fluents that have been terminated by occurrence of an event continue not to hold until occurrence of an initiating event (EC4), and truth values of fluents persist until appropriate initiating and terminating events occur (EC5 and EC6).

Table 2 Domain independent rules of EC

Clipped(t1, f, t2) ≡ ∃a, t[Happens(a, t) /\ t1 ≤ t < t2 /\ Terminates(a, f, t)] (EC1)

Declipped(t1, f, t2) ≡ ∃ a, t[Happens(a, t) /\ t1 ≤ t < t2 /\ Initiates(a, f, t)] (EC2)

HoldsAt(f, t2) ← [Happens(a, t1) /\ Initiates(a, f, t1) /\ t1 < t2 /\ ¬Clipped(t1, f, t2)] (EC3)

¬HoldsAt(f, t2) ← [Happens(a, t1) /\ Initiates(a, f, t1) /\ t1 < t2 /\ ¬Declipped(t1, f, t2)] (EC4)

HoldsAt(f, t2) ← [HoldsAt(f, t1) /\ t1 < t2 /\ ¬Clipped(t1, f, t2)] (EC5)

¬HoldsAt(f, t2) ← [¬HoldsAt(f, t1) /\ t1 < t2 /\ ¬Declipped(t1, f, t2)] (EC6)

In our approach to formalising the arguments, claims are constraints on the combinations of fluent capturing the required states of the system. System context captures the facts and warrants. We now define them more formally.

Definition 5.1: *Claims* consist of a finite conjunction of (¬)HoldsAt predicates. Reference phenomena (Γ) are observations describing the given state of the system, while controlled phenomena (Γ') are observations describing the desired state of the system. A claim is expressed either as

- ground observations Γ', without any reference to the given state of the resource or given action of the processes, or
- as a relationship between the reference and the controlled phenomena, such as a constraint of the form Γ→Γ', or an action precondition axiom of the form (¬)Happens(f1, t) → Γ' where the antecedent is an occurrence of an action in the system (for example, to say that when an event a1 happens at time t, the fluent f1 must be true at t1).

For example, the claim HoldsAt(AircraftOnGround, t) /\ 0 <= t <= 9 says that the aircraft are on the ground between the timepoints 0 and 9 range; the claim HoldsAt(Airborne, t) → HoldsAt(TransponderOn, t) says that as long as the aircraft remains airborne, the transponder is on; and the claim Happens(BreachSD, t) /\ ¬Happens(Clearance, t2) /\ t <= t1 <= t2 → HoldsAt(AlarmRaised, t1) says that as soon as the separation distance is breached, the alarm is raised until the clearance happens.

Definition 5.2: *Facts* are described as ground observations.

Definition 5.3: *Warrants* in this formalisation can be described in a number ways. Firstly, it can be expressed as a finite conjunction of the event occurrence constraints (Ψ) of the form (¬)Happens(a_1, t) /\ (¬)HoldsAt(f, t) → (¬)Happens(a_2, t) where a_1, a_2, t, and f are terms for the action, time point, and fluent respectively.

Secondly, warrants may be expressed as event-to-condition and condition-to-event causality. The first causality deals with what happens to the fluents when events occur, and the second causality deals with the domain properties that lead to the occurrence

of certain events. In the Event Calculus, the event-to-condition causality is described as a finite conjunction positive effect axioms and negative effect axioms (Σ) of the form Initiates(a, f, t) ←Π or Terminates(a, f, t) ←Π where Π has the form (¬)HoldsAt(f_1, t) /\ ... /\(¬)HoldsAt(f_n, t) and t, and f_1 to f_n are terms for the time and fluent respectively. The condition-to-event causality is described as a finite conjunction of trigger axioms (Δ_2) of the form Happens(a, t) ←Π. For example, the following statement says that if the aircraft has transponder, an occurrence of the event interogateTransponder has an effect of making BroadcastACInfo true.

Initiates(interogateTransponder,BroadcastACInfo, t) ← HoldsAt(HasTransponder, t)

Similarly, the following statement says that the fluent OperatorHasWeatherInfo on becoming true, generates the event sendWeatherInfo because of the functionality SendWeatherInfo.

Happens(sendWeatherInfo, t) ← HoldsAt(OperatorHasWeatherInfo, t) /\
¬HoldsAt(OperatorHasWeatherInfo, t -1)

Note that the condition ¬HoldsAt(OperatorHasWeatherInfo, t-1) is necessary to prevent stuttering of the event sendWeatherInfo when the fluent OperatorHasWeatherInfo holds continuously.

Definition 5.4: *Arguments* in this formalism relies on two assumptions. One is the consistency of the domain theory Σ and observations Γ and Γ'. Another is the uniqueness of fluent and event names, meaning that no two names denote the same thing. This uniqueness axiom is represented by Ω. If the system relies on the feedback from the environment (Δ_2) and observations about the environment (Γ), an argument can be formalised as follows:

$$\Sigma /\backslash \Gamma /\backslash \Delta_2 /\backslash \Psi \models \Gamma'$$

That is, given the facts (observations about the environment Γ), warrants (a theory of the domain Σ, feedback from the system environment Δ_2, and a specification Ψ), and an appropriate deductive system, we want to show that the claim can be satisfied.

Definition 5.5: *Rebuttals* are counterexamples to the satisfaction of the claim, and are defined as follows. When restricted to event occurrences, rebuttals are found through logical abduction in the Event Calculus. We first pose a logical abduction problem in order to find all constructive hypotheses (Δ_1) explaining how, given the domain theory (Σ /\ Γ /\ Δ2), the claim (Γ') can be denied, i.e.

CIRC[Σ; Initiates; Terminates] /\
CIRC[Δ_1 /\ Δ_2;Happens] ^ Γ ^ Ω ╞ ¬Γ'

where Δ_1 is consistent with the domain theory and CIRC is the circumscription operator. Δ_1 is a partially ordered sequences of event occurrences that, given the physical domains, leads to the claim not being satisfied. The circumscription operator assumes that no events other than those by Δ_1 and Δ_2 may occur.

Definition 5.6: Mitigations are facts and/or warrants that remove rebuttals from the argument, i.e. Δ_1 above is empty.

3.7 Arguments and the model transformation

Claims can be 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 specific in the field; to refute the supportive evidence for the claims, one can draw on additional (often non-monotonic or negative) facts and domain knowledge to form claim rebuttals.

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

- The facts and domain knowledge rules that cause a *rebuttal* argument are generated into a pattern that match the SeCMER requirements model;
- The new facts and domain knowledge rules introduced by a *mitigation* argument (some of them are new security properties) are generalized into an incremental transformation where the “before” state of the transformation is the SeCMER requirements model before the mitigation, and the “after” state of the transformation is the SeCMER requirements model after the mitigation.

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

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

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

3.8 Tool-support for formal argumentation

An overview of the tool-support for argumentation in OpenPF is given in Figure 8.

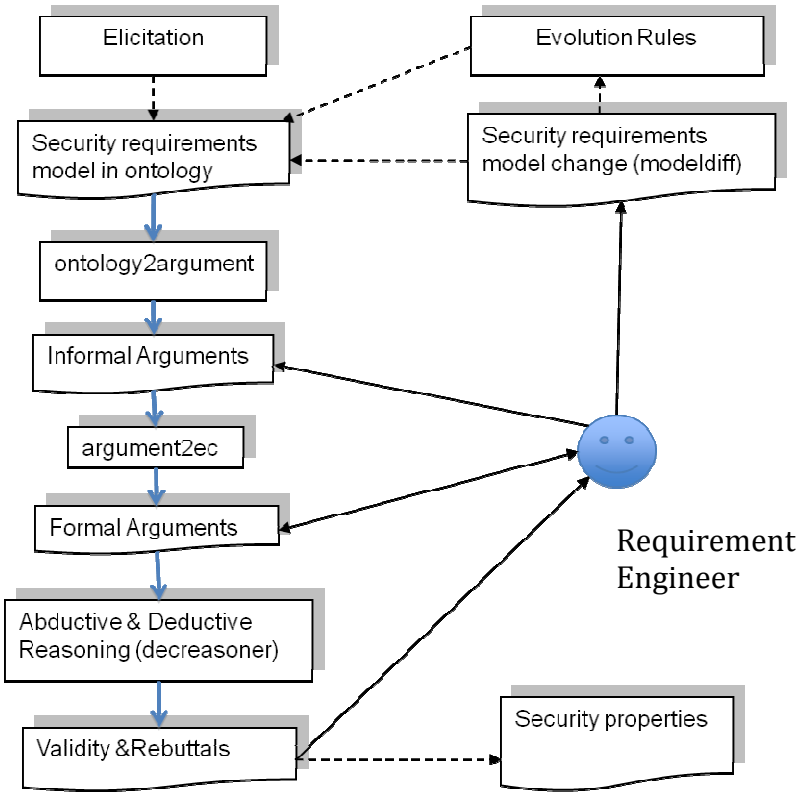


Figure 12 An overview of OpenPF support for argumentation

The workflow starts with the development of a security requirements model in the elicitation stage using the Situation ontology. The OpenPF plug-in ontology2argument generates structure of an informal template, which can be edited by the user. Requirements engineer uses the requirement model to sketch informal arguments for the security goals of the evolving system that will be affected by the proposed change. The informal arguments and the formalized requirements are then used by another plug-in, argument2ec, to generate arguments formalized in the Event Calculus. At this point, two kinds of reasoning can be performed on the arguments: logical *deductive* reasoning to check whether claims in the arguments are valid, and logical *abductive* reasoning to find rebuttals to the claims (see Definitions 5.4 and 5.5). Both types of reasoning are supported through the OpenPF integration of the Event Calculus tool decreasoner.

3.8.1 ATM Example

We now step through OpenPF support for argumentation using the ATM example introduced in Section 2.3. We begin by recalling the structure of the ATM system before the change is introduced (Figure 5). Note that the diagram shows the relevant domains and their connections as they currently are in ATM domains. The diagram shows that the Airport Management is connected to the Meteo Data Center and the Area Control Centre through interfaces ‘a’ and ‘b’ respectively. ‘a’ and ‘b’ are point-to-point communication systems before SWIM is introduced.

After applying the transformation rules to accommodate the change, the diagram in Figure 6 shows another SeCMER requirement model where the point-to-point communication between two specific legacy systems, namely, Airport Management and Meteo Data Center are replaced by with the SWIM network, SWIM boxes and adapters. The diagram also makes explicit the security goal that needs to be maintained after the change has been introduced. This corresponds with the Elicitation and Evolution Rules steps in Figure 12.

ontology2argument is a semi-automated step where the requirement engineer uses the OpenPF tool to create information argumentation diagram. The information argumentation process may go through several rounds. At the beginning, the engineers assume that the current ATM is secure because of certain facts and warrant (Figure 13).

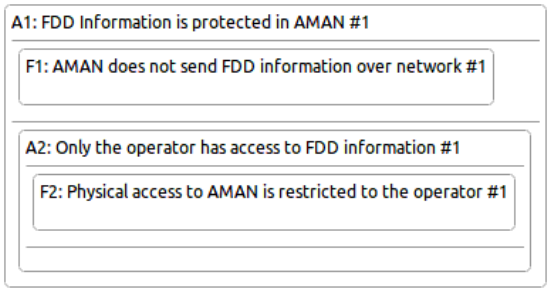


Figure 13 Argument for Security of AMAN before change

Introduction of the SWIM network as shown in Figure 6 adds new domains, facts and warrant which call for the argument to be revised. In particular, we have a new argument that rebuts the orginal argument that the AMAN is secure.

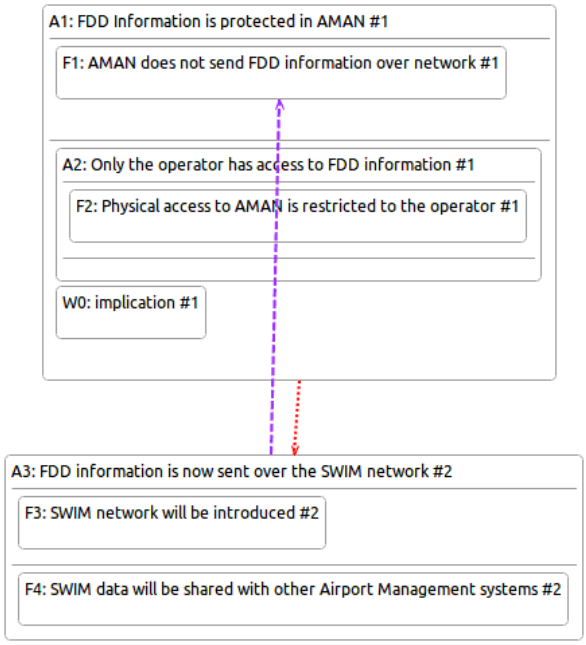


Figure 14 Argument for Security of AMAN before change

Having discovered the vulnerabilities brought about by the proposed introduction of SWIM system, the requirements engineers and the security experts identify various ways to mitigate the rebuttal in order that the initial claim for AMAN security is restored (Figure 15).

Having done the argument analysis, we discovered that the original requirement for system security, namely Protection of FDD (Flight Data Domain) info, couldn't be maintained after the change has been introduced. Mitigations in the arguments led to discovery of additional security properties that need to be discharged in order to maintain the overall system security. Figure 16 shows a SeCMER requirement model for an additional security property. It stresses the necessity to change required security properties in order to accommodate changes while maintain the same security level. The introduction of the AMAN and the SWIM Network requires additional security properties. They include: *'Queue Management Information shall not be accessible by meteo data centres'*, or *'Queue Management Information shall not be accessible by anyone other than those working with AMAN'*. The structured SeCMER's argumentation supports the verification of such additional properties.

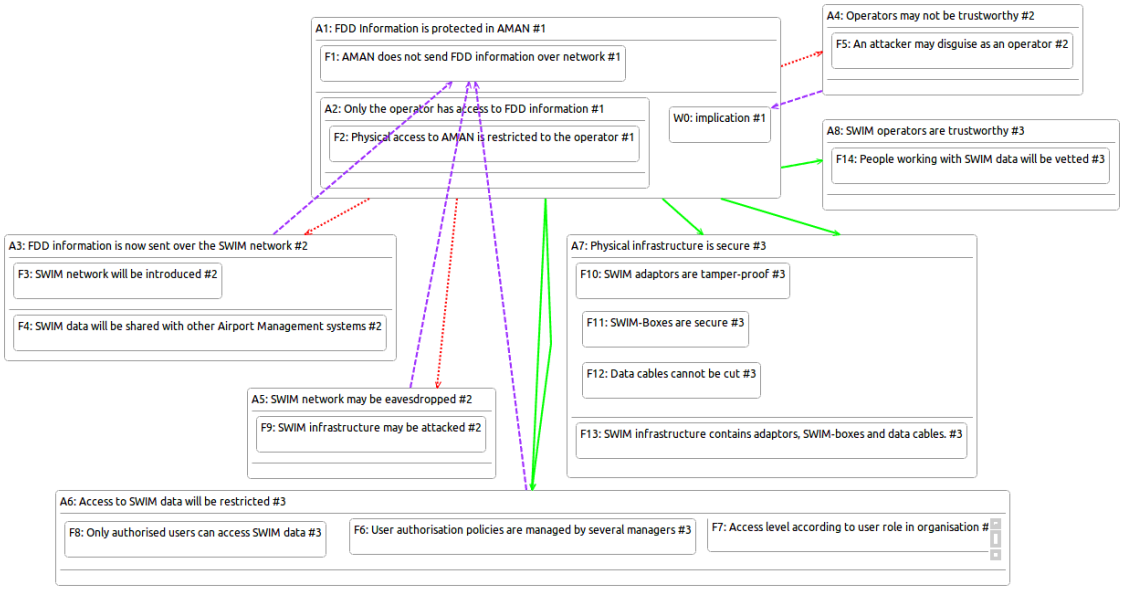


Figure 15 Argument for Security of AMAN after the mitigations

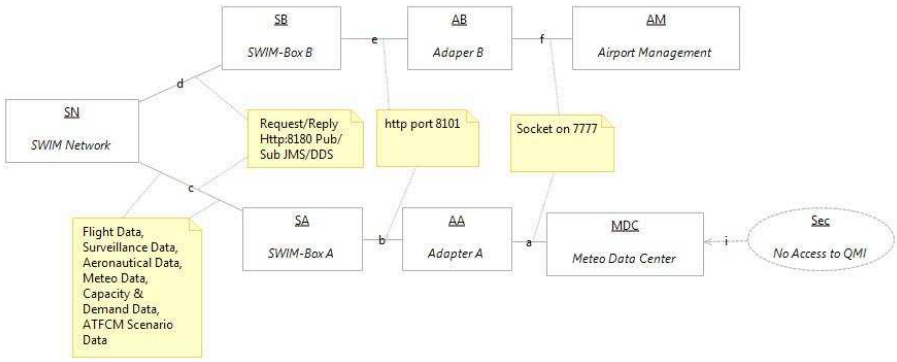


Figure 16 A SeCMER requirement model for a relevant Security Property with respect to Changes

We now illustrate how the initial argument for the system security is broken, by generating counter-examples in the Event Calculus using the abductive reasoning of OpenPF. Our discussion will focus on the protection of FDD data, and in particular the Queue Management Information (QMI), of the Airport Management system. Since we assumed that the existing system before the change is secure, the FDD data is initially protected. In the event calculus, we will write:

!HoldsAt(Accessed_FDD_data_SN(),0).

For the current system, this property can easily be proven. What is of interest is to check whether the property remains true after the change has been introduced. To do this, we input the diagram in Figure 6 into OpenPF. Figure 17 shows the textual input to create the diagram in Figure 6.

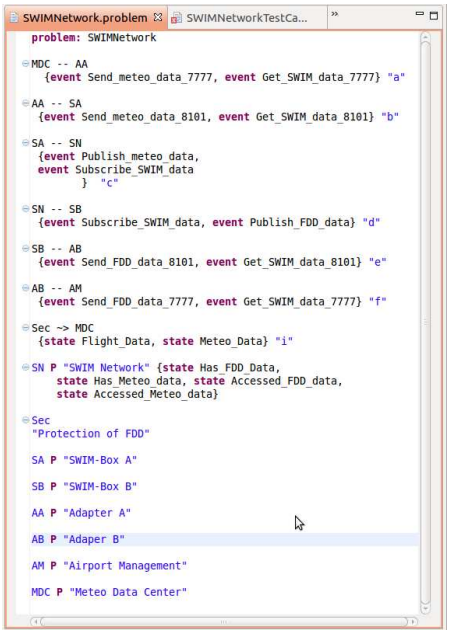


Figure 17 Textual input to create the diagram in Figure 6

In the next step, we generate the diagram shown in Figure 6. We then invoke an OpenPF plug-in that generates the Event Calculus template for the above diagram. A partial template is shown in Figure 6. This corresponds with the argument2ec step of Figure 12.

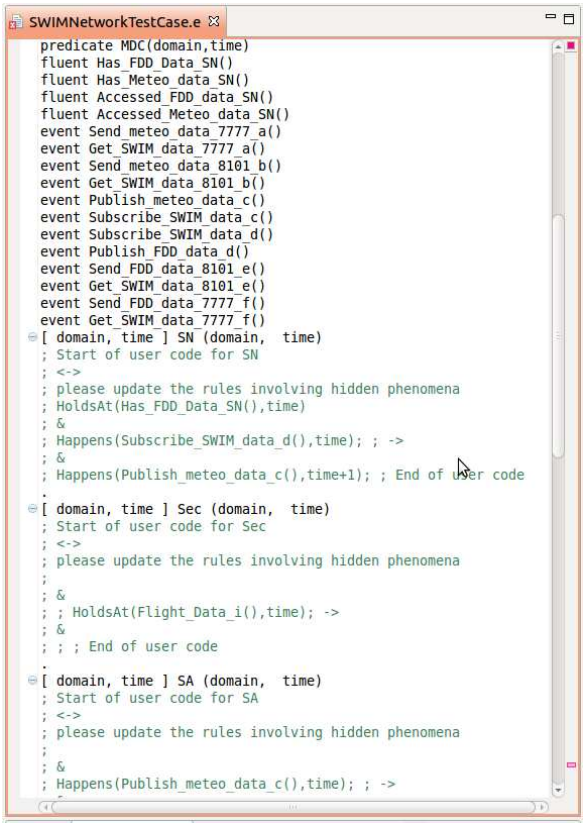


Figure 18 The Event Calculus template generated by the OpenPF tool

In the next step, we describe the behaviour of the domains. For instance, to say that the Adapter B instantly forward FDD data from the Airport Management (via interface f) to the SWIM-Box B (via interface e), we write:

[time] Happens(Send_FDD_data_8101_e(),time+1) <->
Happens(Send_FDD_data_7777_f(),time).

Similarly, to say that the SWIM-Box B instantly publishes the information to the SWIM Network (via interface d) when it receives FDD data from the Adapter B (via interface e), we write:

[time] Happens(Publish_FDD_data_d(),time) <-> Happens(Send_FDD_data_8101_e(),time).

When the FDD data is published with the SWIM Network, the SWIM Network has the FDD data.

[time] Initiates(Publish_FDD_data_d(), Has_FDD_Data_SN(),time).

If SWIM-Box A has subscribed to the SWIM Network, and if the SWIM Network has the SWIM data when SWIM-Box A attempts to get it, then the FDD data has been accessed. This is described by the following rule.

[time,time1] Happens(Subscribe_SWIM_data_c(),time1) & (time1 < time) &
HoldsAt(Has_FDD_Data_SN(),time) ->
Initiates(Get_SWIM_data_c(), Accessed_FDD_data_SN(),time).

Requests by Airport Management and Meteo Data Center for FDD data and Meteo data can be described in the same way.

In the next step, we invoke the abductive reasoned the OpenPF tool to see if the security property !HoldsAt(Accessed_FDD_data_SN(),0) has been broken. The reasoner returns the two models shown in Figure 19. The first model says that the security property Accessed_FDD_data_SN() will become true, i.e. the security is broken, if the Airport Management publishes FDD data to the SWIM Network to which the Meteo Data Center has subscribed for FDD data. The FDD data available to the Meteo Data Center may be outdate because the Airport Management has published more FDD data since the Meteo Data Center has requested it. The second model is similar to the first: the difference being that the FDD data is most up-to-date. This corresponds with the Abductive and Deductive Reasoning step of Figure 12.

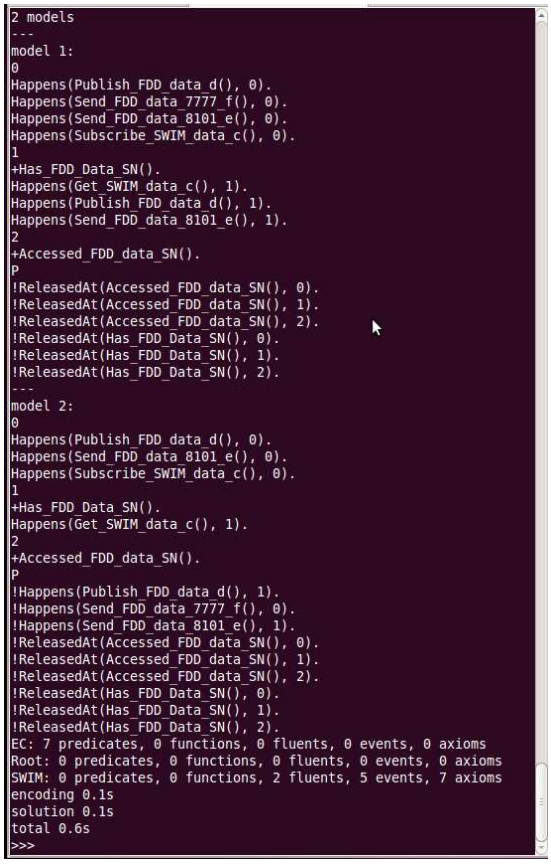


Figure 19 Results of the abductive reasoning on the change

4 Process Automation by Evolution Rules

The SeCMER approach prominently features an automated step, as highlighted in the following figure.

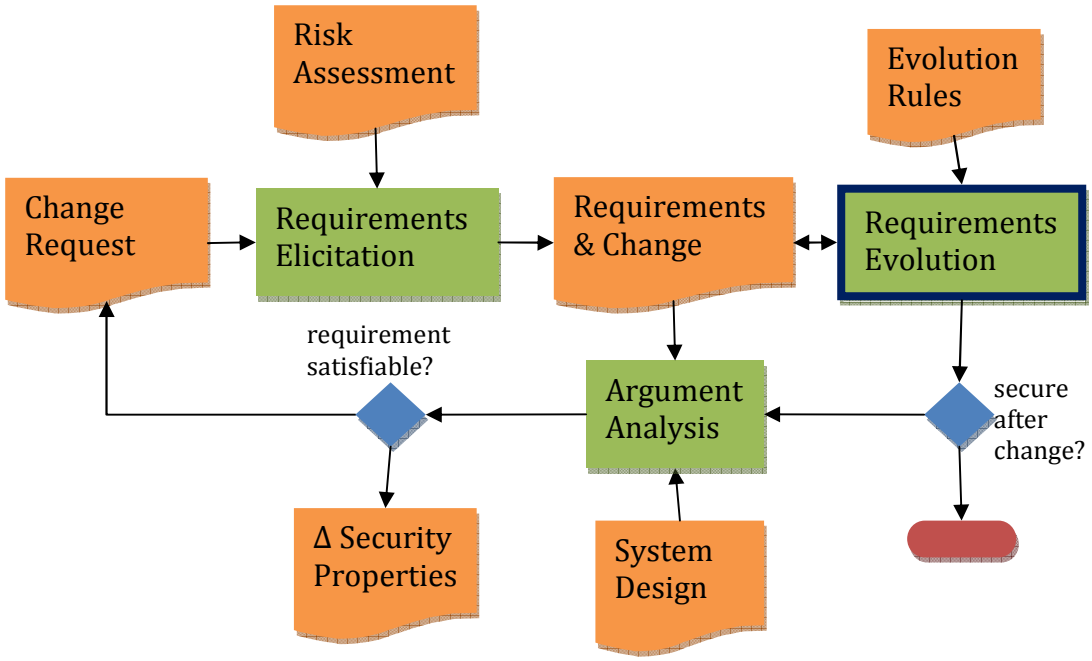


Figure 20 An Overview of SeCMER (Requirements Evolution)

This work phase is carried out automatically by monitoring the existing requirements model (as well as other interconnected models) and reacting to applied changes. Declarative change management artifacts called Evolution Rules define when to intervene, and optionally how to react. With carefully specified evolution rules, the automated rule application can save significant manual effort, e.g. in the argumentation phase.

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

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

4.1 Goals for the evolution rules

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

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

The task of constraint evaluation is not specific to requirements or security engineering, only to the actual conceptual models. Therefore it can be considered out of scope for SecureChange, and will not be discussed here in detail. Results of this approach are shown in [23].

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

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

- By operating over an interconnected requirement model and argumentation model, evolution rules can identify cases when a change in the model influences an evidence in support of a previous argumentation activity, and consequently flag the argument for manual re-evaluation
- Efficient identification of security goals whose satisfaction is implied by the model. Raise alerts (e.g. towards the argumentation staff) if a previously satisfied goal becomes unsatisfied (more precisely, if the satisfaction not provable anymore) due to changes in the model. Cases where the satisfaction of a rule can be determined automatically include the following:
    - There is already a valid (not flagged) argument, constructed in a previous argumentation session that decisively supports the satisfaction of the goal.
    - The goal is decomposed (AND/OR) into subgoals, and its satisfaction is implied by the satisfaction of subgoals.
    - In some cases, model entities connected in a certain way may automatically imply the satisfaction of the goal. For example, if the goal is delegated to an actor, who carries out an action that fulfills the goal, and there is no corresponding attacker with an anti-goal, than the goal can be considered satisfied without manual argumentation. Some of

these rules are expected to be domain-specific (e.g. ATM-only) and to emerge from the argumentation process by carefully scrutinized inductive optimization and rule formalization.

    - Similarly, it can be determined by given (possibly domain-specific) conditions that artifacts in other models (through traceability relations) automatically guarantee the satisfaction of the goals.

- Automatically making decisions and deterministic changes to the requirements model, or instantiating several options (i.e. draft solutions) and offering them to the requirement engineers, if and when such automation is applicable. Once again, such rules are expected to be domain-specific (e.g. for ATM) and to emerge from the argumentation process by carefully scrutinized inductive optimization and rule formalization.

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

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

- support complex structural requirements that are difficult and error-prone to oversee manually;
- allow the capturing of change events in terms of similarly complex structural relations, thereby treating change as a first-class citizen;
- provide automated alerting of criteria that cease to be satisfied;
- allow flexible adaptation to domains, e.g. ATM;
- enable the flexible, scenario-specific definition of the aforementioned complex criteria;
- enable the engineer to define automated reactions to change events where applicable;
- enable the reactions for automatic reconfiguration of the design model; automatic application of security-related design decisions; and automatic reusing of design artifacts (e.g. argumentations), to be filled later by the engineers, that are required for a system evolution to be admissible from a security viewpoint.

4.2 Application of evolution rules in SeCMER

There are two processes where Evolution Rules play a role. The reason for the existence of Evolution Rules is to exert their influence during the “Requirements Evolution” process in the maintenance phase, and there is also a separate process for *defining* evolution rules.

The requirements model and other related models (altogether Integrated Model) may experience an evolution that moves them out of a consistent, secure state. Automatic detection of the undesired nature of change and the potentially automated reaction is

the prime benefit of using evolution rules. This change detection mechanism applies regardless whether the change is merely a simulation of an anticipated future change, or actually initiated by a stakeholder request, or applied as a reaction to a previous change, or caused by external circumstances and merely observed.

Defining evolution rules is driven by the anticipation of such changes. The most important step is formalizing the conditions and events under which a reaction is necessary, using the provided language for Evolution Rules. The formal definition enables the automatic mechanism to detect these changes. In case solution templates can be readily identified in advance, these can also be attached to Evolution Rules as reactions. Some evolution rules will be indentified at the start of the project lifecycle, adapted to the domain and modeling style; others will be established later on the go. The latter case is embedded in or triggered by security engineering processes. We believe argumentation is the most likely subprocess where new evolution rules will be introduced, to reduce future human effort.

A typical example of the proposed workflow would happen the following way. We investigate the possibility of a future external change by simulating it in the model. The change directly results in a model state that is inconsistent with security constraints, as determined by the argumentation process (possibly communicating with risk analysis and architectural modeling processes). The output of the process will be a solution to this specific kind of change, and optionally pre-emptive modifications to brace the system for the effects of the anticipated change. Additionally, it is determined that the decision can be generalized to a range of similar potential changes that are structurally similar and cause similar security concerns. After carefully analyzing all conditions, this class of changes are formally captured by an Evolution Rule. If the resolution in these cases cannot be automated, then the only reaction rule will trigger is to alert the engineers and prompt them to perform analysis; otherwise solution templates can also be created for the rule. Finally, the new Evolution Rule is deployed, and from that point onwards, it will contribute to monitoring and reacting to changes, be they experiments, changes in external factors, stakeholder requests or themselves reactions to preceding changes.

4.3 Underlying model transformation technology

The language and efficient implementation of evolution rules relies on technology pioneered for automated model transformations. As revealed in many surveys and papers during the recent years [7, 8, 12], model transformation (MT) languages and tools play an important role in modern model-driven system engineering in order to query, derive and manipulate large, industrial models.

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

Model synchronization tasks can be formulated as the obligation to keep a model of a source language and a model of a target language consistently synchronized while the underlying source model (and sometimes the target also) is evolving. Model synchronization is frequently captured by transformation rules [3]. When the

transformation is executed, traceability links are also generated to establish logical correspondence between source and target models.

Traditionally, model transformation tools support the batch execution of transformation rules, which means that input is always processed “as a whole”, and output is always regenerated completely. However, in case of large, complex, and continuously evolving models, batch transformations may not be feasible. To address the issue of model evolution, incremental model transformations (i) update existing target models based on changes in the source models [23], and (ii) minimize the parts of the source model that need to be reexamined by a transformation when the source model is changed [5]. In the terminology of [8], these aspects are called *target* and *source incrementality*, respectively.

Since rules are defined in terms of patterns and actions, *pattern matching* plays a key role in the execution of model transformations. The goal of pattern matching is to find the occurrences of a pattern, which imposes structural as well as type constraints on model elements. Source incrementality can be achieved by employing *incremental pattern matching* techniques; for example, the RETE [10] incremental algorithm was used in [5].

The central idea of incremental pattern matching is that occurrences of a pattern are readily available at any time, and they are incrementally updated whenever changes are made. As pattern occurrences are stored, they can be retrieved in constant time – excluding the linear cost induced by the size of the result set itself –, making pattern matching a very efficient process. Benchmarks [4] and practice have shown that incremental pattern matching can improve performance or scalability by up to several orders of magnitude in certain scenarios.

Based on source incrementality, it is also possible to detect the appearance and disappearance of pattern matches efficiently. Ráth et al [23] introduced a live transformation approach where a model change is captured by a change in the match set of a graph pattern, and transformation rules are triggered by such events.

4.4 Conceptual model for evolution rules

Evolution rules control how one model, or an interconnected set of models, follow the evolution of a source model in order to maintain security and other objectives (Figure 21) Evolution rules are defined in conformance with the Event – Condition – Action semantics [2] to specify the desired reaction to changes performed on the model.

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

The Event part of the evolution rule is matched against every change executed on the model. The Condition may restrict the cases where the rule is applicable, and may select multiple ways to apply it. The Action part manipulates the model by issuing

change commands itself; these changes will eventually be processed like any other change operation, and reacted upon by evolution rules.

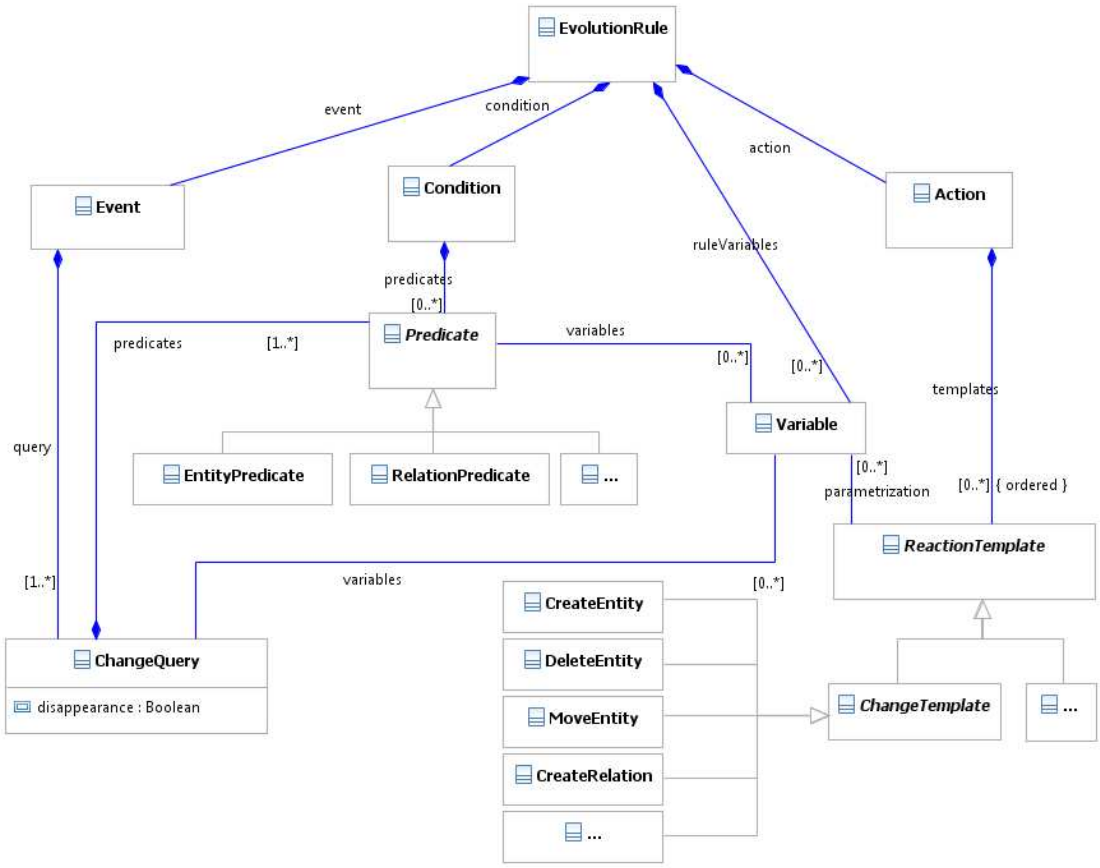


Figure 21 Conceptual model for evolution rules

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

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

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

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

4.5 Mathematical foundations

The notion of Evolution Rules has precise mathematical underpinning based on the theory of graph transformation. For purposes of formalization, we represent the requirement model and other associated models such as design as (attributed) graphs. Whether and when a rule is applicable is determined by the formalism of graph patterns in case of static models; or the more advanced graph change patterns in case changes are taken into consideration. The formal foundations presented here are a simplified version of the definitions in [6].

4.5.1 Graph Patterns

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

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

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

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

the target of the edge. To identify the variables and constraints of a specic pattern P, we use V(P) and C(P), respectively.

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

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

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

A match $m:P\rightarrow G$ is a substitution that satisfies all constraints $c\in C$ of P, which will be denoted by $G,m\models P$. [5]

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

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

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

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

Some graph pattern languages, including the one that will serve as the basis of Evolution Rules, even permit NACs to have NACs of their own. If there is no limit on the number of negations that can be nested within each other, graph patterns (without attribute constraints) become expressively equivalent to first order formulae over the predicates describing the graph model [9].

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

4.5.2 Graph Change Patterns

We define the advanced formalism of Graph Change Patterns (not to be confused with the change pattern concept of WP2) to capture how a graph model changes in an evolution. In addition to conventional graph patterns matched against the current snapshot, a change pattern should also contain constructs for expressing the difference between two graphs, in the form of change queries. An appearance query indicates a graph pattern with a new match in the post-state, while the disappearance query indicates that a match of a given graph pattern is invalidated by the change.

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

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

The intention behind our formalism is that a change pattern should match regardless of the order of elementary model manipulations that ultimately satisfied the appearance / disappearance / update queries, it is therefore irrelevant what the last operation was that e.g. completed the pattern of the appearance query. As a result, a single change pattern compactly captures a large set of different change sequences.

Definition 8 (Graph Change Pattern) Graph Change Patterns (GCP) can be defined as a tuple $GCP=\langle PN,P_+*,P_-*\rangle$, where

- $PN=\langle P,N*\rangle$ is the **main graph pattern** with the positive pattern P and negative application conditions N*.
- P_+* is a set of graph patterns $\{P_i = \langle V_i,C_i\rangle\}$ called **appearance queries**. , Each appearance query $P_i=\langle V_i,C_i\rangle$ represents that a certain graph pattern appears due to the change. P_i is allowed to share variables with P.
- P_-* is a set of graph patterns $\{P_j = \langle V_j,C_j\rangle\}$ called **disappearance queries**. , Each disappearance query $P_j=\langle V_j,C_j\rangle$ represents that a certain graph pattern disappears due to the change. P_j is allowed to share variables with P.
- Appearance and disappearance queries altogether are called **change queries**.
- The set of common variables of a change query and the main pattern is called its **interface**. , and the set of common variables is their interface. $I_i=V_i\cap V(P)$ and $I_j=V_j\cap V(P)$.

- The **pre-state pattern** $P_{pre}(CP) = (\cup_{Pj \in P\text{-}*} P_j) \cup P$ summarizes disappearance queries and the main positive pattern, i.e. all patterns representing existence in the pre-state..

- The **post-state pattern** $P_{post}(CP) = (\cup_{Pi \in P+*} P_i) \cup P$ summarizes appearance queries and the main positive pattern, i.e. all patterns representing existence in the post-state..

GCPs are matched against a pair of graphs G_{pre} and G_{post}, such that G_{post} is derived from G_{pre} by model manipulation. Thus the sets of model entities (Ent_{pre} and Ent_{post}) and relations (Rel_{pre} and Rel_{post}) may intersect on elements that were preserved by the step from G_{pre} to G_{post}..

Definition 9 (Match of Graph Change Pattern) A match of the Graph Change Pattern $GCP=\langle PN,P_+*,P\text{-}*\rangle$ in $\langle G_{pre};G_{post}\rangle$ is the mapping $m=\langle m_P,m_+*,m\text{-}*\rangle$: $GCP \rightarrow \langle G_{pre};G_{post}\rangle$, where

- m_P: $PN \rightarrow G_{post}$ is a match of PN, in the post-state G_{post}.

- For each Pi in P+* the set m+* contains a mapping m_i:$P_i \rightarrow G_{post}$ such that

  - m_i is a match of graph pattern P_i in graph G_{post},

  - $m_i(v) = m_P(v)$ for interface variables $v \in I_i$, i.e. m_i interfaces with the match of the main pattern, and

  - the same m_i is *not* a match of graph pattern P_i in the pre-state G_{pre}.

- For each Pj in P+* the set m+* contains a mapping m_j:$P_j \rightarrow G_{pre}$ such that

  - m_j is a match of graph pattern P_j in graph G_{pre},

  - $m_j(v) = m_P(v)$ for interface variables $v \in I_j$, i.e. m_j interfaces with the match of the main pattern, and

  - the same m_j is *not* a match of graph pattern P_j in the post-state G_{post}.

Note that this definition is deliberately asymmetric for G_{pre} and G_{post}, as the main pattern PN is interpreted on G_{post} only.

4.5.3 On computational complexity

Graph pattern matching can be a computationally intensive process. As it contains the well-known problem of Subgraph Isomorphism, it is NP-hard. However, in most cases the pattern will be of bounded (even small) size, while the model itself may grow big. It is easy to see that with this assumption, the worst-case time complexity is the size of the model to the power of the size of the pattern, therefore polynomial.

Still, this step may take long. The whole point of incremental pattern matching (see Section 4.2) is to avoid the time-consuming full re-computation of the match set after each small modification. Assessing the execution time and space requirements of incremental pattern matcher algorithms is a challenging task, having to take into account pattern structure, graph model structure, match set sizes of patterns and sub-patterns, the extent and consequences of change, etc. Some cost models of RETE, the data structure used in the implementation, are available at [25] and [1].

Having incremental pattern matching available, computing the *change sets* after an evolution from G_{pre} to G_{post}, i.e. the set of new pattern matches and the set of invalidated pattern matches, is a trivial task. GCPs can be thought of as a construct very similar to graph patterns on the union of the post-state (main graph pattern) and the change sets (change queries). Therefore evaluating GCPs has similar complexity characteristics as plain graph patterns.

4.5.4 Rule Formalism

Harnessing the strength of GCPs, a powerful rule-based automation formalism can be defined. Without going into details of how the reactions themselves are defined, such a rule can be characterised by a *guard* that is a GCP; after a change to the model, the actions associated with the rule are executed for each match of the guard. In publications by the authors in the field of model transformation (e.g. [6]), such a rule was referred to as Change-driven Rule (CDR).

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

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

4.6 Examples of evolution rules

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

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

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

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

4.6.1 Graph pattern for expressing the problem

Figure 22 visually depicts the graph pattern (with a negative condition) that characterizes this undesired configuration of elements.

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

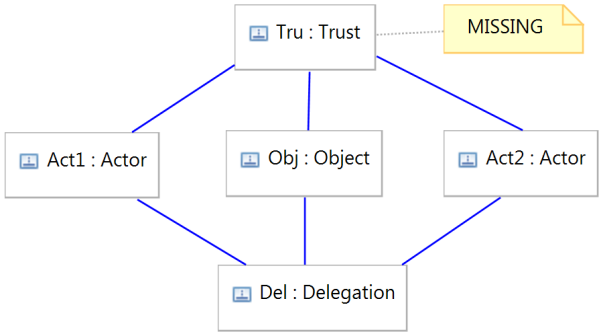


Figure 22 The undesired pattern: untrusted delegation

4.6.2 Solution 1: one rule per elementary change

The first solution would be to create several evolution rules, one for each possible elementary change that can complete the pattern and make an intervention necessary. In this case, two kinds of elementary changes can trigger the rule: the detection of a newly added “delegation” relationship between two actors (and the dependum), or the deletion of an actor-actor trust (over a dependum).

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

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

4.6.3 Solution 2: single coarse-grained rule

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

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

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

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

4.6.4 Solution 3: automatic problem correction

Apart from logging the detection of the pattern and reusing an argumentation, evolution rules can also correct problems present in the model. The difficulty of this approach is

that often there is more than one way to remedy an issue, and the decision is hard to automate. For instance, the problem in this example can be solved by adding a missing trust relationship; or by removing the delegation (and probably implementing something else in its place). Both are valid ways to handle the issue, but engineers should select manually which one should be applied in each concrete case. To achieve this, we introduce two alternate evolution rules that implement these two reactions. Together with the rule `UntrustedDelegation` of Solution 2 introduced in Section 4.6.3, they provide three options that can be automatically offered to the engineers to choose from.

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

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

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

4.6.5 Discussion

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

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

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

5 Application of the Methodology

This section illustrates the different steps of the SecMER methodology based on the process level change and the information protection property of the ATM case study. The Process Level Change is about the introduction of the Arrival Manager (AMAN), which is an aircraft arrival sequencing tool helping to manage and better organize the air traffic flow in the approach phase. The introduction of the AMAN requires new operational procedures and functions (as described in Deliverable D1.1) that are supported by a new information management system for the whole ATM, an IP based data transport network called System Wide Information Management (SWIM) that will replace the current point to point communication systems with a ground/ground data sharing network which connects all the principal actors involved in the Airports Management and the Area Control Centers. The introduction of the AMAN and the SWIM requires suitable security properties to be satisfied which prevent from corruption, accidental or intentional loss of data and guarantee the integrity and confidentiality of the aircraft sensible data against malicious attacks or intrusions.We will focus on information access and information protection properties on the requirements level. In particular, we will show how to achieve information access by enforcing access control policies on Flight Data Domain (FDD) transmission and how to ensure confidentiality of FDD data by using encryption.

5.1 Requirement Elicitation

The first step of our methodology consists of modeling the ATM system before the introduction of the AMAN and the SWIM using the SecMER conceptual model. The resulting requirement model is illustrated in Figure 23.

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

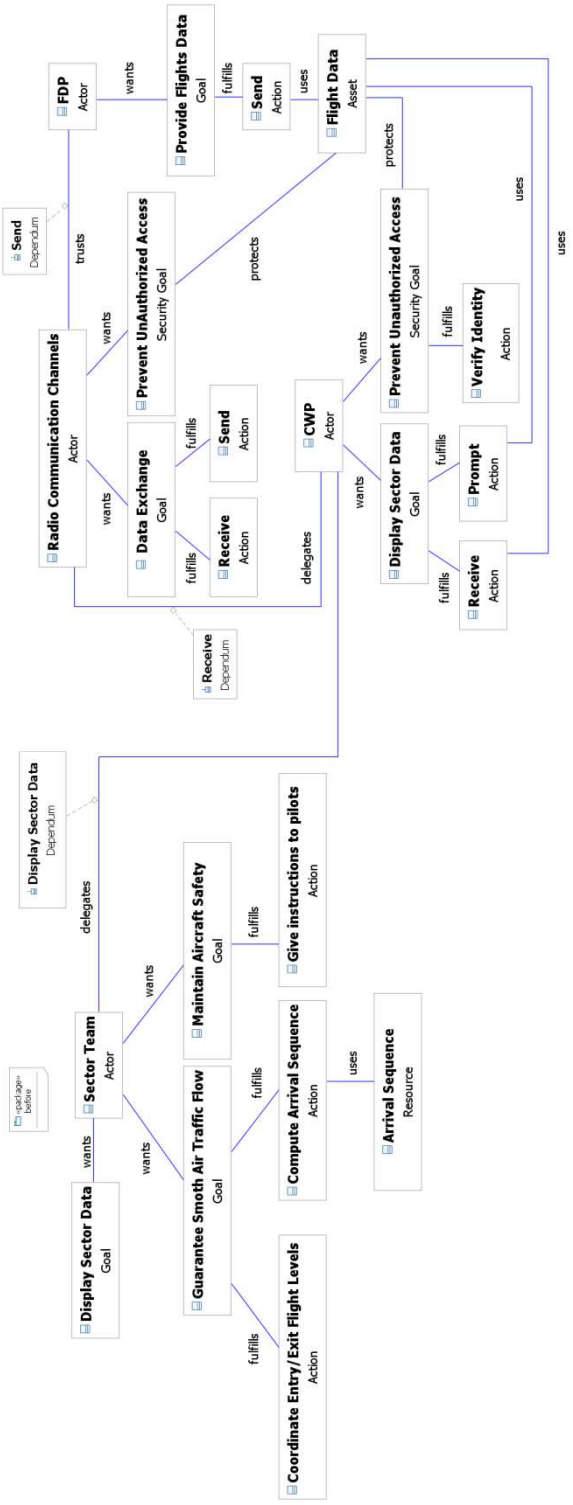


Figure 23 The “before” requirements model

5.2 Requirement Evolution

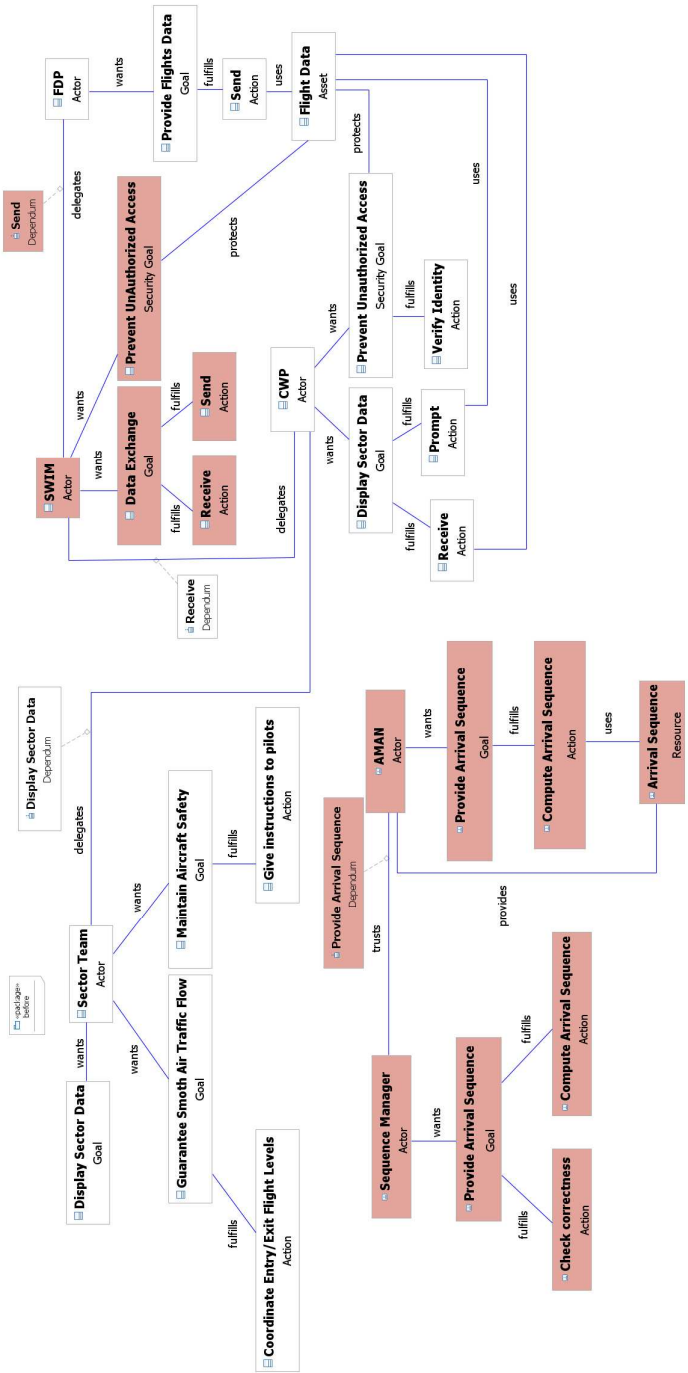


Figure 24 The “after” requirements model

The introduction of AMAN and SWIM triggers a change request that requires the requirement engineer to update the requirement model. The resulting model is illustrated in Figure 24 where the new actors and goals are outlined in red. We have three new actors the Sequence Manager, the AMAN and the SWIM. The Sequence Manager is a new type of ATCO who will monitor and modify the sequences generated by the AMAN and will provide information and updates to the Sector Team. The AMAN main goal is to provide the arrival sequence by interacting with the FDP to get aircraft positions that are necessary to compute the arrival sequence. The communication between the different ATM actors is based on the SWIM, an IP based data transport network which replaces the current point-to-point connections systems. The SWIM actor has replaced the communication lines actor and it wants the security goal of protecting the data flight information from unauthorized access (Figure 24).

5.3 Argumentation for security properties

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

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

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

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

Automation. Security goals often push the system boundary to enclose emergent facts and domain knowledge, some argumentation analysis has to be carried out interactively. On the other hand, conceptual model for argumentation makes it easier to turn the three types of modelled arguments into predicate logic formula that are checked using off-the-shelf reasoning tools [14].

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

5.4 Deriving and using Evolution Rules

In an evolving requirements model, new actors may be introduced, delegation and trust relationships may be changed, all raising security concerns. The ATM evolution case study is an example of this phenomenon: the new SWIM actor is introduced, taking over the responsibility of secure communication, but other actors such as CWP not necessarily trust it. This is exactly the problem that the step described in Section 4.6 addresses; in the following, we will demonstrate how such evolution rules can be derived and applied in the concrete ATM example.

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

To come up with the events and conditions, we first represent a part of the complete requirements model as a graph pattern. For example, when an actor delegates some responsibility (e.g. the security goal of the CWP actor to protect the data communication line from man-in-the-middle attack) to another actor (e.g., SWIM), but does not trust the latter with the same object (e.g., the data communication link). The graph pattern that characterizes the undesired configuration of elements was previously shown in Figure 22. In context of the ATM example, Act1 can be the Actor CWP, which delegates (through a delegation relation captured in variable Del) the Goal Receive (matching the variable Obj) to Actor SWIM (which will be Act2); this variable substitution is a match of the pattern as there is no trust relationship Tru between these two actors over this goal in the model.

After assembling the graph pattern, the event and condition specifications will have to be derived from it. We can create several evolution rules, one for each possible elementary change that can complete the pattern and make an intervention necessary. This will produce an outcome similar to Solution 1 presented in Section 4.6.2. Alternatively, simpler and more concise rules can be used, similar to Solution 2 from

Section 4.6.3, if the mitigation only depends on the after state, and not on the nature of the change itself.

Regardless of the chosen approach, the action part can alert the argumentation engineers, or perform automated intervention by directly manipulating the model if the mitigation is close to deterministic. See Solution 3 from 4.6.4 as an example. Some of the change patterns introduced in D2.1 can be considered as possible candidates for being automated with evolution rules.

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

5.5 Interaction of argumentation and evolution rules

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

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

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

```
        entity SecurityGoal(SecG);
    }
    condition {
        entity Argument(Arg);
        relation Argument.supports(Arg-S1→SecG);
        entity SecurityGoal(SecG);
    }
    action {
        // flag argument as potentially invalid, notify argumentation team
    }
}
```

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

Initial claim:

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

Initial facts:

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

Initial domain knowledge rule:

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

Initial Rebuttals:

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

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

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

Such argumentation can go on until all the facts and domain knowledge rules are refined so that all rebuttals of the root claim are not satisfiable. In other words, a

satisfaction claim is justified as long as all the facts and domain knowledge are true (e.g., trust assumptions in arguing security goals) and all the rebuttals are false. A formal treatment of argumentation using non-monotonic proposition logic can be found in [14]. As one can see, the result of such argumentations would inevitably contribute to changes in the situations of security goals.

6 Integration with other approaches in SecureChange

The SecureChange project is developing a methodology for engineering of secure, long-lived and evolvable systems. Methods and techniques that have been developed in individual work packages are various parts of the methodology. This section discusses how requirements methodology of WP3 integrates with process and architecture methodology of WP2, the design methodology of WP4, and risk assessment methodology of WP5.

The integration is shown both at a conceptual level and at a process level. The conceptual level integration shows how concepts from the requirements methodology relate to those in other methodologies. At the process level the integration shows how the flow of the process across the boundary of work packages. We illustrate and exemplify the integrated the integration based on the Organizational Level Change Requirement in the ATM case study and the Specification Evolution Change Requirement of the POPS case study.

6.1 Integration of Requirements Engineering with the Overall Process and Architecture

In this section, we describe the integration of SeCMER with the Overall Process model, described in WP2, through the artefacts and process. The purpose of the Overall Process model is to describe the abstract development process that can be instantiated by various specific methodologies of the SecureChange project, including the requirements engineering methodology. The purpose of requirements engineering methodology is to describe and analyze the requirements for change, and specify the security properties that need to be implemented in order to effect the change. Therefore, the general relationship between the Overall Process model and the requirements engineering methodology is that of abstract and concrete.

6.1.1 Artefact Integration

The SecureChange report D.2.2 has presented the integrated meta model (Figure Figure 25) showing the artefacts of the SecureChange project and their interdependencies. The artefacts are:

- Integrated Model
  Integrated Model is an aggregated class comprising all other security engineering artefacts. Instances of the class describe the system at all levels of abstraction at a certain point of time. In Integrated Model may be in realised state or planned state.
- System Model
  The System Model comprises all information relevant in WP4 and WP6. In

particular the System Model both comprises the software architecture layer and the code layer including security related information (e.g. formal models of security protocols).

- Risk Model
  The Risk Model comprises information related to WP5
- Requirements Model
  The Requirements Model contains information related to this work package
- Test Model
  The Test Model contains information related to WP7.
- ChangeRequest
  The ChangeRequest class represents a complex change transaction of the system at any level of abstraction, e.g. triggered by modified risks, requirements or components in the System Model. Each change request is associated with a an Integrated Model (=Integrated Meta Model instance) describing the state of the system when initiating the change request (pre), and with arbitrary many Integrated Model instances describing possible states after the change request has been closed (post).
  If the change request is in state closed, exactly one associated Integrated Model instance has to be in state realised. The possible states of a change request are described in detail in D2.2. Links between a change request and its post Integrated Model may be attached with information how the post Model has been constructed based on the pre Model. Two categories are supported and explored within SecureChange: Change Patterns (WP2) and Change Rules (WP3). Various concepts associated with the notion of change are discussed in Section 7.

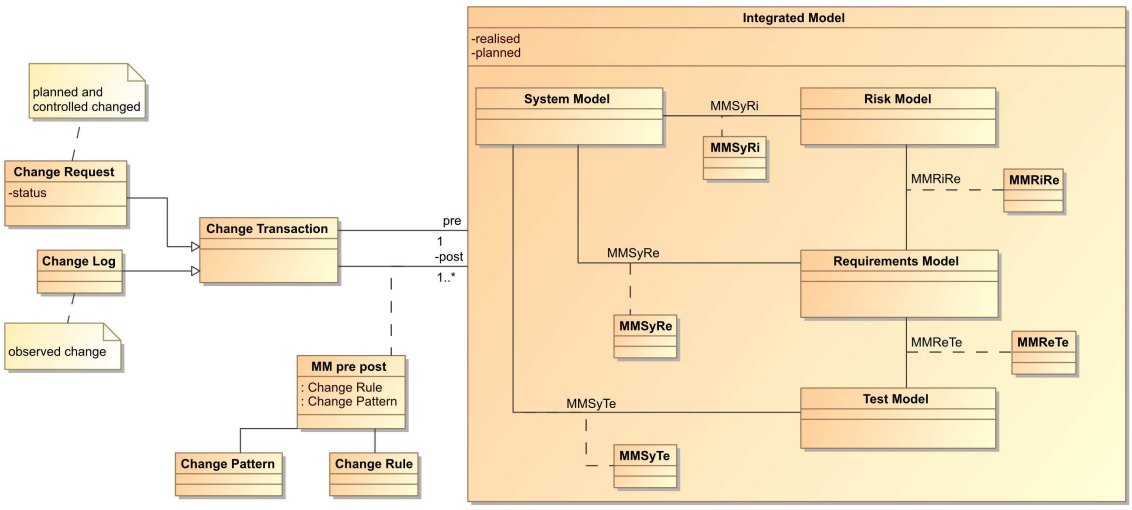


Figure 25 Integrated Meta Model

The SeCMER conceptual model of the evolving requirements discussed in Section 2.1 (reproduced in Figure 26) is a specialisation of the Requirement Model package of the the Integrated Meta Model of D2.2, shown in Figure 25.

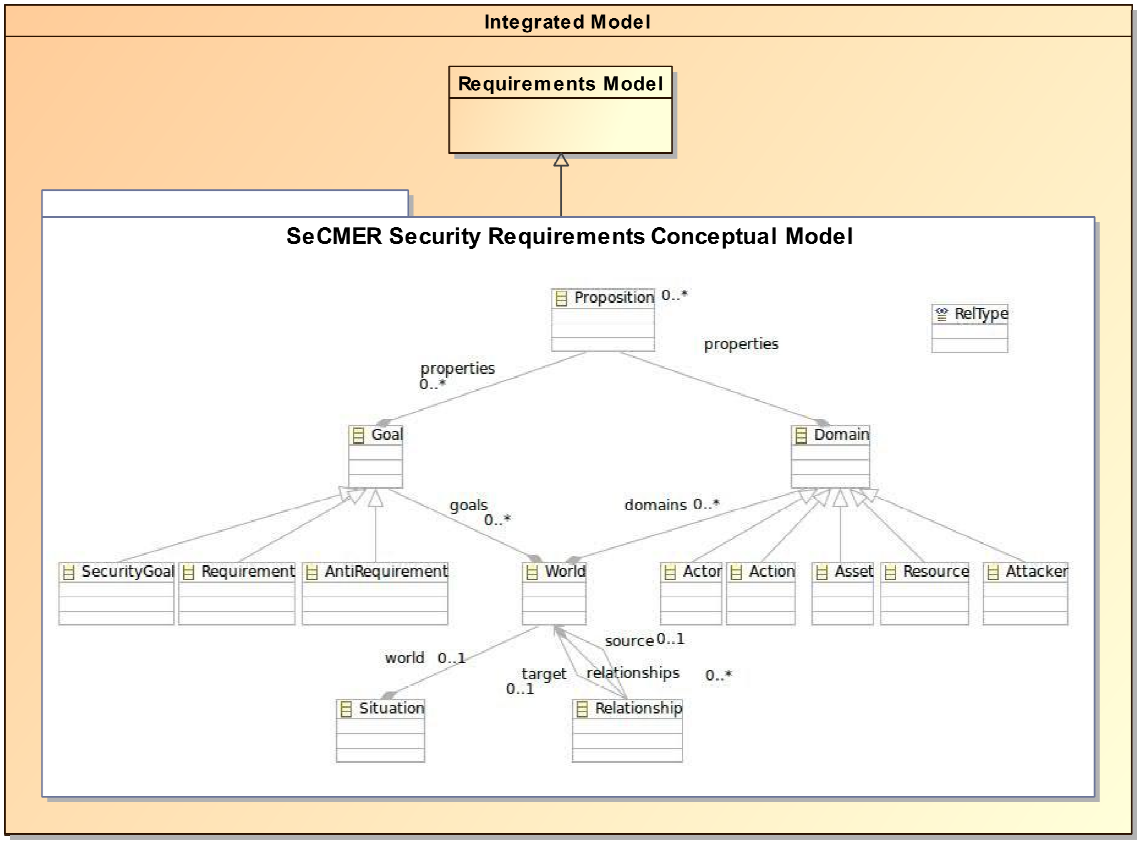


Figure 26 Security Requirements Conceptual Model (in relation to Integrated Meta Model)

In particular, the SeCMER conceptual model of security requirements defines a requirement model in terms of concepts described in Figure 26. A detailed discussion of those concepts is given in Section 2.

The Integrated Meta Model shows that there are three links from the Requirements Model: to the Risk Model, System Model and the Test Model through the links MMSyRe, MMRiRe and MMReTe. Sections 6.2, 6.3 and 6.4 below concretize those links.

6.1.2 Process Integration

According to the Integrated SecureChange Process presented in D2.2., the evolution of the system is determined by a sequence of change requests (for simplicity overlapping change requests are not considered at the current stage). Each change request causes one or several change events. These change events are handled by the state machines of the model elements of the Integrated Model causing state transitions and further events.

Before discussing the state models, Figure 26 shows a sample change story. The events in this sequence diagram are determined by an initial triggering change event (change of the Risk Model) and subsequently by the state machines.

The change story exemplifies a change in the Risk Model (according to a user´s change request) and subsequent actions, starting with a check of the risks, the propagation of changes to the Requirements Model, the System Model and the Test Model, a final check of the risks and propagation to the Requirements Model.

As a general remark change events in the Integrated Process are differentiated in external change events (change) and internally propagated events (propagate), e.g. the external change of the Risk Model (change) and the propagation of this change to the Requirements Model (propagate). The handling of propagation should make use of the respective mapping model in order to determine the affected part of the target model of the change propagation.

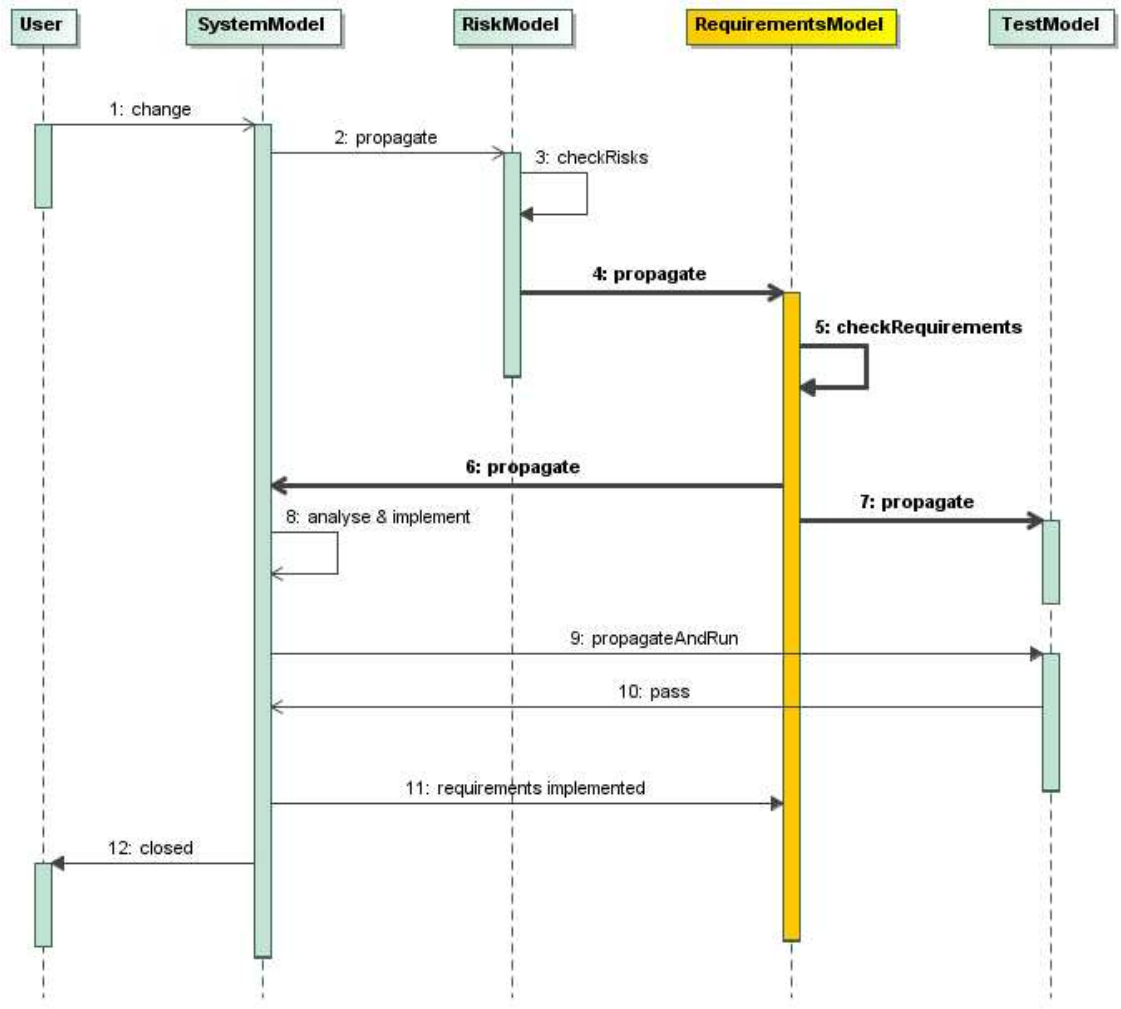


Figure 27 Sample Change Story

The SeCMER methodology, recalled in Figure 28, is an instantiation of the messages 4:propagate and 5:checkRequirements in the change story. 4:propagate is the trigger

for the SeCMER methodology, and the processes Requirements Elicitation, Argument Analysis and Requirement Evolution instantiate 5:checkRequirements. Results from SeCMER can be propagated back to the SystemModel (6:propagate) through the System Design artifact in SeCMER methodology. Similarly, the results from SeCMER can also be propagated to the TestModel (7:propagate), through the incremental security properties that need to be checked in SeCMER. When the requirements have been implemented in SystemModel, the system is in the secure state, and the SeCMER methodology is terminated.

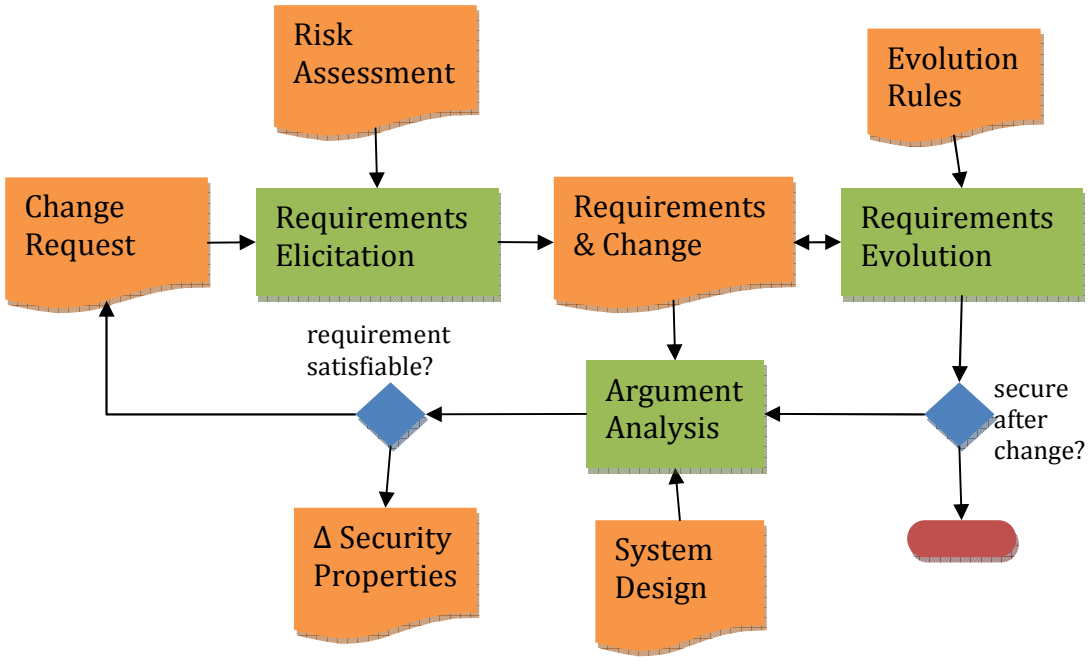


Figure 28 Overview of SeCMER methodology

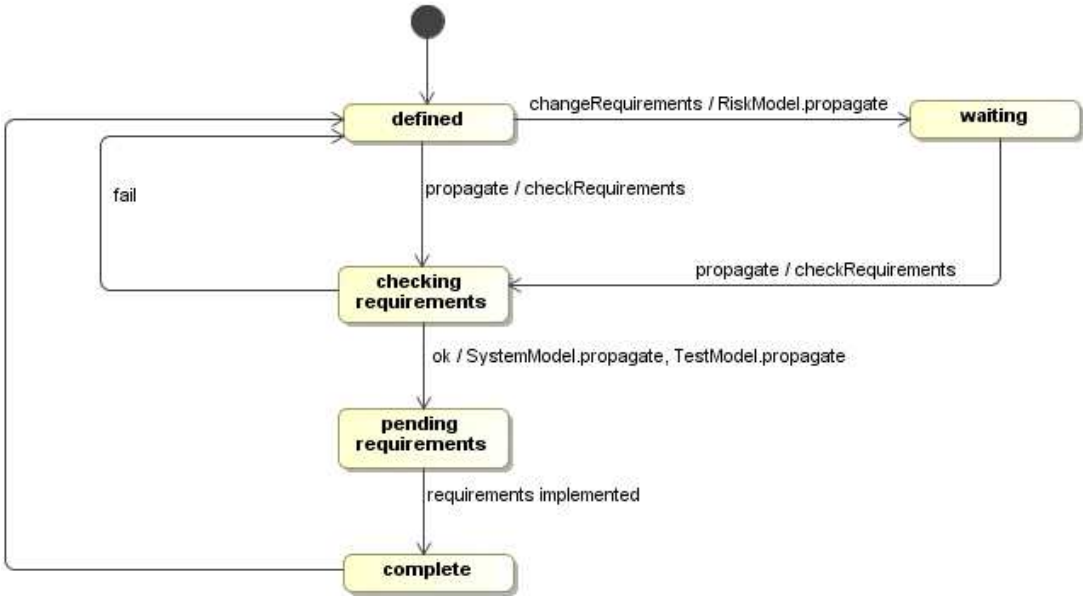


Figure 29 State Diagram of Requirements Model

In the report D2.2, state diagram of the Requirements Model has been defined (recalled in Figure 29). the various states of a requirements model are explained:

- defined: This state reflects that a requirements model has either been newly created or that it is subject to a change which can either be external (changeRequirements) or internal (propagate).
- waiting: This state reflects that the requirements analysis is currently on hold to wait for the results of the risk analysis. That way we ensure that each change to the requirements is always undergoing a risk analysis first.
- checking requirements: This state reflects that if the risk analysis is concluded or changes are propagated internally to the requirements model the requirements model is checked and analyzed. The result of checkRequirements can either be a failure in which case the state is changed back to defined of if it is ok, the changes are further propagated to the system and test model.
- pending requirements: The requirements model remains in this state until the implementation is concluded and the system model fires the trigger requirements implemented.
- complete: This is the target state of the requirements model. It outlines that the requirements have been implemented.

Again, the development process is in the checking requirements state when the SeCMER process is being executed, in the pending requirements state when the changes specified in the incremental security properties are being implemented, and in the complete state when the SeCMER methodology is exited.

6.2 Integration of Requirements Engineering and Risk Assessment

The SecureChange process aims at developing an overall approach to the engineering of secure, long-lived and evolvable systems. Methods and techniques for requirements engineering is one of the cornerstones from the overall approach. In the wider setting of security engineering of changing and evolving systems, the requirements engineering constitutes one part of the overall process.

In this section, we describe the integration between the SeCMER methodology and risk assessment methodology proposed in WP5. The purpose of the risk assessment is to understand the potential security risks that may arise, possibly due to some requirement changes, and to identify treatments for unacceptable risks so as to ensure and maintain an acceptable level of security. The results of a risk assessment (i.e., new treatments) may yield new security requirements that should be included in the requirement model. Moreover, the requirements changes may involve new assets that should be taken into account their risk levels. Therefore, a well-defined process that integrates the two methodologies is required to fully and properly understand both the requirements and the risks.

Moreover, in the setting of changing and evolving systems, there is a need to understand not only how the changes may affect security requirements on the one hand and security risks on the other hand; we also need to understand how changes to requirements and risks affect each other, and how the propagation of changes from the one to the other should be dealt with in a systematic way.

We address the problem both at a conceptual level and at a process level. At the conceptual level we present an integration of concepts and explain how requirement model artifacts should be mapped to risk model artifacts and vice versa. At the process level, we utilize the conceptual level integration and explain how the conceptual integration comes into play in the integration of the respective methodologies.

We illustrate and exemplify the integrated process based on the Organizational Level Change Requirement in the ATM case study. Notice that the risk assessment examples are based on the CORAS instantiation of the method for risk assessment of changing systems as presented in the appendix of deliverable D5.3.

6.2.1 Conceptual Integration

As standalone methods, both requirements engineering and risk assessment may adopt techniques, artifacts and concepts from each other's domain. Requirements analyses may, for example, take into account threats and vulnerabilities, and risk assessments may include requirements identification as a separate task. For the purpose of understanding and describing the potentials for integration of the separate methods, we assume a separation of concern. This means that all risk specific concepts belong to the risk domain, and all requirement specific concepts belong to the requirement domain. The separation of concern assumption ensures that we can identify exactly the actual interface between the domains.

In the following we first separately present the core and basic concepts of requirements engineering and risk assessment. Thereafter we present the conceptual level integration.

6.2.1.1 Requirement Concepts

The UML class diagram of Figure 30 gives an overview of the basic concepts of requirements engineering and the relations between them. We refer to Section 2.1 for a detailed presentation of the conceptual model. In the following we give the definitions of the concepts.

- **Action:** An entity performed by an actor, which can generate events, and can have preconditions and post-conditions
- **Actor:** An entity that can act and intend to want or desire
- **Asset:** an entity of value that can be owned and used
- **Goal:** A proposition an actor wants to make true
- **Proposition:** A statement that can be true or false
- **Resource:** an entity without intention or behavior

- **Security goal:** A proposition that specifies how to prevent harm to an asset through the violation of confidentiality, integrity, and availability security properties
- **Situation:** partial state of the world described by a proposition

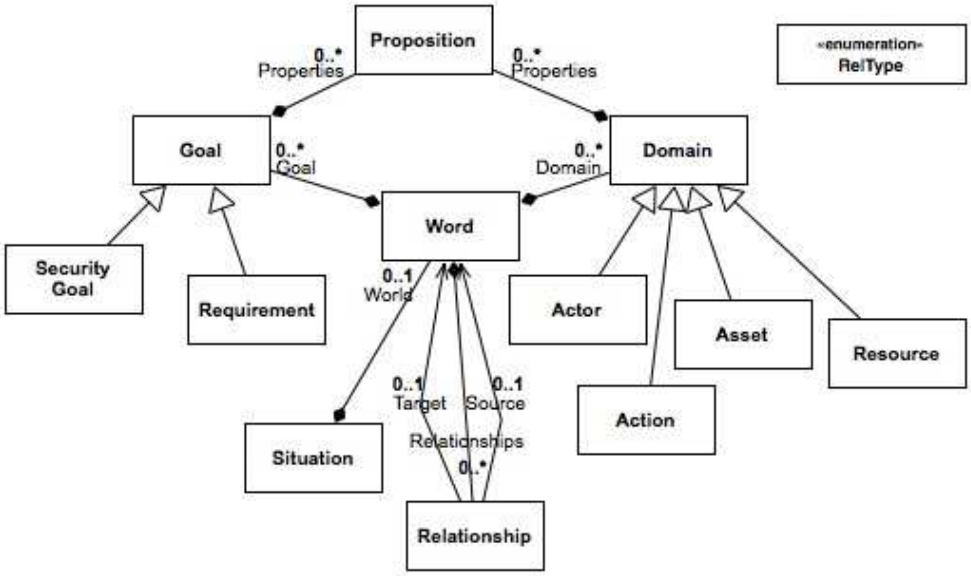


Figure 30 Basic requirements concepts

6.2.1.2 Risk Concepts

The UML class diagram of Figure 31 gives an overview of the basic concepts of risk assessment and the relations between them. We refer to deliverable D5.3 for a more detailed presentation of the underlying concepts of risk assessment. In the following we give the definitions of the concepts.

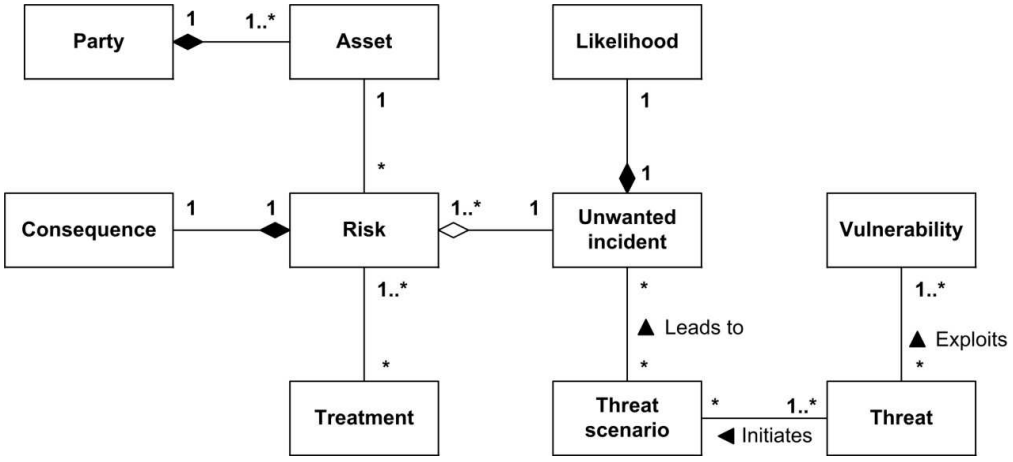


Figure 31 Basic risk concepts

- **Asset:** Something to which a party assigns value and hence for which the party requires protection.
- **Consequence:** The impact of an unwanted incident on an asset in terms of harm or reduced asset value.
- **Likelihood:** The frequency or probability of something to occur.
- **Party:** Stakeholder; an organization, company, person, group or other body on whose behalf a risk analysis is conducted.
- **Risk:** The likelihood of an unwanted incident and its consequence for a specific asset.
- **Threat:** A potential cause of an unwanted incident.
- **Threat scenario:** A chain or series of events that is initiated by a threat and that may lead to an unwanted incident.
- **Treatment:** An appropriate measure to reduce risk level.
- **Unwanted incident:** An event that harms or reduces the value of an asset.
- **Vulnerability:** A weakness, flaw or deficiency that opens for, or may be exploited by, a threat to cause harm to or reduce the value of an asset.

6.2.1.3 Integration

In the conceptual integration we distinguish between what we refer to as *shared elements* on the one hand and *mappable elements* on the other hand. The shared elements are concepts that are common to requirements engineering and risk assessment, with the same semantics in both domains. The mappable elements are concepts from one domain that are not shared by the other, but are nevertheless related to the other domain and can be mapped to concepts of the other domain.

The shared elements are at the core of the integration, and the fact that they are shared means that if a change occurs such that a shared element is affected in one domain, it will inevitably affect the same element in the other domain. Considering the conceptual framework of the two domains, we identify one shared element, namely the **asset**. In both domains, an asset is something of value for a stakeholder. A main objective of requirements engineering is to elicit requirements where they are assured to be achievable securely at runtime, while a main objective of risk assessment is to identify risks with respect to the assets and to identify treatment options for mitigating excessive risks.

The overall objective of the two domains is protecting assets, by maintaining an acceptable level of security/risk to artifacts relevant to some requirements, requires us to identify mappable elements. In the risk assessment domain, a treatment is a process that will reduce the level of risks. In the requirement engineering domain, this corresponds to security goal and action; a security goal specifies what to prevent from the assets, and an action specifies how to fulfill the security goal. An important principle of the conceptual integration is not all concepts in each domain are shared or mappable. Therefore, changes on the element that is not part of the conceptual integration shall have no effect on the other domain.

There are, however, certain artifacts that nevertheless may serve as additional information to the other domain in the sense of explaining or providing the rationale for other elements. Changes to such artifacts in one domain should not have effect on the other domain, other than the effects that are captured via the shared and mappable elements.

In the integration between requirements engineering and risk assessment, we include the achievement matrix and the risk matrix, respectively, as such explanatory elements. The achievement matrix is produced as a result of a requirements analysis, and specifies the continuity and the achievement level for each requirement. The risk matrix is used both in the input to and in the output from a risk assessment. As input, the risk matrix is used to define the risk evaluation criteria for each asset and the specification of acceptable risks. As output, the risk matrix is used to evaluate the identified risks and to document the changes of risk levels due to changes to the target system, including the implementation of treatments. Note that acceptable risks do not always lead to a "sufficient" level of the achievement matrix in the requirement model. For instance, there might be a situation where all risks are treated, but some requirements cannot be fulfilled because the organization lacks of required capabilities.

An overview of the conceptual integration is given in Table 3.

Requirement concept	Risk concept	Kind of integration
Asset	Asset	Shared concept
Security goal	Treatment	Risk concept mapped to requirement concepts
Action		
Achievement matrix	Risk matrix	Risk concept mapped to requirement

Table 3 Conceptual integration of requirement and risk modeling

6.2.2 Integrated Process

In this section we present various options for integrating the respective processes of requirements engineering and risk assessment. The two processes should still be understood as separate processes with their own iterations, activities and techniques for managing change. The integrated process explains at which steps of the respective processes that the conceptual level interface can or should be invoked. The integrated process is hence based on the conceptual integration presented in the previous section.

The integrated process presented here can be understood as an instantiation of the more general and project wide integration presented in deliverable D2.2. In this section we present a more detailed integration, however maintaining the consistency with the more high-level integration of D2.2.

The integration of the requirements engineering process and the risk assessment process is based on the principle that the respective domains are oblivious to the elements of the other domain that are not part of the conceptual integration. This

means that whenever data is processed, model elements are modified, analyses are conducted, etc. in one domain, and these activities do not affect the elements of the conceptual integration, there is no need to interact with the other domain.

For example, if the requirement engineers keep modifying and remolding satisfaction arguments for achieving functional goals while there are no changes to the assets, there is no need to invoke the risk assessment process. And if the risk analysts modify the risk models by including new threats and adjusting likelihood estimates, and these modifications have no effect on the treatments, there is no need to invoke the requirement engineering process.

6.2.2.1 Overview of Process

The UML activity diagram of Figure 32 gives a high-level overview of the integrated process. The diagram is divided into three partitions to distinguish between the activities and objects under the control of the user, the risk analyst and the requirement engineer. The user is typically the client commissioning the analyses, and may, for example, be the owner of the system that is the target of analysis.

The integrated process is defined such that the risk assessment process and the requirements engineering process are conducted separately. Depending on the change request and the analysis needs, the respective processes may optionally invoke each other at different stages of the overall process. In the diagram, the diamonds specifies branching of the sequence of activities. When there is no guard condition on the branching (specified by the notes with Boolean expressions), the process proceed along one or both of the branches. This gives a wide flexibility on how the overall process may be conducted. Some of the potential scenarios that are described by the diagram are the following:

- A change transaction is planned, and the user makes a change request that is passed to the risk analyst who is asked to update a previous risk assessment. The risk analyst uses the previous risk model (RiM before) and the change request as input to the risk assessment. As a result of the risk assessment, the risk analyst passes treatment options for unacceptable risks back to the user, and the process ends.
- A change transaction is planned, and the user makes a change request that is passed to the requirement engineer who is asked to update a previous requirements analysis. The requirement engineer uses the previous requirement model (ReM before) and the change request as input, and updates the ReM. Based on the updated ReM, a requirements analysis is conducted. The results are passed back to the user, and the process ends.
- The scenario is initiated as one of the previous ones, but during the process the risk analyst and the requirement engineer interact by invoking each other without going through the user. For example, the requirement analysts identify new assets that are passed to risk assessment, and the risk analysts report back by passing relevant treatments to the requirement engineer who updates the ReM accordingly before the requirements analysis is conducted.

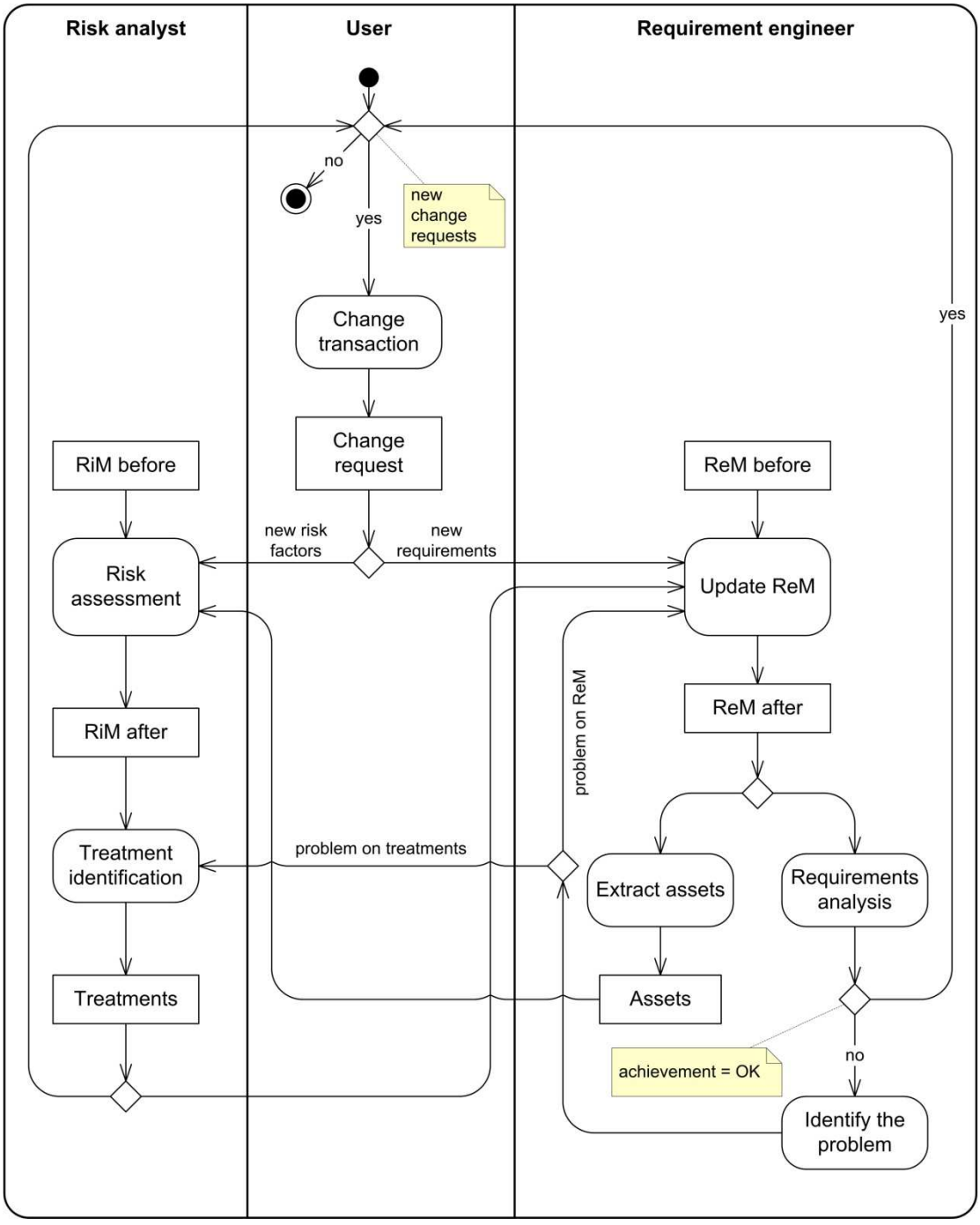


Figure 32 Overview of integrated process

We can think of the integrated process as interactions that are triggered by an external event, namely the change request from the user. When this change request is passed to one or both of the risk analyst and requirement engineer, there are a number iterations where the changes propagate back and forth between the two until a stable state (equilibrium) is reached and the results can be passed back to the user.

As seen from the diagram, it is the user that initiates and terminates the overall activity. For long-lived, evolving systems there may be many iterations of the overall integrated process, each of them triggered at different points in time. In the following we focus on a single iteration of the overall process (which of course may consist of several internal iterations without the user), explaining in more detail the integrated process.

In the description of the integrated process, we assume that the risk analyst and the requirement engineer share a common representation and description of the target of analysis before the changes. Such a description may, for example, be a set of UML diagrams as exemplified by the documentation of the ATM target of analysis in the appendix of deliverable D5.3. Once the user has passed on the description of the change request, the interactions between the risk analyst and the requirement engineer in the integrated process is then conducted without consulting the user or other stakeholders/externals.

In explaining the integrated process, we begin with the requirement engineer partition. The requirement engineer uses the previous requirement model (ReM before) and the change request to update the requirements model, producing ReM after. Based on the ReM after, new assets are extracted if relevant. At this point, the requirement engineer may invoke the risk assessment in order to have the risk analyst to identify related risks and pass back relevant treatments that should be taken into account in the requirements analysis.

Receiving the extracted assets, the risk analyst use this input as a kind of change request and combines it with the previous risk model (RiM before) to conduct a new risk assessment. The risk assessment includes the identification of risks regarding the new assets, as well as the estimation and evaluation of these risks. For the unacceptable risks, a treatment identification is conducted. Without consulting the user, the risk analyst passes a specification of treatment options back to the requirement engineer who now proceeds.

Using the treatments as input, the requirement engineer specifies corresponding security goals and actions to fulfill these. This yields a new update of the ReM, which serves as the basis for the requirements analysis.

The requirement analysis must determine whether the achievement of the specified goals is acceptable. If it is acceptable, the requirement engineering process concludes and reports back to the user. If it is not acceptable, the requirement engineer must identify the problem.

If there is a problem with the treatments, for example that some of the corresponding security goals cannot be fulfilled, the risk assessment is invoked with a request for alternative options for treatments. The risk analyst passes new treatments back, after which the requirement engineer makes a new update of the ReM and conducts a second requirement analysis. If there is a problem with the ReM, the requirement analyst must backtrack and search for an alternative way of updating the ReM when considering the change request that was initially passed from the user.

6.2.2.2 Detailing the Process with the Rationales

The most basic form of the interactions between the risk assessment process and the requirement analysis process is by the passing of assets from the latter to the former, and the passing of treatments from the former to the latter. In some cases this will

suffice for the respective processes to proceed with their respective activities. In other cases there is a need to include also the rationales for the artifacts that are exchanged.

For example, when only a set of treatments is passed from the risk analyst to the requirement engineer, this may not suffice for the requirement engineer to determine how to most adequately update the requirement model, or to determine which treatments to select. The risk analyst can then specify for each treatment the set of assets for which the treatment provides protection. If even further rationales are required, the risk analyst can provide for each pair of treatment and asset the risk evaluation matrix where the estimated reduction of risk levels by implementing the treatment is documented. The risk matrix shows the reduction of risk level by depicting the reduction of likelihood and/or consequence, and also shows whether the levels of the risks are acceptable.

Conversely, it may not be sufficient for the risk analyst to know the new assets alone. In order to determine the severity of identified risks, the risk analyst may need to know the priorities of the assets. For this purpose the requirement engineer can provide the achievement matrix for the goals, where the achievement matrix shows for each combination of continuity level and achievement level whether the combination is acceptable or not.

6.2.2.3 External Integration

As mentioned above, the integrated process described in this section can be understood as a more detailed instantiation of the integration of risk assessment and requirement engineering that is presented in deliverable D2.2. Whereas deliverable D2.2 explains the wider integration of design, testing, verification, risk assessment and requirements engineering, we focus here on the latter two.

The UML sequence diagram of Figure 33 is adopted from D2.2 and shows a sample change story that illustrates the global integration. The change story is initiated by a change request from the user, and the subsequent interaction exemplifies how the change may propagate. We refer to D2.2 for a more detailed explanation.

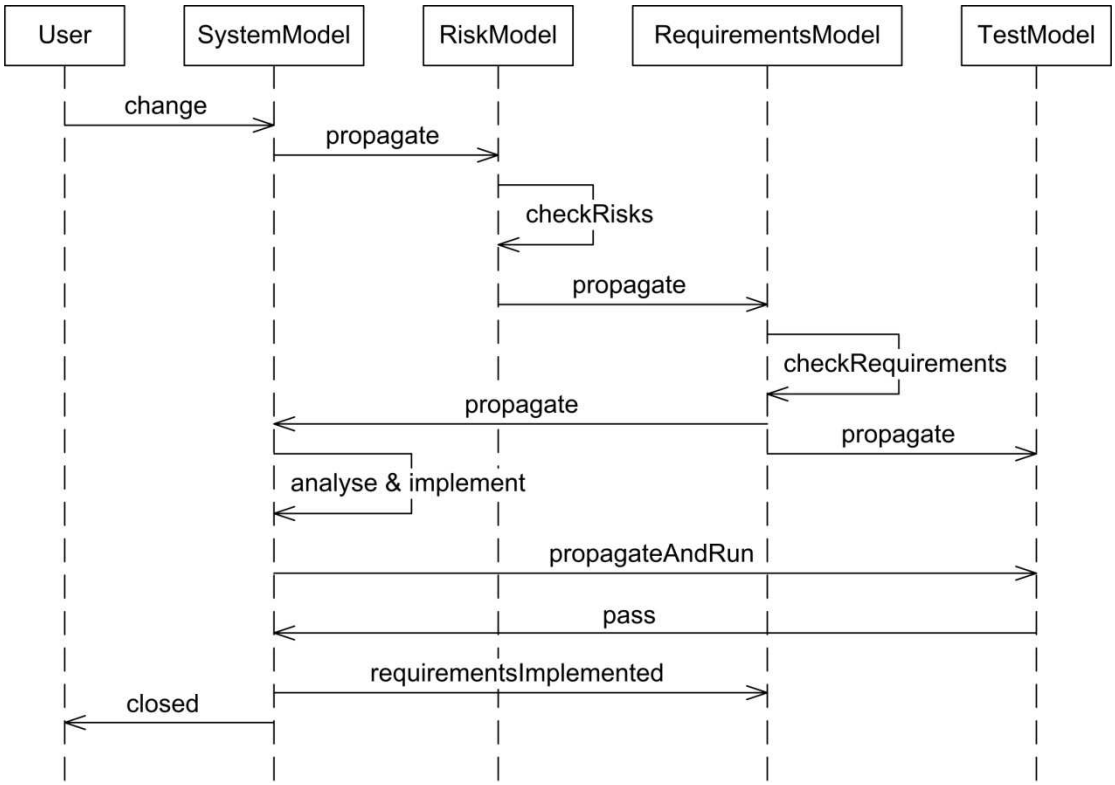


Figure 33 Simple change story

Comparing this sample of the global integration with the integrated process presented in this section, we see that the sample change story is supported by the latter. The main difference is that the integrated process described in this section is more detailed and supports a wider set of interactions.

In the global setting we can conceive the integration of the risk assessment and requirements engineering as representing one artifact. Expressing this in the sequence diagram, we can compose the lifelines Risk Model and Requirements Model, thus representing the other artifacts, i.e. the other lifelines, as external to this local integrated process. This composition is illustrated by the sequence diagram of Figure 34 with the lifeline RiM&ReM.

In this diagram we have purposely hidden the internal messages of the RiM&ReM lifeline to convey that the internal interactions can be any of those that are supported by the integrated process as described by the activity diagram of Figure 32.

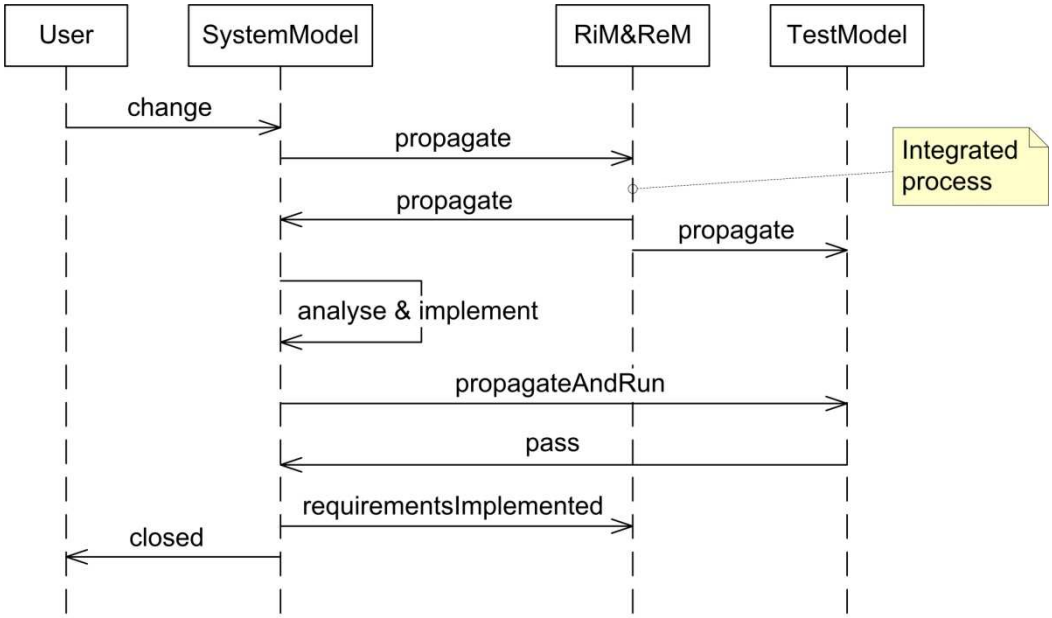


Figure 34 Integrated process in the global setting

In this global setting, we can understand the interaction with the other artifacts as an instantiation of the global integration. From the global perspective, the initial change request triggers a sequence of activities, possibly with several iterations, that should result in a stable state (global equilibrium). From the local perspective of the RiM&ReM, any input from the other lifelines are external input that triggers an internal sequence of activities that should result in a local equilibrium before the results are propagated externally. During the global process that continues towards global equilibrium, it may be that the RiM&ReM integrated process is invoked anew. In that case, the integrated process should again reach a local equilibrium before the global process continues.

6.2.3 Application to ATM Case Study

In the following we illustrate and exemplify some of the steps of the integrated process of risk assessment and requirements engineering. We address the Air Traffic Management (ATM) case study, and particularly the change requirement of Organizational Level Change.

6.2.3.1 Change Requirement and Security Properties

The Organizational Level Change introduces changes both at process and at organizational level. At organizational 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. The Sector Team consists of two Air Traffic Controllers (ATCOs), namely the Tactical Controller (TCC) and the Planner

Controller (PLC). The Sector Team is responsible for managing the air traffic of an allocated sector of the airspace.

The introduction of the AMAN requires the addition of a new type of ATCO, called Sequence Manager (SQM), who will monitor and modify the sequences generated by the AMAN and will provide information and updates to the Sector Team. The SQM replaces the ATCO role of Coordinator (COO) before the Organization Level Change.

In addition to the introduction of the AMAN, we consider the adoption of the Automatic Dependent Surveillance Broadcasting (ADS-B) which is a GPS-based system for determining aircraft positions.

The security properties we consider are Information Protection and Information Provision. For the purpose of keeping the example simple, we restrict Information Protection to the confidentiality of ADS-B data, and we restrict Information Provision to the availability of arrival sequences. In the example, we often refer to these simply as confidentiality and availability, respectively. As the ADS-B is introduced as part of the changes, we address only the availability property before the changes. We address both availability and confidentiality after the changes.

For more detailed descriptions of the target of analysis and the changes, we refer to D1.1 and to the appendix of D5.3.

6.2.3.2 Requirement and Risk Modelling before Change

We assume that the requirement model for the ATM system before the introduction of AMAN is the one illustrated in Figure 35. The model consists of four main actors, namely the TCC, the PLC, the Radar and the Flight Data Processing System (FDPS). The asset in the requirement model is the availability of the arrival sequence manually computed by the PLC.

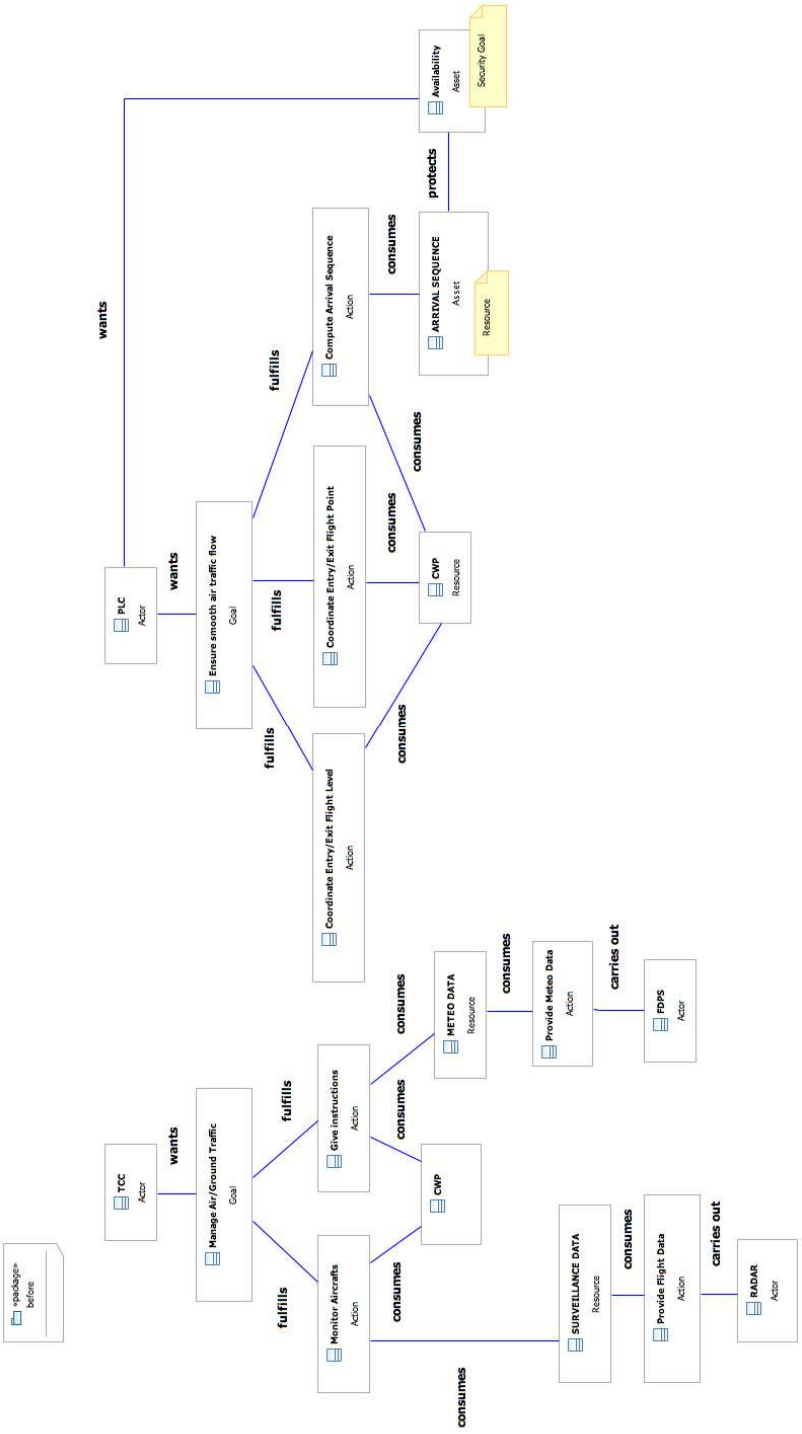


Figure 35 Requirement Model before the introduction of the AMAN

On the risk assessment side, the risk analyst has previously conducted and documented a risk assessment before the changes. The threat diagram of Figure 36 shows a sample of the documentation of the risks that were identified.

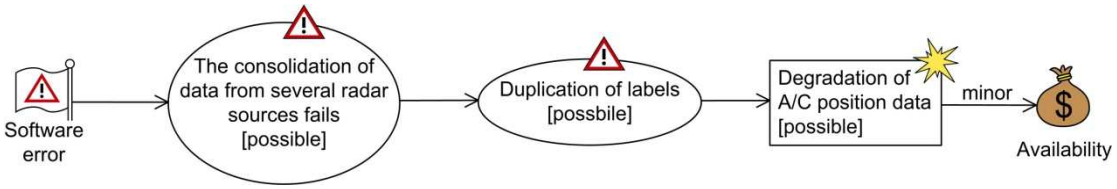


Figure 36 Risk Model before the introduction of the AMAN

The threat diagrams documenting the risks before the changes address the availability asset, as this is the only asset that is considered at this point. The threat diagrams furthermore document the results of the risk estimation by the likelihood and consequence annotation. For example, the unwanted incident of degradation of aircraft (A/C) position data occurs with likelihood possible and has a minor consequence for the availability asset. The likelihood and consequence in combination determines the risk level.

6.2.3.3 Requirement and Risk Modelling after Change

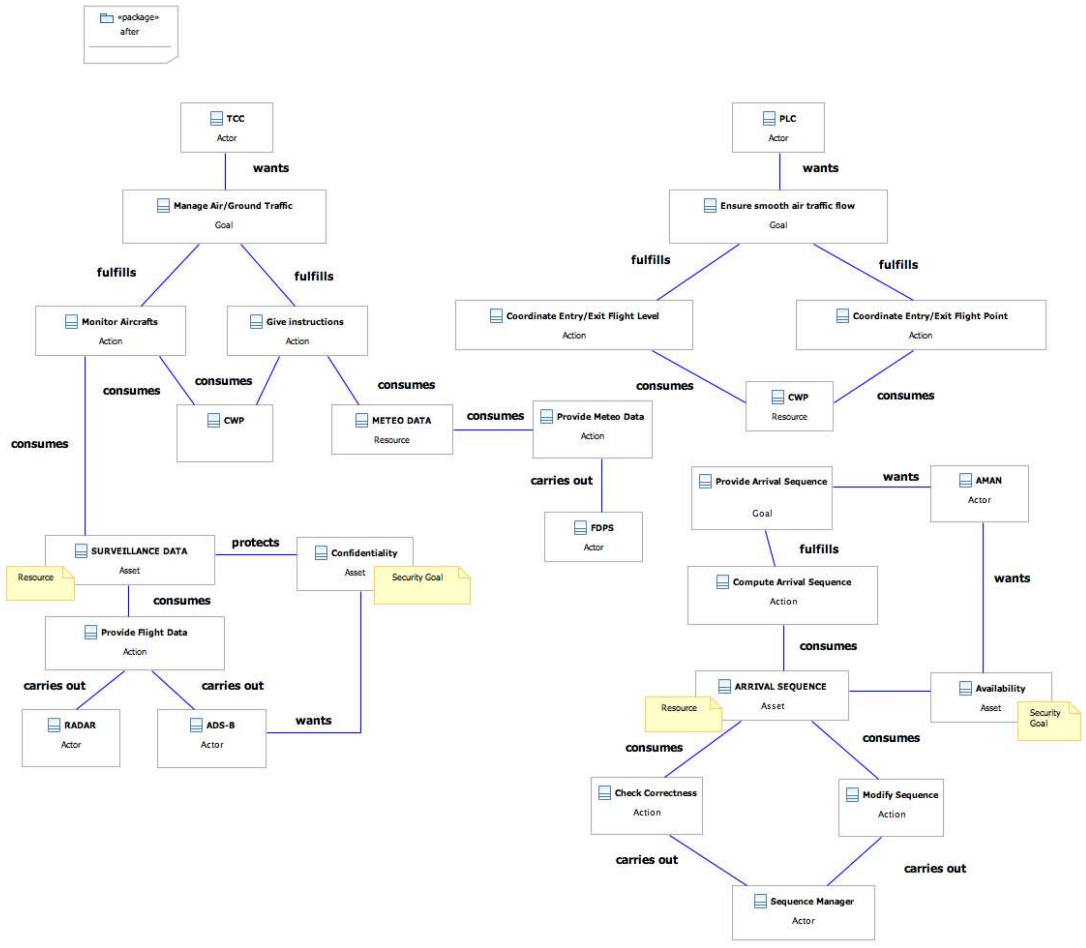


Figure 37 Requirement Model after the introduction of the AMAN

The user (e.g. the designer or the ATM service provider) decides to introduce two new components to the ATM system, namely the AMAN to support the PLC in the computation of flights arrival sequences, and the ADS-B provides more highly accurate information about aircraft position. The change request submitted by the user requires the previous requirement model to be updated with the introduction of two new actors, the AMAN and the Sequence Manager as illustrated in Figure 37.

The ADS-B has several benefits for air traffic management, but it raises several new security concerns; the ADS-B transmissions can be easily corrupted, and the signal can moreover be eavesdropped as they are openly broadcasted. Thus, because of the introduction of the ADS-B, we consider also the asset of confidentiality of the surveillance data provided by the ADS-B.

After having updated the requirement model and extracted the new asset, the requirement engineer decides to invoke the risk assessment in order see if there are new treatments that should be taken into account in the requirement model given the change requirement and the new asset.

According to the integrated process, the requirement engineer provides the list of assets to the risk analyst to help him in identifying the target of analysis and the assets. We assume that the risk analyst previously has conducted and documented a risk assessment before the changes, and that the risk analyst has access to the specification of the change requirement provided by the user. The new asset that is passed from the requirement engineer serves to increasing the details of the target description and in more precisely defining the focus of the risk assessment after the changes.

Based on the risk models before the changes, the description of the change requirement, as well as the new asset that is passed from the requirement engineer, the risk analyst updates the risk assessment documentation. The risk assessment is conducted by following the method for the risk assessment of changing systems as described in D5.3.

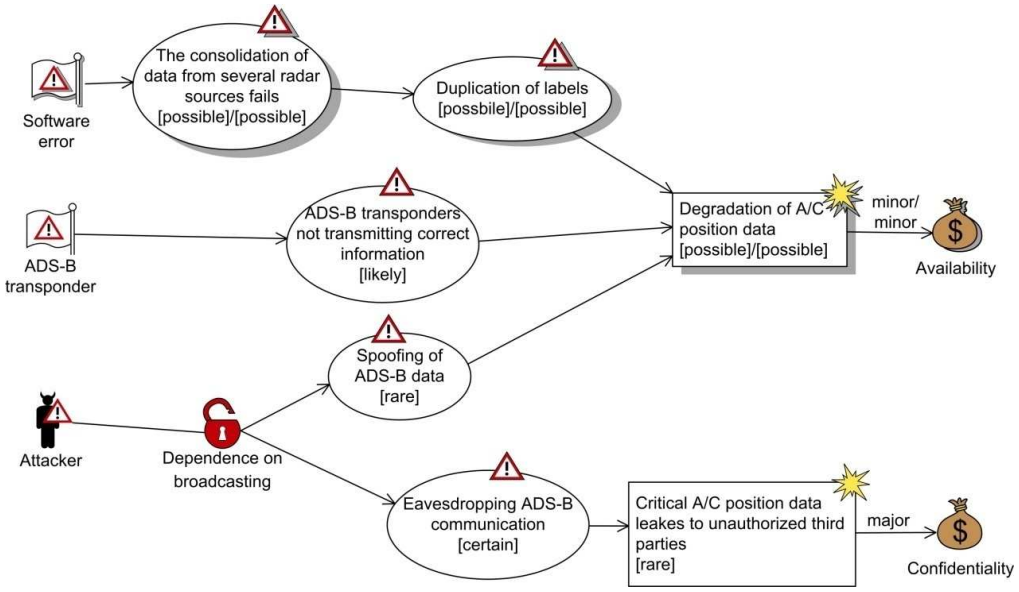


Figure 38 Risk Model after the introduction of the AMAN

The threat diagram of Figure 38 shows a sample of the results of the updated risk identification and risk estimation. The diagram models the changes to risks by distinguishing between risks before changes and risk after changes. The two-layered elements are aspects that are present both before and after, whereas the other elements are aspects that are present only after the changes.

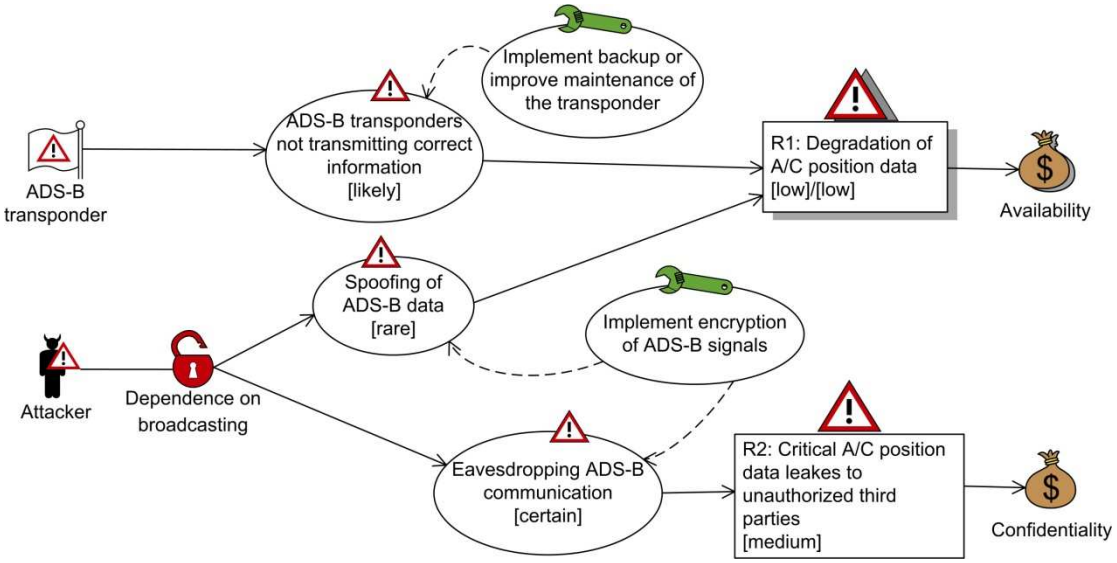


Figure 39 Treatment options after the introduction of the AMAN

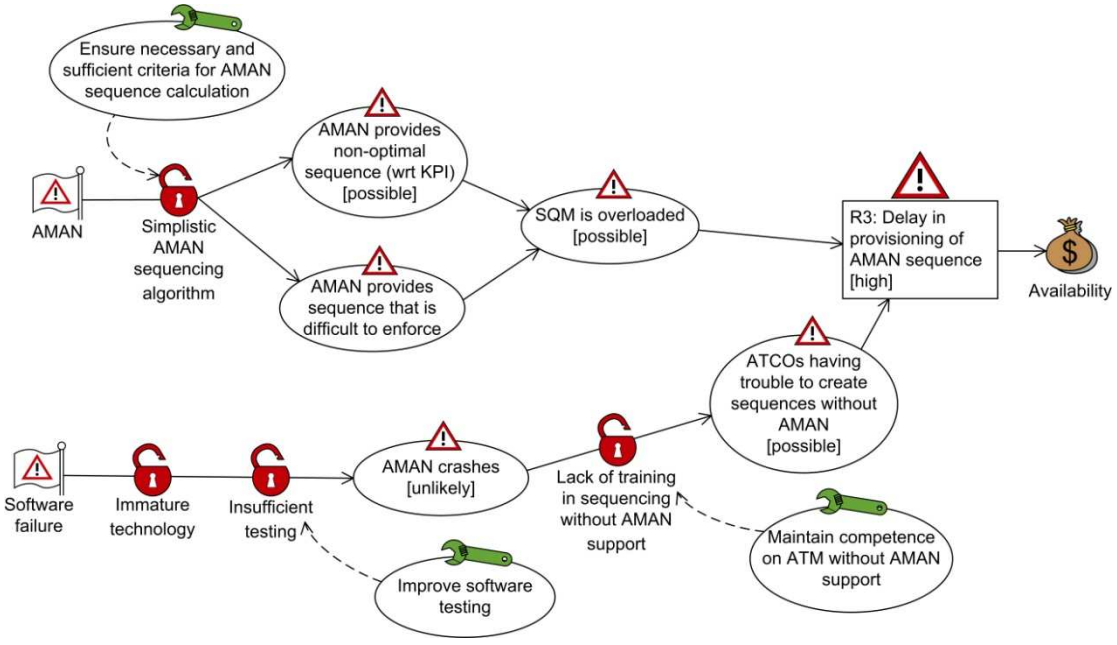


Figure 40 Treatment options after the introduction of the AMAN

From the before-after threat diagram we see that the unwanted incident of degradation of A/C position data occurs both before and after the changes. The likelihood is

possible both before and after, and the consequence for availability is minor both before and after. The unwanted incident of leakage of critical A/C position data occurs only after the changes, and has consequence for the asset of confidentiality that was passed from the requirement engineer.

After the risk assessment is concluded, the risk analyst identifies and documents treatments for the unacceptable risks, addressing risks for both of the assets. Some of the identified treatments are documented in the treatment diagrams of Figure 39 and Figure 40.

According to the integrated process, the risk analyst passes the treatment options to the requirement engineer. Focusing on the asset of confidentiality that the requirements engineer initially passed to the risk analyst, the requirement engineer conducts an update of requirement model of Figure 37. The resulting requirement model is illustrated in Figure 41 to by adding the action “Encrypt Data” that fulfills the security goal “Confidentiality” which protects the resource “Surveillance Data”.

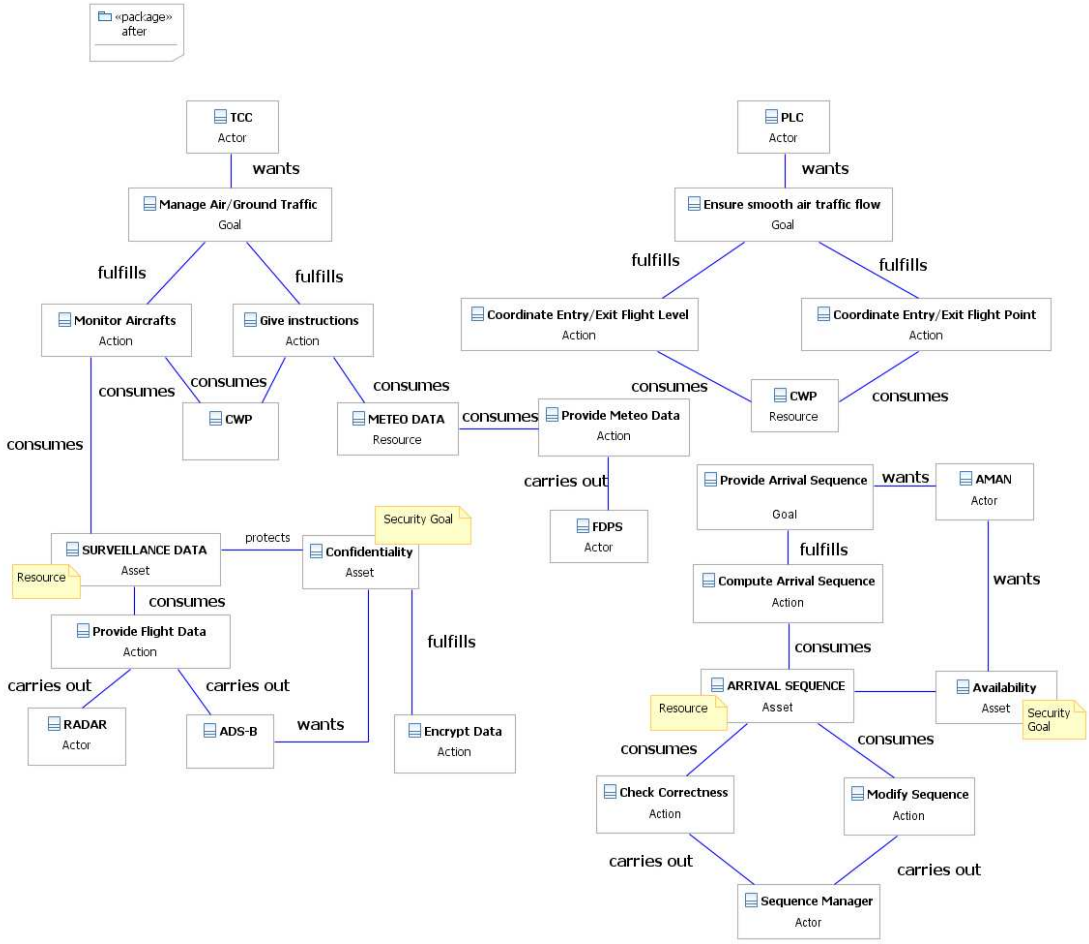


Figure 41 Requirement model updated with treatment actions

6.3 Integration of Requirements Engineering with Design

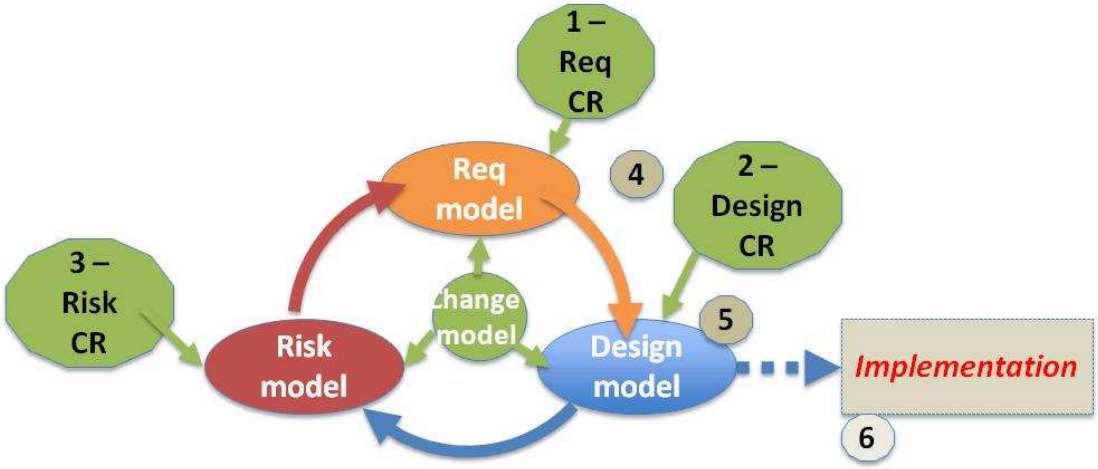
Industrial Requirements Management tools such as DOORS T-REK enable to manage the elicited requirements by providing an efficient traceability with the Product Breakdown Structure, that is to say the elements of the system.

Model Driven Engineering brings new Design tools that provide new means to assess the architecture by validating properties on the model. Security is considered as a non-functional property of the system leading to specific constraints or needs to which the system must conform.

The interaction between WP3 and WP4 lies in the way security requirements are used to influence system design, and how the conformance of the system with respect to the high-level security objectives is evaluated.

An integration is proposed by the Security DSML developed at Thales after EU-FP6 Modelplex project. This tool shall be regarded as a security viewpoint of a system model design tool in the sense where viewpoint is intended as a technology to provide non functional properties tooling integrated to a system engineering workbench, and as it is studied in French research project Movida (ANR – Call 8).

Security DSML focuses on a risk management process at system design phase, which provides security requirements as output. Starting from a model design, the Security DSML tooling enables to perform a risk analysis. The management of the risks leads to define Security Objectives, which are in their turn refined in Security Requirements. These security requirements lead to an evolution of the model since security solutions shall be implemented to complete or make the model evolve.



4 – Req CR ; 5 – Design CR ; 6 – Implementation

The WP3-WP4 link is demonstrated in D4.2 on the ATM use case of the introduction of the AMAN, which addresses the Organizational Level Change and at least the Information Access and Information Protection security properties.

The exchange of information between the requirement and system models is illustrated with a role-based access control setting. In the Thales methodology, information flows naturally from requirements to system design through semi-formal contractual requirement specifications managed with DOORS T-REK. However, dealing with the opposite direction, that is verifying that requirements are actually met by a system and that they are complete with respect to high-level security objectives are difficult tasks that are not automated in general. We show how UMLseCh can be used to help with the latter in a simple example. The scenario developed in D4.2 purposely presents an incomplete set of requirements derived through a flawed risk analysis, and then how UMLseCh detects and identifies those shortcomings. This allows the introduction of additional requirements to the system.

Industrial practices are still highly informal therefore formal verification is not feasible in general. What can be reasonably achieved, however, is a traceability link between requirements and system elements, such that any evolution in one of the models triggers a notification in the other. This is a crucial element to ensure the consistency of the models, which is all the more important that it involves distinct areas of expertise and consequently distinct actors.

Further description of the integration proposed by Thales fits in D4.4 prototype.

6.4 Integration of Requirements Engineering and Testing

The SecureChange process aims at developing an overall approach to the engineering of secure, life-long and evolvable systems. Methods and techniques for requirements engineering is only one of the cornerstones of the overall approach, and in the wider setting of security engineering of changing and evolving systems the requirements engineering constitutes one part of the overall process.

Moreover, in the setting of changing and evolving systems, there is a need to understand not only how the changes may affect security requirements on one hand and testing models on the other hand; we also need to understand how changes to requirements affects test models, and how the propagation of changes from the one to the other should be dealt with in a systematic way.

We address the problem both at a conceptual level and at a process level. At the conceptual level we present an integration of concepts and explain how requirements artifacts should be mapped to test artifacts and vice versa. At the process level, we utilize the conceptual level integration and explain how the conceptual integration comes into play in the integration of the respective methodologies.

We illustrate and exemplify the integrated process based on the Specification Evolution Change Requirement of the POPS case study.

6.5 Conceptual Integration

In this section, we describe how the SeCMER conceptual model concepts are mapped on the test concepts for the purpose of integration so to identify an interface between the two domains.

In the following we first separately present the core and basic concepts of requirements engineering and testing. Thereafter we present the conceptual level integration.

6.5.1 Requirement Concepts

The UML class diagram of Figure 42 gives an overview of the basic concepts of requirements engineering and the relations between them. We refer to Section 2 for a detailed presentation of the conceptual model. In the following we give the definitions of the concepts.

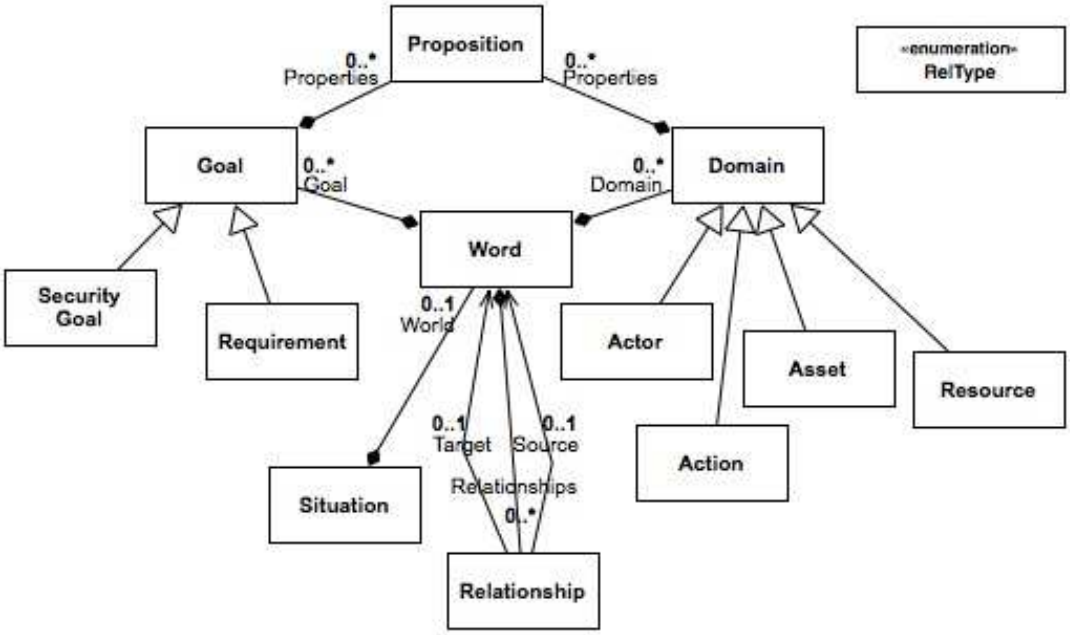


Figure 42 Basic requirements concepts

- **Action:** An entity performed by an actor, which can generate events, and can have preconditions and post-conditions
- **Actor:** An entity that can act and intend to want or desire
- **Asset:** An entity of value that can be owned and used
- **Goal:** A proposition an actor wants to make true
- **Proposition:** A statement that can be true or false
- **Resource:** An entity without intention or behavior
- **Security goal:** A proposition that specifies how to prevent harm to an asset through the violation of confidentiality, integrity, and availability security properties
- **Situation:** partial state of the world described by a proposition

- **Requirement:** Statement about what the system should do
- **Test Model:** Dedicated model for capturing the expected SUT behavior (Class diagram, State machine)
- **Test Case:** A finite sequence of test steps
- **Test Intention:** User's view of testing needs
- **Test Suite:** A finite set of test cases
- **Test Script:** Executable version of a test case
- **Test Step:** Operation's call or verdict computation
- **Test Objective:** High level test intention

6.5.2 Integration

In the conceptual integration we distinguish between what we refer to as *shared elements* on the one hand and *mappable elements* on the other hand. The shared elements are concepts that are common to requirements engineering and testing, with the same semantics in both domains. The mappable elements are concepts from one domain that are not shared by the other, but are nevertheless related to the other domain and can be mapped to concepts of the other domain.

We identify one share element that is Requirement. A Requirement in both domains represents a statement by a stakeholder about what the system should do.

The concepts of Actor, Goal and Action are mapped on the Test Model. In particular, the concept of Actor is used to identify the system under test (SUT). The concept of Goal and Action are used by the testing engineer to build the Test Model which represents the expected behavior of the SUT. The Test Model is usually represented using UML Class Diagrams, Instance Diagrams and State Machine Diagrams. The dynamic behaviors of those diagrams are described using OCL (Object Constraint Language). The goals and actions in the Requirement Model are identified by a unique name that is used to annotate the State Machine of the Test Model and the OCL code in order to achieve traceability between the Requirement Model and the Test Model.

In the integration between requirement engineering and testing, we also use the artifacts achievement matrix and test results. The achievement matrix is produced as the result of requirement analysis, and specifies the continuity and achievement level for each requirement. A test result is the outcome of the execution of a test case which can be *fail*, *pass* or *inconclusive*. The test results are used to determine the level of requirement coverage. If the test result for a given test case is “fail” or “inconclusive” the corresponding requirement under test is not satisfied by the SUT or it is not correctly implemented.

An overview of the conceptual integration is given in Table 1.

Table 4 Conceptual Integration

Requirement concept	Testing concept	Kind of integration
Goal	Test Model (State Machine, OCL code)	Requirement concept mapped to Testing concept
Action	Test Model (State Machine, OCL code)	Requirement concept mapped to Testing concept
Achievement matrix	Test result	Requirement concept mapped to Testing concept
Actor	SUT	Requirement concept mapped to Testing concept
Requirement	Requirement	Shared concept

6.6 Integrated Process for Change Management

In this section we present the integration between requirements engineering and testing engineering processes. The two processes should still be understood as separate processes with their own iterations, activities and techniques for managing change. The integrated process explains at which steps of the respective processes the conceptual level interface can or should be invoked. The integrated process is hence based on the conceptual integration presented in the previous section.

The integrated process presented here can be understood as an instantiation of the more general and project wide integration presented in deliverable D2.2.

6.6.1 Overview of Process

The UML activity diagram of Figure 42 gives a high-level overview of the integrated process. The diagram is divided into three partitions to distinguish between the activities and objects under the control of the user, the requirement engineer, and the test engineer. The user is typically the client commissioning the testing and may, for example, be the owner of the system that is under test.

The integrated process is defined such that the requirements engineering process and the testing process are conducted separately. In the diagram, the diamonds specifies branching of the sequence of activities. When there is no guard condition on the branching (specified by the notes with Boolean expressions), the process proceeds along one or both of the branches. This gives a wide flexibility on how the overall process may be conducted.

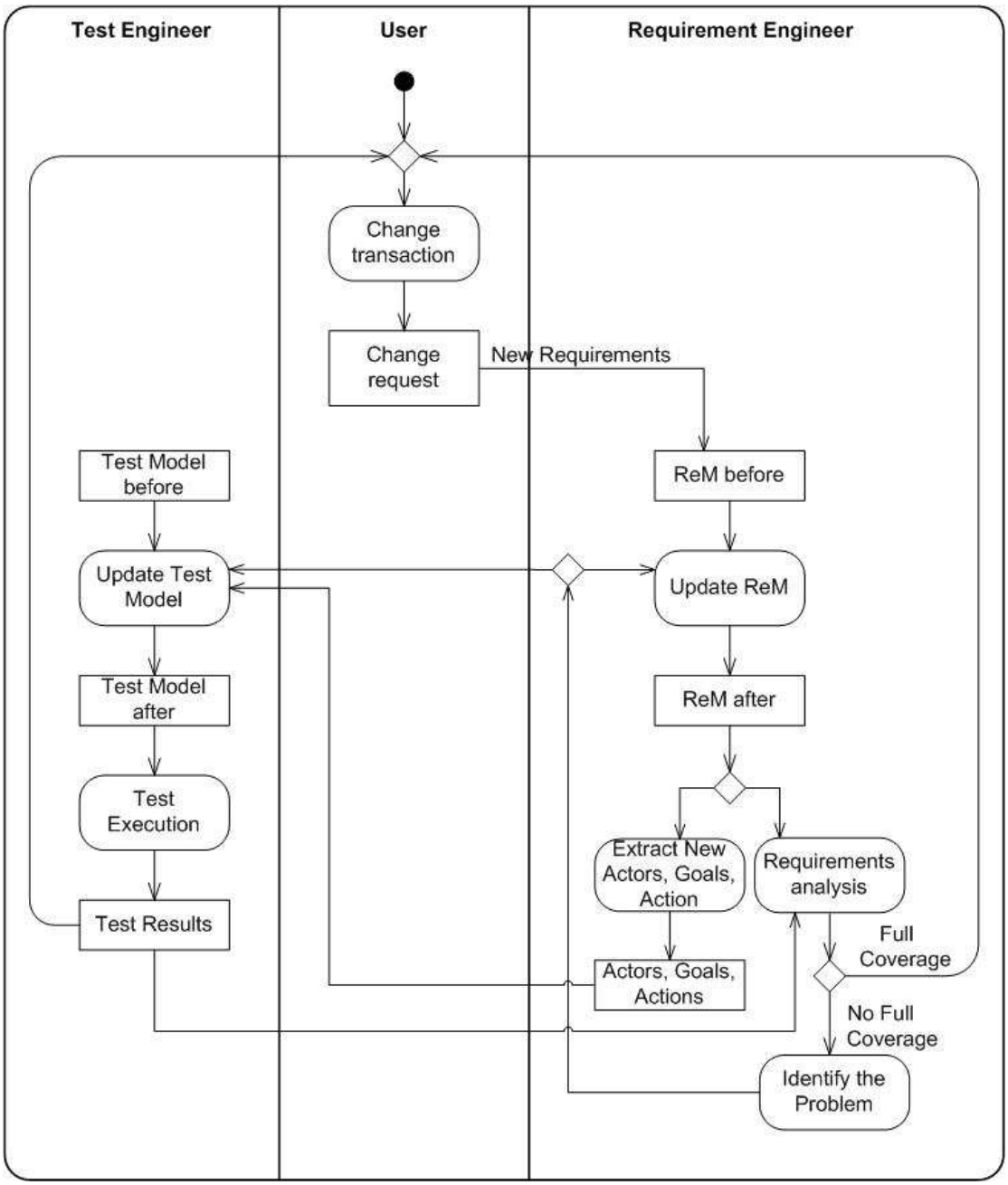


Figure 42. Integrated process

The process starts when the user makes a change request that is passed to the requirement engineer. The requirement engineer uses the previous requirement model (ReM before) and the change request to update the requirement model, producing ReM after. Based on the ReM after, new actors, goals and actions are extracted if relevant. At this point, the requirement engineer may interact with the test engineer in order to have the test engineer to identify which are the part of the test model that are affected by the change in the requirement model. Then, the testing engineer checks

which tests are reusable or are obsolete and if there is the need to produce new test cases.

If this is the case the new tests are executed and the result of the execution is passed back to the requirement engineer that should take them into account in the requirements analysis.

The requirement analysis must determine whether requirements have been fulfilled based on the test results. If all the test results are successful, the requirements are all satisfied and, thus, the requirement engineering process concludes and reports back to the user. If some of the requirements are not fulfilled by the SUT, the requirement engineer must identify the problem.

If there is a problem with the ReM, the requirement analyst must backtrack and search for an alternative way of updating the ReM when considering the change request that was initially passed from the user.

If there is a problem with testing, the test engineer must determine whether there is the need to produce new test cases or not.

The UML sequence diagram of Figure 43 is taken from D2.2 and shows a sample change story that illustrates the global integration. The change story is initiated by a change request from the user, and the subsequent interaction exemplifies how the change may propagate. We refer to D2.2 for a more detailed explanation.

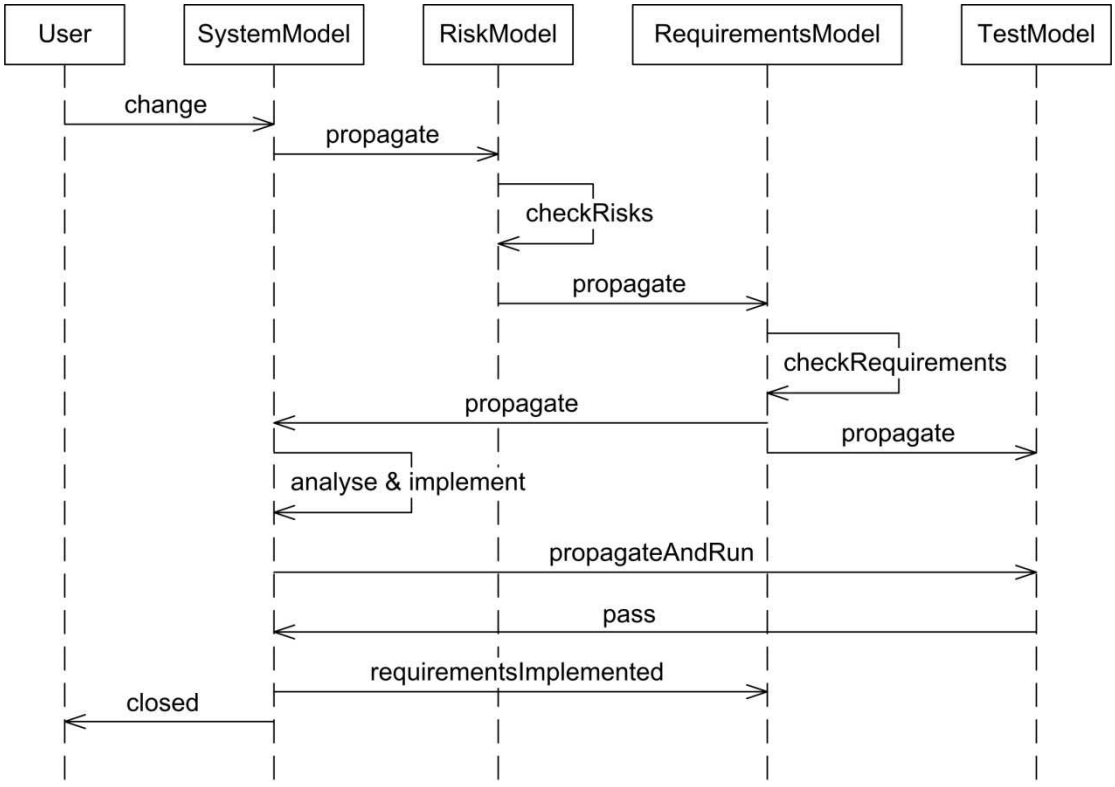


Figure 43 Simple change story

Comparing this sample of the global integration with the integrated process presented in this section, we see that the sample change story is supported by the latter. The main difference is that the integrated process described in this section is more detailed and supports a wider set of interactions.

6.7 Application to POPS Case Study

In the following we illustrate and exemplify some of the steps of the integrated process of testing and requirements engineering based on change requirement “Specification Evolution” of the POPS case study.

6.7.1 Change Requirement

The Specification Evolution Change Requirement is about the changes in the card life cycle that have been introduced in GP 2.2 with respect to GP 2.1.1. The card life cycle differences in the two versions of GP specification are illustrated in Figure 44.

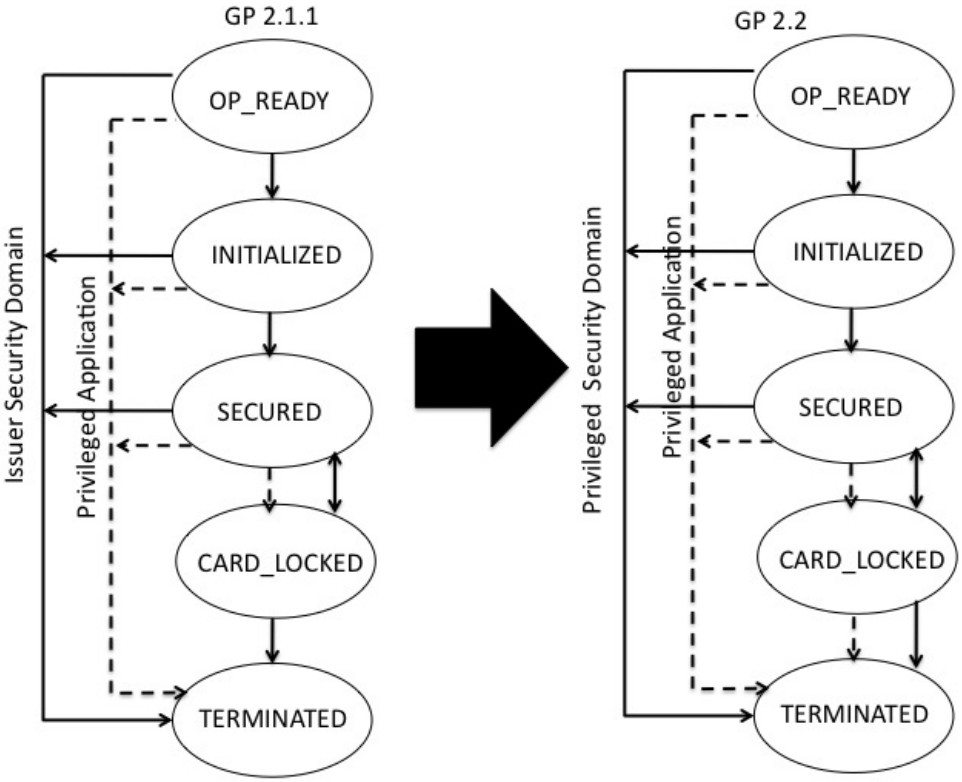


Figure 44 Card Life Cycle in GP 2.1.1 and GP 2.2

The main differences between the two GP specification versions are that in GP 2.2 not only the Issuer Security Domain can perform any state transition in the card life cycle but any Privileged Security Domain and that a Privileged Application can terminate the card from the CARD-LOCKED state.

6.7.2 Requirement and Test Modeling for GP 2.1.1

The requirement model for Card Lifecycle Management of Global Platform 2.1.1 is illustrated in Figure 45. The model consists of four actors namely: OPEN (Global Platform Environment), Privileged application, Privileged SD (Security Domain), and Issuer SD (ISD). Privileged application can only terminate card lifecycle by setting card status from any state (except CARD_LOCKED) to CARD_TERMINATED. Additionally, privileged application can lock the card by changing card state from SECURED to CARD_LOCKED. Security Domain is a special kind of privileged application, and therefore, has exactly the same behavior of privileged application in terms of card lifecycle management. The other transitions in the card life cycle can only be performed by the ISD.

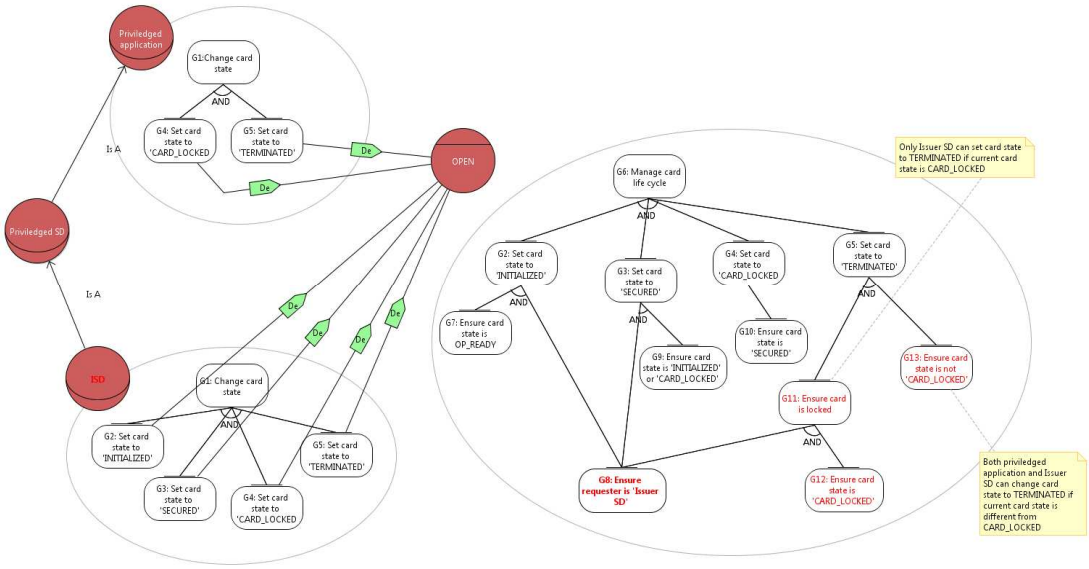


Figure 45 Requirement Model for GP 2.1.1

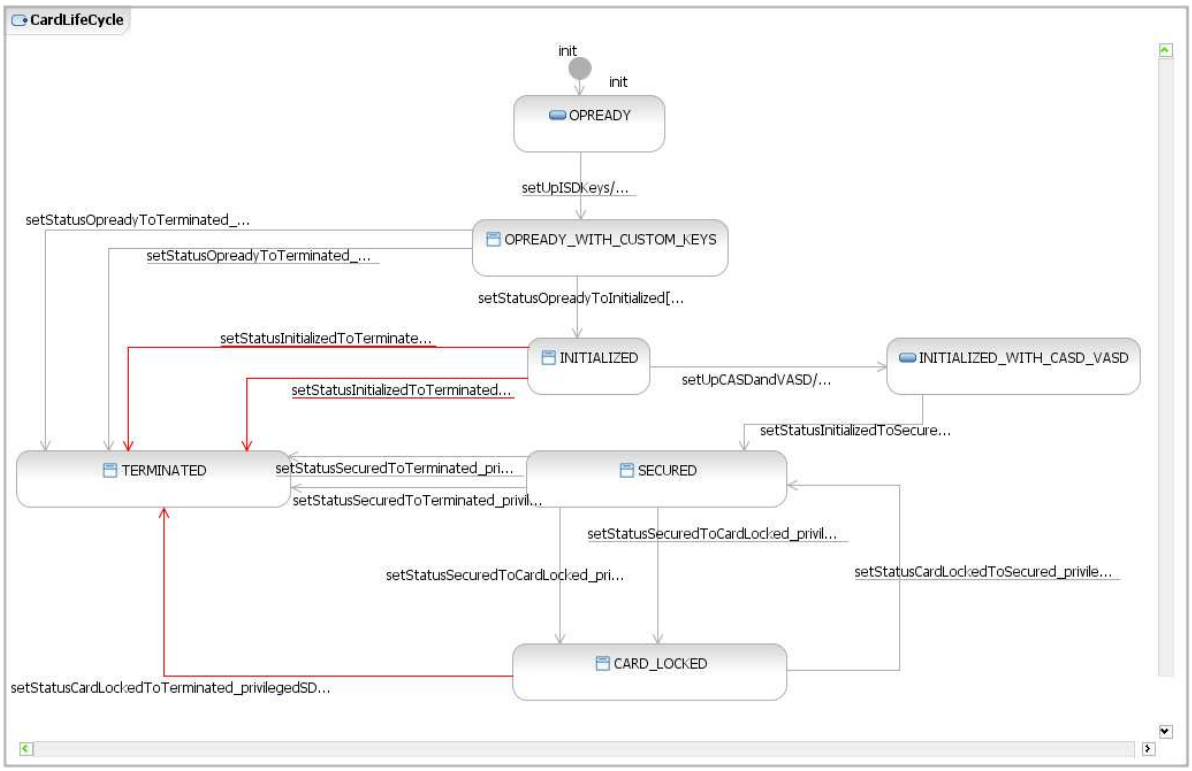


Figure 46 Test Model for GP 2.1.1

Figure 46 depicts the state machine for the card lifecycle in GP 2.1.1. For the sake of this example we only focus on Goal G5 (Set card state to TERMINATED). This goal is detailed in 3 transitions (colored in red) of the state machine. They provide test cases for 4 sub goals: G8, G11, G12 and G13.

Guard - setStatusCardLockedToTerminated_privilegedS

```
/*APDU: SetStatus GP2.1.1
Type: Guard of transition

Current state is CARD_LOCKED ,
Application = Issuer SD */

(
    self.lcs->exists(lc : LogicalChannel |
        IN_lcNumber = lc.number and
        IN_claSMLevel = lc.secureChannelSession.secureMessagingLevel and
        /**@AIM: G8 */ /*Issuer SD*/
        lc.selectedApp.aid = ALL_AIDS::aid_ISD
    ) and
    IN_option = ALL_SET_STATUS_OPTIONS::CARD and
    /**@REQ: G11, G12*/ /*state is CARD_LOCKED*/
    self.state = ALL_STATES::CARD_LOCKED and
    /**@REQ: G5*/ /* new state is TERMINATED*/
    IN_state = ALL_STATES::TERMINATED and
    IN_appAid = ALL_AIDS::aid_ISD
) = true
```

Figure 47 OCL code for transition from CARD_LOCKED to TERMINATED

Figure 47 shows how the OCL code for the transition from the status CARD_LOCKED to the status TERMINATED in the State Machine has been tagged with the names of the goals G11 and its sub goals G8 and G12 that are part of the requirement model in Figure 44.

Guard - setStatusInitializedToTerminated_privilegedApp

```
/*APDU: SetStatus GP2.1.1
Type: Guard of transition

Current state is INITIALIZED ,
Application = application with cardTerminate privilege */
(
    self.lcs->exists(lc : LogicalChannel |
        IN_lcNumber = lc.number and
        IN_claSMLevel = lc.secureChannelSession.secureMessagingLevel and
        /**@AIM: G13_a */ /* application with cardTerminate privilege */
        lc.selectedApp.privileges.cardTerminate = true and
        lc.selectedApp.privileges.securityDomain = false
    ) and
    IN_option = ALL_SET_STATUS_OPTIONS::CARD and
    /**@AIM: G13*/ /* Current state is INITIALIZED */
    self.state = ALL_STATES::INITIALIZED and
    /**@REQ: G5*/ /* new state is TERMINATED */
    IN_state = ALL_STATES::TERMINATED and
    IN_appAid = ALL_AIDS::aid_ISD
) = true
```

Figure 48 OCL code for setStatus APDU command for Privileged Application

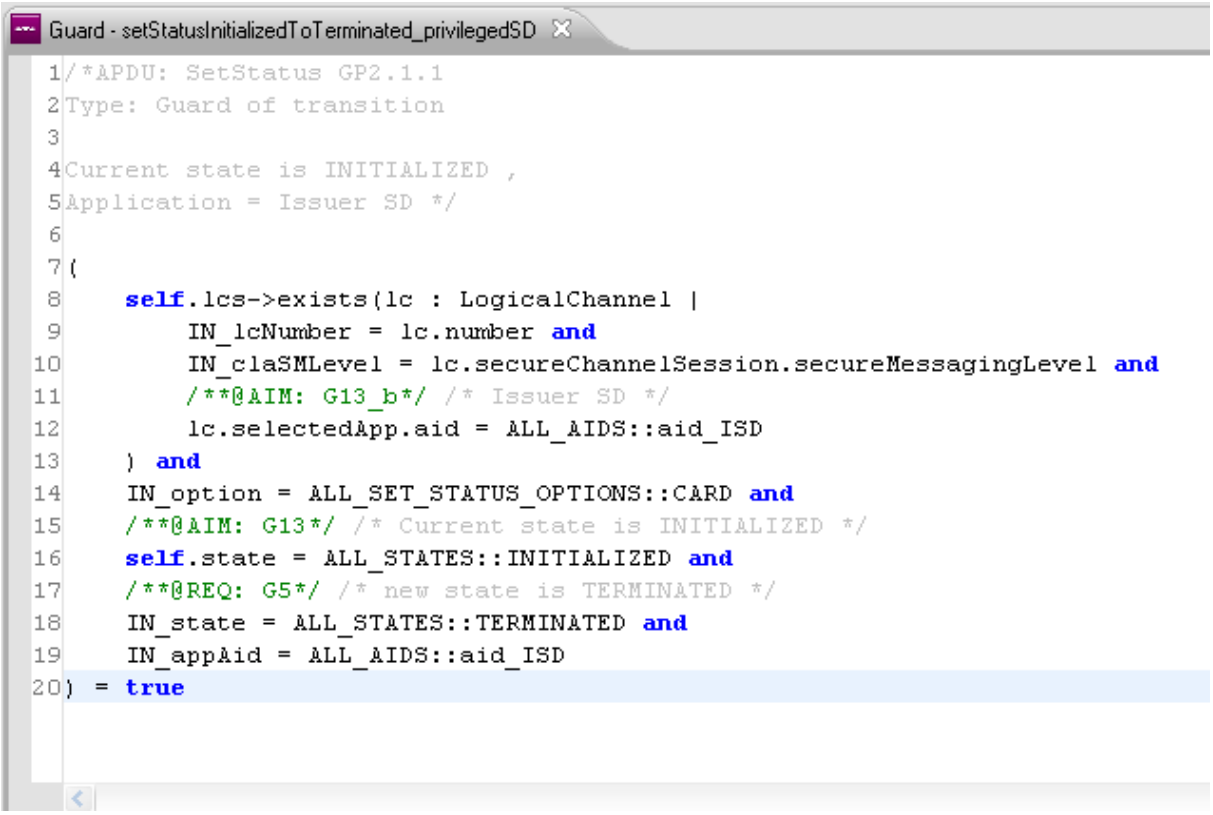


Figure 49 OCL for setStatus APDU command for Privileged Security Domain

Figure 48 and Figure 49 provide a test model for goal G13 that is divided in two separate transitions, one for an issuer SD and one for an application with cardTerminate privilege.

6.7.3 Requirement and Test Modeling after Change

Figure 50 describes the requirement model for Card Lifecycle Management in GP 2.2, where the actors are same as of GP 2.1.1. There are two changes in card lifecycle management. First, privileged application can now terminate card lifecycle from any state if the application has appropriate privileges. Second, a security domain is more powerful since it can now perform all card state transitions which can only be done by issuer security domain in the previous version.

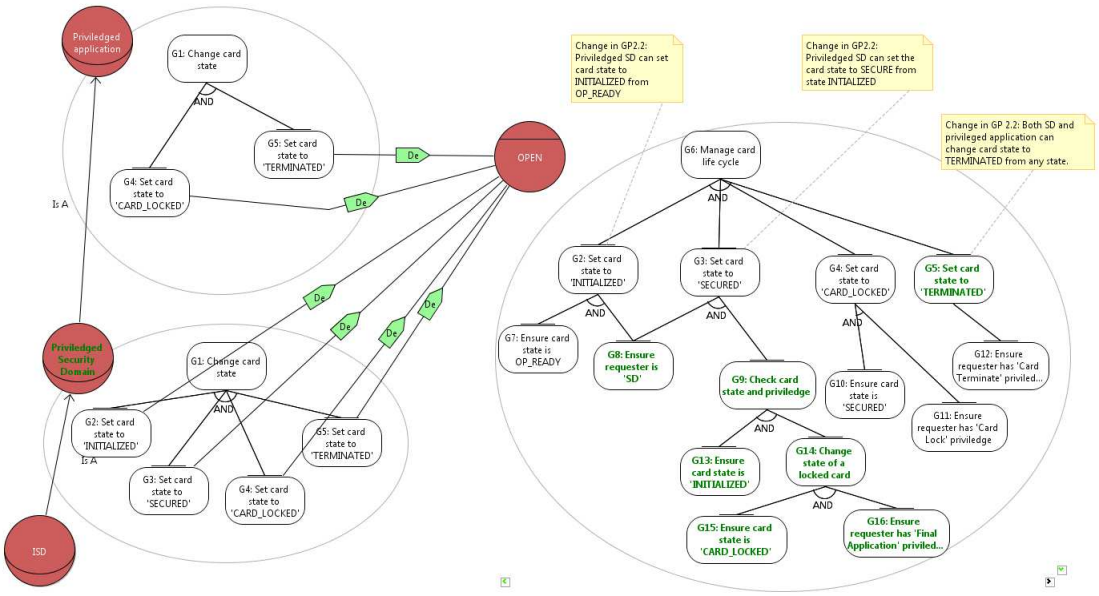


Figure 50 Requirement Model for GP 2.2

Figure 51 depicts the test model for card lifecycle in GP 2.2, with specific transitions red colored. They provide test cases for Goal G5. They are detailed in the OCL code reported in Figure 52 and Figure 53. The two figures reflect the difference between two types of actors: an application with cardTerminate privilege and a SD with cardTerminate privilege.

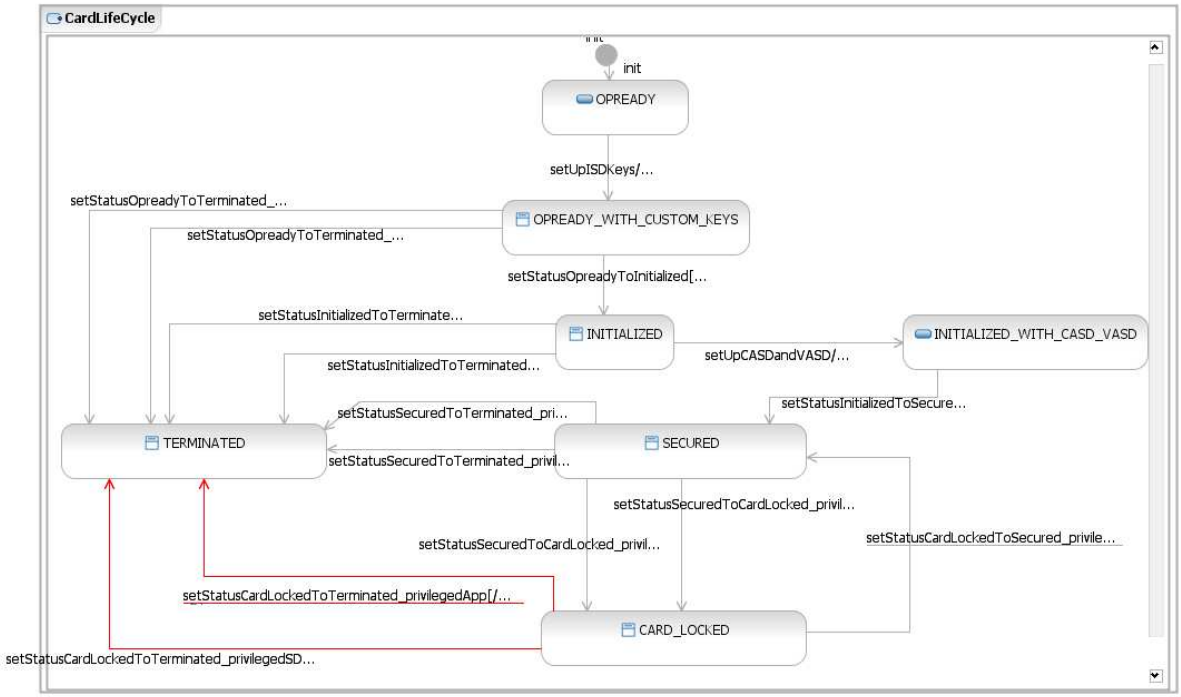


Figure 51 Test Model for GP 2.2

Guard - setStatusCardLockedToTerminated_privilegedApp

```
/*APDU: SetStatus GP2.2
Type: Guard of transition

Current state is CARD_LOCKED ,
Application = not an SD but with cardTerminate privilege */
(
    self.lcs->exists(lc : LogicalChannel |
        IN_lcNumber = lc.number and
        IN_claSMLevel = lc.secureChannelSession.secureMessagingLevel and
        /**@REQ: G12*/ /*application with cardTerminate privilege*/
        lc.selectedApp.privileges.cardTerminate = true and
        lc.selectedApp.privileges.securityDomain = false
    ) and
    IN_option = ALL_SET_STATUS_OPTIONS::CARD and
    self.state = ALL_STATES::CARD_LOCKED and
    /**@REQ: G5*/ /*new state is TERMINATED*/
    IN_state = ALL_STATES::TERMINATED
) = true
```

Figure 52 OCL code for setStatus APDU command for privileged application

Guard - setStatusCardLockedToTerminated_privilegedSD

```
/*APDU: SetStatus GP2.2
Type: Guard of transition

Current state is CARD_LOCKED ,
Application = SD but with cardTerminate privilege */
(
    self.lcs->exists(lc : LogicalChannel |
        IN_lcNumber = lc.number and
        IN_claSMLevel = lc.secureChannelSession.secureMessagingLevel and
        /**@REQ: G12*/ /*SD with cardTerminate privilege*/
        lc.selectedApp.privileges.cardTerminate = true and
        lc.selectedApp.privileges.securityDomain = true
    ) and
    IN_option = ALL_SET_STATUS_OPTIONS::CARD and
    self.state = ALL_STATES::CARD_LOCKED and
    /**@REQ: G5*/ /*new state is TERMINATED*/
    IN_state = ALL_STATES::TERMINATED
) = true
```

Figure 53 OCL code for setStatus APDU command for privileged SecurityDomain

According to the integrated process, test results from test cases of two models in both GP 2.1.1 and GP 2.2 are fed back to the requirement engineer. Then he can evaluate the coverage of requirements and test results are propagated to the corresponding goals to estimate their satisfaction.

In POPS case study, all the tests are reported as successful. Hence, full requirement coverage is achieved.

7 Evolution of Security Models

This section discusses various concepts associated with the notion of change. To avoid restricting the generality of the change model, it will not be bound to requirements modeling, but to the Integrated Model introduced by WP2 in D.2.1, consisting of requirement, architectural, risk, etc. domain models. In a passive view of the classification, model snapshots are observed to change over time. The change model will show the properties and relationships of model snapshots, and the different kinds of changes that span between them.

The first part of the Section introduces our proposed generic, domain and application independent Change Model. The second part discusses a concrete industrial change model used by Thales to manage the changes to requirements and related models. We also provide a mapping between the two different terminologies.

7.1 Generic Model of Change Concepts

This Section identifies a number of concepts to describe changes experienced by an engineering model. An UML Package representation of the concepts discussed here is presented in Figure 54.

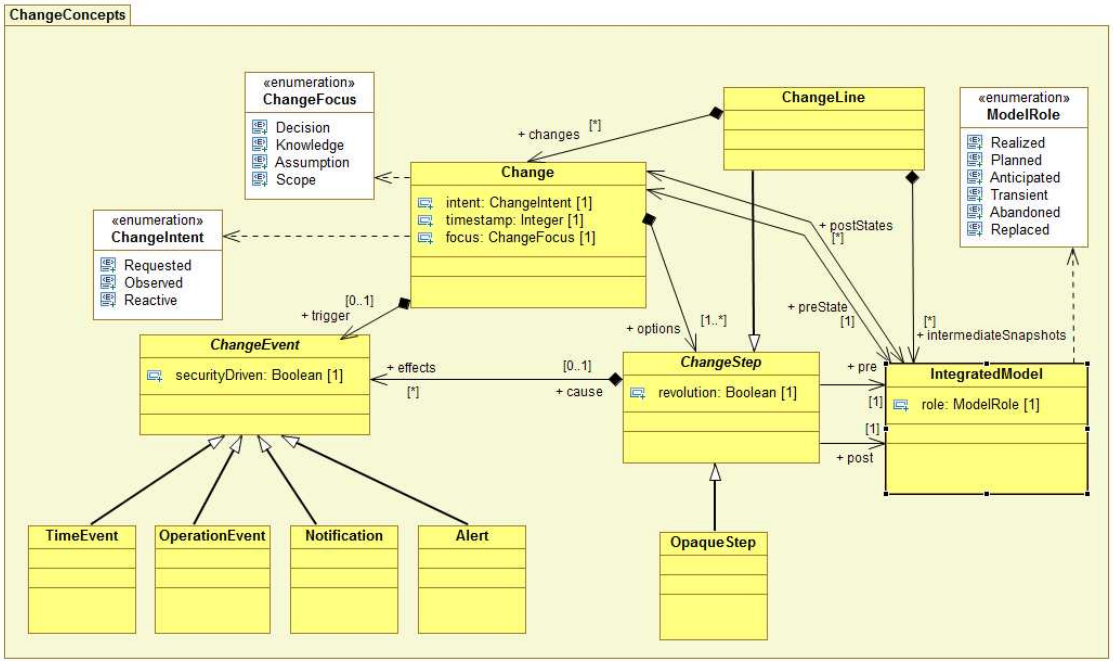


Figure 54 Change Concepts

7.1.1 Integrated Model

Before discussing change, we have to find a representation for the engineering model that are subject to change. Borrowing from the terminology of WP2, we use the concept of **Integrated Model** to refer to the set of engineering models from various domains (Architecture, Risk, Requirements, etc. interlinked through traceability) as a single entity. An Integrated Model instance represents a single snapshot of these models and their linkages.

From the point of view of model versioning and change history, each Integrated Model snapshot assumes a certain **role**. As this role itself can change also, the set of possible roles form the state machine depicted in Figure 55. The model snapshot that represents the current reality is the **Realized** model. If reality changes and a new Realized model emerges, the older version is marked as **Replaced**. Unrealized model snapshots that are created for analyzing possible uncontrolled (provisional, expected) future changes have the **Anticipated** role, while a model that show the result of a candidate outcome of a deliberate decision is a **Planned** model. Both kinds can be Realized eventually, or **Abandoned** otherwise. If security deficiencies, inconsistencies or other problems are discovered in a Realized, Planned or Anticipated model (either at the creation of the snapshot or later), and it would have to be superseded by newer versions to address the problem, the model is marked **Transient**.

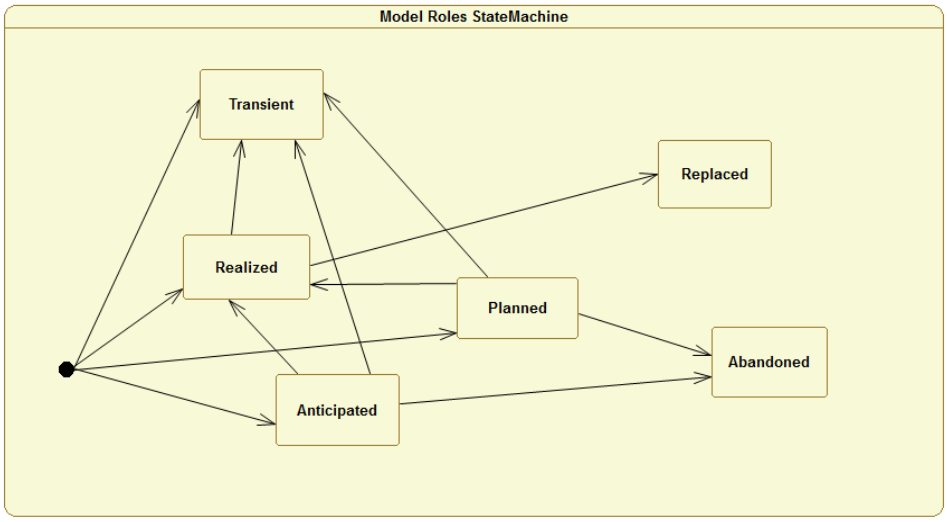


Figure 55 State Machine of Roles of an Integrated Model

7.1.2 Change

The key concept of the Change Model is **Change**. A Change instance represents the updating (or planning to update) of an Integrated Model. The snapshot that is updated is called the **preState** of the Change, while the resulting model is the **postState**. Sometimes the Change encompasses several **options** (alternatives that the engineers or outside forces can choose from) leading to different postStates; each option is a single **ChangeStep**. A **timestamp** can also be associated to the Change.

As indicated by the **intent** attribute, some Changes are **Requested** by stakeholders, others are passively **Observed** and cannot be influenced, while the rest are **Reactive** changes driven by engineers to restore desired properties of the model after earlier changes. On a related note, a Change can have a single triggering **Change Event** that is the cause of the Change, but it is possible that no trigger is identified for an Observed change. An optional **focus** attribute tells us conceptually what type of information is subjected to Change: either our **Knowledge** about the system and its context is changed, or our **Assumptions** are changed, or the change is in the **Scope** of the analysis producing the model, or it reflects a design **Decision** being made or revised.

7.1.3 Change Line and Change Step

A **Change Line** describes a longer trajectory from a single **preState** model to a single **postState** model, through several successive intermediate models. It also contains the Changes that happen between each model (except the postState) and its successors. The Changes can even have Abandoned options.

A **Change Step** is the transition from a single preState model to a single postState model. Each option of a Change is described by a Change Step. The **revolution** attribute indicates whether the step is considered evolutionary (gradual change over time) or revolutionary (large parts of the model removed and/or created from scratch). Based on granularity, there are two kinds of Change Steps. An **Opaque Step** does not contain any further information about how the postState is derived from the preState. It is also possible, on the other hand, to consider a composite step that consists of intermediate stages, changes between them, or even explored and rejected alternatives; therefore the previously introduced Change Line is also one kind of Change Step. Furthermore, it is possible to use these composite steps as options for a Change, as well as the opaque ones; this way a long engineering process of considering alternatives and their consequences can be condensed into the best and final version.

Refer to Section 7.1.5 for an illustration of how the concepts of Change Line, Change Steps and the options of a Change are related to each other.

7.1.4 Change Event

While some Observable Changes may have an unknown cause, the majority of Changes are triggered by a well-defined **Change Event**. Some Change Events are initiated externally; others are caused by *previous* changes to the model, so that a new, Reactive Change is triggered by the Change Step leading up to its preState. If the reason of the Change Event is associated with security concerns, it can be marked as **securityDriven**; the triggered Change in this case is especially in the focus of SecureChange.

We distinguish four kinds of Change Events.

- A **Time Event** is triggered by reaching a certain point of time; e.g. a pre-scheduled revision of security risks after one year of operation, or a periodical renewal/replacement of supplier Actor contracts.

- An **Operation Event** is a human intervention into the system, e.g. a stakeholder requesting new functionality.
- A **Notification** represents the elementary model manipulations performed by the previous Change Step: addition, deletion, attribute modification, replacing, etc. depending on the modeling technology used.
- **Alert** is a higher level mechanism that conveys a more abstract, domain-specific report of the modifications performed by the preceding Change Step.

7.1.5 Illustrative Example

This Section illustrates the introduced concepts by an example evolution story.

A fraction of the lifecycle of an Integrated Model is shown on Figure 56.

1. At the beginning, there is stable version of the model.
2. Due to an uncontrolled change in circumstances (e.g. a new regulation), an Observed Change happens (without offering multiple options), leading to a postState Integrated Model. As opposed to the initial snapshot, the new model has some security problems, that are indicated by Alerts.
3. They trigger a Reactive Change to fix the problems in this Transient state. The Change has two options, corresponding to design alternatives. Each option is defined by an Opaque Step leading to a different Planned version of the Integrated Model.
4. However, one of the proposed solutions raises further security issues, so it also becomes a Transient state. A second Reactive Change with a single option is initiated, leading to a final and consistent Planned model.

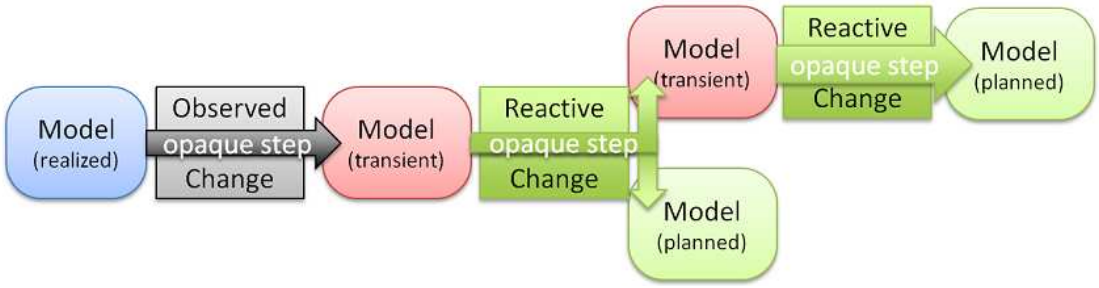


Figure 56 Example evolution, phase one (exploring alternatives)

Before a decision can be made between the two viable choices, the set of final candidate solutions has to be represented as the options of the first Reactive Change. One of the options is an Opaque Step leading to a problem-free Integrated Model. As the other immediate Opaque Step only leads a Transient version, it can be wrapped together with the next successive one as a Change Line, leading all the way to the candidate solution from the first Transient state. This way, the sequence of intermediate stages (the second Transient model in this case) can be collapsed and abstracted away. The first Reactive Change will have two options, an Opaque Step and a Change Line, leading to the two candidate solutions. This makes the Change

with multiple options ready to support the decision. The new state is indicated in Figure 57.

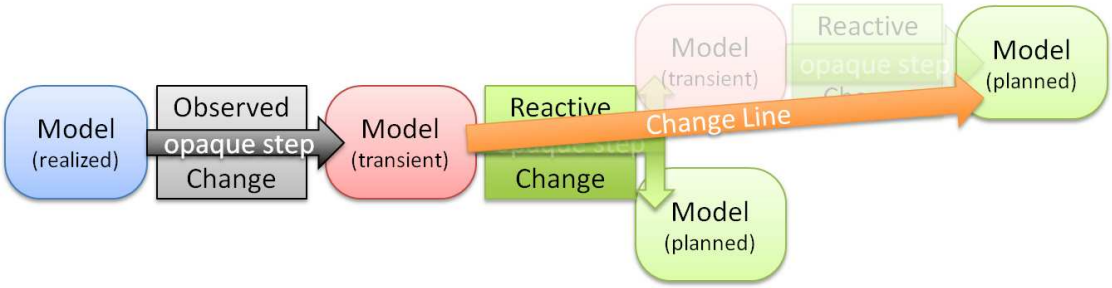


Figure 57 Example evolution, phase two (reduced to stable candidates)

Finally, a decision is made and one of the options is selected. The disfavored Proposed model becomes Abandoned. The chosen solution is implemented, and the corresponding model becomes Realized. The superseded snapshot will be marked as Replaced. This whole period of evolution is condensed into a Change Line leading from the old reality to the current one, hiding obsolete details, as indicated in Figure 58.

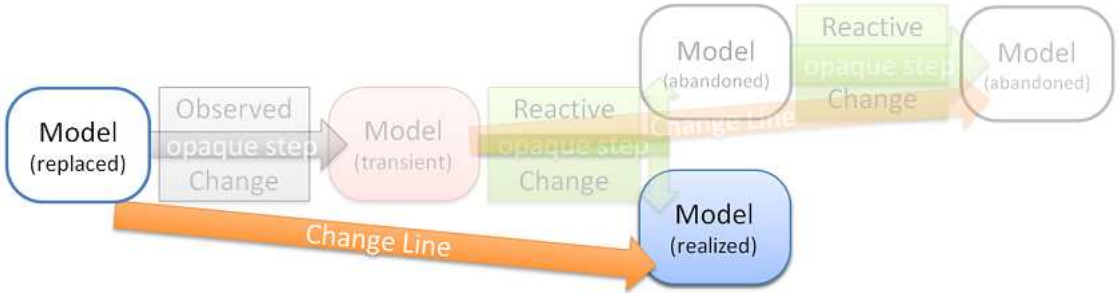


Figure 58 Change Line, phase three (after decision)

7.2 Definition of Change Control

Based on the generic notions of change introduced in Section 7.1, we now introduce our terminology regarding the mechanisms of change control. An UML Package representation of the concepts discussed here is presented on Figure 59.

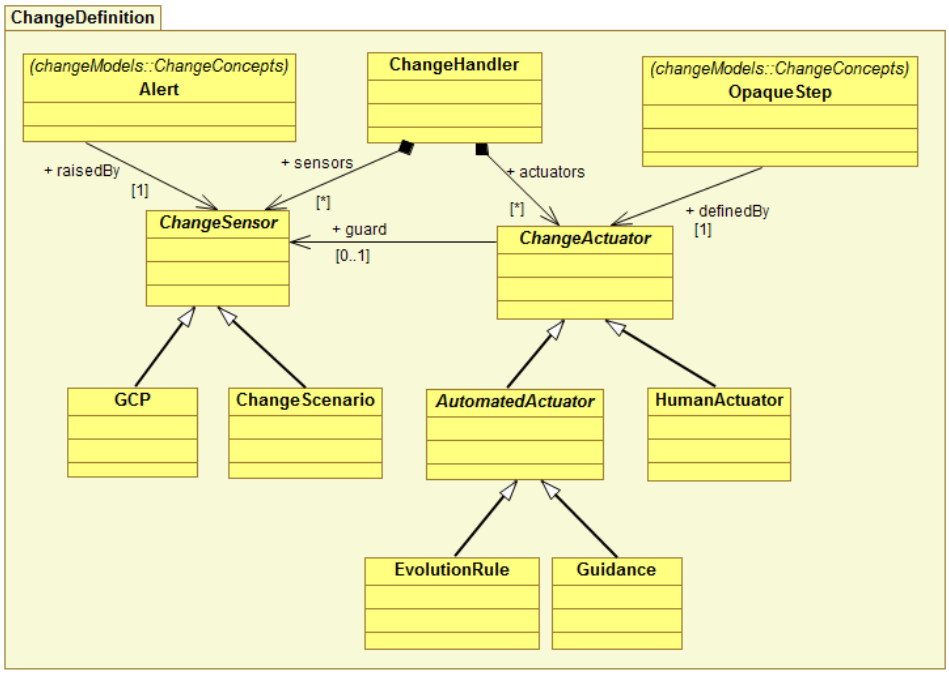


Figure 59 Definition of Change Control Facilities

7.2.1 Change Handler

A key concept of change control is **Change Handler**, which governs the changes over the lifetime of a system. The Change Handler is composed of the body of engineers of various domains (risk analysts, system architects, verification experts, etc.) and the entirety of automated mechanisms and other supportive technology at their disposal.

We use a control theory metaphor, in which the Change Handler entity acts as a controller, observing and reacting to Changes, as well as performing Changes itself. In accord with this metaphor, the two control mechanisms of great importance are called the sensors and actuators provided by the Change Handler.

7.2.2 Change Sensor

The role of **Change Sensor**s is to monitor Changes (Observed, Requested and Reactive alike) and Change Steps, compare the preState against the postState, filter the large volume of elementary Notifications; and raise high-level Alerts when security problems are detected. These Alerts can trigger Reactive Changes to correct the issues in the Transient model.

The Graph Change Pattern (GCP) formalism that will be introduced in Section 4.5, as well as the Change Scenario concept related to the Change Pattern approach of WP2 can be regarded as examples for automated Change Sensor formalisms.

7.2.3 Change Actuator

While Alerts produced by Change Sensors and other Change Events define *when* a Change is necessary, the role of a **Change Actuator** is to define *how* to change the model, *what* the proposed Change (Steps) should be. As discussed earlier, Change Steps can either be composite Change Lines or Opaque Steps; the contents of the latter can be defined by a Change Actuator. When providing Reactive behavior, a Change Actuator can be associated with a **guard** Change Sensor that raises the alerts that the actuator will react to.

Some Change Actuators are **Human Actuators**, reflecting decisions and proposals made by a board of engineers and experts. It is also possible to employ software components that can draft proposed modifications, typically specializing in Reactive interventions; these are **Automated Actuators**.

The Evolution Rules formalism that will be introduced in Section 4, as well as the Guidance concept related to the Change Pattern approach of WP2 can be regarded as examples for Automated Actuator formalisms.

7.3 Correspondence of Change Model Concepts

The various Work Packages of SecureChange use their own concepts to model change and to represent the aspects of change that are used in their methodology and perspective. Figure 60 shows the correspondence of the concepts used in various WPs (as well as the Thales change model presented in Appendix A.4) with the concepts of the generic model introduced here. The first column lists those concepts that were introduced here, but also have an exact or loose equivalent defined by another WP or the Thales model. The second column lists the numbers of WPs that have a similar concept (or T for Thales), and the respective local names are shown in column 3.

Generic Change Concept	WP	WP-specific Concept
Integrated Model	2 | T	Integrated Model | Static Model
Realised / Planned	2	realised / planned
Planned, Anticipated	2,5	~before-after perspective
Change Line	T	Change Line
revolution	4,5,7	Evolution/Revolution
Change	2	Change Transaction
Reactive	2,4,5,T	~ maintenance perspective
Observed	4	~ unplanned perspective
Observed / Requested	2	Change Log / Change Request
Change Focus	5	~ “Change Kind”
Change Step	T	Change
Change Event	2, T	Change Trigger / Change Event
securityDriven	4	“Change Kind”
Time Event	2	“time events”
Operation Event	2	“action events initiated by the stakeholders”
Notification	2	“change events caused by the modification/creation/deletion of some model element”
Alert	2	“conditions on the system state”
Change Control	2,4,5,T	~ continuous perspective
Change Sensor	T	Evolution Function

Figure 60 Correspondence of Change Concepts

8 Conclusions

In summary, this report has described SeCMER, a requirements engineering methodology for addressing evolutionary security goals. The approach is grounded in the Jackson-Zave framework and has three interleaving stages: requirements elicitation, argument analysis and requirements evolution.

For the requirements elicitation stage, we have proposed a meta-model of evolving security requirements, and a light weight process, aided by a tool, to support elicitation of requirements.

For the argument analysis stage, we have proposed a meta-model of arguments, which has been implemented in our OpenPF tool to support arguments visualization, formalization of arguments using proposition logical to check the validity of rebuttal and mitigation relationships between arguments, and formalization of arguments using the Event Calculus to reason about arguments using deductive and abductive reasoning.

For the requirement evolution stage, we have proposed a meta-model of evolution rules, and implement them in order to detect and apply incremental changes to the requirement models. The formalizing and computation aspects of the transformation have also been discussed.

We have discussed how these three stages relate to each other, and apply the whole methodology to a significant example from the ATM case study.

Furthermore, we have discussed how our methodology integrates with the process and architecture, risk, and design methodologies developed in the SecureChange project, whilst also providing a survey of how the notion of change is used in different SecureChange methodologies.

9 Acknowledgement

We thank external and interview reviewers of the SecureChange project for their insightful comments and constructive criticisms.

References

[1] Albert, L., *Average Case Complexity Analysis of Rete Pattern-Match Algorithm and Average Size of Join in Databases*, in *Foundations of Software Technology and Theoretical Computer Science*. 1989, Springer Berlin / Heidelberg. p. 223-241.

[2] Alferes, J.J., F. Banti, and A. Brogi. *An Event-Condition-Action Logic Programming Language*. in *10th European Conference Logics in Artificial Intelligence (JELIA)*. 2006. Liverpool, UK: Springer.

[3] Becker, S.M., T. Haase, and B. Westfechtel, *Model-Based a-Posteriori Integration of Engineering Tools for Incremental Development Processes.* Software and System Modeling, 2005. **4**(2): p. 123-140.

[4] Bergmann, G., A. Horvath, I. Rath, and D. Varro, *A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation*, in *Proceedings of the 4th international conference on Graph Transformations*. 2008, Springer-Verlag: Leicester, United Kingdom.

[5] Bergmann, G., A. Okros, I. Rath, D. Varro, and G. Varro, *Incremental Pattern Matching in the Viatra Model Transformation System*, in *Proceedings of the third international workshop on Graph and model transformations (GraMoT 2008)*. 2008, ACM: Leipzig, Germany.

[6] Bergmann, G., I. Rath, G. Varro, and D. Varro, *Change-Driven Model Transformations. Change (in) the Rule to Rule the Change.* Software and System Modeling (under review).

[7] Bezivin, J., *On the Unification Power of Models.* Software and System Modeling, 2005. **4**(2): p. 171-188.

[8] Czarnecki, K. and S. Helsen, *Feature-Based Survey of Model Transformation Approaches.* IBM Systems Journal, 2006. **45**(3): p. 621-645.

[9] Ehrig, H., G. Engels, F. Parisi-Presicce, G. Rozenberg, and A. Rensink, *Representing First-Order Logic Using Graphs*, in *Graph Transformations*. 2004, Springer Berlin / Heidelberg. p. 187-190.

[10] Forgy, C.L., *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*, in *Expert Systems*, G.R. Peter, Editor. 1990, IEEE Computer Society Press. p. 324-341.

[11] Gangemi, A., N. Guarino, C. Masolo, A. Oltramari, and L. Schneider, *Sweetening Ontologies with Dolce*, in *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*. 2002, Springer-Verlag.

[12] Gerber, A., M. Lawley, K. Raymond, J. Steel, and A. Wood, *Transformation: The Missing Link of Mda*, in *Proceedings of the First International Conference on Graph Transformation*. 2002, Springer-Verlag.

[13] Gunter, C.A., E.L. Gunter, M. Jackson, and P. Zave, *A Reference Model for Requirements and Specifications.* IEEE Softw., 2000. **17**(3): p. 37-43.

[14] Haley, C., R. Laney, J. Moffett, and B. Nuseibeh, *Security Requirements Engineering: A Framework for Representation and Analysis.* IEEE Trans. Softw. Eng., 2008. **34**(1): p. 133-153.

[15] Jackson, M., *Problem Frames: Analyzing and Structuring Software Development Problems*. 2001: Addison-Wesley Longman Publishing Co., Inc.

[16] Kowalski, R. and M. Sergot, *A Logic-Based Calculus of Events.* New Gen. Comput., 1986. **4**(1): p. 67-95.

[17] Lamsweerde, A.V., *Goal-Oriented Requirements Engineering: A Guided Tour*, in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*. 2001, IEEE Computer Society.

[18] Lin, L., B. Nuseibeh, D. Ince, and M. Jackson, *Using Abuse Frames to Bound the Scope of Security Problems*, in *Proceedings of the Requirements Engineering Conference, 12th IEEE International*. 2004, IEEE Computer Society.

[19] Lin, L., B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett, *Introducing Abuse Frames for Analysing Security Requirements*, in *Proceedings of the 11th IEEE International Conference on Requirements Engineering*. 2003, IEEE Computer Society.

[20] Massacci, F., J. Mylopoulos, and N. Zannone, *Computer-Aided Support for Secure Tropos.* Automated Software Engineering, 2007. **14**(3): p. 341-364.

[21] Miller, R. and M. Shanahan, *The Event Calculus in Classical Logic - Alternative Axiomatisations.* Electron. Trans. Artif. Intell., 1999. **3**(A): p. 77-105.

[22] Nhlabatsi, A., B. Nuseibeh, and Y. Yu, *Security Requirements Engineering for Evolving Software Systems: A Survey.* International Journal of Secure Software Engineering (IJSSE), 2010. **1**(1): p. 54–73.

[23] Rath, I., G. Bergmann, A. Okros, and D. Varro, *Live Model Transformations Driven by Incremental Pattern Matching*, in *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*. 2008, Springer-Verlag: Zurich, Switzerland.

[24] Shanahan, M., *The Event Calculus Explained*, in *Artificial Intelligence Today*, J.W. Michael and V. Manuela, Editors. 1999, Springer-Verlag. p. 409-430.

[25] Tun, T.T., R. Chapman, C. Haley, R. Laney, and B. Nuseibeh, *A Framework for Developing Feature-Rich Software Systems*, in Proceedings of the 2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems. 2009, IEEE Computer Society.

[26] Wang, Y.-W. and E.N. Hanson, *A Performance Comparison of the Rete and Treat Algorithms for Testing Database Rule Conditions*, in *Proceedings of the Eighth International Conference on Data Engineering*. 1992, IEEE Computer Society.

[27] Yu, E.S.K. and J. Mylopoulos, *Understanding "Why" in Software Process Modelling, Analysis, and Design*, in *Proceedings of the 16th international*

conference on Software engineering. 1994, IEEE Computer Society Press: Sorrento, Italy.

[28] Zave, P. and M. Jackson, *Four Dark Corners of Requirements Engineering.* ACM Trans. Softw. Eng. Methodol., 1997. **6**(1): p. 1-30.

Glossary

A *claim* is a (probably grounded) predicate whose truth-value will be established by an argument, 30

A goal is a concept found in GORE approaches, and represents a proposition an actor wants to make true, 26

A *proposition* is an object representing a true/false statement, 25

A *resource* is an entity without intention or behavior, 26

A *situation* is a partial state of the world described by a proposition (its description in [11]), 25

An *action* is an entity performed by an actor, which can generate events, and can have preconditions and post-conditions, 26

An *actor* is an entity that can act and intend to want or desire, 26

An *argument* contains one and only one claim. It also contains facts and rules in domain knowledge, 30

An *asset* is an entity of value that can be owned and used, 26

Domain is specialized into *Actor, Action, Asset,* and *Resource*, 26

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, 31

Facts are grounded predicates -- something that is either true or false where terms in these predicate must be constant, 30

A. Appendix: State of the Practice

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

A.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 61 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 over the engineering problem:

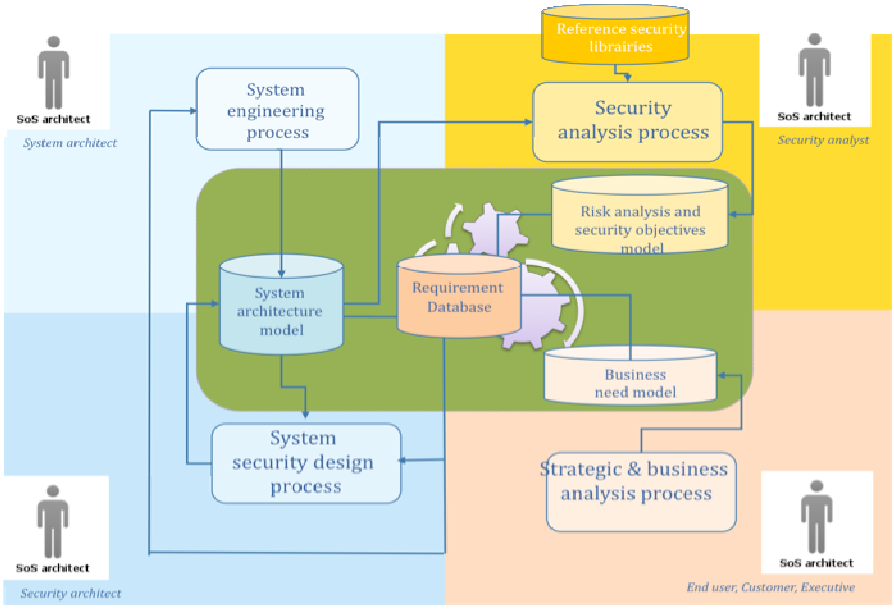


Figure 61 The security analysis method in Thales context – big picture

- 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 XXX). The security risk analysis information is defined as annotations or related new concepts added over the system architecture elements. The risk analysis model and security objectives model may also be traced to elements of information defined in the Business need model.
- The Requirement Database captures all kinds of systems requirements (Security, Safety, Maintainability, Cost, etc.). Security goals are derived from security objectives model of dedicated DSML (see XXX). This mapping enables to add security goals with other kind of requirement addressed for a complex system. Requirement Database is traced back and maintained in synchronisation with the system architecture model and Business need model.

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

A.2. DOORS T-REK

Rational DOORS XXX (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 62, 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.

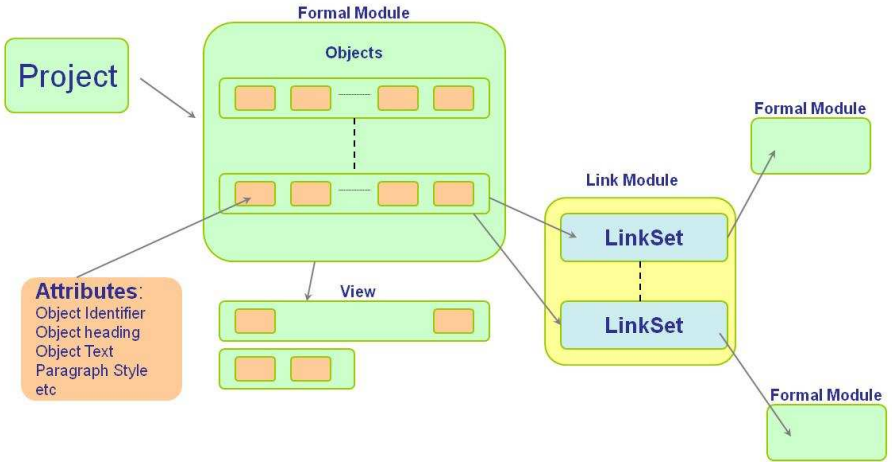


Figure 62 DOORS project structure

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

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

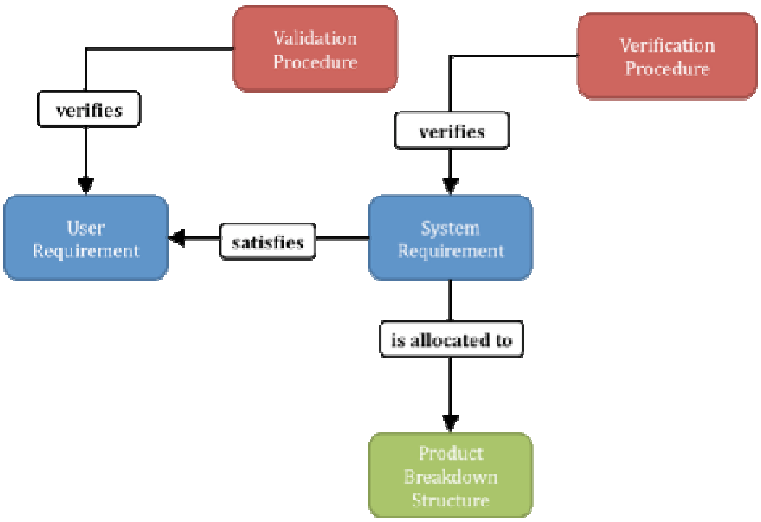


Figure 63 Simplified Datamodel in T-REK

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.

A.3. APPLICATION IN THALES REQUIREMENT WORKBENCH

This deliverable cannot be the place for a detailed presentation of the conceptual model and syntax of DSML. We are providing below representative extracts. More details are provided in XXX. The core part of the conceptual model[6] is represented in Figure 64.

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

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

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

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

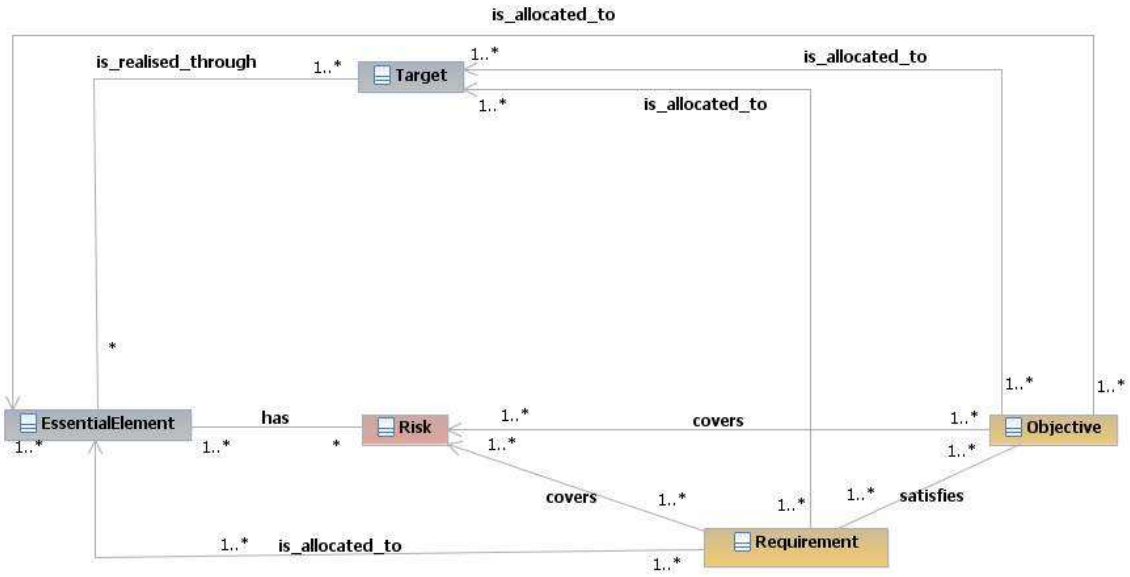


Figure 64 Conceptual model of Security Objectives and Requirements in Security DSML

In current Security DSML, we distinguish three kinds of static models[7] as shown by Figure 65:

- ***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).

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

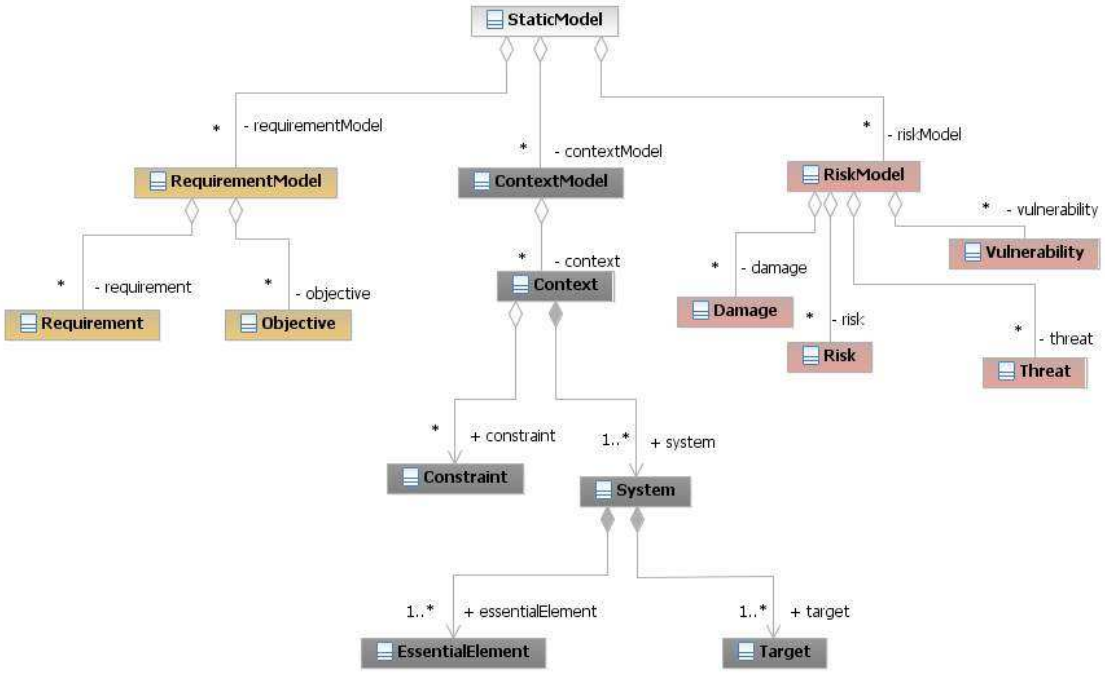


Figure 65 Security DSML Static Model description

Figure 66 shows how to realize the mapping between Thales Security DSML (or Other DSML for Need Analysis) and DOORS T-REK, to do this we must consider a **Traceability relation** between Security Goal of Security DSML and DOORS Requirements.

This relation enables to connect other kind of requirement (Safety, Maintainability, Cost, etc.) with Security Goals expressed in DSML. Requirements are stored in a common requirement Database (DOORS Database). This communication is realized via a Model Bus (Bidirectional interface XML to DXL[8]) for Traceability needs between DOORS and Security DSML.

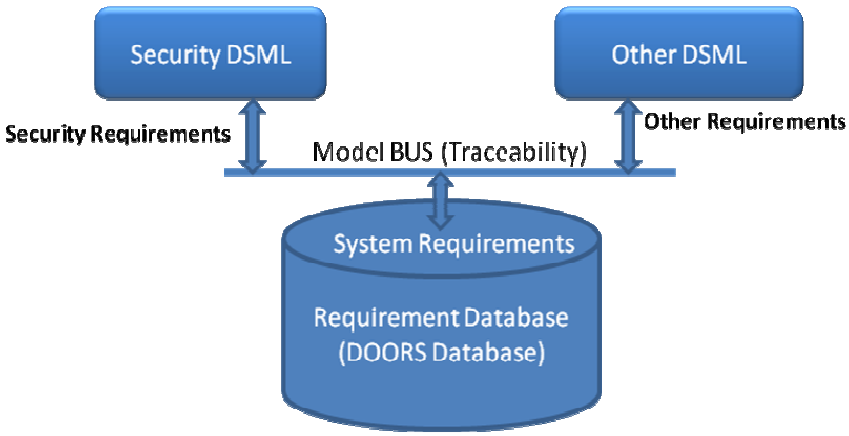


Figure 66 Mapping between DSML and DOORS

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

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

Figure 67 presents the extended conceptual model including DOORS connections. Two kinds of entities are mapped with DOORS: Requirements and Target that are respectively represented by Requirement and Product Breakdown Structure object in DOORS. To ensure traceability between DSML and DOORS, we add a PUID (Product Unique IDentifier) attribute, PUID is the reference name of a DOORS object.

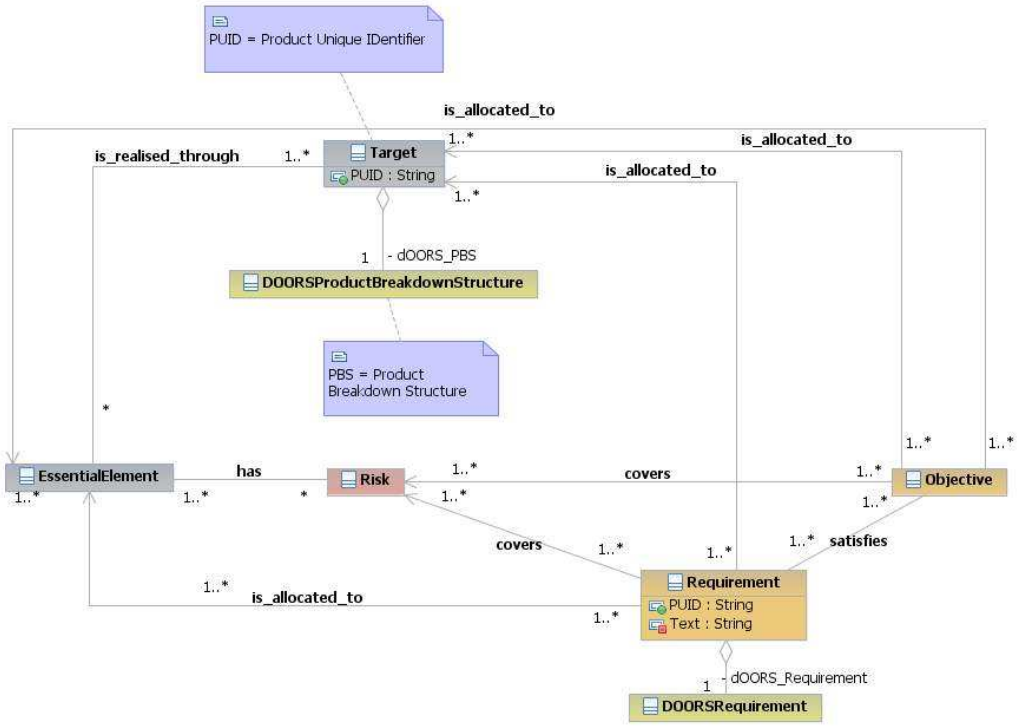


Figure 67 Extended Conceptual model including DOORS connections

Figure 68 depicts the properties view on Security Objective O6 (Identifiers should be chosen so that they do not compromise user's privacy). Figure 69 presents the requirement derived from security objective in DOORS.

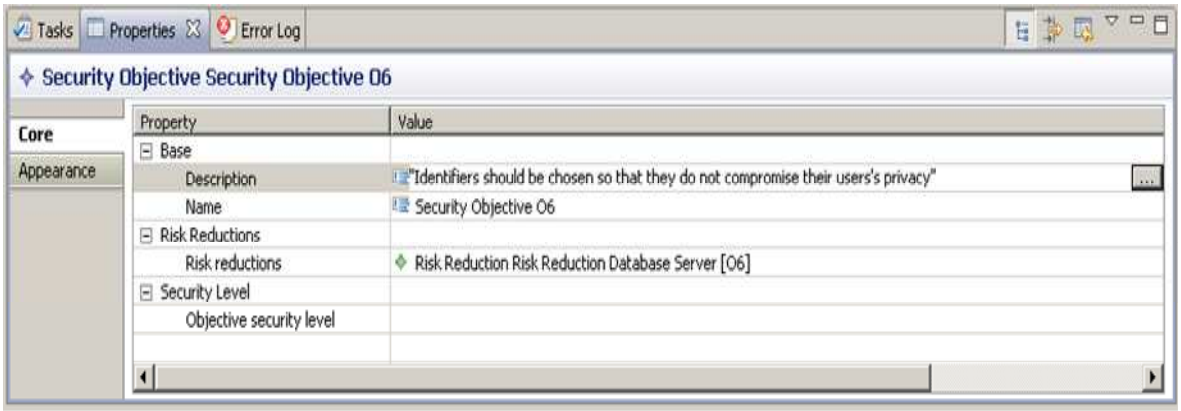


Figure 68 Close view on the Security Objectives

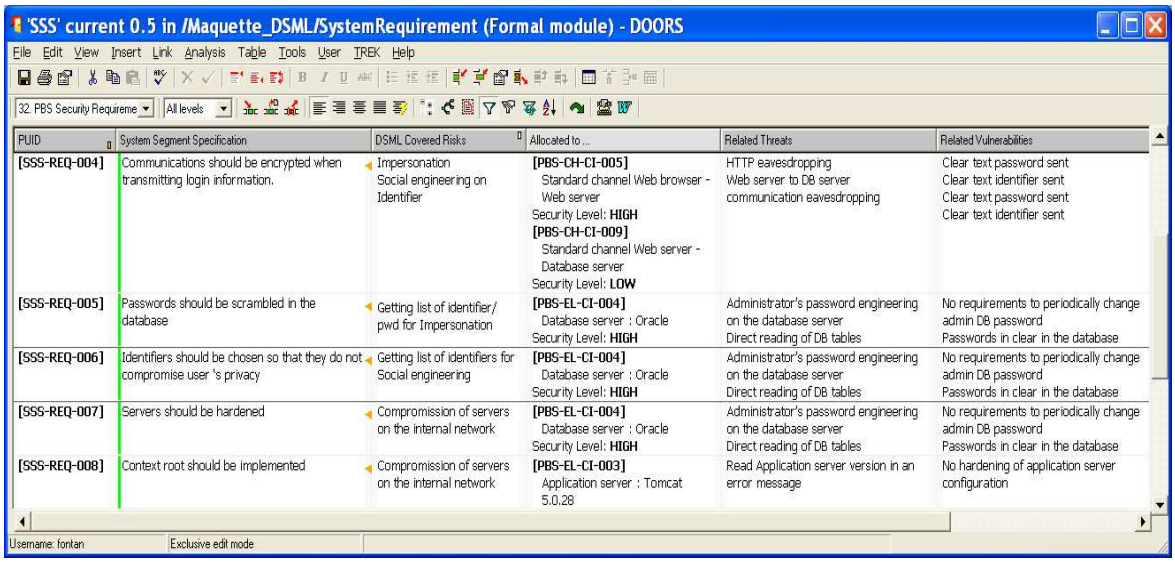


Figure 69 Derived Requirements expressed in DOORS

The information of target can be consulted in the Properties View (Description, constraints applied on it), as can be seen in Figure 70. This properties view of Target is also defined in DOORS as shown by Figure 71.

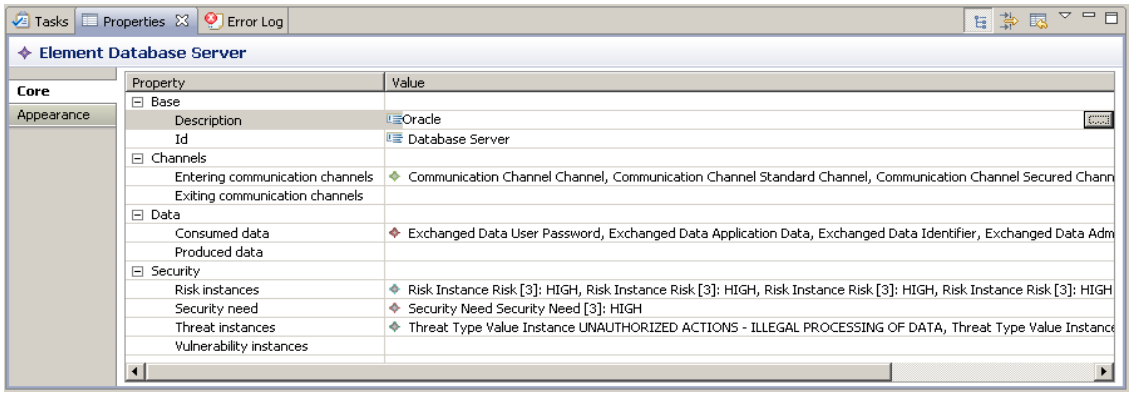


Figure 70 Properties of the Database Server in DSML

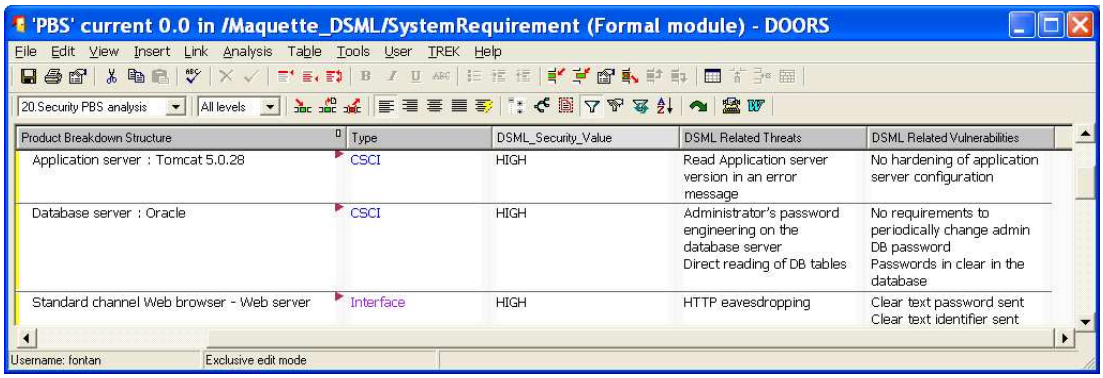


Figure 71 Database Server description in DOORS

A.4. THE THALES CHANGE MODEL

This Section introduces the industrial Change Model used by Thales. The concepts discussed here provided partial inspiration for the proposed generic change model in Section 7.1, as well as the change control model in 7.2. At certain points in the text we indicate the equivalent terminology in the Generic Change Model of WP3 (Section 7); a detailed table of corresponding concepts is located in Section 7.3.

A.5. CHANGELINE CONCEPTUAL MODEL

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

To represent traceability between changes and versioning of change, Thales has a further conceptual model: a ***Change Model*** is composed by several ***Change Lines***. A ***Change Line*** is considered as set of ***Changes*** and ***Change Transitions*** to preserve links and grant consistency between successive changes which compose a Change Line. ***Change*** is caused by a **Change Trigger** (e.g. discovery of a fault or a new threat); this concept corresponds to Change Event defined in Section 7. The 'contents' of the change is documented in a ***Change Request***, that describes what and how should be changed; in the terminology of the SecureChange change model in Section 7, this is equivalent to the Opaque Change Step of a Requested Change as defined by a Human Actuator. It's possible to activate a Change Trigger by a threshold defined in an ***Evolution Function*** (Change Sensor in the terminology of Section 7) which monitors the static model of the system. Evolution functions enable to represent Continuous Perspective of change. Change Lines enable to represent both the maintenance perspective and the before-after perspective.

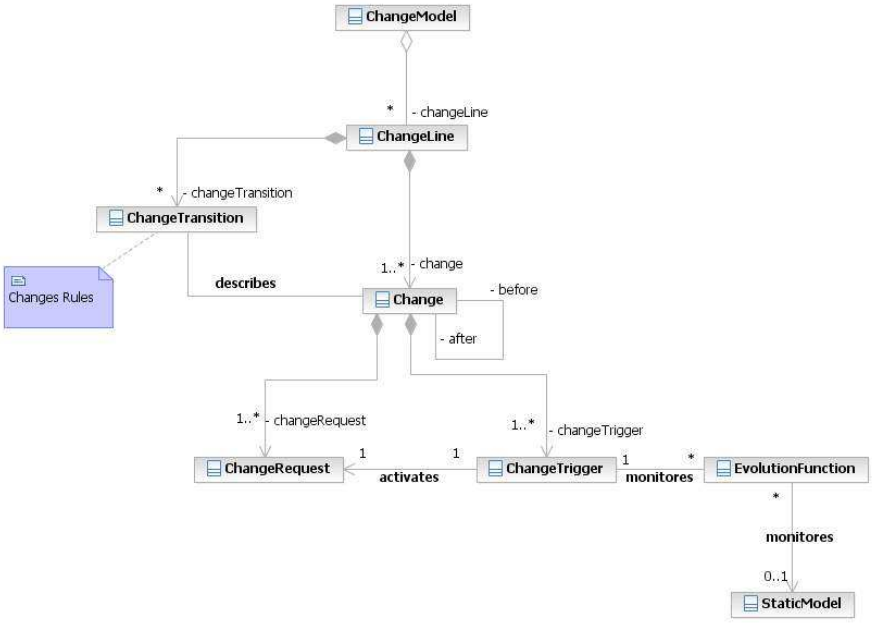


Figure 72 DSML Change Model conceptual model

A.6. CHANGEREQUEST CONCEPTUAL MODEL

As shown by Figure 72, a ***Change Request*** contains a PUID to identify it and a status representing the state of Change request. After the activation of Change Request by the Change Trigger, Change Request status is first defined in CCB (Configuration Control Board). The configuration (or change) control board (CCB) is a meeting between all actors of a development team (client, manager, quality, design, integration, …) to define the change request status (e.g. accepted, refused or postponed in the next version of system). The detailed behavior of Requirement Change Request is described in next section.

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

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

These three kinds of Change Request are dependants; a Requirement Change Request could impact on Risk Change Request and Context Change Request and vice versa. This is why we consider a traceability relation between those Change Requests. This relation is described by an association called "impacts_on" (see Figure 73).

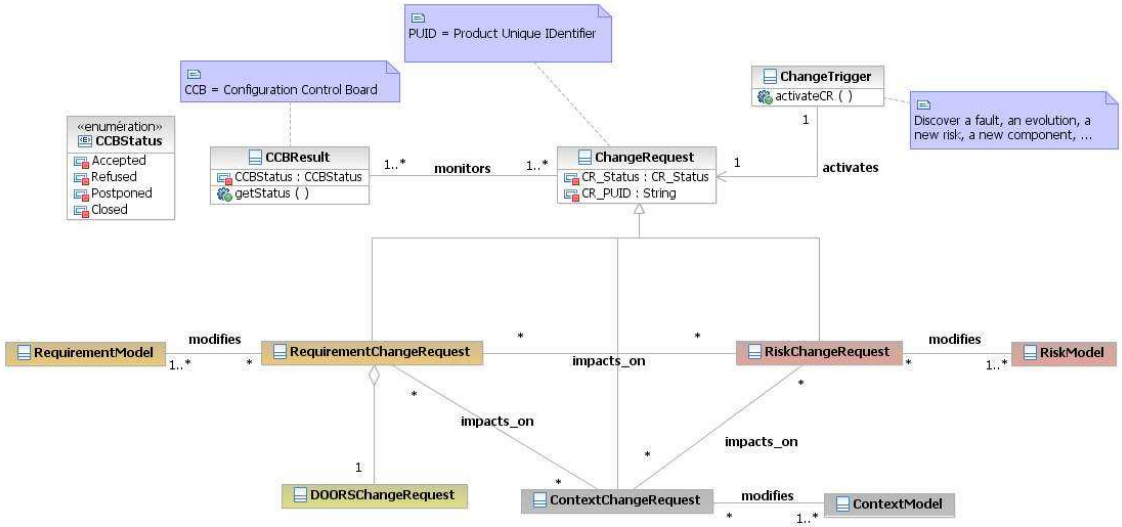


Figure 73 DSML Change Request Conceptual model

A.7. BEHAVIOR OF CHANGE REQUEST

For the sake of readability, the generic Change Request Behavior is described by UML Statechart Diagram (see Figure 74a). We present on the one hand the generic behavior of Change Request including CCB status relations. On the second hand we describe the specific behavior of Requirement Change Request.

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

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

- ***Refused***, CR is not relevant; it is not integrated in system. Change Request is ended in this state. In SecureChange terminology (Section 7), the Planned model version becomes Abandoned.
- ***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. In SecureChange terminology, this is not distinguished from the case where the CR is refused and later an identical CR is accepted,since the focus is on managing the changes of the model and not the CR process.
- ***Accepted***, CR is integrated in current version of system. In SecureChange terminology (Section 7), the Planned model becomes Realized.

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

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

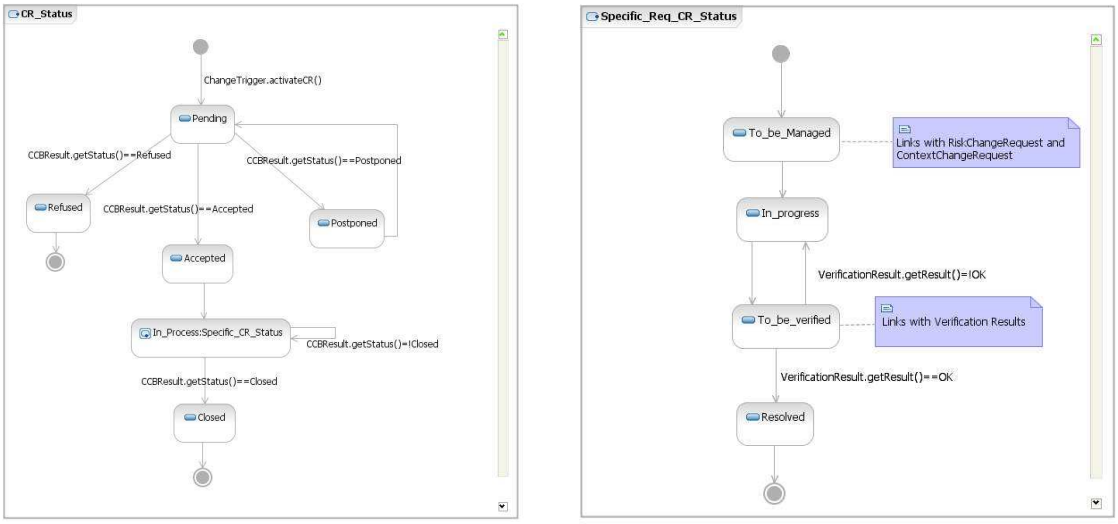


Figure 74 Change Request Status Behavior (a) generic (b) requirements-specific

Specific ***Requirement Change Request*** (RCR) Behavior starts after ***Accepted*** state in generic behavior. As shown by Figure 74b, 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.