

PROPRIETARY RIGHTS STATEMENT

THIS DOCUMENT CONTAINS INFORMATION, WHICH IS PROPRIETARY TO THE CRYSTAL CONSORTIUM. NEITHER THIS DOCUMENT NOR THE INFORMATION CONTAINED HEREIN SHALL BE USED, DUPLICATED OR COMMUNICATED BY ANY MEANS TO ANY THIRD PARTY, IN WHOLE OR IN PARTS, EXCEPT WITH THE PRIOR WRITTEN CONSENT OF THE CESAR CONSORTIUM THIS RESTRICTION LEGEND SHALL NOT BE ALTERED OR OBLITERATED ON OR FROM THIS DOCUMENT. THE RESEARCH LEADING TO THESE RESULTS HAS RECEIVED FUNDING FROM THE EUROPEAN UNION'S SEVENTH FRAMEWORK PROGRAM (FP7/2007-2013) FOR CRYSTAL – CRITICAL SYSTEM ENGINEERING ACCELERATION JOINT UNDERTAKING UNDER GRANT AGREEMENT N° 332830 AND FROM SPECIFIC NATIONAL PROGRAMS AND / OR FUNDING AUTHORITIES.



CRritical **SY**STem Engineering **Acce**Leration

First MSE SEE (Prototype)

D203.020

DOCUMENT INFORMATION

Project	CRYSTAL
Grant Agreement No.	ARTEMIS-2012-1-332830
Deliverable Title	First MSE SEE (Prototype)
Deliverable No.	D203.020
Dissemination Level	CO
Nature	R
Document Version	V1.00
Date	2014-02-10
Contact	Ralf BOGUSCH
Organization	EADS-CAS
Phone	+ 49 7545.8-2745
E-Mail	ralf.bogusch@cassidian.com

AUTHORS TABLE

Name	Company	E-Mail
Ralf BOGUSCH	EADS-CAS	ralf.bogusch@cassidian.com

REVIEW TABLE

Version	Date	Reviewer
V0.10	2014-01-31	Inga BINDER (EADS-CAS)
V0.20	2014-02-05	Frédéric AUTRAN (EADS-CAS), Adeline SCHÄFER (FhG), Herbert KLENK (EADS-CAS), Marc MALOT (SAGEM), Andreas MITSCHKE (EADS IW-G)

CHANGE HISTORY

Version	Date	Reason for Change	Pages Affected
V0.10	2014-01-31	Initial Version for internal review	All
V0.20	2014-02-05	Version for external review	All
V1.00	2014-02-10	Version for release	

CONTENT

1	INTRODUCTION	9
1.1	ROLE OF THE DELIVERABLE	9
1.2	RELATIONSHIP TO OTHER CRYSTAL DOCUMENTS	9
1.3	STRUCTURE OF THIS DOCUMENT	9
2	OVERVIEW OF THE SEE PROTOTYPE	11
2.1	LANDING SYMBOLOGY FUNCTION	11
2.2	SUPPORTED USAGE SCENARIOS	12
2.3	CREATED LIFECYCLE DATA AND USED TOOLS.....	13
3	DESCRIPTION OF THE TOOL CHAIN	15
3.1	OVERVIEW	15
3.2	VEDIT	15
3.3	FEATUREIDE	16
3.4	DOORS	16
3.5	RQA.....	17
3.6	RAT	17
3.7	κM.....	18
3.8	RHAPSODY.....	18
3.9	RPE	19
3.10	EPF COMPOSER.....	19
4	DESCRIPTION OF THE USAGE SCENARIOS	20
4.1	OVERVIEW	20
4.2	SCENARIO SC1 – DEFINE PRODUCT FAMILY SCOPE AND VARIABILITY MODEL	21
4.2.1	Step 1.1 – Define product family scope.....	22
4.2.2	Step 1.2 – Create variability model.....	22
4.3	SCENARIO SC2 – DEVELOP DOMAIN SYSTEM REQUIREMENTS	24
4.3.1	Step 2.1 – Define structure of the requirements repository.....	25
4.3.2	Step 2.2 – Define data model for requirements modules	25
4.3.3	Step 2.3 – Create requirements.....	26
4.3.4	Step 2.4 – Allocate requirements to features.....	27
4.4	SCENARIO SC3 – ANALYZE AND IMPROVE REQUIREMENTS QUALITY	28
4.4.1	Step 3.1 – Select requirements module.....	30
4.4.2	Step 3.2 – Configure quality analysis for requirements module	31
4.4.3	Step 3.3 – Define ontologies.....	34
4.4.4	Step 3.4 – Analyze quality of requirements module	34
4.4.5	Step 3.5 – Provide findings.....	36
4.4.6	Step 3.6 – Create requirements quality report.....	38
4.4.7	Step 3.7 – Improve requirements	39
4.5	SCENARIO SC4 – CREATE PRODUCT SYSTEM REQUIREMENTS.....	39
4.5.1	Step 4.1 – Configure product.....	39
4.5.2	Step 4.2 – Create product system requirements	41
4.6	SCENARIO SC5 – PERFORM SYSTEM FUNCTIONAL ANALYSIS	42
4.6.1	Step 5.1 – Create initial model setup.....	43
4.6.2	Step 5.2 – Import input requirements to the model	44

4.6.3	Step 5.3 – Define system context and system-level use cases.....	45
4.6.4	Step 5.4 – Allocate requirements to use cases	46
4.6.5	Step 5.5 – Analyze the use case uninterrupted flow	47
4.6.6	Step 5.6 – Define black box scenarios	47
4.6.7	Step 5.7 – Create system external ports and interfaces.....	48
4.6.8	Step 5.8 – Define state-based behaviour	49
4.6.9	Step 5.9 – Verify model by model execution	50
4.7	SCENARIO SC6 – PERFORM REPORT GENERATION	55
4.7.1	Step 6.1 – Define template for document generation.....	56
4.7.2	Step 6.2 – Create document.....	56
4.8	SCENARIO SC7 – PROVIDE PROCESS GUIDANCE.....	56
4.8.1	Step 7.1 – Create method contents and practice library	59
4.8.2	Step 7.2 – Define delivery process.....	60
4.8.3	Step 7.3 – Publish delivery process.....	60
5	CONCLUSIONS AND WAY AHEAD.....	63
5.1	PRELIMINARY EVALUATION AND PLANNED FUTURE WORK	63
5.1.1	US202 – Safety Analysis	63
5.1.2	US203 – Variability Management	63
5.1.3	US204 – Ontology-based Requirements Engineering.....	63
5.1.4	US205 – Process Automation, Guidance and Monitoring	64
5.1.5	US206 – Project compliance monitoring based on advanced traceability.....	64
5.2	ENVISAGED SEE	65
6	TERMS, ABBREVIATIONS AND DEFINITIONS	67
6.1	ABBREVIATIONS.....	67
6.2	GLOSSARY	69
7	REFERENCES.....	74

List of Figures

Figure 2-1 Degraded visual environments.....	11
Figure 2-2 Display of the Landing Symbolology.....	11
Figure 2-3 Overview of the supported usage scenarios.....	12
Figure 2-4 Overview of the integrated tool chain.....	14
Figure 3-1 Ontology layers [CRYSTAL D607.041].....	18
Figure 4-1 Define the scope of the product family with product feature matrix.....	22
Figure 4-2 Create variability model using OVM.....	23
Figure 4-3 Create variability model using feature trees.....	24
Figure 4-4 Define the structure of the RM repository	25
Figure 4-5 Define the data model for requirements modules	26
Figure 4-6 Create requirements	27
Figure 4-7 Allocate requirements to features	28
Figure 4-8 Select a requirements module for analysis	31
Figure 4-9 Configure quality metrics in RQA	33
Figure 4-10 Configure vague phrases in RQA	33
Figure 4-11 Define new boilerplates in kM	34
Figure 4-12 Analyze requirements quality of requirements set.....	35
Figure 4-13 Identify requirements that need to be improved.....	36
Figure 4-14 Provide the quality summary for a requirement	37
Figure 4-15 Analyze the findings for a requirement	37
Figure 4-16 Create a quality report.....	38
Figure 4-17 Store the quality evaluation results in DOORS	38
Figure 4-18 Improve requirement using RAT	39
Figure 4-19 SferiAssist300 configuration.....	40
Figure 4-20 SferiAssist500 configuration.....	40
Figure 4-21 SferiAssist300 requirements	41
Figure 4-22 SferiAssist500 requirements	42
Figure 4-23 Model setup in Rhapsody.....	44
Figure 4-24 Import input requirements from DOORS to Rhapsody using Rhapsody Gateway	45
Figure 4-25 Define system context and system level use cases (use case diagram)	46
Figure 4-26 Allocate system requirements to system-level use cases (use case diagram)	46
Figure 4-27 Analyze the use case uninterrupted flow (activity diagram).....	47
Figure 4-28 Define black box scenarios (sequence diagram).....	48
Figure 4-29 Create system external ports and interfaces (internal block diagram)	49
Figure 4-30 Define state-based behaviour (state chart diagram).....	50
Figure 4-31 Enhanced model setup using UML testing profile	51
Figure 4-32 Generate test context for the system under test (internal block diagram)	52
Figure 4-33 Define test scenarios for the system under test (sequence diagram)	53

Version	Nature	Date	Page
V1.00	R	2014-02-10	6 of 74

Figure 4-34 Execute tests in Rhapsody using simulation of state-based behaviour.....	54
Figure 4-35 Analyze model-based coverage of test cases	55
Figure 4-36 Define the RPE template for document export in RPE Document Studio	56
Figure 4-37 Define method contents using EPF Composer.....	59
Figure 4-38 Define delivery process using EPF Composer	60
Figure 4-39 View published method contents	61
Figure 4-40 View published practice including additional guidance.....	62
Figure 5-1 Envisaged SEE	66



List of Tables

Table 2-1 Overview of the created lifecycle data and the used tools 13

Table 3-1 Tools integrated in the SEE prototype 15

Table 4-1 Relation of usage scenarios with user stories and engineering methods 20

Table 6-1 Abbreviations 69

Table 6-2 Terms 73

Table 7-1 References 74

1 Introduction

1.1 Role of the Deliverable

This document has the following major purposes:

- Describe the model-based systems engineering approach and the technical status of the implemented Systems Engineering Environment (SEE) prototype for the Mission Support Equipment (MSE) use case:
 - a. Provide an overview of the current tool chain of the SEE prototype
 - b. Describe the usage of the tool chain in the frame of process activities and engineering methods covered by the SEE prototype
 - c. Exemplify the artefacts created for the MSE use case, e.g. system requirements and model entities
 - d. Describe the envisaged SEE and planned future work
- Provide input to WP601 (IOS Development) required to derive specific IOS-related requirements
- Provide input to WP602 (Platform Builder) required to derive adequate meta models
- Establish the technology baseline with respect to the MSE use case, and the expected progress beyond, i.e. existing functionalities vs. functionalities that are expected to be developed in CRYSTAL.

1.2 Relationship to other CRYSTAL Documents

This document is the first in a series of three reports:

- D203.020 – First MSE SEE (Prototype) – this document
- D203.030 – Enhanced MSE SEE
- D203.040 – Final MSE SEE

The description of the SEE prototype is linked to D203.011, which provides a detailed definition of the process activities and engineering methods, selected to support the model-based systems engineering paradigm adopted in the MSE use case:

- D203.011 – MSE Report V1

1.3 Structure of this Document

This document is composed of three main chapters:

- Chapter 1 gives an overview of the scope of the deliverable, relationship with other CRYSTAL documents and this description of the document structure.
- Chapter 2 provides an introduction to the SEE prototype. It briefly describes the Landing Symbolology function, which is used for prototyping and evaluation purposes. Further, it provides an overview of the usage scenarios covered by the prototype as well as the created lifecycle data and the related tool chain.

Version	Nature	Date	Page
V1.00	R	2014-02-10	9 of 74

-
- Chapter 3 describes the tool chain of the SEE prototype in detail. This includes descriptions of the individual tool functions.
 - Chapter 4 provides in-depth descriptions of the usage scenarios of the prototype that exemplify the use of the SEE prototype in the frame of the related process activities and engineering methods as defined in D203.011.
 - Chapter 5 provides the conclusions including an initial assessment of the current technical status. It describes the envisaged SEE and the planned way ahead.

2 Overview of the SEE Prototype

2.1 Landing Symbology Function

For demonstration and evaluation purposes the Landing Symbology function, which is part of a situational awareness suite Sferion™, is applied in this use case. The Landing Symbology supports helicopter pilots during the final landing approach in degraded visual environments which can be caused by e.g. rain, fog, sand, dust and snow (see **Figure 2-1**). Many accidents can be directly attributed to such degraded visual environments where pilots often loose spatial and environmental orientation.



Figure 2-1 Degraded visual environments

The Landing Symbology function allows to mark the landing point on ground using a head-tracked helmet mounted display. During the final landing approach it enhances the spatial awareness of flying crews by displaying 3D conformal visual cues on a helmet-mounted display (see **Figure 2-2**). In addition it employs a surface grid conformal to the measured terrain for the landing area.

The Landing Symbology function provides the following functionality:

- Display 3D conformal visual cues on a helmet mounted display visualizing the helicopter attitude and position relative to the intended landing point.
- Determine and visualize the condition of the anticipated landing zone with respect to roughness and slope based on real time 3D data.
- Display obstacles on a helmet mounted display relevant for the start and landing phase. The obstacles are taken from the real-time obstacle fusion, thus considering obstacles from the obstacle data bases and from real-time sensor obstacle classification.

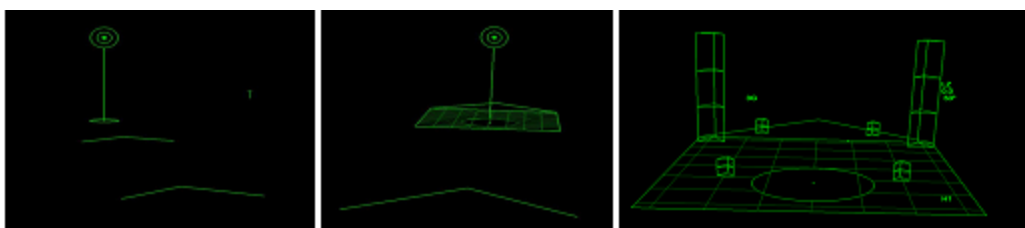


Figure 2-2 Display of the Landing Symbology

Version	Nature	Date	Page
V1.00	R	2014-02-10	11 of 74

2.2 Supported Usage Scenarios

Usage scenarios illustrate how the tool chain of the SEE prototype can be used in order to perform certain engineering activities. In the current status of the SEE prototype, the early phases of systems engineering are covered. This includes the following usage scenarios:

- SC1 – Define Product Family Scope and Variability Model
- SC2 – Develop Domain System Requirements
- SC3 – Analyze and Improve Requirements Quality
- SC4 – Create Product System Requirements
- SC5 – Perform System Functional Analysis

Two further usage scenarios are transversal, i.e. they may be used at any time of the development lifecycle:

- SC6 – Perform Report Generation
- SC7 – Provide Process Guidance

Figure 2-3 provides an overview of the usage scenarios and relates them to the user stories covered by the SEE prototype. The user stories are introduced in [CRYSTAL D203.011].

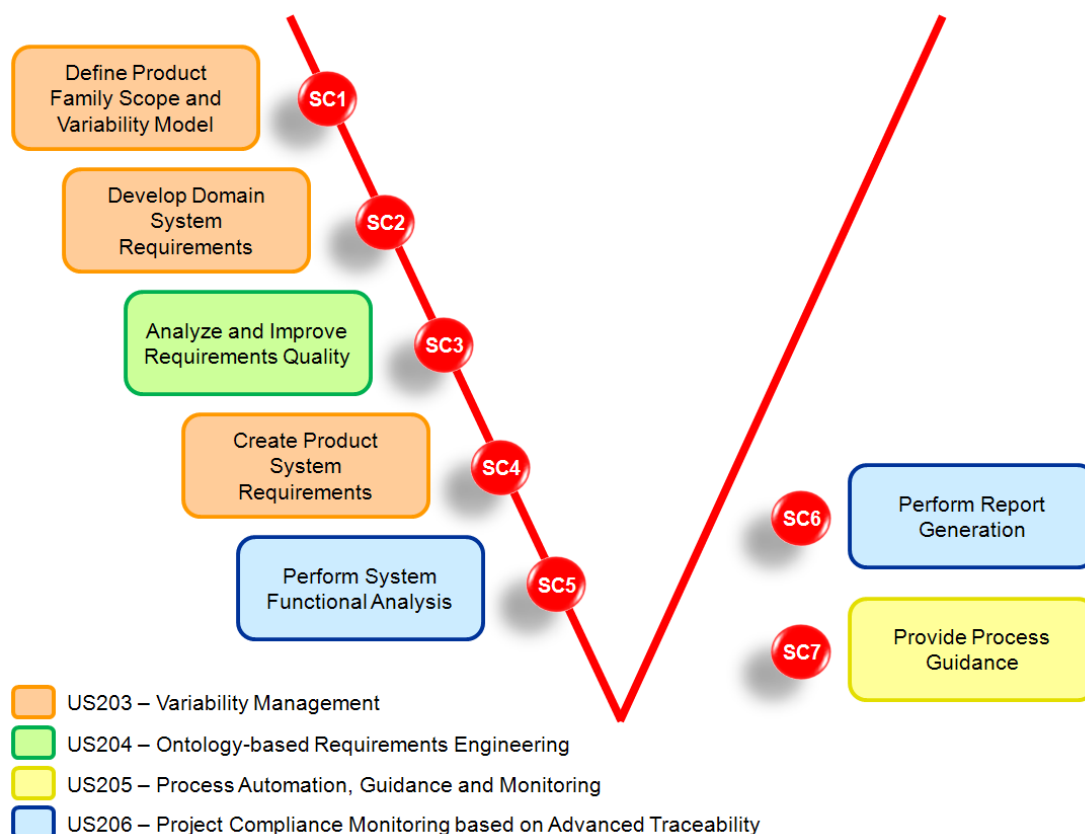


Figure 2-3 Overview of the supported usage scenarios

2.3 Created Lifecycle Data and Used Tools

For the Landing Symbology function a data set has been created, which will be enhanced in future demonstrator versions. The following table lists the created artefacts and the related tools:

Usage Scenario	Artefacts	Tools
SC1 – Define Product Family Scope and Variability Model	Variability models	FeatureIDE, Vedit
SC2 – Develop Domain System Requirements	Top-level system requirements for product family	DOORS
SC3 – Analyze and Improve Requirements Quality	Quality metrics, Boilerplates	DOORS, RQA, RAT, kM
SC4 – Create Product System Requirements	Top-level system requirements for product variant	DOORS
SC5 – Perform System Functional Analysis	Top-level functional analysis model including test cases	DOORS, Rhapsody
SC6 – Perform Report Generation	Template for report generation	RPE
SC7 – Provide Process Guidance	Method contents describing the systems engineering process	EPF Composer

Table 2-1 Overview of the created lifecycle data and the used tools

Figure 2-4 shows the SEE prototype setup as of January 2014. The current prototype is mainly based on IBM Rational Software products and the Requirements Quality Suite from The Reuse Company. Colours illustrate how the individual tools are related with the user stories introduced in [CRYSTAL D203.011].

The individual tools are described in chapter 3. The created lifecycle data is illustrated in the description of the usage scenarios in chapter 4.

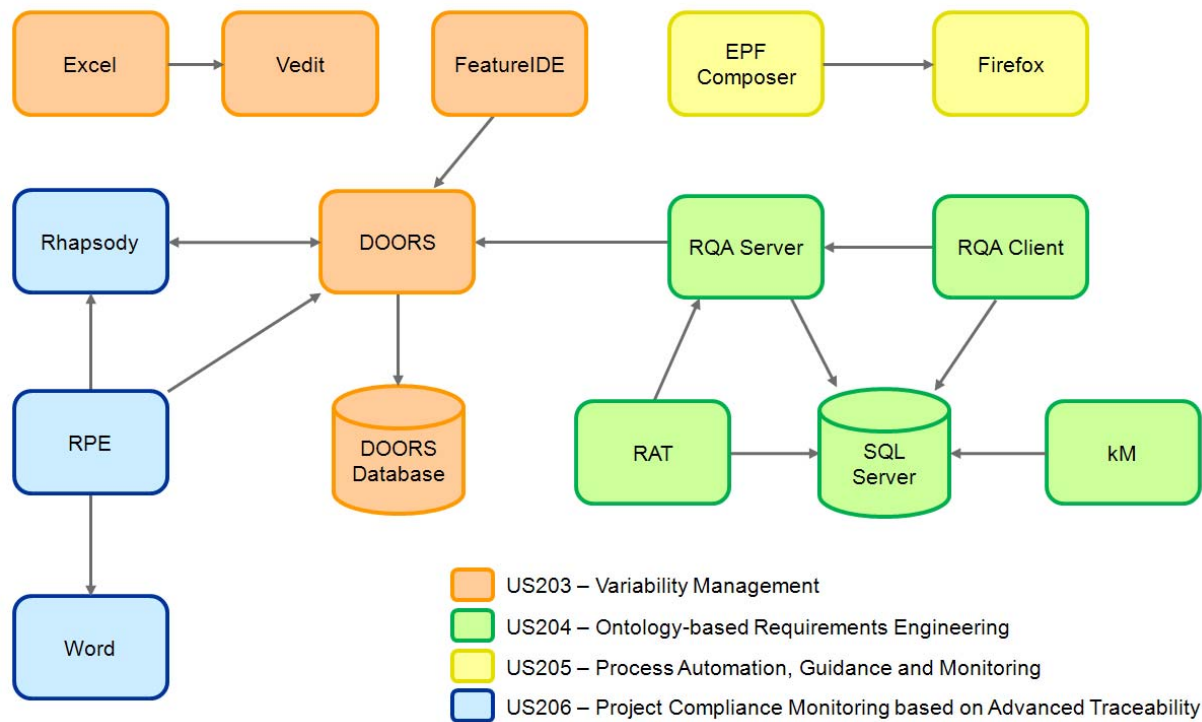


Figure 2-4 Overview of the integrated tool chain

3 Description of the Tool Chain

3.1 Overview

Table 3-1 provides information on tool versions and tool vendors for all tools integrated in the first SEE prototype.

Tool	Short Name	Version	Tool Vendor
IBM Rational DOORS	DOORS	9.2.0.5	IBM Rational
IBM Rational Rhapsody including the add-ons Gateway and TestConductor	Rhapsody	8.0	IBM Rational
IBM Rational Publishing Engine Document Studio	RPE	1.1.2.2	IBM Rational
Requirements Quality Analyzer	RQA	4.1.4892	The Reuse Company
Requirements Authoring Tool	RAT	4.1	The Reuse Company
knowledgeManager	kM	6.1	The Reuse Company
Microsoft SQL Server 2008 Express Edition	SQL Server	10.0.5500	Microsoft
VarMod Editor	Vedit	N/A	Paluno, The Ruhr Institute for Software Technology, University of Duisburg-Essen
FeatureIDE	FeatureIDE	2.6.5	University of Magdeburg
Eclipse Process Framework Composer	EPF Composer	1.5.1.5	Eclipse Foundation
Mozilla Firefox	Firefox	26.0	Mozilla
Microsoft Word	Word	2007	Microsoft
Microsoft Excel	Excel	2007	Microsoft

Table 3-1 Tools integrated in the SEE prototype

In the following sections a short description of each tool is given.

3.2 Vedit

The Vedit tool is an Eclipse plug-in that is based on the EMF and GMF. It supports the specification of product family variability models which are documented in the OVM notation (Orthogonal Variability Modelling). The editor has been developed based on the variability meta model of the OVM and the OVM notation [OVM]. Vedit provides a graphical editor and performs syntax checks based on the OVM language model. It provides the following tool functions:

- Definition of variation points and variants: name, short description, artefact reference, internal / external (visible to customer) variation points

Version	Nature	Date	Page
V1.00	R	2014-02-10	15 of 74

- Definition of variability constraints: optional, mandatory, range of alternatives
- Definition of constraint dependencies: requires, excludes
- Import and export of variability models

The Vedit tool was developed at the Software Systems Engineering institute of the Ruhr Institute for Software Technology (PALUNO), University of Duisburg-Essen.

Within EADS-CAS, Vedit is not used yet.

3.3 FeatureIDE

FeatureIDE is a development tool that supports feature-oriented programming. The idea of feature-oriented programming is that features realize functionalities which are implemented in program fragments. In a software product line a family of program variants can be defined by a composition of selected features. Different composition engines such as AHEAD, FeatureC++ or FeatureHouse are available in FeatureIDE, which support different programming languages and paradigms. The selection of features can be done in a tree-style configuration dialog. During the configuration process it is assured that only valid configurations are defined. Valid configurations are specified in the feature model, which is graphically represented by a feature diagram. The feature diagram contains mandatory or optional features which can be grouped with and / or / alternative relations. Logical constraints can be added such as "feature A implies feature B". The following functions of FeatureIDE are used in the SEE prototype:

- Definition of a feature model including cross-tree constraints
- Creation of configurations

3.4 DOORS

IBM Rational DOORS is a requirements management tool which aims to support requirements-based engineering activities. DOORS offers the following functions:

- Requirements management in a centralized location (DOORS database)
- Requirements capture by importing requirements from other sources (e.g. MS Word or MS Excel)
- Requirements definition by creating and editing requirements objects which can be further described by a customizable set of attributes
- Traceability by linking requirements with other requirements or test cases (if present in the DOORS database)
- Custom views to aid traceability and impact analysis
- Creation of requirements documents

Based on custom DXL (DOORS Extension Language) scripts several company-specific extensions have been introduced:

- Completeness and consistency checks, e.g. have all attributes been specified according to the applicable requirements management plan?
- Simple linguistic checks, e.g. have vague phrases been avoided in the requirements statements?
- Calculation of metrics and KPIs
- Improved report generation

Version	Nature	Date	Page
V1.00	R	2014-02-10	16 of 74

- Support for product family management. This includes integration with feature models and automatic generation of requirements modules for given product configurations.

Within EADS-CAS, DOORS is used as standard tool to manage requirements in all large projects and programmes.

3.5 RQA

Requirements Quality Analyzer (RQA) belongs to the Requirements Quality Suite (RQS), a set of tools aimed to customize, manage and improve the quality of a set of requirements. The main goals of RQA are summarized from [CRYSTAL D607.021]:

- Allowing the customization of quality metrics for the whole suite, so that the suite could provide recommendations to different end-users
- Forcing the re-check of the quality for a set of requirements
- Editing individual requirements by following a set of quality hints
- Generating quality reports:
 - Correctness report: including the quality hints for a set of metrics measured individually, i.e. requirement by requirement
 - Completeness report: this report is based on boilerplates and lists all the boilerplates defined to represent a set of different types of requirements, together with the list of requirements matching any of those boilerplates
 - Consistency report: based on the measurement units used in different requirements
 - Coupling analysis: showing those requirements with a similar semantic graph

Within EADS-CAS, RQA is not used yet.

3.6 RAT

Requirements Authoring Tool (RAT) belongs to Requirements Quality Suite (RQS), a set of tools aimed to customize, manage and improve the quality of a set of requirements. The main goals of RAT are summarized from [CRYSTAL D607.031]:

- Typing (either adding or editing) requirements on top of a requirements management tool
- Generating correctness information on the fly
- Highlighting the defects (or order relevant information) found during the quality analysis
- Accessing the details of the quality metrics: actual quantitative value, qualitative value, expressions found in the requirement which raised the metric...
- Assistance in writing requirements by following a set of agreed upon boilerplates
- Use of the right vocabulary by showing suggestions coming from domain ontologies
- Consistency information based on measurement units
- Similar requirements based on their semantic graphs
- Suggestion management: that allows to send suggestions to the “owners” of the ontology about new concepts or even new boilerplates

Version	Nature	Date	Page
V1.00	R	2014-02-10	17 of 74

Within EADS-CAS, RAT is not used yet.

3.7 kM

knowledgeMANAGER (kM) belongs to Requirements Quality Suite (RQS), a set of tools aimed to customize, manage and improve the quality of a set of requirements. The main goals of kM are the following are summarized from [CRYSTAL D607.041]:

- Managing the indexing process based on Natural Language tools (NL tools)
- Manage controlled vocabulary to be used in RAT and RQA
- Manage thesaurus and links among the concepts in the controlled vocabulary
- Manage requirements patterns needed to generate the proper formalization of requirements (or any other text-based artefact)
- Manage the communication between the team in charge of creating the requirements and the team in charge of managing all the layers on the ontology. This communication is modelled as a suggestion system for the requirements authors.

All the previous information defines an ontology formed by the following layers:

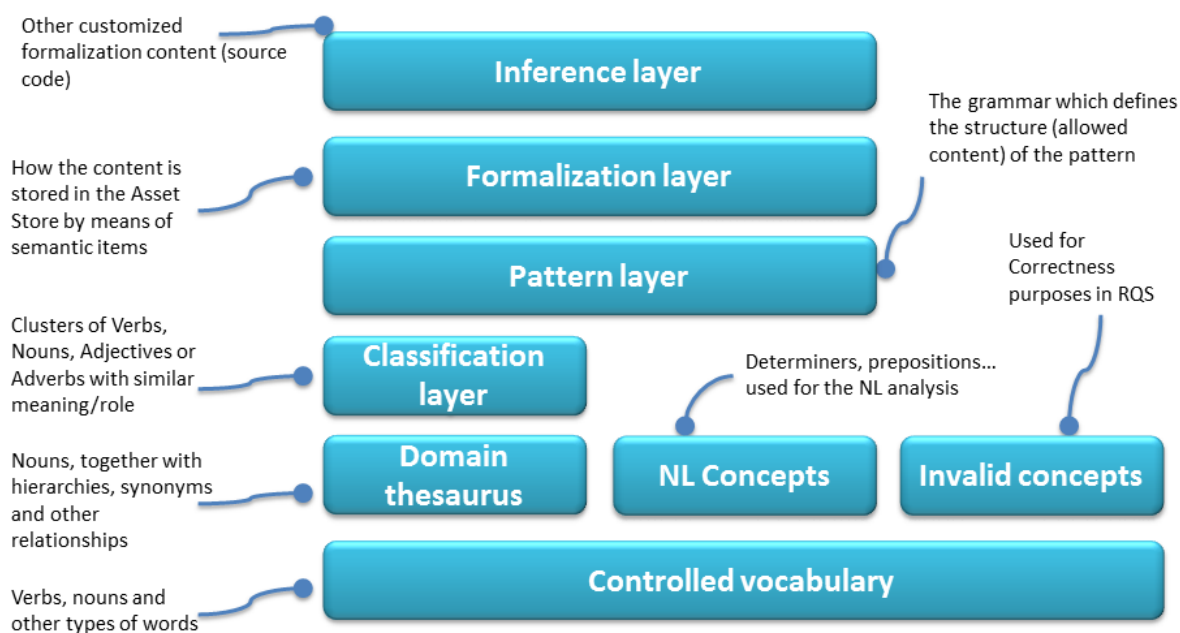


Figure 3-1 Ontology layers [CRYSTAL D607.041]

Within EADS-CAS, kM is not used yet.

3.8 Rhapsody

IBM Rational Rhapsody is a UML/SysML modelling tool which supports model-based development (MBD) and model-based systems engineering (MBSE) for real-time or embedded systems engineering. It can improve productivity, quality, and communication by abstracting complex designs and assisting in finding defects early. It supports the requirements analysis and traceability to design and allows automate the

Version	Nature	Date	Page
V1.00	R	2014-02-10	18 of 74

testing process with visualization of test cases and execution of automated tests using the IBM Rational TestConductor add-on. It is able to import requirement objects from the requirements management database DOORS via the IBM Rational Rhapsody Gateway add-on. It can generate C, C++, Java and ADA code which can be executed together with the Rhapsody Object Execution Framework (OXF).

Within EADS-CAS, Rhapsody is used in some projects and programmes, which have deployed MBD and/ or MBSE.

3.9 RPE

IBM Rational Publishing Engine automates document generation from Rational products and select third-party tools. RPE can be used to automate the generation of documents for ad hoc use, formal reviews, contractual obligations or regulatory compliance. Built-in capabilities extract data from a range of data sources to help reduce manual work and risk of errors. RPE provides the following features:

- Documents and reports: generate high-quality documents with flexible formatting as well as composite reports containing data from multiple sources
- Outputs: support multiple output formats and concurrent document generation to multiple target formats from a single template.
- Templates: include predefined templates and provide a graphical template editing environment for custom report design.
- Data sources: extract data from a single source or combine data from multiple sources.

Within EADS-CAS, RPE is used in combination with Rhapsody in some projects and programmes.

3.10 EPF Composer

The Eclipse Process Framework (EPF) is an open source project that is managed by the Eclipse Foundation. It has two goals:

- To provide an extensible framework and exemplary tools for process engineering - method and process authoring, library management, configuring and publishing a process.
- To provide exemplary and extensible process content for a range of development and management processes supporting iterative, agile, and incremental development, and applicable to a broad set of development platforms and applications.

By using EPF Composer you can create your own development process by structuring it in one specific way using a predefined schema. This schema is based on the SPEM OMG specification and is referred to as the Unified Method Architecture (UMA). The UMA and SPEM schemata support the organization of large amounts of descriptions for methods and processes. Such method content and processes do not have to be limited to software or systems engineering, but can also cover other design and engineering disciplines.

Within EADS-CAS, EPF Composer is not used in the operational environment.

Version	Nature	Date	Page
V1.00	R	2014-02-10	19 of 74

4 Description of the Usage Scenarios

4.1 Overview

This chapter describes the engineering activities applied with the integrated tool chain of the first SEE prototype. It provides an overview of the data that has been defined for specifying the Landing Symbology feature. **Table 4-1** provides references to the related user stories and engineering methods as described in [D203.011].

Usage Scenario	Related User Story	Related Engineering Method
SC1 – Define Product Family Scope and Variability Model	US203 – Variability Management	N/A
SC2 – Develop Domain System Requirements	US203 – Variability Management	EM203_01_01 – Develop Domain System Requirements
SC3 – Analyze and Improve Requirements Quality	US204 – Ontology-based Requirements Engineering	EM204_01_03 – Analyze Requirements Quality
		EM204_01_01 – Define Requirements
SC4 – Create Product System Requirements	US203 – Variability Management	EM203_02_01 – Create Product System Requirements
SC5 – Perform System Functional Analysis	US206 – Project Compliance Monitoring based on Advanced Traceability	EM206_01_01 – Retrieve Valid Traces
		EM206_01_02 – Analyse Trace
		EM206_02_01 – Perform Coverage Analysis
		EM206_03_01 – Create Verification Objective
		EM206_03_02 – Create Verification Case
		EM206_03_03 – Create Verification Procedure
SC6 – Perform Report Generation	US206 – Project Compliance Monitoring based on Advanced Traceability	Will be defined in the next issue of [CRYSTAL D203.011]
SC7 – Provide Process Guidance	US205 – Process Automation, Guidance and Monitoring	Will be defined in the next issue of [CRYSTAL D203.011]

Table 4-1 Relation of usage scenarios with user stories and engineering methods

In the following sections the scenarios, applied in the first SEE prototype, are described in detail. For each scenario the related user story, engineering methods and the tool chain is given. The scenarios are exemplified using the data set created for the Landing Symbology function.

4.2 Scenario SC1 – Define Product Family Scope and Variability Model

Related user story: US203 – Variability Management

Related engineering methods: N/A

Related tool chain: Vedit, FeatureIDE

Product family domain engineering consists of two main activities: scoping and variability modelling. During the scoping, relevant product features and future product configurations are identified based on business and market information. This involves:

- Identify features which could provide value to at least one customer.
- Classify features according to the categories mandatory, optional, customizable, range of alternatives.
- Assess the relevance of each feature in terms of value (ability to contribute to customer satisfaction), risk (maturity of development) and cost (effort required for development).
- Define the scope of the product family by including only feature with high relevance.

The result of the scoping can be formalized using the Orthogonal Variability Model (OVM) approach [OVM]. The Vedit tool is employed for developing variability models in the OVM language. The main modelling elements of OVM are:

- Variation points: representation of a variable item or property of an item. What varies?
- Variant: particular instance of a variable item or property. How does it vary?
- Variability dependency: the specified variation point allows choosing the specified variant. An optional variant can be, but does not have to be bound for a variation point. A mandatory variant must be bound whenever its variation point is considered. An alternative choice groups a set of optional variability dependencies. In this case the allowed number of variants that can be selected has to be defined.
- Variability constraints: requires (selection of one variant requires another variant to be selected), excludes (selection of one variant excludes selection of another variant).

In the next step, variabilities have to be related to development artefacts. These are the outputs of the product family domain engineering such as requirements, system design, components, or tests. For this purpose traceability links that relate variants to development artefacts, need to be created. The formal representation of OVM allows reasoning about the consistency and correctness of the OVM and the derived products. However, in the current version of the SEE prototype this step is not considered yet.

Instead, a second approach has been implemented based on feature models. The configuration of valid variants is well supported by the feature approach. Selected features can easily be allocated to requirement artefacts. The derivation of requirements for different product variants can then be achieved by composition of the requirements corresponding to selected features.

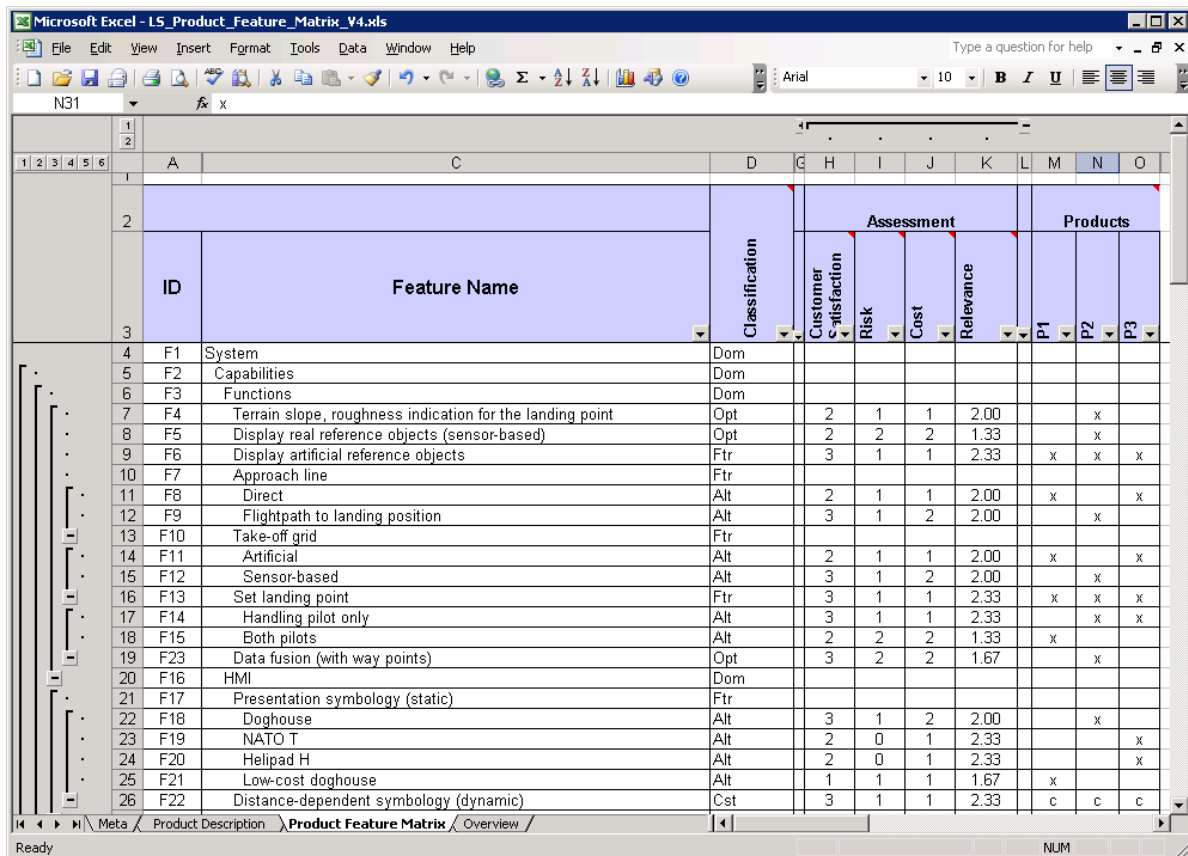
Version	Nature	Date	Page
V1.00	R	2014-02-10	21 of 74

The following steps are performed in this scenario:

- Step 1.1 – Define the scope of the product family
- Step 1.2 – Create variability model

4.2.1 Step 1.1 – Define product family scope

In this step the results of the scoping process are recorded in an Excel sheet – the Product Feature Matrix (see **Figure 4-1**). First, the characteristics of potential products (e.g. low-cost or high-end variants) are collected. Then, features of these products are identified by analyzing the capabilities (functions and performance), interfaces (to human operator or other systems), and technologies (development approaches and tools) of the potential products. An initial classification of each feature is performed (e.g. feature is optional; feature is refined into a range of alternatives). Finally, each feature is assessed by the stakeholders with respect to customer satisfaction, development risk and cost yielding a relevance factor. A threshold for the relevance is defined in order to get the list of features which should be in the scope of the product family.



ID	Feature Name	Classification	Assessment				Products		
			Customer satisfaction	Risk	Cost	Relevance	P1	P2	P3
F1	System	Dom							
F2	Capabilities	Dom							
F3	Functions	Dom							
F4	Terrain slope, roughness indication for the landing point	Opt	2	1	1	2.00		x	
F5	Display real reference objects (sensor-based)	Opt	2	2	2	1.33		x	
F6	Display artificial reference objects	Ftr	3	1	1	2.33	x	x	x
F7	Approach line	Ftr							
F8	Direct	Alt	2	1	1	2.00	x		x
F9	Flightpath to landing position	Alt	3	1	2	2.00		x	
F10	Take-off grid	Ftr							
F11	Artificial	Alt	2	1	1	2.00	x		x
F12	Sensor-based	Alt	3	1	2	2.00		x	
F13	Set landing point	Ftr	3	1	1	2.33	x	x	x
F14	Handling pilot only	Alt	3	1	1	2.33		x	x
F15	Both pilots	Alt	2	2	2	1.33	x		
F23	Data fusion (with way points)	Opt	3	2	2	1.67		x	
F16	HMI	Dom							
F17	Presentation symbology (static)	Ftr							
F18	Doghouse	Alt	3	1	2	2.00		x	
F19	NATO T	Alt	2	0	1	2.33			x
F20	Helipad H	Alt	2	0	1	2.33			x
F21	Low-cost doghouse	Alt	1	1	1	1.67	x		
F22	Distance-dependent symbology (dynamic)	Cst	3	1	1	2.33	c	c	c

Figure 4-1 Define the scope of the product family with product feature matrix

4.2.2 Step 1.2 – Create variability model

Based on the Product Feature Matrix, two variability models have been setup in order to formalize the results of the product family scoping.

The variability model depicted in **Figure 4-2** is based on the Orthogonal Variability Modelling (OVM) approach. The identified features are represented by variation elements. Dependencies are added

expressing options and alternatives. Constraints are introduced between variation elements. For example, the feature “Mark landing position” can be realized in two ways: either the handling pilot only or both pilots are allowed to set the landing position. All product variants provide the feature “Check for no ground”. However, the feature “Check for obstacle” is optional. If selected, an Obstacle Warning System (OWS) is required. In this case, one of the sensor equipments ELOP or HELLAS has to be selected.

The main advantage of the OVM approach is that the variation elements can be linked with other development artefacts (e.g. requirements, system model elements, or test cases). This will be further investigated when mature IOS adapter for accessing development artefacts in other tools are available.

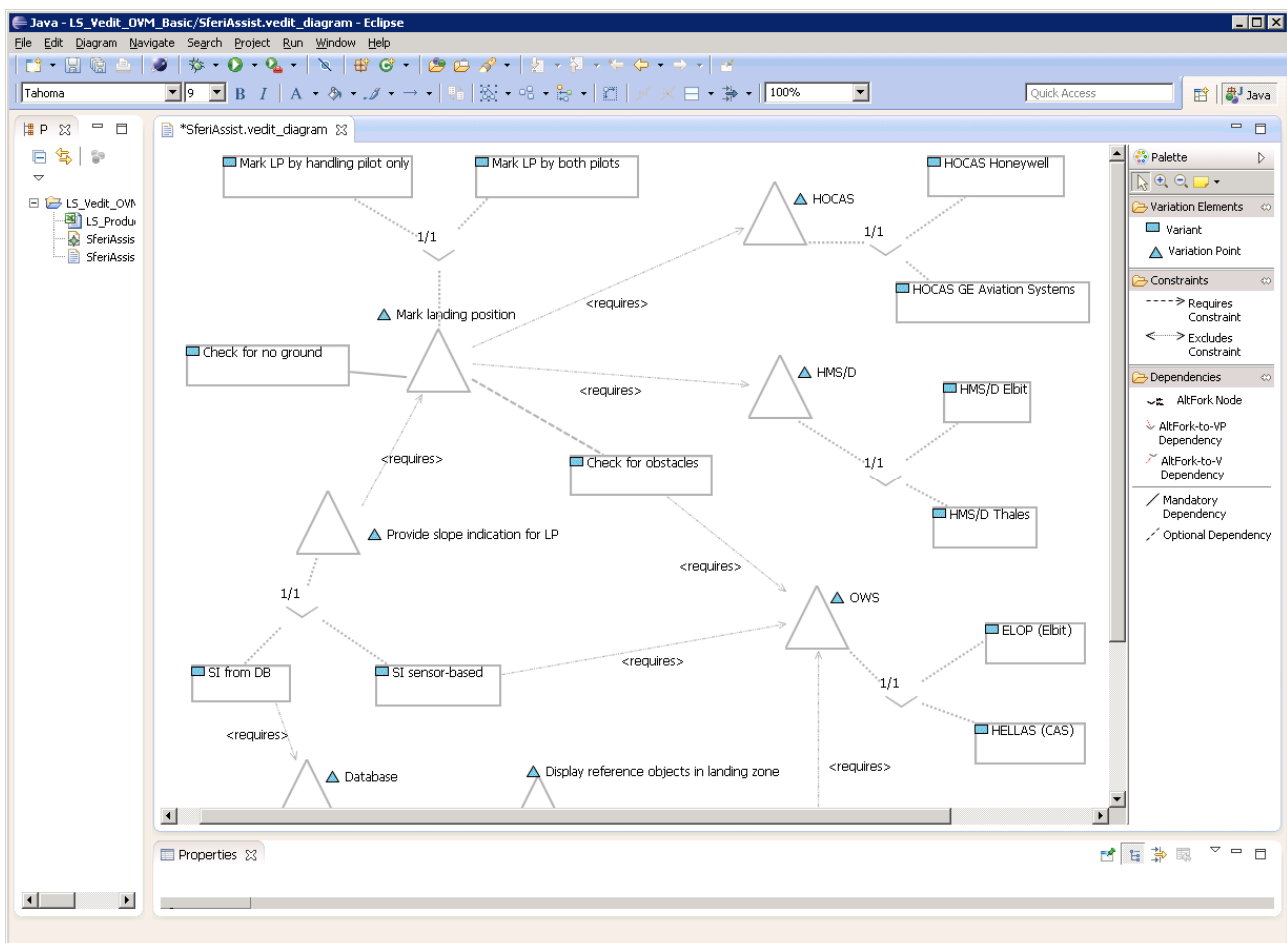


Figure 4-2 Create variability model using OVM

The second approach is based on feature modelling. The identified features have been introduced in a feature tree (see **Figure 4-3**). Subsequently, feature dependencies (graphical) and cross-tree constraints (textual) have been added. The main advantage of the feature modelling approach is that based on the feature model, configuration dialogs are generated automatically which assure that only valid configurations are defined. This has been further exploited in the SEE prototype (see section 4.5.1).

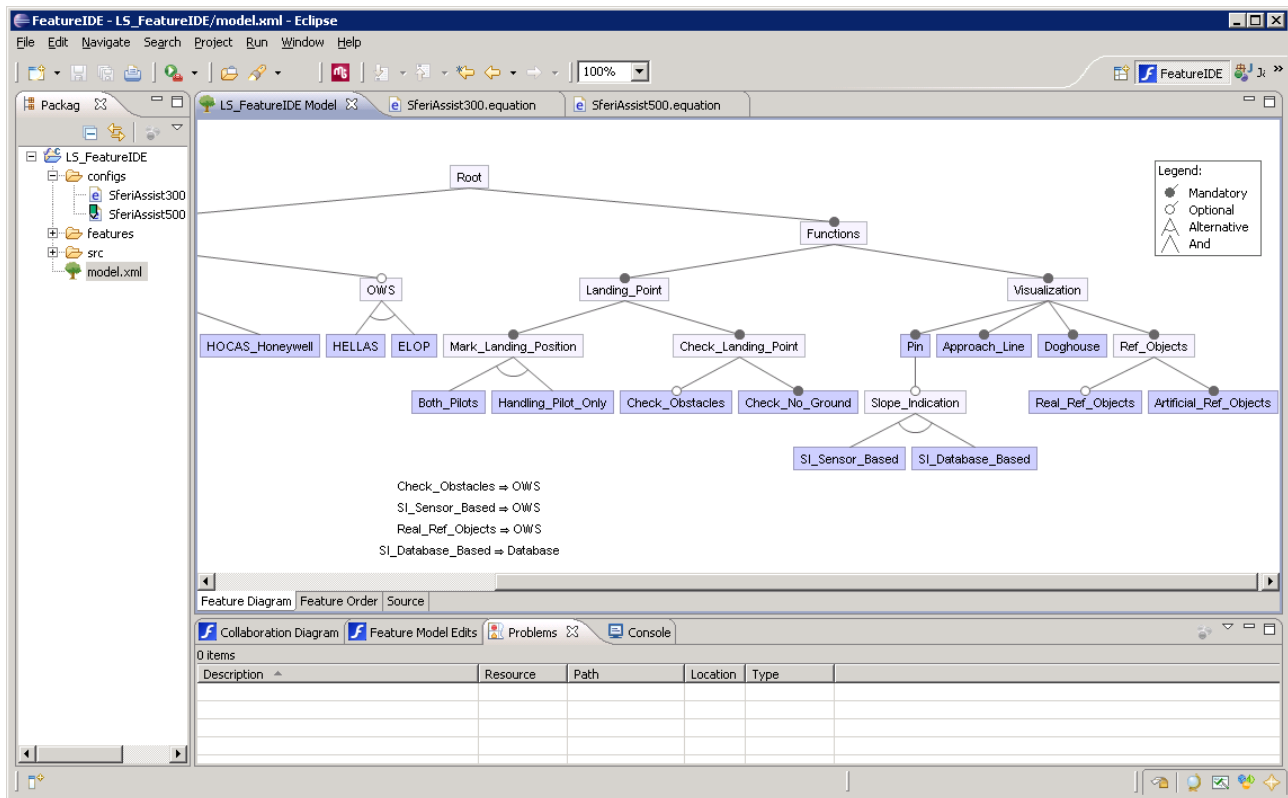


Figure 4-3 Create variability model using feature trees

4.3 Scenario SC2 – Develop Domain System Requirements

Related user story: US203 – Variability Management

Related engineering methods: EM203_01_01 – Develop Domain System Requirements

Related tool chain: DOORS

The scenario "Develop Domain System Requirements" describes the systems engineering activities related to the elicitation, development and analysis of system requirements. The purpose is to transform the stakeholder, user-oriented view of desired capabilities into a technical view of a solution that meets the operational needs of the user. It involves creating a set of verifiable system requirements that specify what characteristics, attributes, and functional and performance requirements the system is to possess, in order to satisfy stakeholder requirements.

System requirements shall be recorded in a form suitable for requirements management throughout the life cycle. These records establish the system requirements baseline. System requirements are the basis for traceability to stakeholder requirements and subsequent system elements. In order to fulfil these needs a DOORS requirements management repository has been setup and an initial set of top-level system requirements for the Landing Symbology function has been defined. Since the system requirements are supposed to specify all supported variants of the Landing Symbology function in the frame of the Sferion product family, features identified in Step 1.1 and formalized Step 1.2 are allocated to the system requirements.

The following steps are performed in this scenario:

- Step 2.1 – Define structure of the requirements management repository
- Step 2.2 – Define data model for requirements modules
- Step 2.3 – Create requirements
- Step 2.4 – Allocate requirements to features

4.3.1 Step 2.1 – Define structure of the requirements repository

A DOORS database has been setup. Different folders are introduced in order to separate different concerns (e.g. domain engineering and product realization) and different levels of decomposition (e.g. stakeholder, system, system element, and component level). Requirements modules and link modules defining the traceability scheme are created (see **Figure 4-4**).

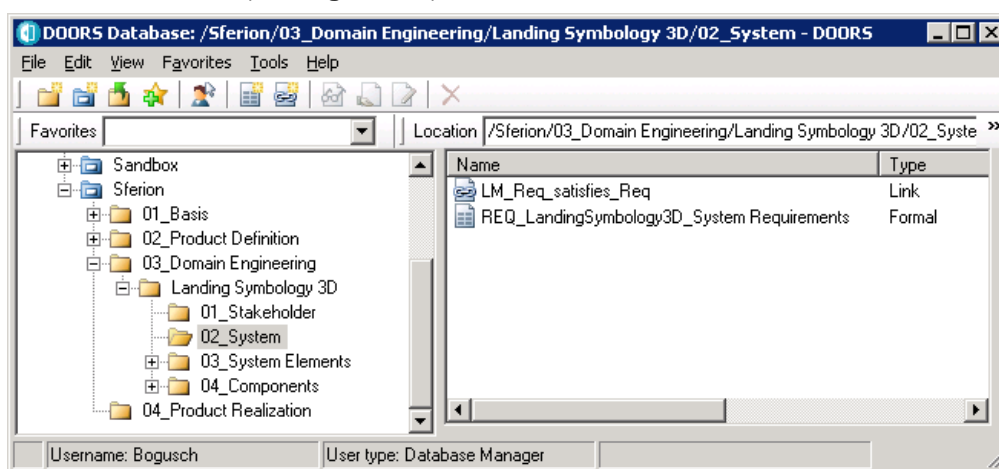


Figure 4-4 Define the structure of the RM repository

4.3.2 Step 2.2 – Define data model for requirements modules

In this step the data model for requirements modules is defined. Attributes are used to provide additional information for each requirement object. Examples are rationale, status, involved stakeholders, and verification methods (see **Figure 4-5**). In addition, views are created, which present the attributes relevant for performing certain engineering activities such as "Agree input requirements", "Generate requirements" or "Validate output requirements".

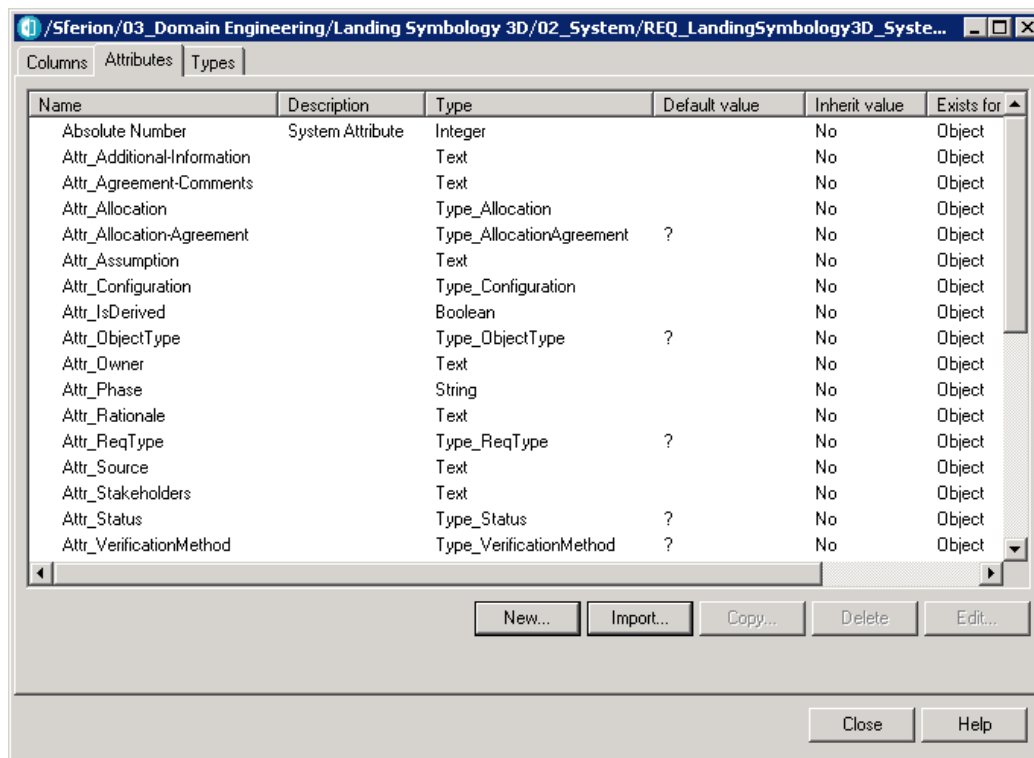


Figure 4-5 Define the data model for requirements modules

4.3.3 Step 2.3 – Create requirements

After organizing the requirements management repository structure and data model in DOORS, a first set of system requirements for the Landing Symbology function has been created (see **Figure 4-6**). The requirements have been written in structured (tabular) form in order to ease formalization and analysis. The requirement type is assigned. This enables filtering requirements according to their type, e.g. gathering all functional requirements as input for the functional analysis (see Step 5.2).

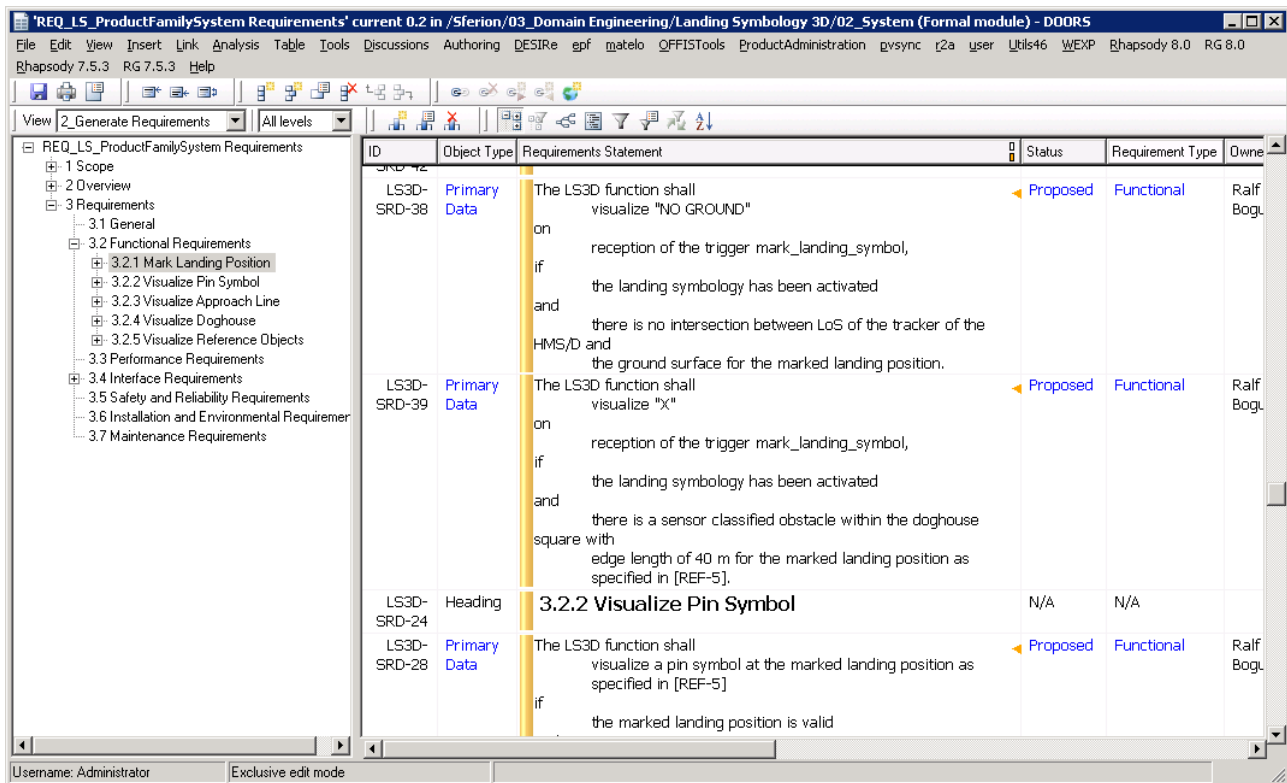


Figure 4-6 Create requirements

4.3.4 Step 2.4 – Allocate requirements to features

In this step the system requirements are allocated to features. A DOORS add-on has been developed that provides support for product family management [Stocker 2011]. The add-on comprises the following functions for this step:

- Read a feature model (see Step 1.2) and identify all concrete features (omitting abstract features)
- Define a new attribute “feature” in the DOORS requirements module and specify the value range such that only valid features can be allocated to requirements
- Synchronize with changes in the feature model on request

When a feature is allocated to a requirement by choosing a feature, the requirement type is automatically changed from “Common Platform” to “Variable Platform” indicating that this requirement is not realized in all product configurations (see **Figure 4-7**).

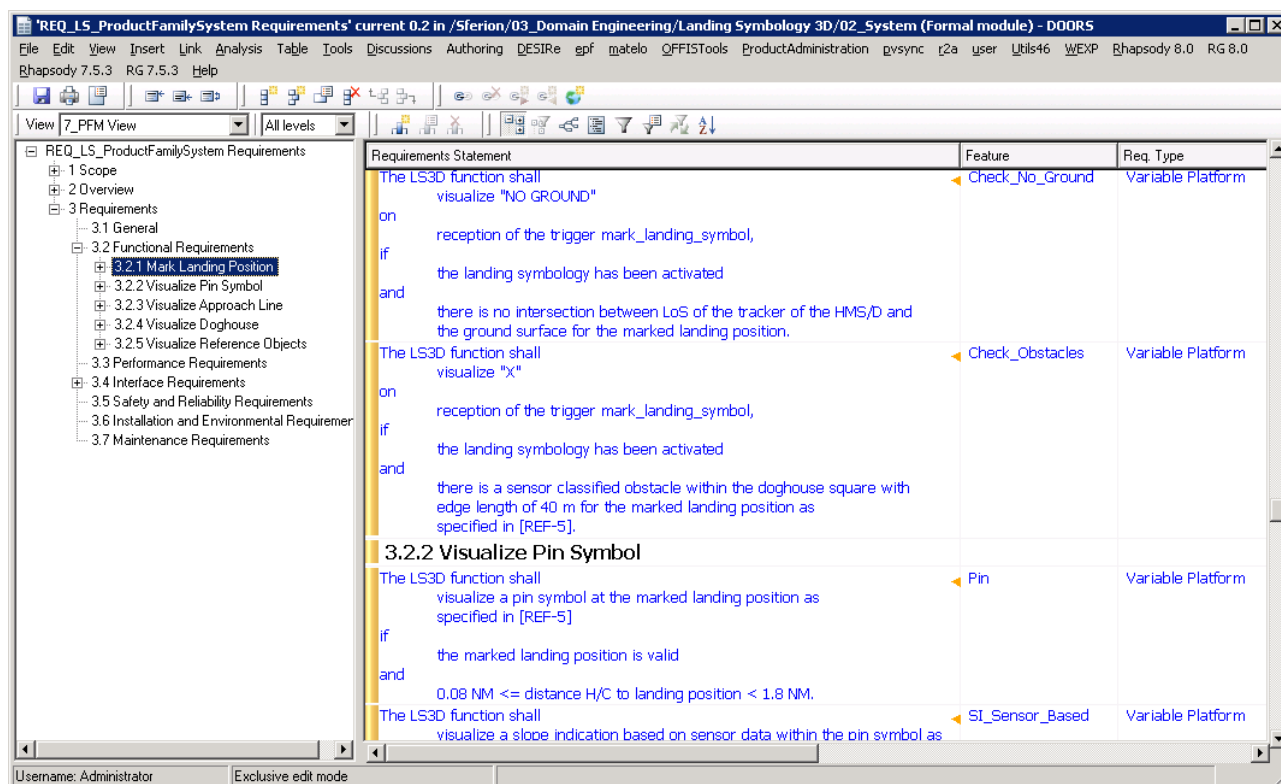


Figure 4-7 Allocate requirements to features

4.4 Scenario SC3 – Analyze and Improve Requirements Quality

Related user story: US204 – Ontology-based Requirements Engineering

Related engineering methods: EM204_01_03 – Analyze Requirements Quality, EM204_01_01 – Define Requirements

Related tool chain: DOORS, RQM, RAT, kM

The quality of requirements has a major impact on the three main project constraints for every project: time, cost and scope. Badly written requirements are a well-known source of project failure. For this reason, project management requires collecting and analyzing metrics or Key Performance Indicators (KPIs) on a regular basis to measure quality of requirements and support evaluation of the effectiveness of the requirements engineering process.

According to [ISO/IEC 29148] the result of requirements engineering is a hierarchy of requirements that enable an agreed understanding between stakeholders, is validated against stakeholders' needs, and provides a basis for verifying designs and accepting solutions. Well-formed requirements contribute to requirements validation with the stakeholders, and ensure that the requirements accurately capture stakeholder needs. If a requirement is expressed in natural language, the statement should be formulated in active voice and comprise a subject, a verb and a complement. The keyword "shall" indicates that a requirement is a mandatory binding provision. Requirement boilerplates should be used, since they provide standardized requirement syntax and ease communication with stakeholders.

When writing requirements, the quality of requirements shall be assessed. According to [ISO/IEC 29148] individual requirements shall possess the following quality characteristics:

Version	Nature	Date	Page
V1.00	R	2014-02-10	28 of 74

- *Necessary*. The requirement defines an essential capability, characteristic, constraint, and/or quality factor. If it is removed or deleted, a deficiency will exist, which cannot be fulfilled by other capabilities of the product or process.
- *Implementation free*. The requirement, while addressing what is necessary and sufficient in the system, avoids placing unnecessary constraints on the architectural design. The objective is to be implementation-independent. The requirement states what is required, not how the requirement should be met.
- *Unambiguous*. The requirement is stated in such a way so that it can be interpreted in only one way. The requirement is stated simply and is easy to understand.
- *Consistent*. The requirement is free of conflicts with other requirements.
- *Complete*. The stated requirement needs no further amplification because it is measurable and sufficiently describes the capability and characteristics to meet the stakeholder's need.
- *Atomic*. The requirement statement includes only one requirement with no use of conjunctions.
- *Feasible*. The requirement is technically achievable and fits within system constraints (e.g. cost, schedule, technical, legal, regulatory).
- *Traceable*. The requirement is upwards traceable to specific documented stakeholder statement(s) of need, higher tier requirement, or other source (e.g. a trade or design study). The requirement is also downwards traceable to the specific requirements in the lower tier requirements specification or other system definition artefacts. That is, all parent-child relationships for the requirement are identified in tracing such that the requirement traces to its source and implementation.
- *Verifiable*. The requirement has the means to prove that the system satisfies the specified requirement. Verifiability is enhanced when the requirement is measurable.

According to [ISO/IEC 29148] each set of requirements shall possess the following quality characteristics:

- *Complete*. The set of requirements needs no further amplification because it contains everything pertinent to the definition of the system or system element being specified. In addition, the set contains no To Be Defined (TBD), To Be Specified (TBS), or To Be Resolved (TBR) clauses.
- *Consistent*. The set of requirements does not have individual requirements which are contradictory. Requirements are not duplicated. The same term is used for the same item in all requirements.
- *Affordable*. The complete set of requirements can be satisfied by a solution that is feasible within life cycle constraints (e.g. cost, schedule, technical, legal, and regulatory).
- *Bounded*. The set of requirements maintains the identified scope for the intended solution without increasing beyond what is needed to satisfy user needs.

When writing textual requirements, the following considerations will help ensure that good requirements characteristics are employed. Vague and general terms shall be avoided. They result in requirements that are often difficult or even impossible to verify or may allow for multiple interpretations. The following are types of unbounded or ambiguous terms, see [ISO/IEC 29148]:

- Superlatives (such as 'best', 'most')
- Subjective language (such as 'user friendly', 'easy to use', 'cost effective')
- Vague pronouns (such as 'it', 'this', 'that')
- Ambiguous adverbs and adjectives (such as 'almost always', 'significant', 'minimal')
- Open-ended, non-verifiable terms (such as 'provide support', 'but not limited to', 'as a minimum')
- Comparative phrases (such as 'better than', 'higher quality')

Version	Nature	Date	Page
V1.00	R	2014-02-10	29 of 74

- Loopholes (such as 'if possible', 'as appropriate', 'as applicable')
- Incomplete references (not specifying the reference with its date and version number; not specifying just the applicable parts of the reference to restrict verification work)
- Negative statements (such as statements of system capability not to be provided)

A more theoretical treatment of requirement quality and a comprehensive set of rules how to write requirements is provided by [INCOSE RWG]. In addition, significant work has done in the CESAR project, which has been taken as a starting point. This work refers to completeness, consistency and correctness of requirements [CESAR CCC] as well as boilerplates, patterns and ontologies [CESAR RSL].

The following steps are performed in this scenario:

- Step 3.1 – Select requirements module
- Step 3.2 – Configure quality analysis for requirements module
- Step 3.3 – Define ontologies
- Step 3.4 – Analyze quality of requirements module
- Step 3.5 – Provide findings
- Step 3.6 – Create requirements quality report
- Step 3.7 – Improve requirements

4.4.1 Step 3.1 – Select requirements module

The purpose of this step is to select the DOORS module that is to be analyzed by RQA. When RQA is started, a connection to the DOORS server is established. The structure of DOORS projects and folders containing DOORS modules is displayed as shown in **Figure 4-8**. When a DOORS module is selected, RQA will perform quality analysis for the selected module.

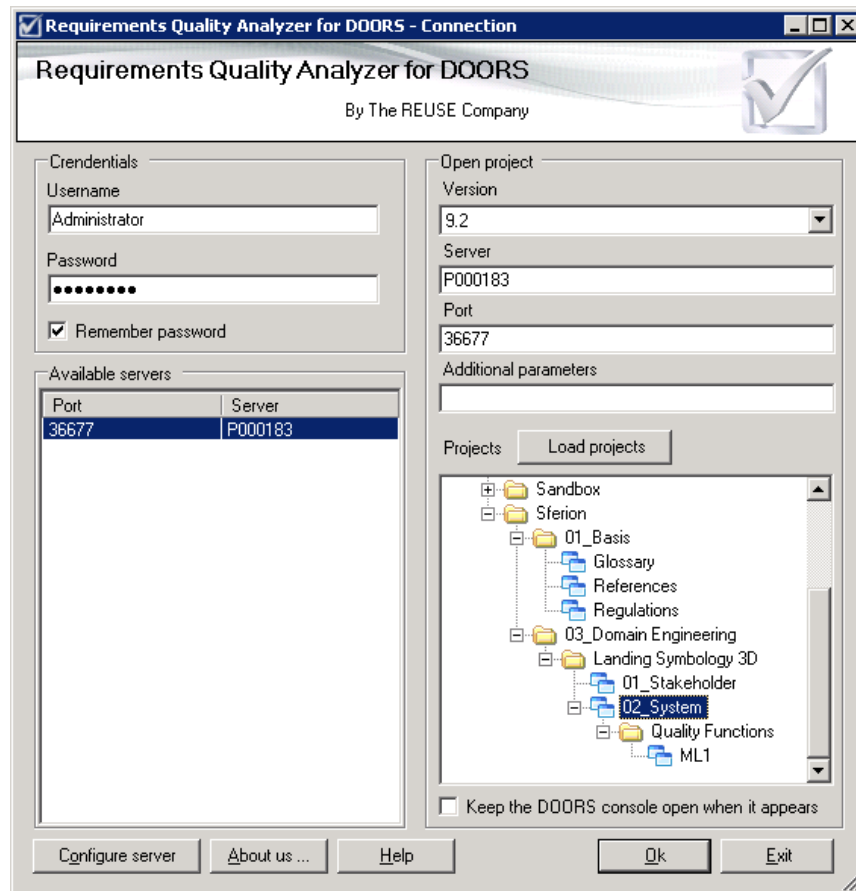


Figure 4-8 Select a requirements module for analysis

4.4.2 Step 3.2 – Configure quality analysis for requirements module

In this step the quality analysis process that should be employed for the selected DOORS module is configured. The customization takes into account the needs of different projects, teams, or types of requirements modules. A filter can be specified to identify requirements to be analyzed and ignore headlines and other information contained in the DOORS module. Moreover, it can be defined whether RQA stores the results of the analysis in the RQA database only or in the analyzed DOORS module additionally.

A set of more than 30 pre-defined correctness metrics is provided by RQA. A sub-set of correctness metrics may be chosen for quality analysis. During the evaluation, RQA takes every individual requirement, one by one, and gets a series of indicators for every requirement (e.g. text length, readability, ambiguous sentences...). Every indicator is now transformed into a qualitative value thanks to the associated quality function. During the correctness checking process, every metric rated as medium or low quality will generate a hint that leads the requirement author or reviewer in the best way to get rid of the problem and enhance the quality of the requirement. The quality functions may be customized for each metric. Moreover, weights can be assigned to selected metrics to change the sensitivity of the indicators. **Figure 4-9** depicts the configuration of correctness metrics.

According to [CRYSTAL D607.021] the following correctness metrics are provided:

- Size: expressed in paragraphs, chars, nouns or verbs. Long requirements will be difficult to understand

Version	Nature	Date	Page
V1.00	R	2014-02-10	31 of 74

- Readability: number of letters between punctuation marks and some other formulas that indicate whether the requirement will be easy to read. Ease to read requirements generates less problems all over the project
- Conditional sentences vs. imperative sentences: avoid “would” and use “shall”, “should” and “will” in the right way
- Active vs. passive voice: avoid using passive voice to increase the readability of the requirement
- Optional sentences: maybe... Optional requirements must be stated by an attribute, never in the body of the requirement
- Ambiguous sentences: fast, user-friendly... Analysts, developers and customers understand ambiguous sentences in different ways
- Subjective sentences: in my opinion, I think that... Don't show your ideas, but what the system should do
- Implicit sentences: it must be provided by them... Too many pronouns make your requirements difficult to understand
- Abuse of connectors: and, or. Many times connectors reveal different needs enclosed within the same requirement, losing the atomic characteristic
- False friends: customized according to “mother language” of your project
- Negations: no, never... Two or more negations in the same sentence make it difficult to understand
- Speculative sentences: usually, almost always... Make the requirement imprecise
- Design terms: loop, hash... Remember, avoid How, concentrate on What
- Flow terms: while, if, else... Remember avoid How, concentrate on What
- Number of domain nouns and verbs: domain terms and verbs should be involved in the requirement specification, nevertheless, too many different terms in the same requirement often means multiple needs
- Acronyms: avoid those that don't belong to the domain representation
- Hierarchical levels: don't complicate your specification with too many indentation levels
- Volatility: if a requirement suffers many changes, you must be very careful with it
- Number of dependencies: the same if your requirement is the source of too many dependences

In addition, special sentences such as ambiguous phrases, design terms, speculative sentences can be defined, see **Figure 4-10**. Further, measurement units (magnitudes) can be customized. This is used in the consistency analysis based on measurement units (e.g. both “miles” and “km” are used in the same set of requirements). Domain verbs and domain nouns can be defined in the so-called light ontology. This is used in some of the correctness metrics.

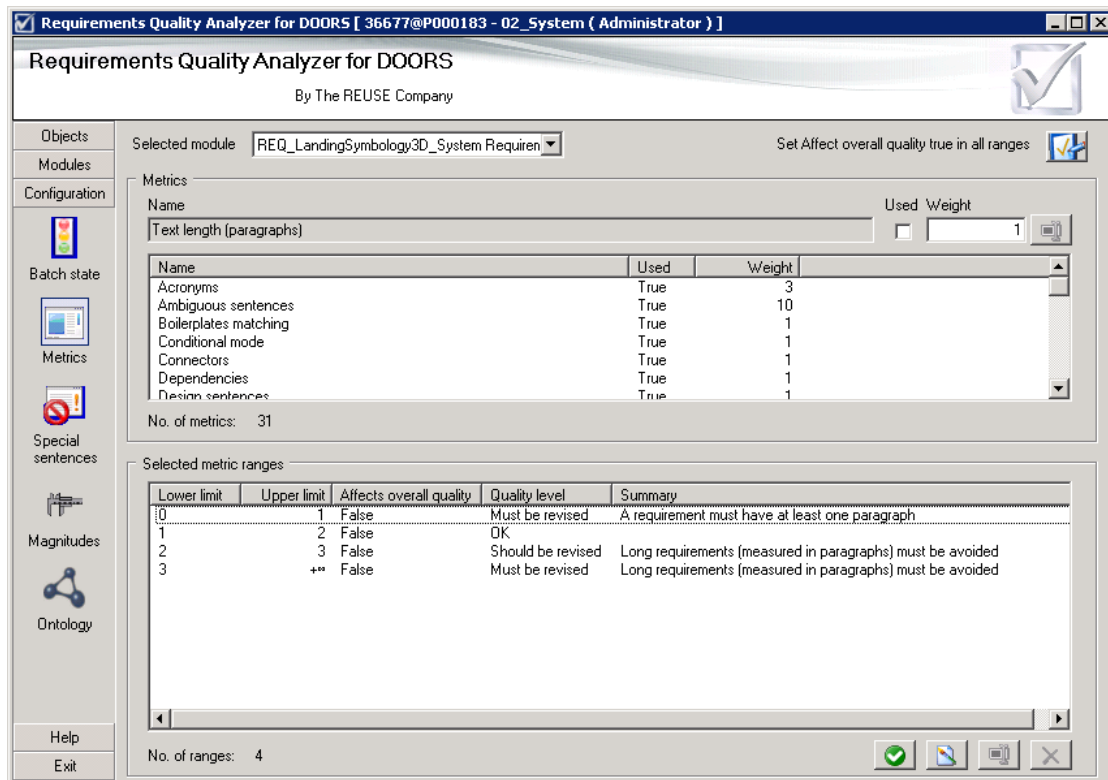


Figure 4-9 Configure quality metrics in RQA

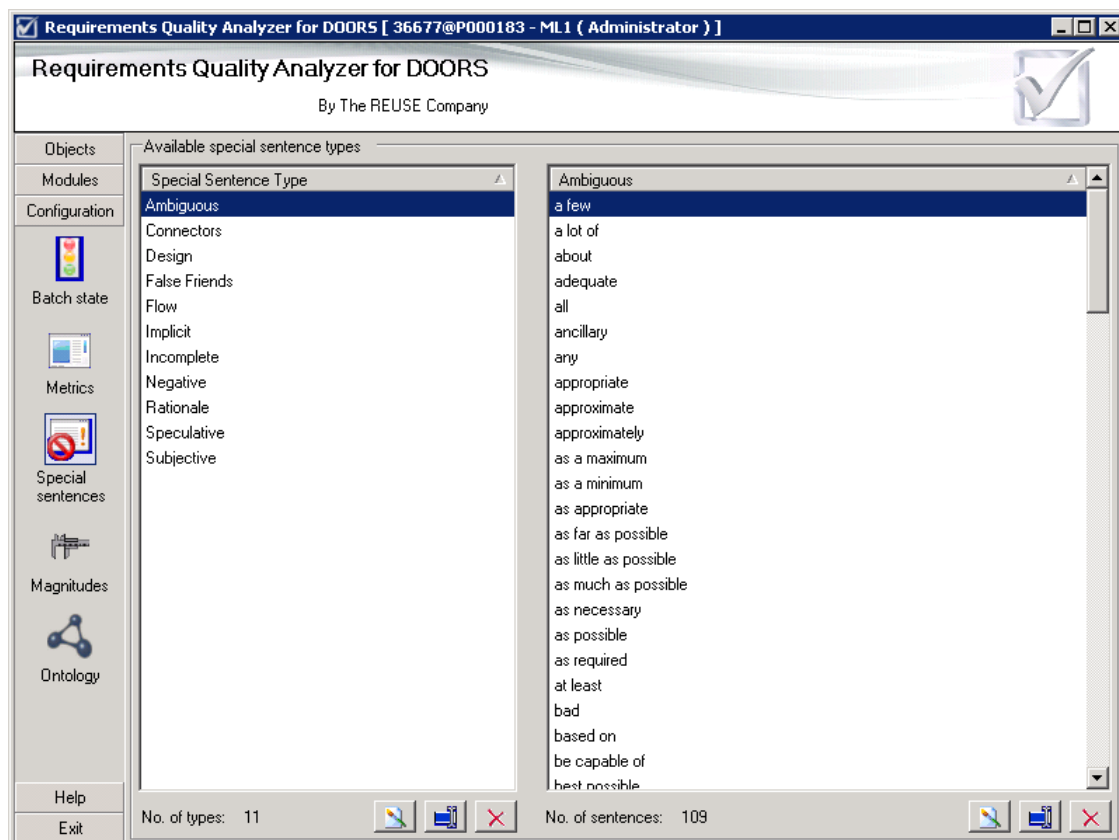


Figure 4-10 Configure vague phrases in RQA

4.4.3 Step 3.3 – Define ontologies

The requirements quality analysis can be extended towards more advanced metrics by

- Managing the controlled vocabulary (concepts) to be used in RAT and RQA
- Managing the thesaurus (links between concepts in the controlled vocabulary)
- Definition of boilerplates which support the proper formalization of requirements

Figure 4-11 exemplifies the definition of a simple boilerplate that follows a trigger-action pattern. For more information on KM, see [CRYSTAL D607.041].

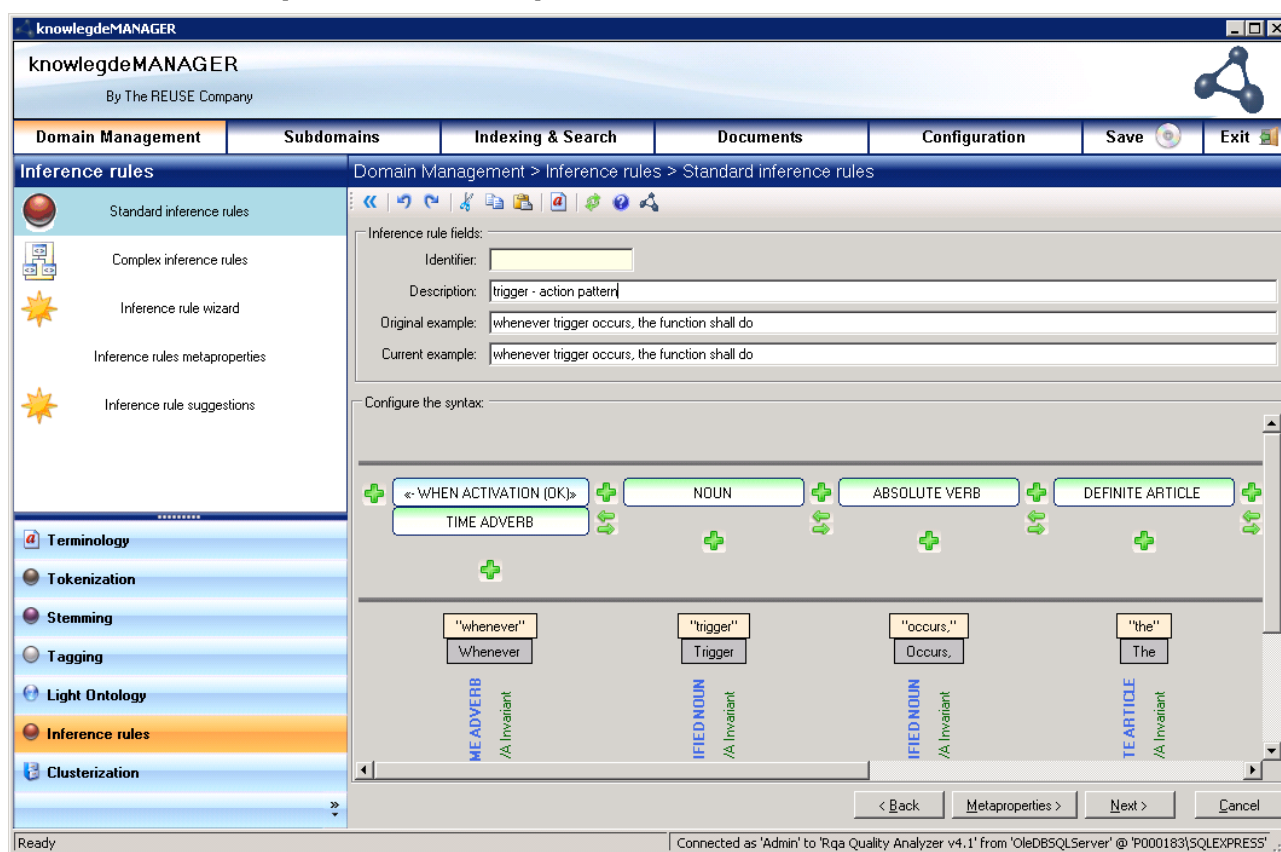


Figure 4-11 Define new boilerplates in kM

4.4.4 Step 3.4 – Analyze quality of requirements module

RQA is able to create a range of different reports. The metric report (see **Figure 4-12**) shows all the correctness metrics, the number of requirements assessed as high/medium/low quality on that metric, maximum and minimum values, average and standard deviations. This allows getting a quick overall view of the requirements quality of the selected DOORS module. In addition the most frequent errors are shown which allows creating focused corrective actions (e.g. team training).

In the prototype setup we have injected a requirement with multiple language defects:

LS3D-SRD-60: An "X" is visualized when an obstacle is close to the landing position.

This requirement has been detected by RQA and is rated as “low quality” requirement that “must be revised” (see global statistics in **Figure 4-12** and selected requirement in **Figure 4-13**).

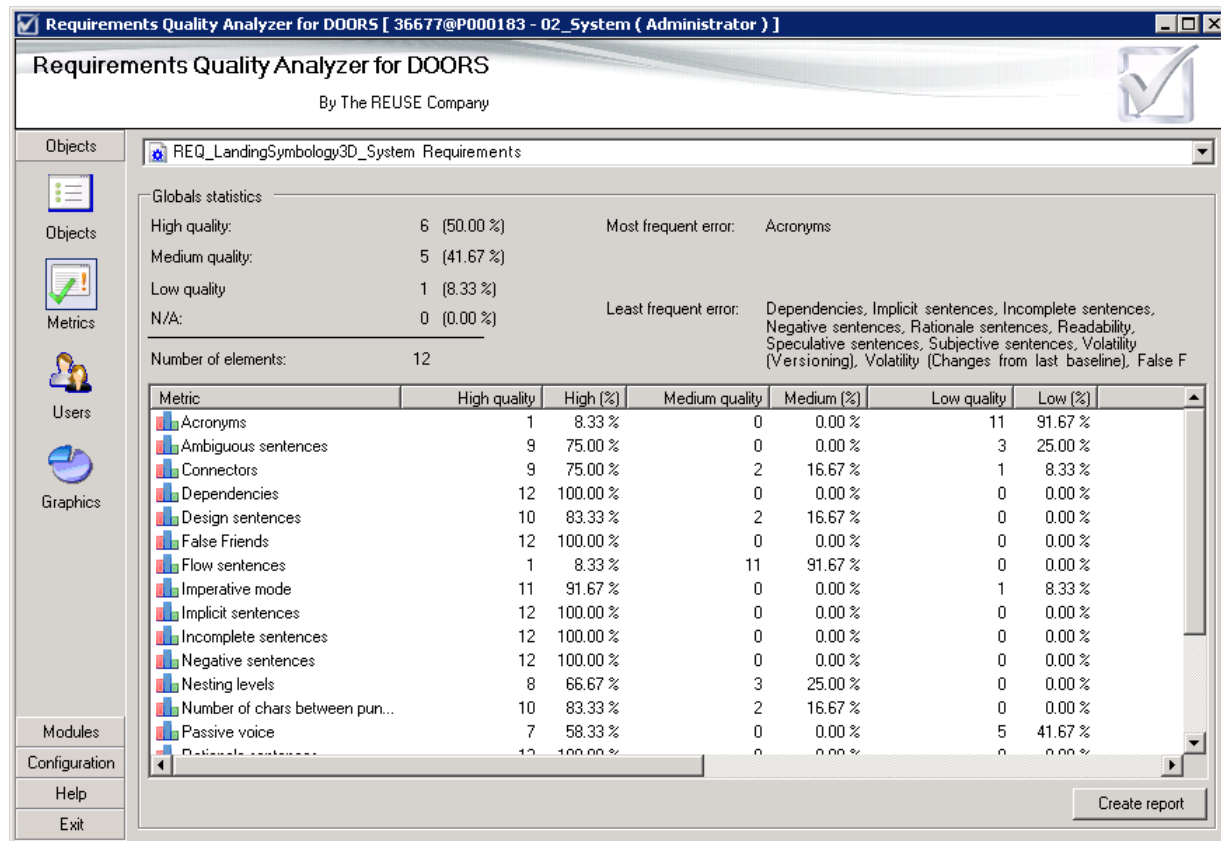


Figure 4-12 Analyze requirements quality of requirements set

In the “Objects view” the quality summary of each requirement is given qualitatively (OK, should be revised, must be revised) and quantitatively. The injected bad quality requirement LS3D-SRD-60 can easily be identified since it is rated “must be revised” and is associated with a very low quality value of 0.003 as shown in **Figure 4-13**.

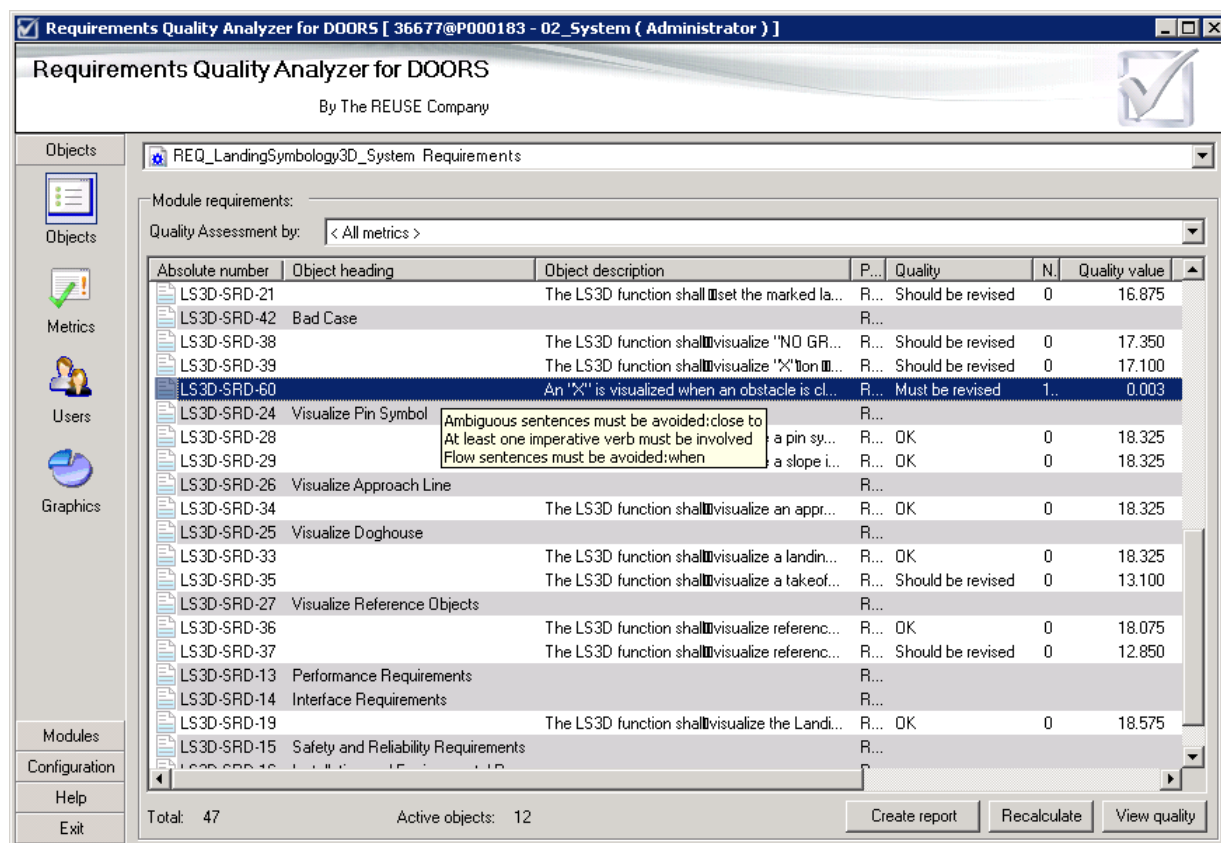


Figure 4-13 Identify requirements that need to be improved

4.4.5 Step 3.5 – Provide findings

In this step a more in-depth quality assessment is performed for each requirement that is rated as “must be revised” or “should be revised”. As shown in **Figure 4-14**, each metric is evaluated for the selected requirement. For the injected bad quality requirement the most relevant findings are:

- It contains ambiguous phrases
- It lacks an imperative verb (shall)

As illustrated in **Figure 4-15**, each metric can be further investigated. In this case, the ambiguous phrase is localized and marked with red colour.

Object quality assessment

Requirements Quality Analyzer for DOORS
By The REUSE Company

Requirements Quality Analyzer for DOORS assessment | Manual Assessment

Original quality assessment: Must be revised
Original quality date: 30/01/2014 10:28:39
Original quality summary: Ambiguous sentences must be avoided:close to
At least one imperative verb must be involved
Flow sentences must be avoided:when

New quality assessment: Must be revised
New quality date: 30/01/2014 10:30:24
New quality summary: Ambiguous sentences must be avoided:close to
At least one imperative verb must be involved
Flow sentences must be avoided:when

Object original data | Object data | **Metrics** | Textual metrics | Quality forums

Metric	Value	Quality	Recommendation	Affects overall quality
<input checked="" type="checkbox"/> Acronyms	0.00	OK		<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Ambiguous sentences	1.00	Must be revised	Ambiguous sentences must be avoided	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Connectors	0.00	OK		<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Dependencies	0.00	OK		<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Design sentences	0.00	OK		<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> False Friends	0.00	OK		<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Flow sentences	1.00	Should be revised	Flow sentences must be avoided	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Imperative mode	0.00	Must be revised	At least one imperative verb must be involved	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Implicit sentences	0.00	OK		<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Incomplete sentences	0.00	OK		<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Negative sentences	0.00	OK		<input checked="" type="checkbox"/>

Help | Create report | < Previous | Next > | Reload | Calculate quality | Open in DOORS | Save in DOORS | Close

Figure 4-14 Provide the quality summary for a requirement

Object quality assessment

Requirements Quality Analyzer for DOORS
By The REUSE Company

Requirements Quality Analyzer for DOORS assessment | Manual Assessment

Original quality assessment: Must be revised
Original quality date: 30/01/2014 10:28:39
Original quality summary: Ambiguous sentences must be avoided:close to
At least one imperative verb must be involved
Flow sentences must be avoided:when

New quality assessment: Must be revised
New quality date: 30/01/2014 10:30:24
New quality summary: Ambiguous sentences must be avoided:close to
At least one imperative verb must be involved
Flow sentences must be avoided:when

Object original data | Object data | **Metrics** | Textual metrics | Quality forums

An "X" is visualized when an obstacle is **close to** the marked landing position.

- < None >
 - Ambiguous sentences
 - CLOSE TO**
 - Flow sentences
 - WHEN**
 - Unclassified concepts

Help | Create report | < Previous | Next > | Reload | Calculate quality | Open in DOORS | Save in DOORS | Close

Figure 4-15 Analyze the findings for a requirement

4.4.6 Step 3.6 – Create requirements quality report

Different reports can be created, which summarize the results of the quality analysis. This is useful for project managers and quality assurance. An example is provided in **Figure 4-16**.

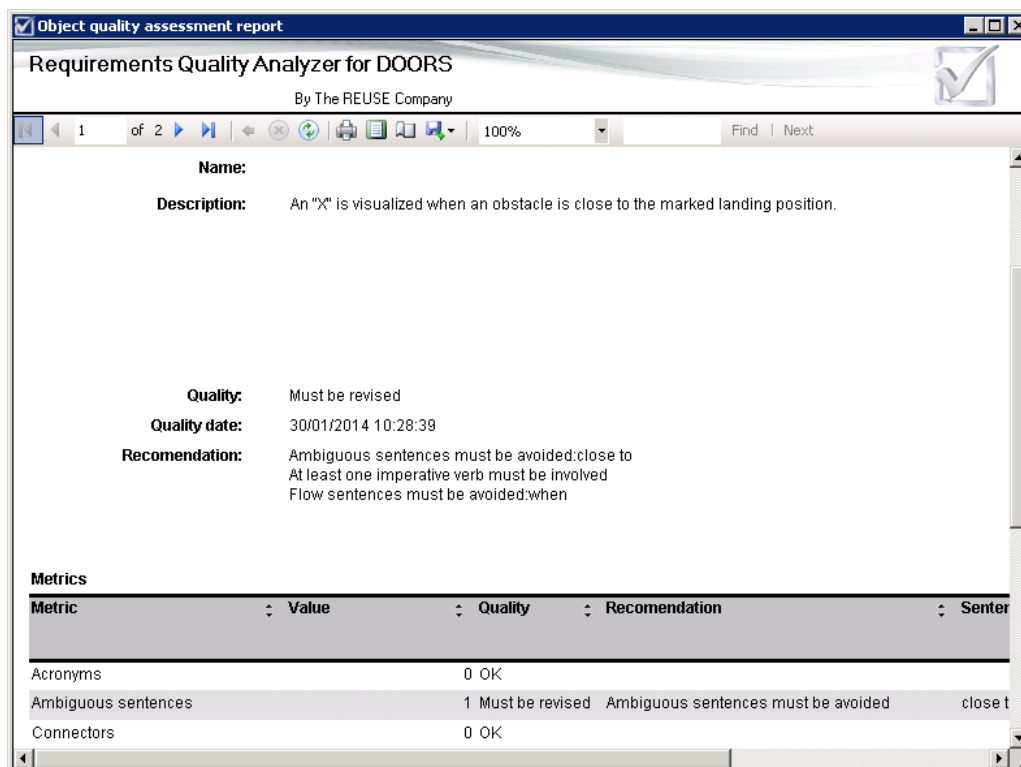


Figure 4-16 Create a quality report

Depending on the quality analysis configuration, the results can be stored in the DOORS module. This is particular useful for requirements authors working in the DOORS environment (see **Figure 4-17**).

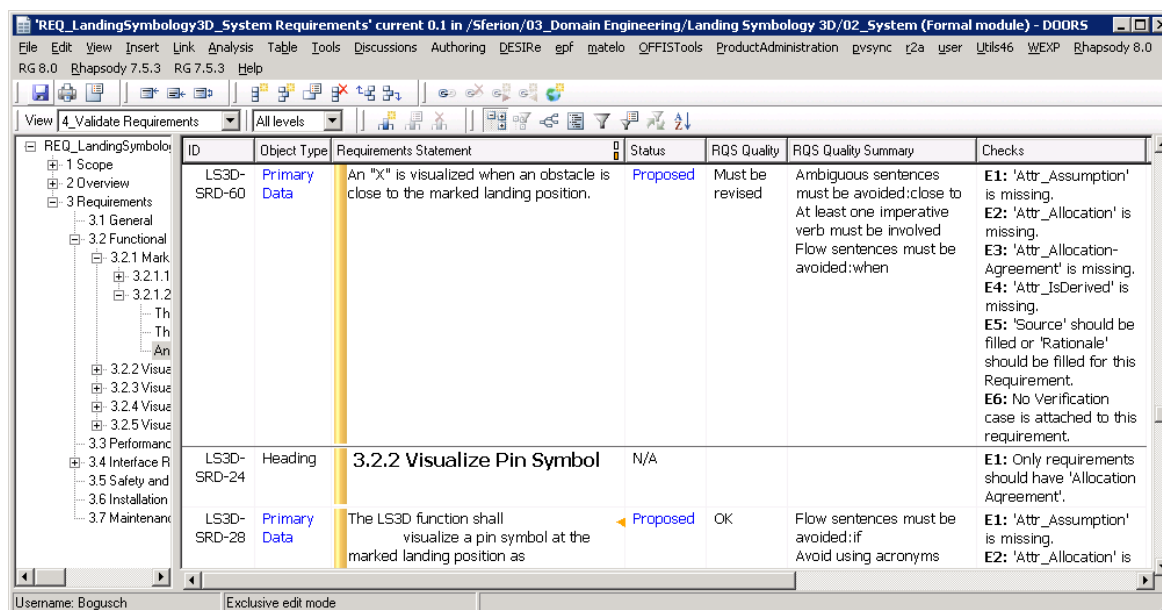


Figure 4-17 Store the quality evaluation results in DOORS

4.4.7 Step 3.7 – Improve requirements

During requirements quality assessment, requirements that must or should be improved are identified. These requirements have to be updated in accordance with the findings. The update can be performed in RQA, but in this case the RAT tool is used. RAT is integrated with the DOORS environment. One can select a requirement for editing in DOORS and open the RAT dialog as shown in **Figure 4-18**. For example, one finding was that the imperative “shall” is missing. After adding the “shall” RAT still complains that the requirement statement is in passive voice. In this way a requirement can be improved until all language defects have been removed. Finally, the improved requirement is stored in the DOORS requirements management repository.

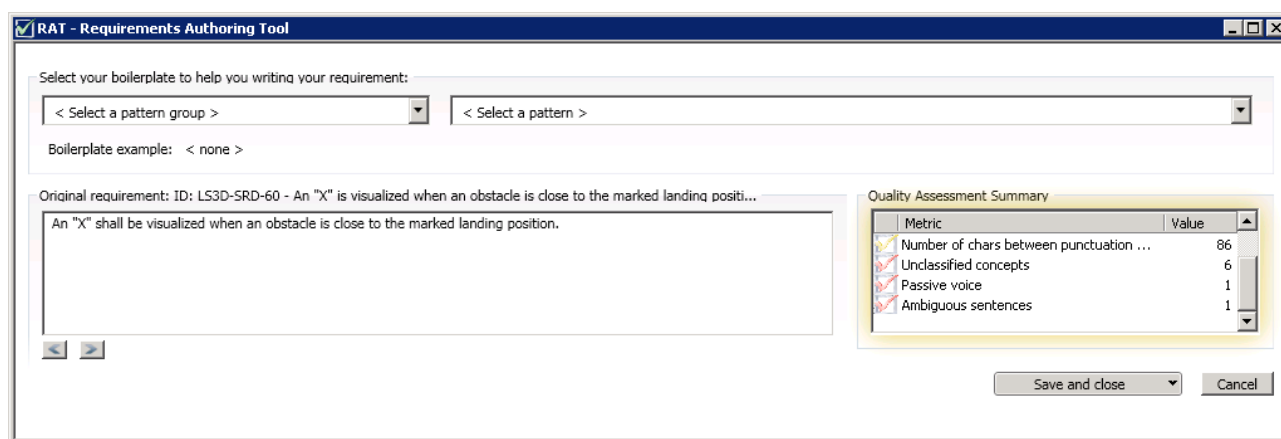


Figure 4-18 Improve requirement using RAT

4.5 Scenario SC4 – Create Product System Requirements

Related user story: US203 – Variability Management

Related engineering methods: EM203_02_01 – Create Product System Requirements

Related tool chain: DOORS, FeatureIDE

The scenario “Create Product System Requirements” covers the activities involved in the product realization. The main goal is to reuse assets previously be developed during domain engineering. In this SEE prototype only system requirements are considered for reuse.

The following steps are performed in this scenario:

- Step 4.1 – Configure product
- Step 4.2 – Create product system requirements

4.5.1 Step 4.1 – Configure product

In this step two potential products are configured – SferiAssist300 (low-cost variant) and SferiAssist500 (high-end variant). **Figure 4-19** and **Figure 4-20** illustrate how the configuration process is performed. Mandatory features like “Check_No_Ground” are automatically selected. A degrees of freedom analysis indicates the number of possible configuration. Text colouring (see green colour in **Figure 4-20**) guide the user through pending decisions.

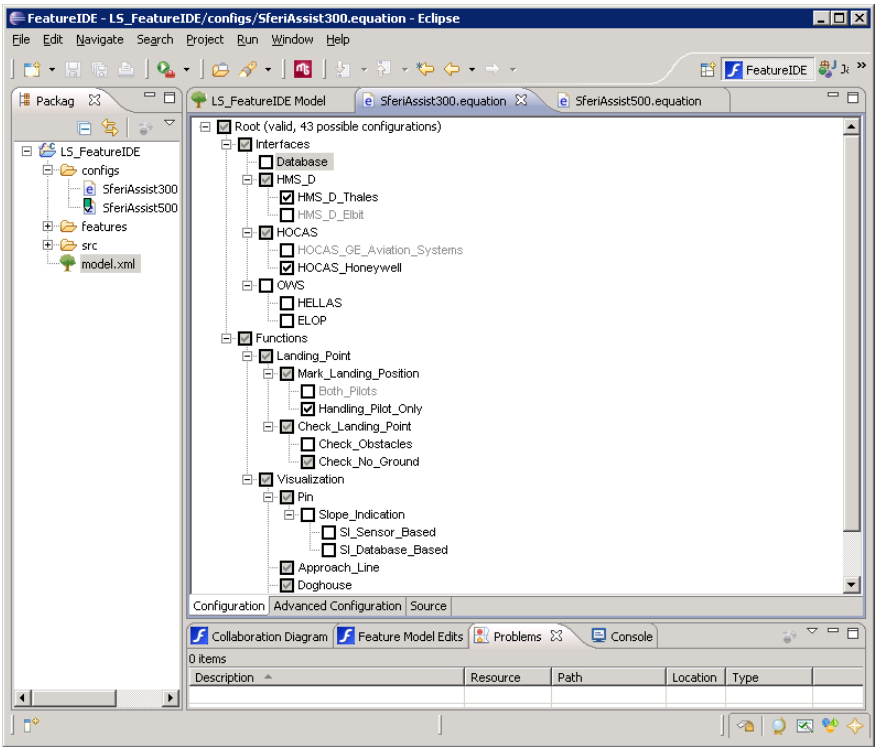


Figure 4-19 SferiAssist300 configuration

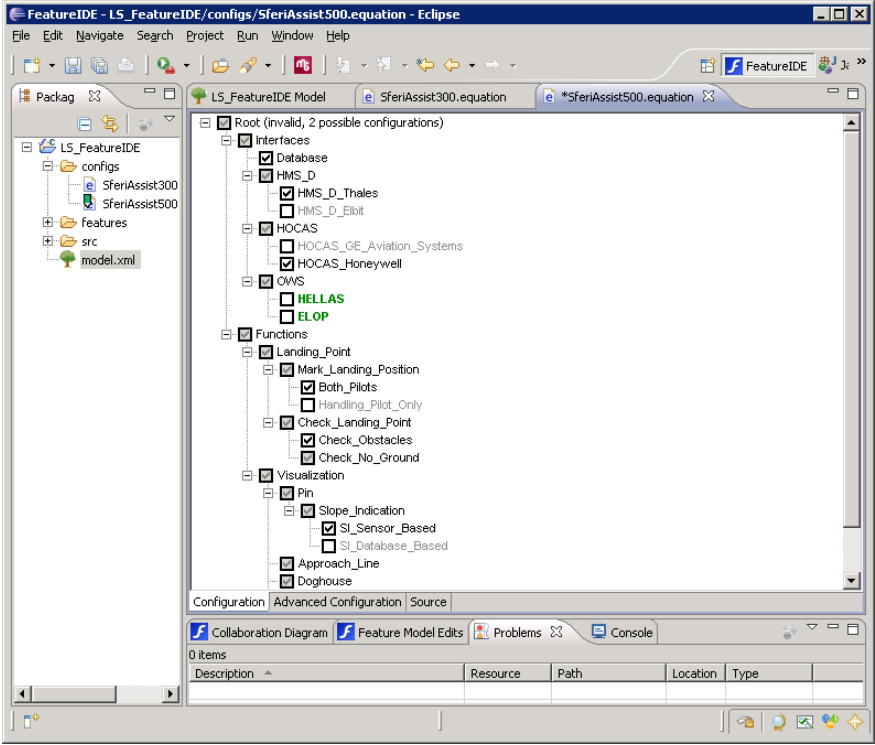


Figure 4-20 SferiAssist500 configuration

4.5.2 Step 4.2 – Create product system requirements

In this step the system requirements that need to be considered for a given product variant are automatically created. A DOORS add-on has been developed that provides support for product family management [Stocker 2011]. The add-on comprises the following functions for this step:

- Read a configuration model (see Step 4.1) and identify all selected features
- Create a new DOORS module containing all requirements relevant for the configured product
- Establish traceability between product requirements and product family requirements
- Synchronize with changes in the feature model, configuration and product family requirements on request

Figure 4-21 and **Figure 4-22** show the generated requirements modules corresponding to the SferiAssist300 and SferiAssist500 configurations. For example, for SferiAssist300 the requirement corresponding to the optional feature “Check_Obstacles” is not present.

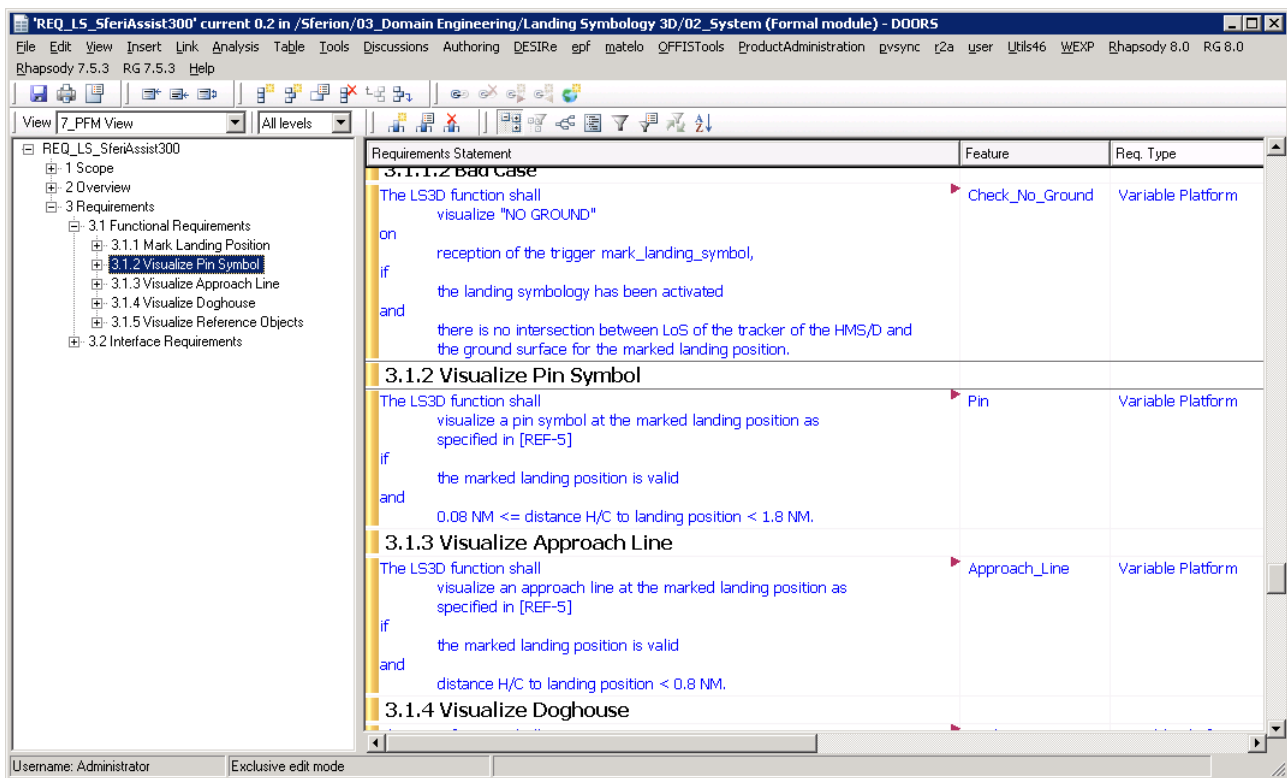


Figure 4-21 SferiAssist300 requirements

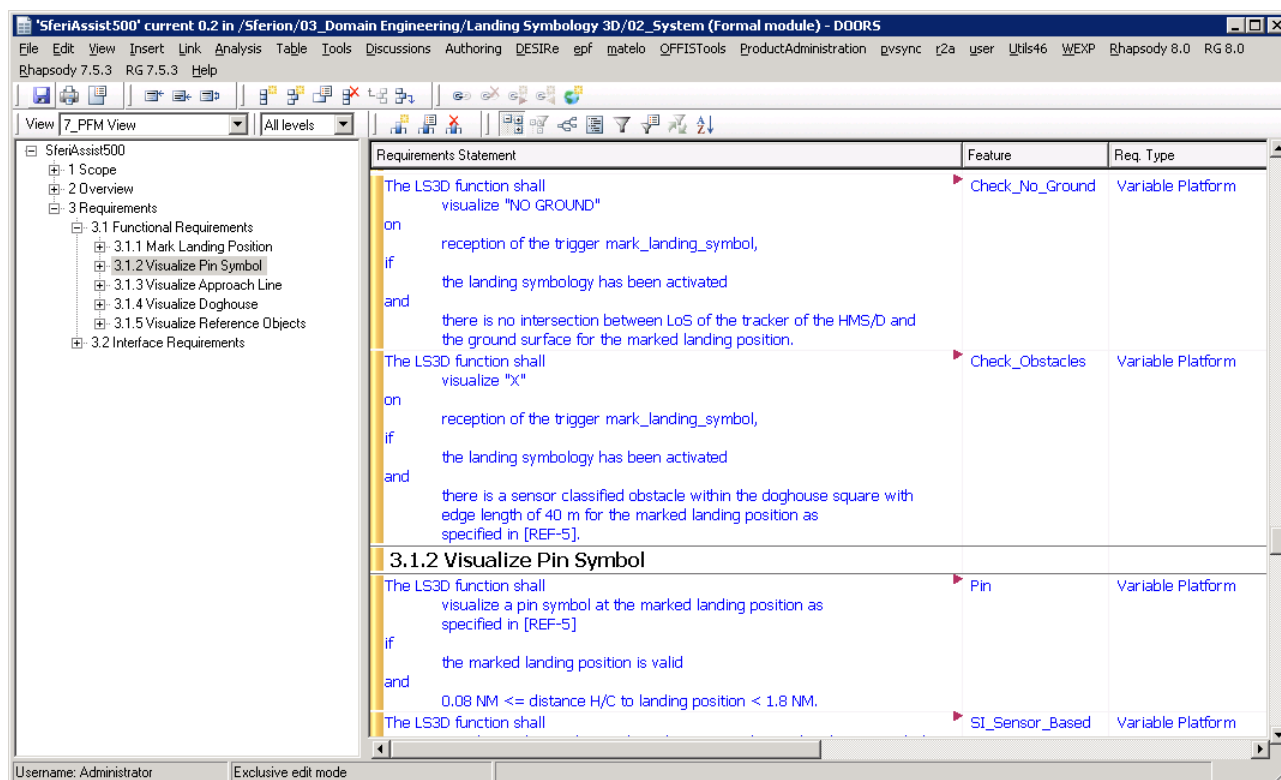


Figure 4-22 SferiAssist500 requirements

4.6 Scenario SC5 – Perform System Functional Analysis

Related user story: US206 – Project Compliance Monitoring based on Advanced Traceability

Related engineering methods: EM206_01_01 – Retrieve Valid Traces, EM206_01_02 – Analyse Trace, EM206_02_01 – Perform Coverage Analysis, EM206_03_01 – Create Verification Objective, EM206_03_02 – Create Verification Case, EM206_03_03 – Create Verification Procedure

Related tool chain: Rhapsody, Rhapsody Gateway, Rhapsody TestConductor, DOORS

The aim of functional analysis is to describe in detail from a technical perspective the functions and behaviour the intended system shall provide, the interaction via identified interfaces with external systems, users and operators, and the interaction and dependencies between the different functions. The main concerns of the functional analysis are:

- What is the functional scope of the system of interest?
- How is the system interacting with the identified operators and external systems?
- Which information is exchanged between the system and the identified operators and external systems?
- What are the sub-functions implementing associated requirements?
- How is the system reacting in normal cases and in abnormal cases?
- What are the critical, top-level performance requirements?

The modelling language is SysML:

- Requirements diagrams to capture the input requirements
- Functional context diagram (top-level use case diagram), showing the system of interest embedded in its environment as well as identifying external systems and operators
- Activity diagrams, depicting detailed activity flows showing how the system (black box) is satisfying the user requirements
- Sequence diagrams, depicting functional interactions between the system of interest and operators or external systems
- Sequence diagrams depicting the performance requirements
- State charts depicting the system modes and their transitions
- Internal block diagrams to identify the interfaces of each function and to depict the decomposition of the function into sub-functions.

The goal of the functional analysis is to identify all functions required by the system in order to perform the use cases and to ensure the coherence to the input requirements. As a result a functional architecture is created for each individual use case. The functional analysis consists of two parts: the use case analysis and the functional analysis for each identified use case.

The use case analysis is performed with the following steps:

- Step 5.1 – Create initial model setup
- Step 5.2 – Import input requirements to the model
- Step 5.3 – Define system context and system-level use cases
- Step 5.4 – Allocate requirements to use cases

The functional analysis is performed for each identified system-level use case with the following steps:

- Step 5.5 – Analyze the use case uninterrupted flow (activity diagram)
- Step 5.6 – Define black box scenarios (sequence diagrams)
- Step 5.7 – Create system external ports and interfaces (internal block diagram)
- Step 5.8 – Define state-based behaviour (state chart)
- Step 5.9 – Verify model by model execution (simulation)

4.6.1 Step 5.1 – Create initial model setup

First, a Rhapsody model is setup in a standardized way. Therefore a common package structure is established to ensure readability and the possibility for report generation. Additionally a CASSIDIAN specific profile based on SysML is applied. **Figure 4-23** shows the resulting model structure in Rhapsody.

Version	Nature	Date	Page
V1.00	R	2014-02-10	43 of 74

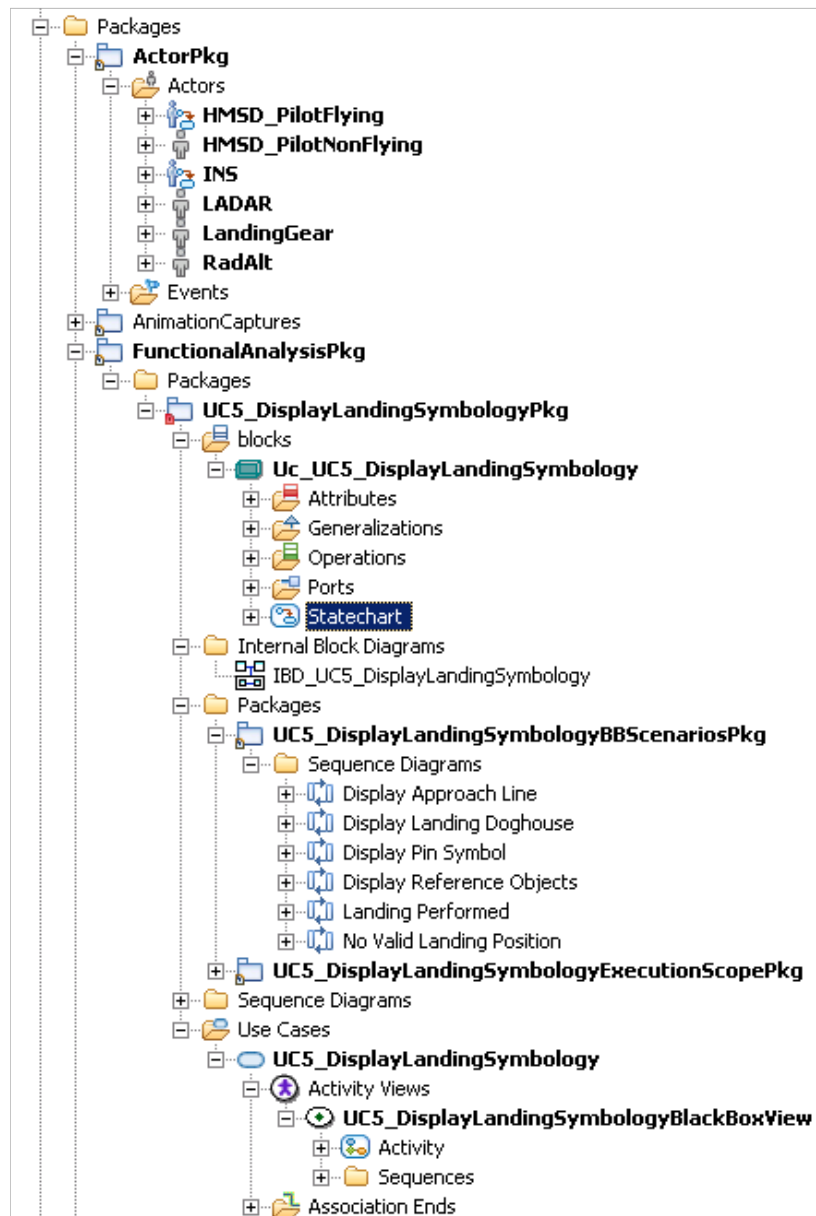


Figure 4-23 Model setup in Rhapsody

4.6.2 Step 5.2 – Import input requirements to the model

In this step system requirements are imported from DOORS into Rhapsody. The goal is to prove the complete consideration of all relevant input requirements during the functional analysis process. The relevant input requirements have been classified as functional requirements in DOORS by using a respective attribute that denotes the requirements type. This allows automatic synchronisation of requirements between DOORS and Rhapsody. The Rhapsody Gateway is used to establish the traceability between the system requirements in DOORS and the model elements describing the functional behaviour in Rhapsody.

With the Rhapsody Gateway coverage and impact analysis can be performed. **Figure 4-24** shows that the functional analysis model covers 83.3% of the input requirements. Two requirements which are not covered

Version	Nature	Date	Page
V1.00	R	2014-02-10	44 of 74

by the model are highlighted in red colour. In addition, when a requirement is selected, the model elements which trace to this requirement are listed in the pane “Downstream Coverage Information”.

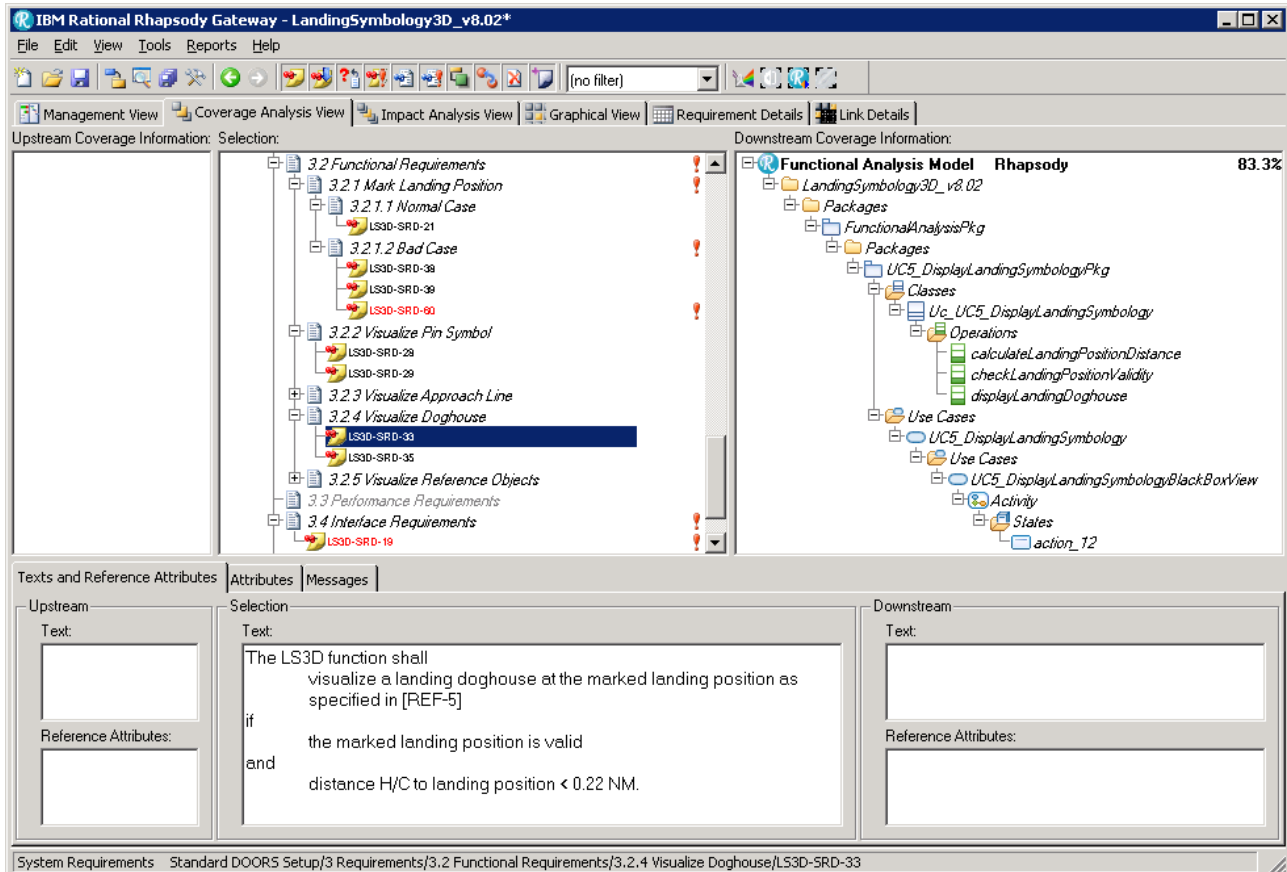


Figure 4-24 Import input requirements from DOORS to Rhapsody using Rhapsody Gateway

4.6.3 Step 5.3 – Define system context and system-level use cases

The purpose of this step is to define the system context with interfacing actors, identify system-level use cases and associate use cases with actors. This modelling step is performed by creating a use case diagram as shown in **Figure 4-25**. Other use case diagrams have been created to identify additional system-level use cases and group the functional scope accordingly. Actors representing external users or systems have been identified.

Each use case comprises additional information including:

- Purpose: a short statement what the primary actor wants to accomplish by performing the use case
- Pre-condition: the system and environment conditions, states and modes present before the use case has been started
- Post-condition: the system and environment conditions, states and modes present after the use case has been completed
- Constraints: any constraints like timing or quality of service identified for the use case
- Description: a detailed description of the activities and interactions for the use case
- Trigger: the event that leads to the activation of the use case

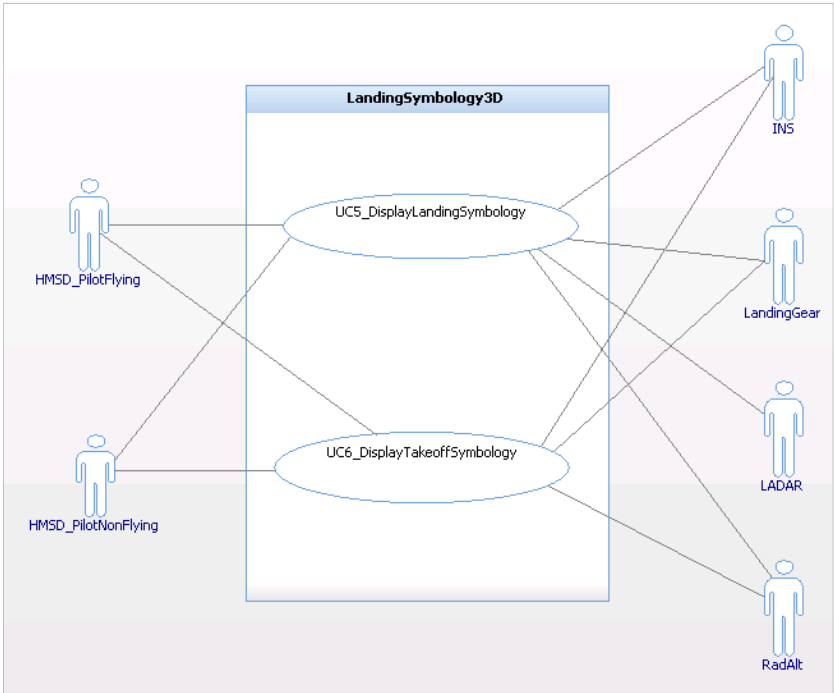


Figure 4-25 Define system context and system level use cases (use case diagram)

4.6.4 Step 5.4 – Allocate requirements to use cases

The functional system requirements are linked to the use case with a <<trace>> dependency. This is the basis to perform coverage and impact analysis. **Figure 4-26** depicts an excerpt of a use case diagram showing the use case and some of its allocated requirements.

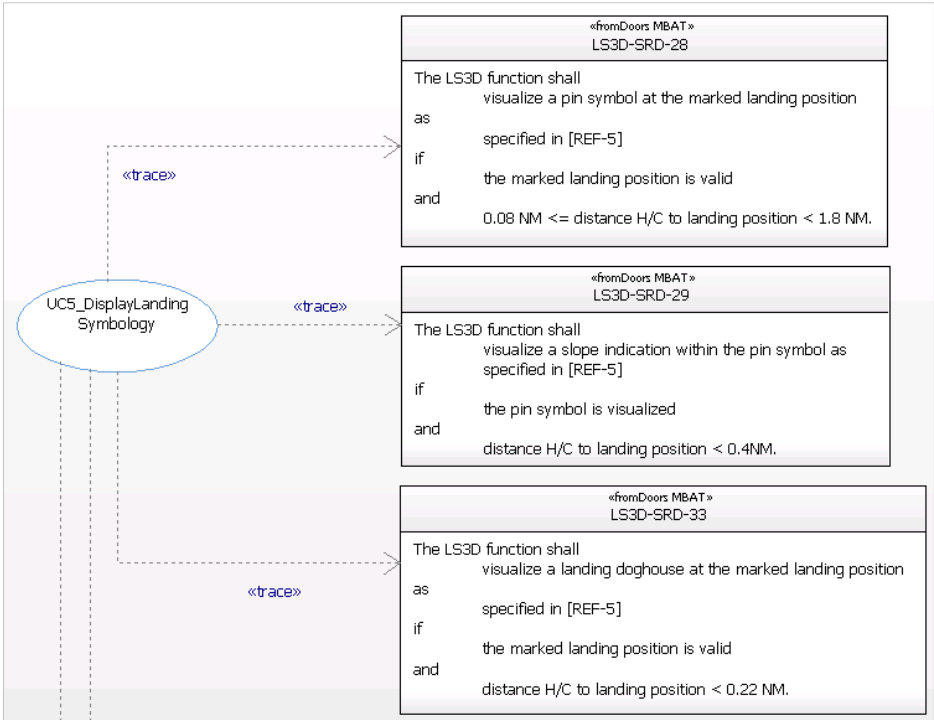


Figure 4-26 Allocate system requirements to system-level use cases (use case diagram)

The steps described in the subsequent sections are performed for each use case.

4.6.5 Step 5.5 – Analyze the use case uninterrupted flow

Within this modelling step all functions required to execute a specific use case as well as the interaction with the associated actors (external interfaces) are identified. Therefore, the uninterrupted flow is captured with an activity diagram. The focus of activity diagrams is to capture the different functional flows from start to end, capturing the involved functions without considering function loops and timeouts, which will be captured later.

Before completing this step, a first set of <<satisfy>> dependencies from actions to requirements is established. In this way it can easily be checked whether one of the <<traced>> requirements has been forgotten in the use case. **Figure 4-27** shows an example.

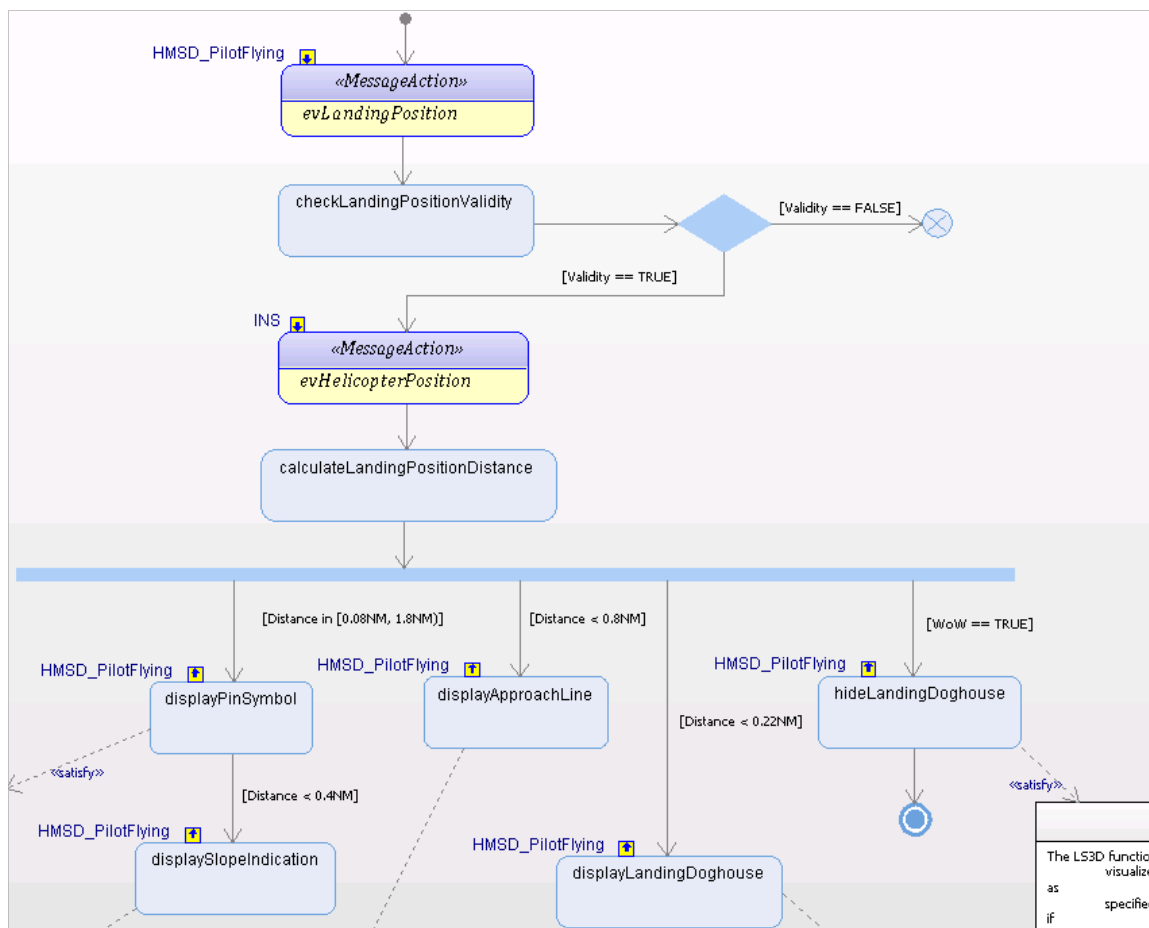


Figure 4-27 Analyze the use case uninterrupted flow (activity diagram)

4.6.6 Step 5.6 – Define black box scenarios

The purpose of this step is to derive a set of sequence diagrams consistent with the activity diagram. Operations are derived from actions allocated to the block representing the use case. Further, events and related event receptions are defined in relation with external actors. The identified events are linked to data items that are transported across physical interfaces.

Initial sequence diagrams are created automatically from the activity diagram representing the functional flows using a toolkit which is based on HarmonySE toolkit.

Within the sequence diagram neither model elements (operations and events) shall be created, nor shall the order of model elements be changed, inconsistently with the activity diagram. Interaction operators may be used to indicate loops, optional or alternative sequences.

Each scenario describes the behaviour of the system in a particular situation. Therefore, a scenario describes the behaviour for a specific use case with defined assumptions and pre-conditions. The following information is provided for each scenario:

- Purpose: a short statement that describes the purpose of the scenario.
- Pre-condition: the system and environment conditions, states and modes present before the sequence starts
- Post-condition: the system and environment conditions, states and modes present after the sequence is finished
- Scenario type: sunny day / rainy day.

Figure 4-28 depicts an example of a black box scenario.

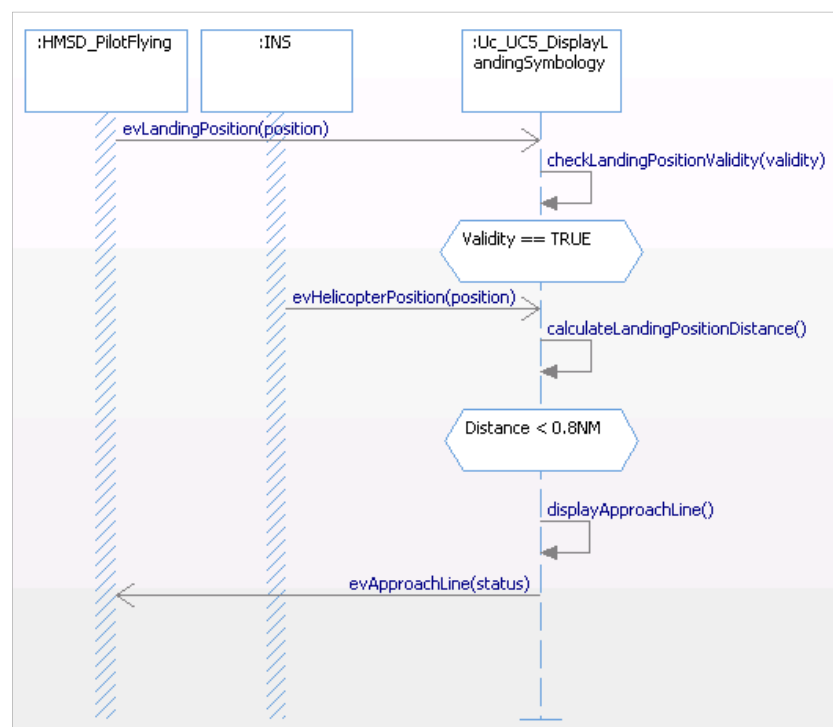


Figure 4-28 Define black box scenarios (sequence diagram)

4.6.7 Step 5.7 – Create system external ports and interfaces

The goal of this modelling step is the definition of ports and interfaces, including all in- and outgoing events. The result is shown in an internal block diagram.

With all identified sequence diagrams the system ports and interfaces can be defined. In Rhapsody this is done automatically using a toolkit which is based on HarmonySE toolkit. An example is shown in **Figure 4-29**.

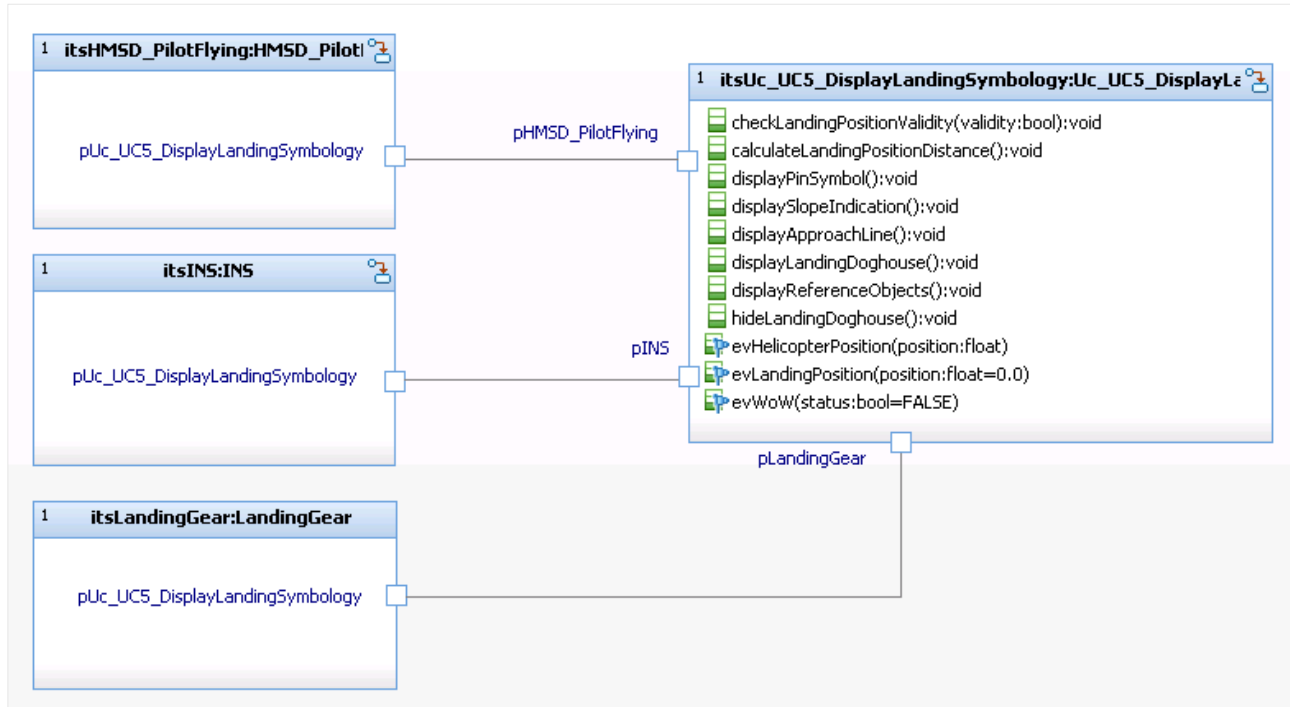


Figure 4-29 Create system external ports and interfaces (internal block diagram)

4.6.8 Step 5.8 – Define state-based behaviour

To complete the functional analysis, the use case is described by its state-based behaviour. In the Service Request Driven Model workflow the state machine diagram is considered as the most important behaviour diagram, as it aggregates the information from both, the activity diagram (functional flow) and the sequence diagrams (interactions with the environment), and adds to it the event driven block behaviour (loops, failure states, timeouts,...). As the semantic of state charts is formally defined, the correctness and completeness of the resulting behaviour can be verified through model execution. State chart diagrams are finite state machines that are extended by the notation of

- Hierarchy
- Concurrency

Basically, a state chart diagram is composed of a set of states joined by transitions and various connectors. An event may trigger a transition from one state to another. Actions can be performed on transitions and on state entry/exit. A part of the state chart defined for the Landing Symbology function is shown in **Figure 4-30**.

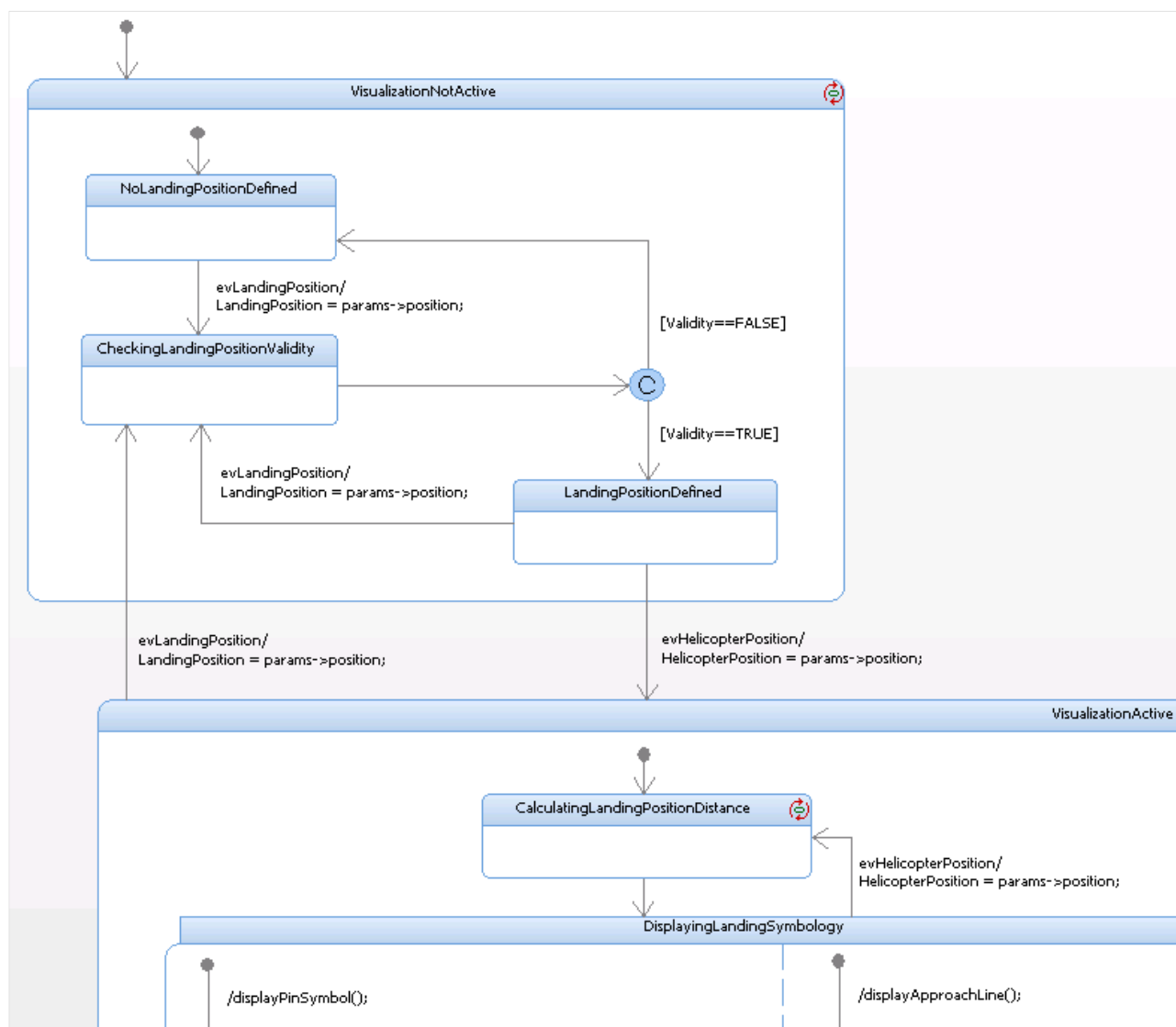


Figure 4-30 Define state-based behaviour (state chart diagram)

4.6.9 Step 5.9 – Verify model by model execution

Model execution is a powerful method to check the semantic correctness as well as the completeness of the functional analysis and is based on the visual inspection of the model behaviour (animated state machines and sequence diagrams or equivalent methods). Verification of the state-based behaviour for the use cases provides evidence that the behaviour is as expected and consistent with the previously defined activity and sequence diagrams and that no deadlocks occur.

TestConductor, the test execution and verification engine of the Rhapsody environment, is employed in order to verify that the simulation is compliant with the expected behaviour as defined by the input requirements. It executes test cases defined by sequence diagrams, flow charts, state charts or source code. During test execution TestConductor verifies the results against defined requirements.

TestConductor implements the UML Testing Profile (UTP) and provides wizards that support creating test architectures for a selected system under test. **Figure 4-31** exemplifies the structure of the Rhapsody model which is enhanced by the test architecture. The test architecture comprises all artefacts needed for test

automation. For example, test components are created for stubbing of external interfaces. They allow generating test stimuli and observing the behaviour of the system under test.

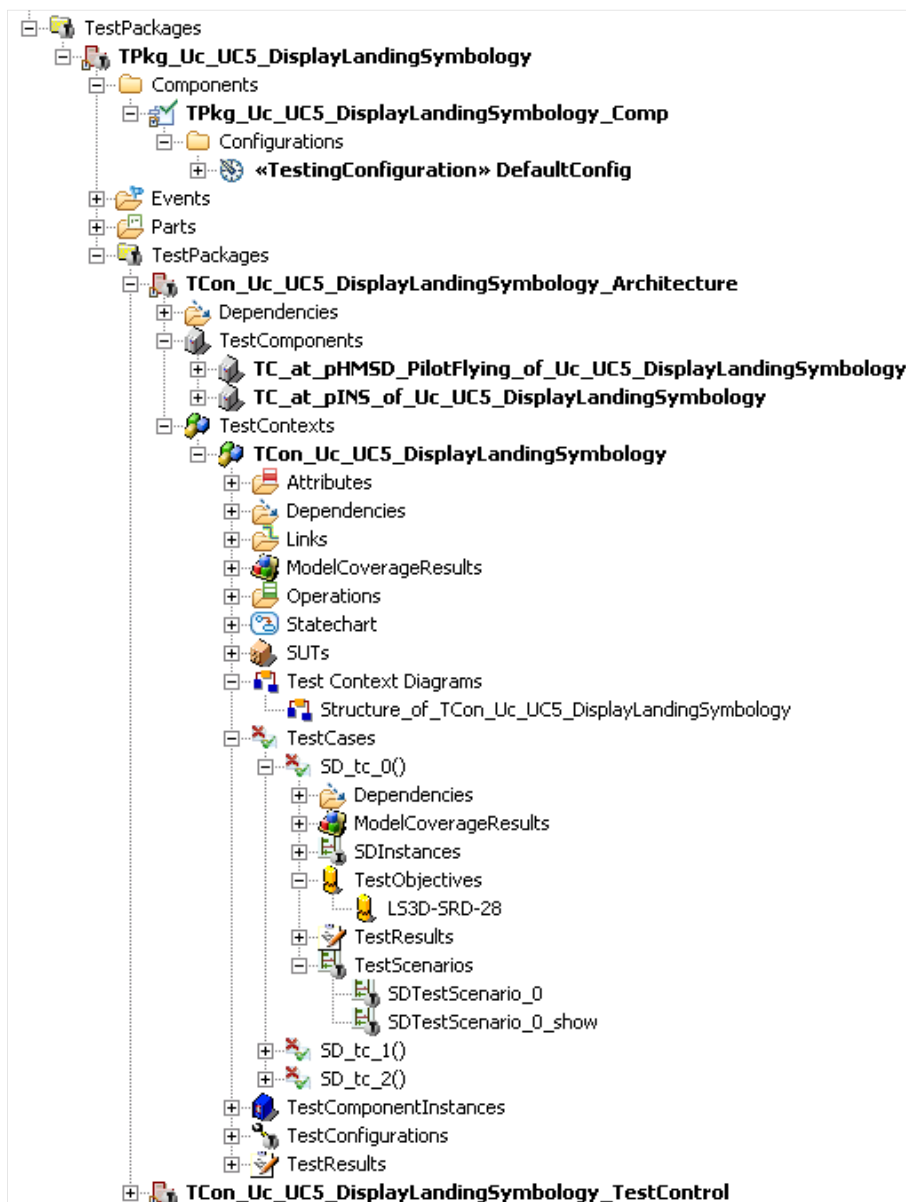


Figure 4-31 Enhanced model setup using UML testing profile

A test context provides a definition of the test environment, see **Figure 4-32**. Moreover, it contains all created test cases. Each test case consists of

- Test objectives: requirements that are covered by the test case
- Test scenarios: definition of the test steps including test stimuli and expected results
- Test results: test verdict

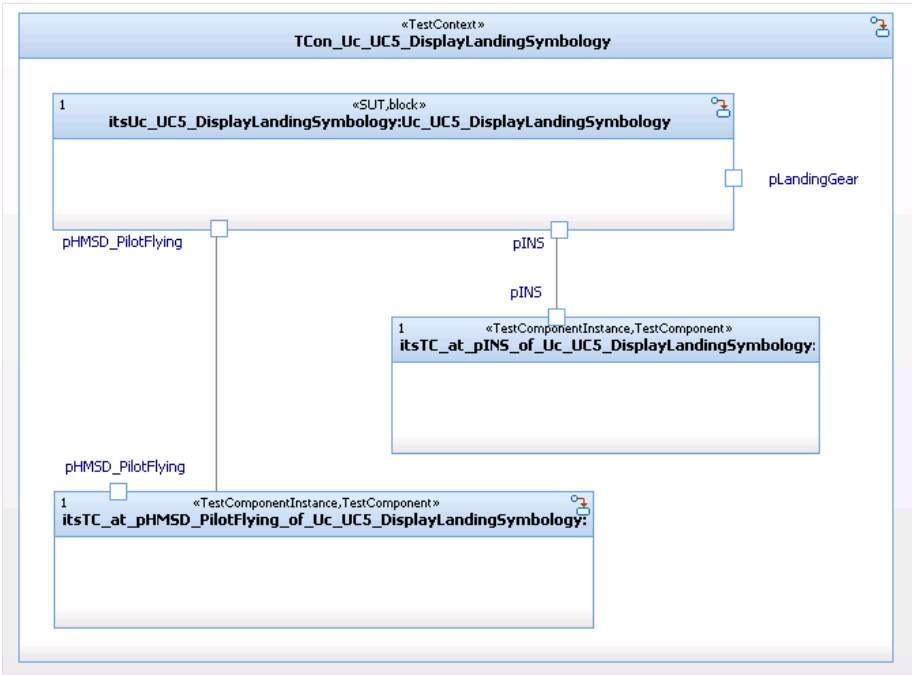


Figure 4-32 Generate test context for the system under test (internal block diagram)

Based on the test context various test scenarios can be defined. **Figure 4-33** depicts a sequence diagram test case. In this test case events sent from external actors are defined as test stimuli. In addition the expected system reaction of the system under test (marked with red colour) is defined. During test execution the state chart is animated to indicate the current state of the system. The events of the test scenario are coloured in green (test step is passed) or red (test step is failed), respectively. This is shown in **Figure 4-34**. The test verdict (test case is passed / failed) is indicated after termination of the test case. Due to the test automation test suites can be defined for regression testing. When the model has been changed, regression testing allows confirming that the model still satisfies its requirements after the change.

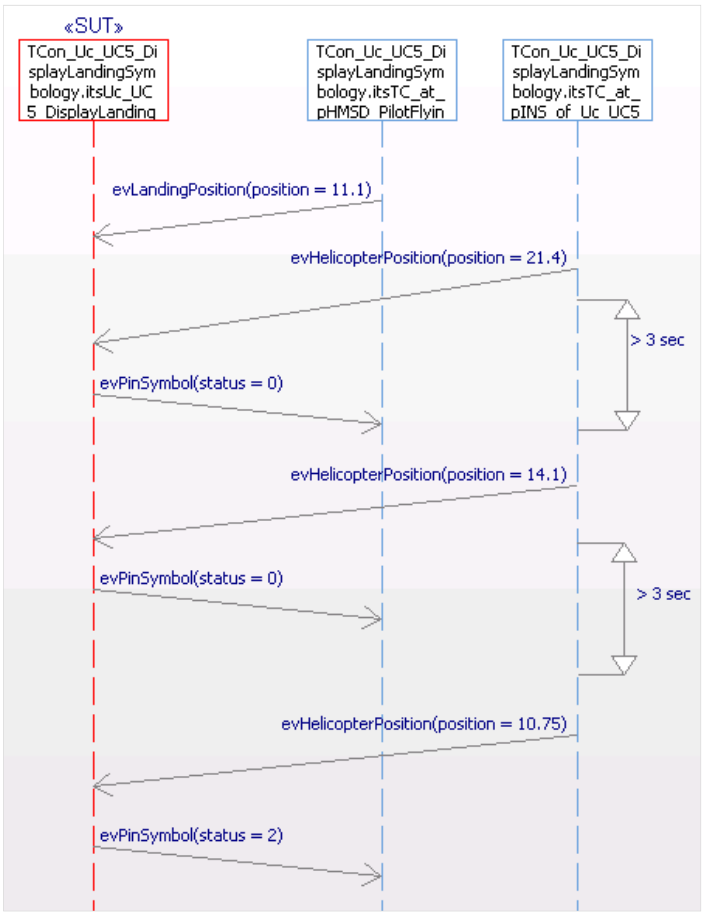


Figure 4-33 Define test scenarios for the system under test (sequence diagram)

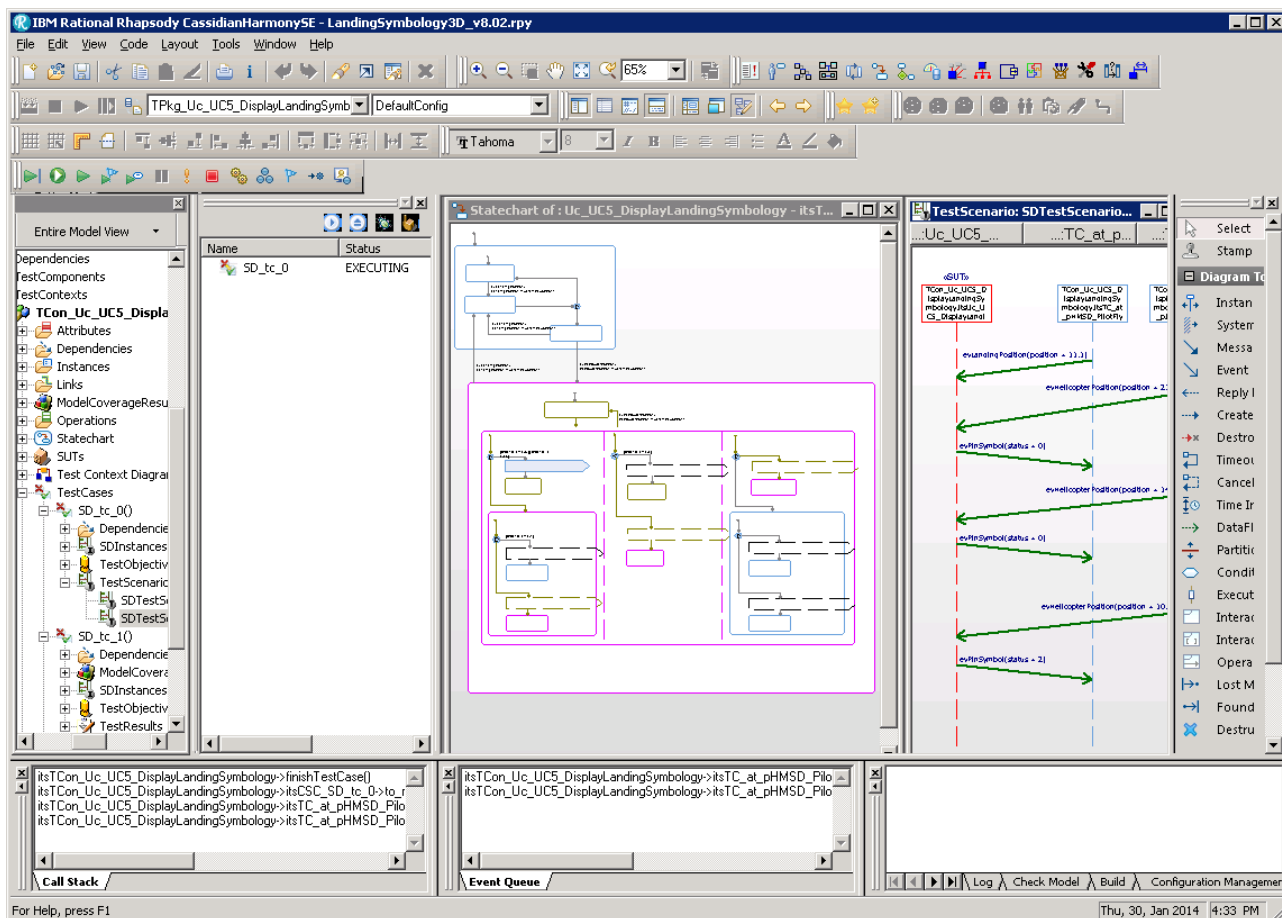


Figure 4-34 Execute tests in Rhapsody using simulation of state-based behaviour

Requirements coverage as well as model coverage (model elements covered / not covered during test execution) can be shown for all test cases of the text context. See **Figure 4-35** for an example.

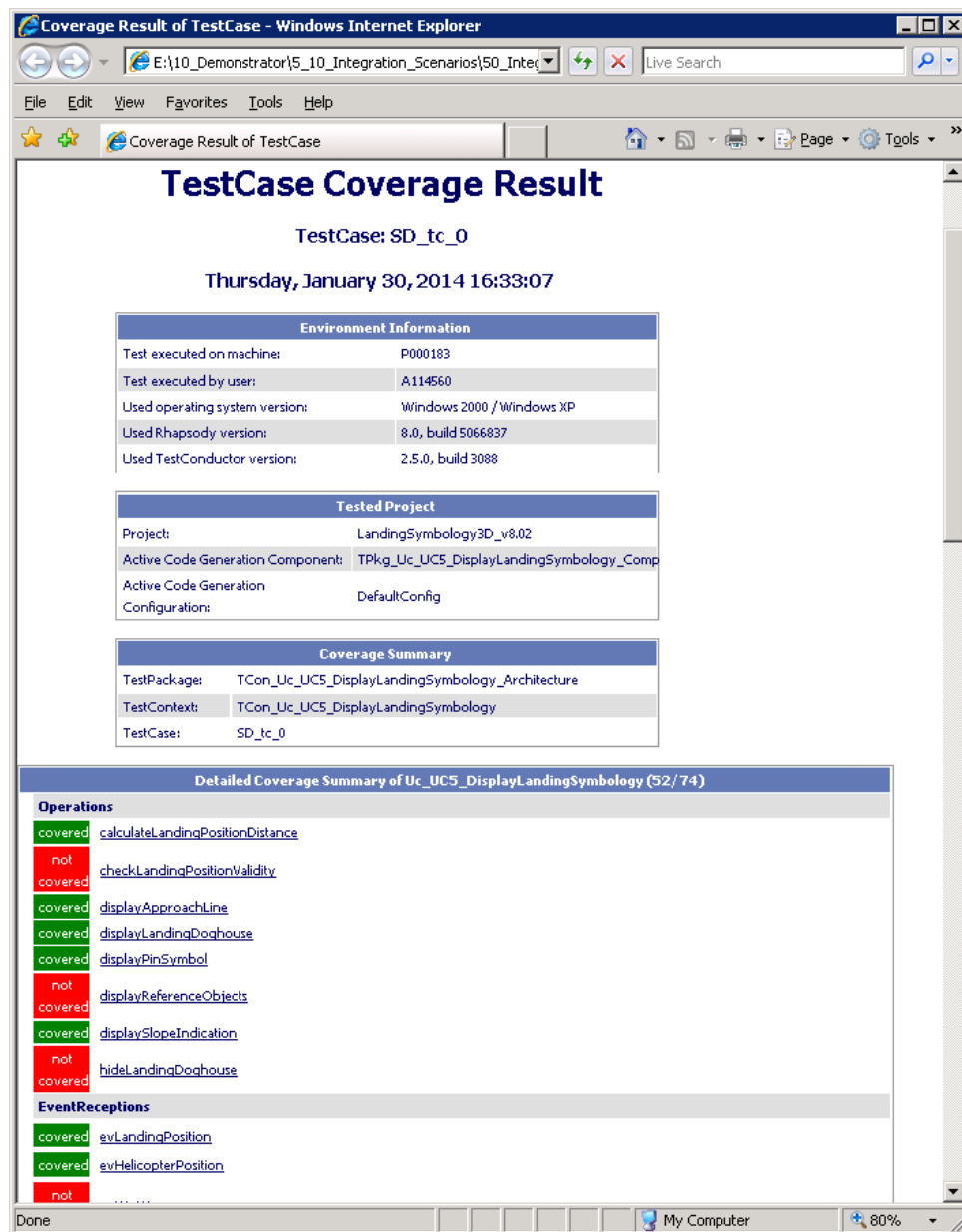


Figure 4-35 Analyze model-based coverage of test cases

4.7 Scenario SC6 – Perform Report Generation

Related user story: US206 – Project Compliance Monitoring based on Advanced Traceability

Related engineering methods: N/A

Related tool chain: RPE

The aim of this scenario is the generation of specification documents capturing the information gathered during the functional analysis (see section 4.6) in a structured way. The documents can be used to perform formal reviews, fulfil contractual obligations or show regulatory compliance.

The following steps are performed in this scenario:

- Step 5.1 – Define template for document generation
- Step 5.2 – Create document

4.7.1 Step 6.1 – Define template for document generation

RPE provides a graphical template editing environment for custom report design. An example of the RPE Document Studio is shown in **Figure 4-36**.

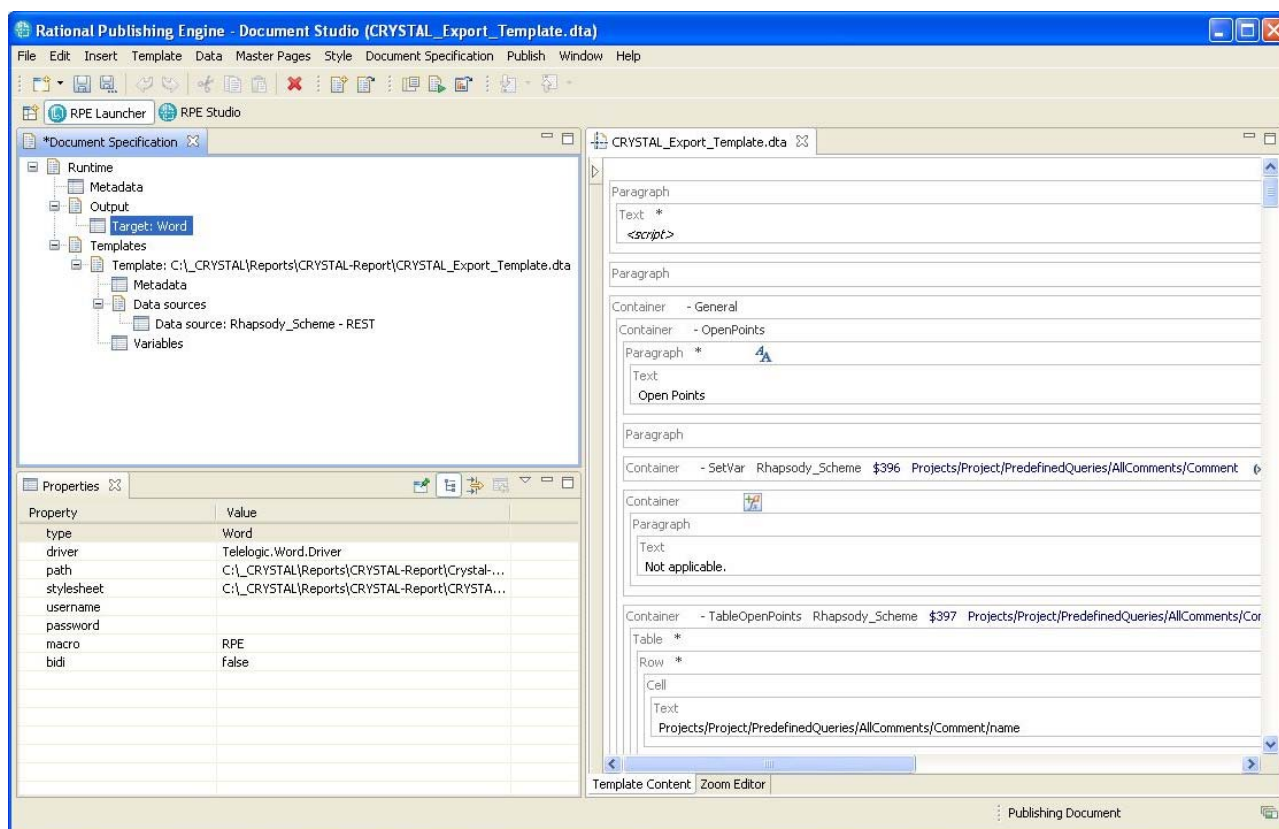


Figure 4-36 Define the RPE template for document export in RPE Document Studio

4.7.2 Step 6.2 – Create document

During report generation data is extracted from a single source or combined from multiple sources. Different output formats such as Microsoft Word, Adobe PDF or HTML are supported.

4.8 Scenario SC7 – Provide Process Guidance

Related user story: US205 – Process Automation, Guidance and Monitoring

Related engineering methods: N/A

Related tool chain: EPF Composer

In this section the systems engineering process and guidance framework is described. The conceptual framework used for the definition of the systems engineering process and the associated guidance is based on SPEM 2.0, EPF, Practices, UMF and MAM. The basic principles are summarized below:

SPEM – Software & Systems Process Engineering Meta-Model

The SPEM 2.0 specification was released by the Object Management Group (OMG) in 2008. It constitutes a process engineering meta-model as well as conceptual framework, which can provide the necessary concepts for modelling, documenting, presenting, managing, interchanging, and enacting development methods and processes. An implementation of this meta-model is targeted at process engineers, project leads, project and program managers who are responsible for maintaining and implementing processes for their development organizations or individual projects.

The usage of a SPEM 2.0 implementation is:

- Manage libraries of reusable method content: Developers need to understand the methods and key practices of development. They need to be familiar with the basic development tasks, such as how to elicit and manage requirements. They further need to understand the work products from such development tasks as well as which skills are required to perform those tasks. SPEM 2.0 enables development practitioners to manage and deploy their knowledge using a standardized format.
- Develop and manage processes for performing specific development lifecycles: Development teams need to define how to apply their development methods and best practices throughout a project lifecycle. For example, requirements management methods have to be applied in one fashion during the early phases of a project, where the focus is more on elicitation of stakeholder needs and requirements and scoping a vision. The same methods have to be performed in a different fashion during later phases, where the focus is on managing requirements updates and changes and performing impact analysis of these requirements changes. The same requirements methods might also be applied differently if the project develops a new system or maintains an existing system as well as depending on the teams and distribution of the teams. A development process model needs to support expressing these differences. SPEM 2.0 supports the systematic creation of processes based on reusable method content. Lifecycle independent method content can be placed into a process for a specific development lifecycle. Such processes can be represented as workflows and/or breakdown structures.
- Configure and deploy a process framework customized for the project needs: No development project is exactly like another. Organizations can provide libraries of reusable method content and processes. Team leads can then select and tailor the method content and processes they require. They can then describe these selections and customizations with a SPEM 2.0 method configuration, which they can deploy to their teams, only providing the content they really need.
- Create project plan templates for enactment of the process in the context of the project: Processes as well as guiding method content need to be available in the context of daily work of project managers, technical leads, and developers. They therefore need to be deployed in formats that are ready for enactment. Typical enactment systems are project and resource planning systems, work backlog tracking systems, and workflow engines. SPEM 2.0 provides process definition structures that allow process engineers to express how a process shall be enacted within these systems. For example, SPEM 2.0 process definition can include information that indicates that modelled work definitions shall be repeated several times in a project or that there could be multiple occurrences of work definitions that can be performed in parallel.

Version	Nature	Date	Page
V1.00	R	2014-02-10	57 of 74

EPF – Eclipse Process Framework

The EPF project (www.eclipse.org/epf) is an Eclipse Technology open source project that aims to provide an extensible framework and exemplary tools based on SPEM 2.0 concepts for defining and managing development processes. Within this framework, the project develops extensible process content for a range of software development and management processes supporting iterative, agile, and incremental development, and applicable to a broad set of development platforms and applications.

Practices

Practices enable a compositional approach to building methods. They are intended as process chunks for adoption, configuration and enactment. The practice approach offers the following benefits:

- **Adaptability and Scalability:** Practices can be adapted to support a range of solutions. In particular, practices can be adapted to suit your organization and supplemented by your own practices.
- **Incremental Adoption:** Each practice is described as a standalone capability that can be adopted by an organization or project. Each practice may include enablement materials that explain how to get started.
- **Easy to Configure and Use:** Creating a method is as simple as selecting the practices that you wish to adopt, and then publishing the results. Each practice adds itself into the framework so that content can be viewed by practice or across practices by work product, role, task and so on.
- **Community Development:** Since a practice can be easily authored on its own, practices are ideal for community development.

UMF – Unified Method Framework

The UMF is a practice framework in which different practices from many different contexts and developed by many different organizations can co-exist, sharing a common infrastructure for the interoperation of practices. This ultimately supports the objective to develop re-usable method content which can be integrated in a knowledge base sharing process knowledge. The benefits of the UMF are the same as for any practice framework:

- Defines a *consistent approach* for how plug-ins are structured that allows *plug-and-play* between content authored by different groups and ensures that remotely authored content integrates seamlessly into the overall library.
- *Reduces complexity* and *increases understandability* of the methods as all methods are constructed/structured in a similar way.
- *Maximizes reuse* as common elements are shared across practices and practices are shared across processes. Practices also provide a coarser-grained unit of reuse and customization than just work products.
- Increases configurability as practices can be easily configured to produce many different types of method assets to match specific needs. Practices are *loosely coupled* and *interchangeable*. Practices are easily “swapped out” and can be “mixed and matched” to create the best solution. Specifically, processes can be assembled to best suit the end user's needs.
- Supports *incremental method authoring*. Practices are written independently from each other. Practices are dependent on a shared core and not on each other.
- Supports *incremental adoption of a process*. The process is divided into practices that can be adopted individually and incrementally. You can start small with a few practices and then grow/scale, adopting one practice at a time.

Version	Nature	Date	Page
V1.00	R	2014-02-10	58 of 74

MAM – Method Authoring Method

The MAM is a practice-based approach to method authoring. It provides guidelines for authoring methods compliant with the UMF and is directly supported by the EPF Method Composer. The scope of the MAM includes all content from taking set of method requirements to producing a method that is ready for deployment.

The following steps are performed in this scenario:

- Step 7.1 – Create method contents and practice library
- Step 7.2 – Define delivery process
- Step 7.3 – Publish delivery process

4.8.1 Step 7.1 – Create method contents and practice library

In this step the basic building blocks of the process model are defined according to the SPEM meta model. Such building blocks are tasks, work products, roles or tools. Tasks can be aggregated to capability patterns, which constitute reusable process chunks. For each building block detailed information is provided.

An EPF model has been created that defines the process activities according to company-specific systems engineering guidelines. **Figure 4-37** illustrates the definition of method contents using the EPF Composer.

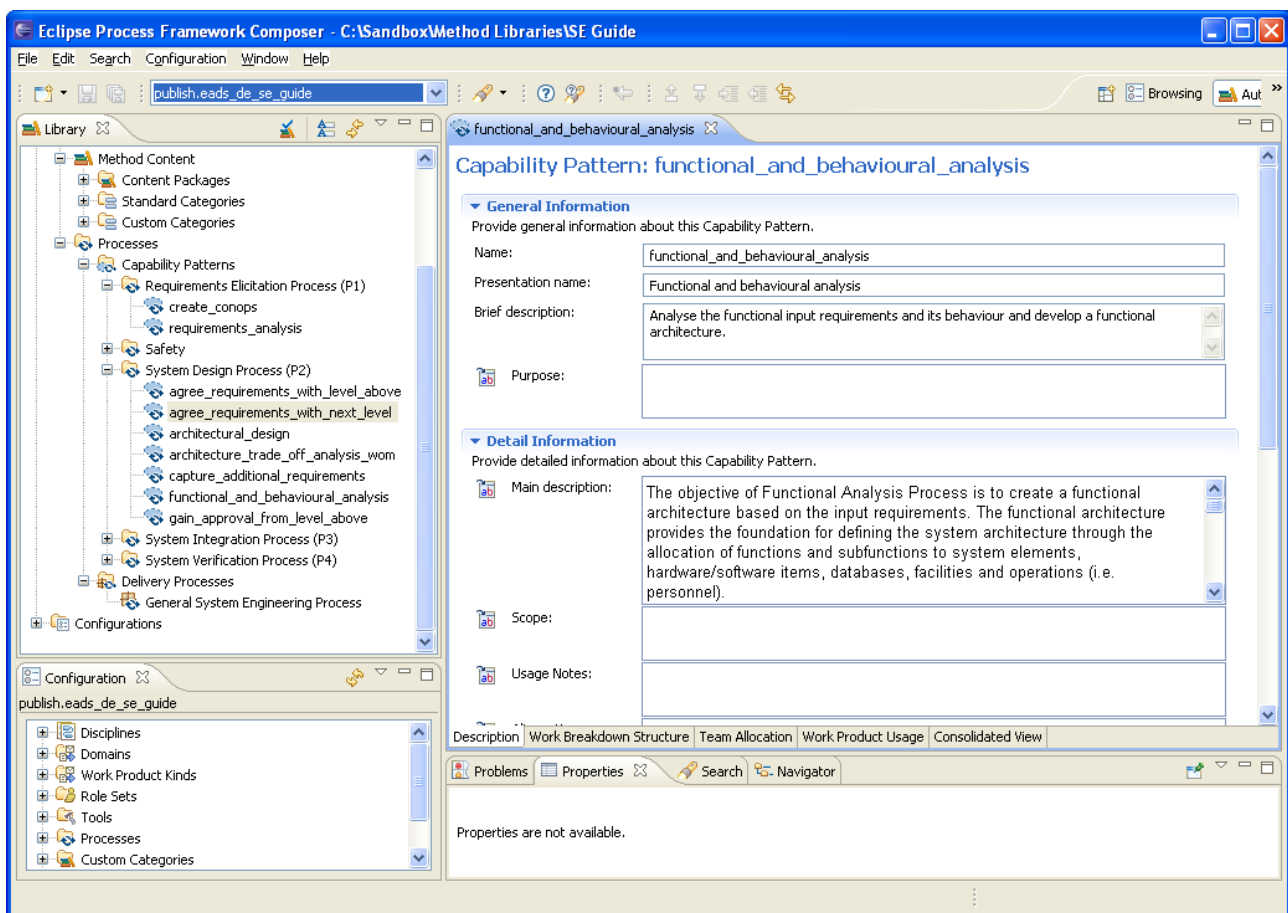


Figure 4-37 Define method contents using EPF Composer

4.8.2 Step 7.2 – Define delivery process

In this step existing capability patterns are combined in order to build an overall process. The resulting work breakdown structure is depicted in **Figure 4-38**.

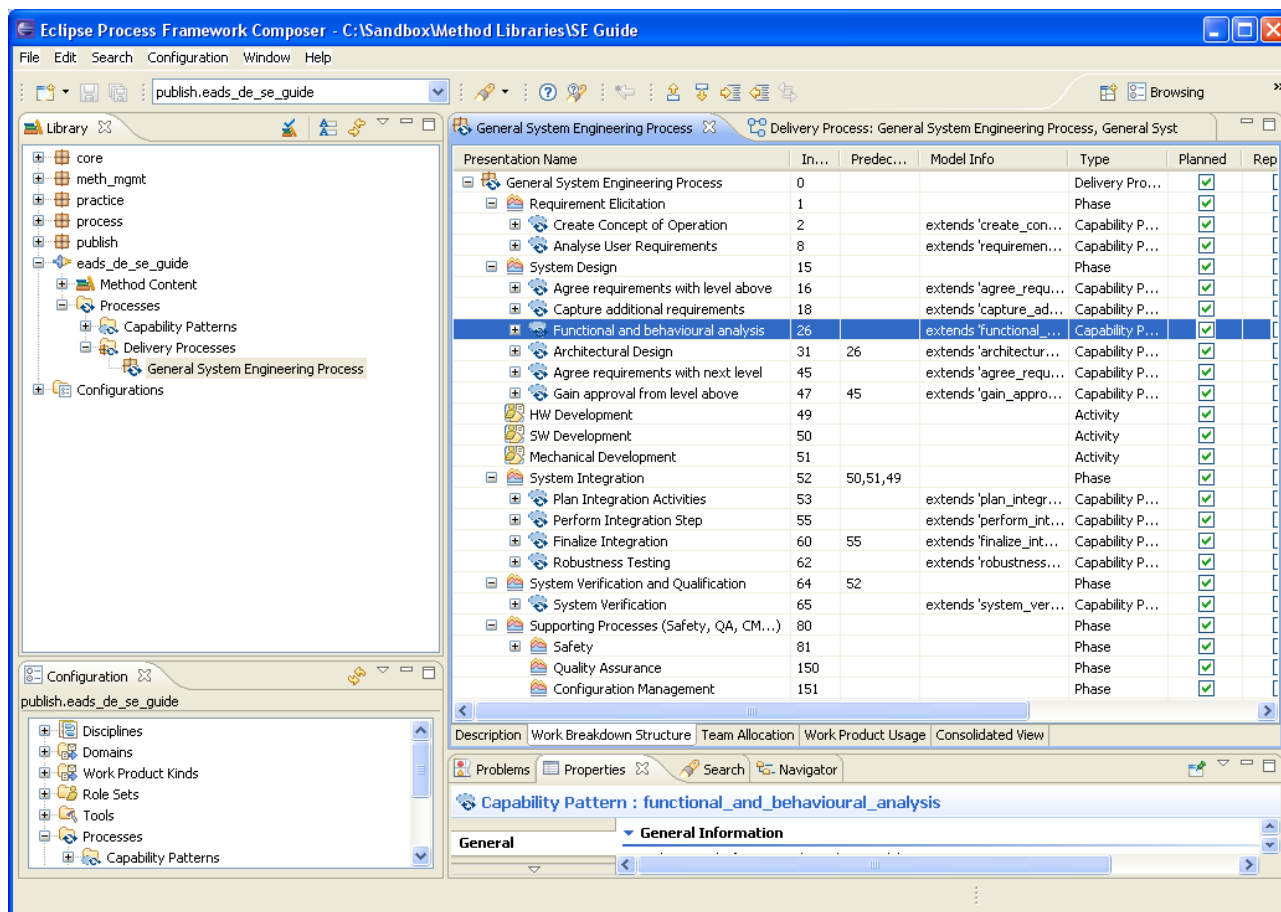
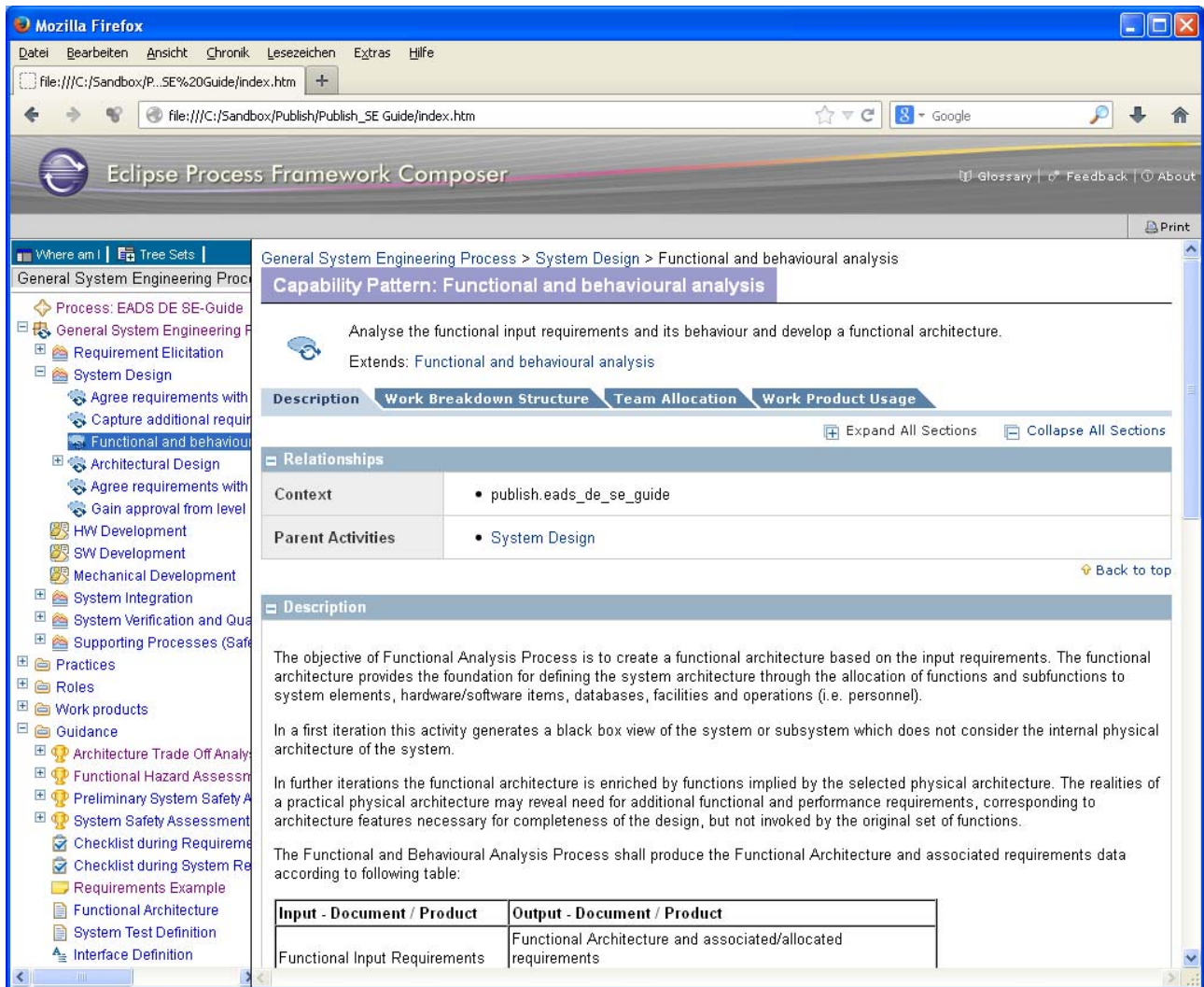


Figure 4-38 Define delivery process using EPF Composer

4.8.3 Step 7.3 – Publish delivery process

In this step the defined delivery process (see Step 7.2) is published into a collection of HTML pages. These HTML pages provide the documentation of the process for the development team, see **Figure 4-39**. The published delivery process may also contain practises which provide guidance on specific topics (e.g. methods like Functional Hazard Assessment) as shown in **Figure 4-40**. Practices can be accompanied by other guidance items, e.g. examples, guidelines, and checklists.



General System Engineering Process > System Design > Functional and behavioural analysis

Capability Pattern: Functional and behavioural analysis

Analyse the functional input requirements and its behaviour and develop a functional architecture.
Extends: Functional and behavioural analysis

Description | **Work Breakdown Structure** | **Team Allocation** | **Work Product Usage**

Expand All Sections | Collapse All Sections

Relationships

Context	• publish.eads_de_se_guide
Parent Activities	• System Design

Back to top

Description

The objective of Functional Analysis Process is to create a functional architecture based on the input requirements. The functional architecture provides the foundation for defining the system architecture through the allocation of functions and subfunctions to system elements, hardware/software items, databases, facilities and operations (i.e. personnel).

In a first iteration this activity generates a black box view of the system or subsystem which does not consider the internal physical architecture of the system.

In further iterations the functional architecture is enriched by functions implied by the selected physical architecture. The realities of a practical physical architecture may reveal need for additional functional and performance requirements, corresponding to architecture features necessary for completeness of the design, but not invoked by the original set of functions.

The Functional and Behavioural Analysis Process shall produce the Functional Architecture and associated requirements data according to following table:

Input - Document / Product	Output - Document / Product
Functional Input Requirements	Functional Architecture and associated/allocated requirements

Figure 4-39 View published method contents

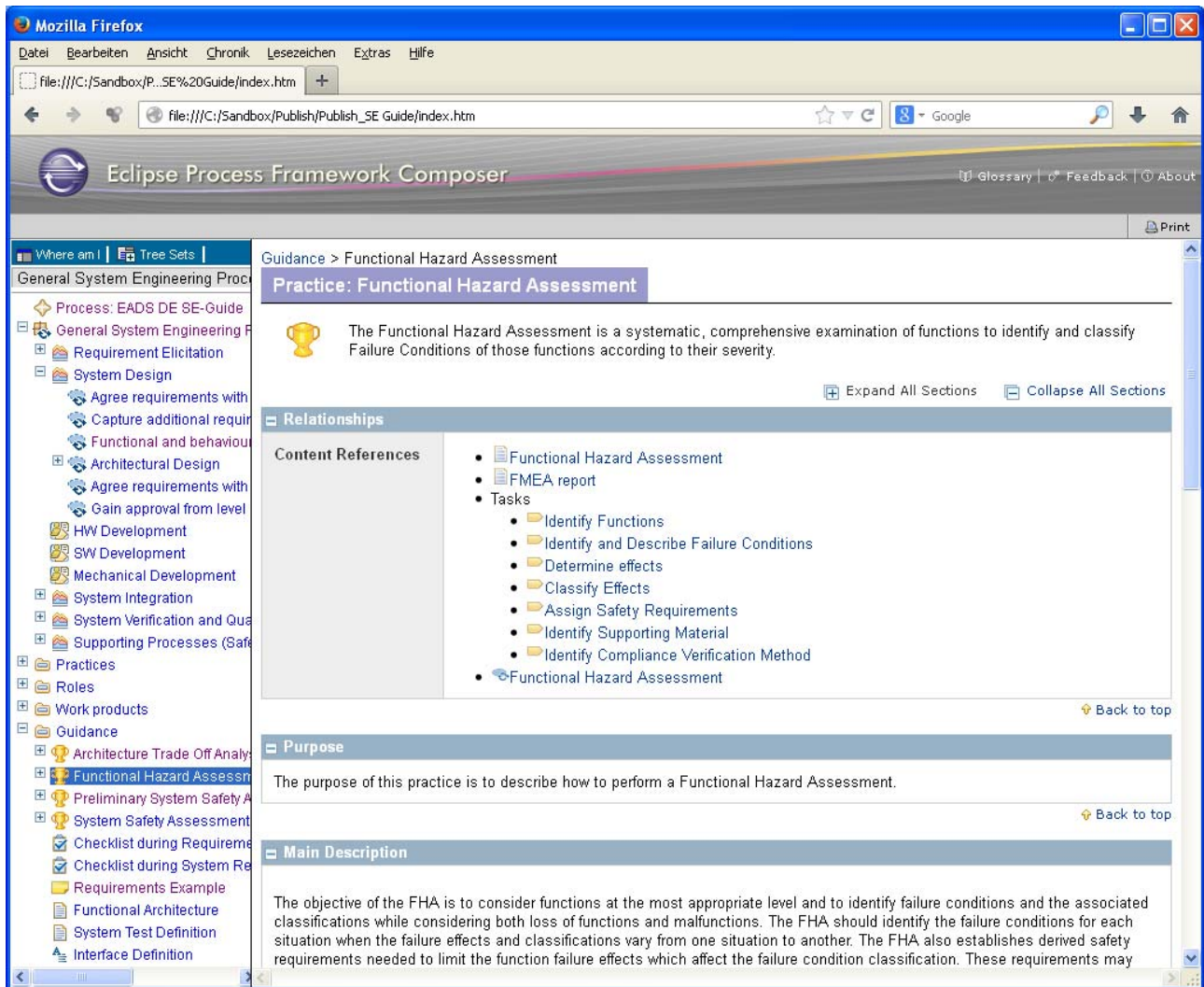


Figure 4-40 View published practice including additional guidance

5 Conclusions and Way Ahead

5.1 Preliminary Evaluation and Planned Future Work

In the following sections a brief evaluation of the SEE prototype and the work planned for the next SEE demonstrator is presented for each user story defined in [CRYSTAL D203.011].

5.1.1 US202 – Safety Analysis

The user story “Safety Analysis” is not supported in the first SEE prototype. For the next version of the SEE demonstrator a scenario will be setup that supports functional safety analysis. The safety analysis will be based on the Rhapsody system model, which will be annotated accordingly. Two safety tools, namely FaultTree+ and QuantUM, will be integrated with Rhapsody. They provide the following functions:

- FaultTree+ (Isograph): define fault trees and identify common cause failures, perform quantitative analysis
- QuantUM (University of Konstanz): perform probabilistic model checking in order to prove safety properties

For both tools, artefacts need to be exchanged between Rhapsody and the safety tool. The CRYSTAL IOS is expected to be a major enabler.

5.1.2 US203 – Variability Management

The following usage scenarios contribute to this user story:

- SC1 – Define Product Family Scope and Variability Model
- SC2 – Develop Domain System Requirements
- SC4 – Create Product System Requirements

Feature-based variability management has been established for system requirements in this SEE prototype. However, the approach is quite limited: only individual requirement objects can be reused. One possible extension is to introduce parameterized requirements, which provide variation points for elements of requirement statements.

Feature models and the related configuration well support external variability (stakeholder view). But it is very difficult to link features with variation points in system models. For this reason OVM has been evaluated as an alternative approach. It seems to be much better suited to support internal variability (development view). However, it is required to define links between variation points and development artefacts inside the OVM modelling tool. Again, the CRYSTAL IOS is seen as a major enabler to allow linking variation points with development artefacts across tool boundaries.

In the next version of the SEE demonstrator we plan to add the reference architecture and system design to the lifecycle data created for the Landing Symbolology function. We will investigate the potential of the Common Variability Language (CVL) and the Domain-specific Language (DSL) brick proposed by Task 6.10.8 to improve variability management for system models.

Moreover, pure::variants will be integrated in the tool chain of the SEE demonstrator.

5.1.3 US204 – Ontology-based Requirements Engineering

The following usage scenario contributes to this user story:

Version	Nature	Date	Page
V1.00	R	2014-02-10	63 of 74

- SC3 – Analyze and Improve Requirements Quality

Up to now, requirements are analysed manually using checklists and peer reviews. Some proprietary add-on tools have been developed which allow calculating basic quality metrics. With RQA it is now possible to automate requirements quality assessment to a large extent by providing a comprehensive set of quality metrics and findings, which guide the improvement of requirements. However, it is difficult to customize the quality assessment to project or team needs. A lot parameters can be customized, e.g.

- Which metric shall be used?
- What is the weight (priority) of each metric?
- What are acceptable ranges for each metric (quality function)?

Hence, a method is needed that supports the quality metric configuration in a more structured way.

In the next version of the demonstrator, ontologies will be created for the Landing Symbology function. This includes the used domain terms (controlled vocabulary), the relations between terms (thesaurus) as well as boilerplates (templates). Based on the ontology and boilerplate approach the potential of additional quality metrics will be evaluated.

Moreover, it is expected that initial IOS adapters will be available for RQA, RAT and kM.

5.1.4 US205 – Process Automation, Guidance and Monitoring

The following usage scenarios contribute to this user story:

- SC7 – Provide Process Guidance

In this SEE prototype an EPF model has been created that describes the engineering activities to be performed in systems engineering. This process model has been published to HTML documentation, which provides guidance to the users. Up to now, no specific method content has been integrated that describes in detail the new methods and tools being developed in CRYSTAL. This needs to be considered when mature CRYSTAL results are available.

Additionally, for the next version of the SEE demonstrator the following topics, which are not covered yet, will be investigated:

- User and access management
- Configuration management
- Process automation
- Change control

The planned functionality mainly relies on the integrated services of the envisaged SEE (e.g. VVC, RTC, SSO, RELM) as depicted in the bottom part of **Figure 5-1**. Therefore, it will be a major activity to explore the IBM Rational solution for Systems and Software Engineering (SSE) in the next SEE demonstrator.

5.1.5 US206 – Project compliance monitoring based on advanced traceability

The following usage scenarios contribute to this user story:

- SC5 – Perform System Functional Analysis
- SC6 – Perform Report Generation

In this SEE prototype traceability of artefacts is only available within an engineering tool. Linking artefacts across tool chains is not yet possible. For example, Rhapsody and DOORS are able to import a copy of the artefacts to be linked. The imported artefacts can then be linked and are traceable within the respective tool.

Version	Nature	Date	Page
V1.00	R	2014-02-10	64 of 74

In the usage scenario “Perform System Functional Analysis” system requirements have been imported from DOORS into Rhapsody with the Rhapsody add-on Gateway. Within the model, the imported requirements have been linked and are traceable. Using the Rhapsody Gateway coverage analysis (do the model elements cover all input requirements?) or change impact analysis (what model elements are affected by a requirements change?) is supported. In this SEE prototype only traceability between requirements and model elements is achieved. This needs to be extended to other artefacts, e.g. requirements quality reports, fault trees, variation points, etc. In addition, IOS adapters are a pre-requisite in order to enable cross-tool traceability.

A first version of a traceability meta model has been defined in [Binder 2014]. This will be implemented in the envisaged SEE, see **Figure 5-1**. IOS adapters will be provided by DOORS Next Generation and Design Manager for Rhapsody. Adapters for other tools such as FaultTree+, pure::variants and RQA will be added as soon as they are available to enhance the cross-tool traceability capabilities.

In the usage scenario “Perform Report Generation” a RPE template has been defined that allows generating the contents of a Rhapsody model into a Microsoft Word document. The mapping of the RPE template with the employed Rhapsody MBSE profile is described in [Binder 2013].

In the next version of the SEE demonstrator the report generation needs to be extended to DOORS and potentially other tools such that reports can be composed by artefacts created in different tools.

5.2 Envisaged SEE

Figure 5-1 illustrates the Systems Engineering Environment (SEE) for the MSE use case as it is currently envisaged. Please note that the list of tools and types of tools is not yet complete and may be updated in the future.

Taking into account the perimeter of the MSE use case, the envisaged SEE will have to include tools and databases at least for Requirements Management, Functional Models, Safety Models, Product Life Cycle Management, and Application Lifecycle Management. These tools and databases can be deployed at different company sites.

In order to realise interoperability, each tool and database has to provide a connector that is based on open standards. The connector approach as well as the open standard for interoperability will be defined in WP6.1. The communication between the tools (e.g. sending of requests to other tools, receiving data from tools) can be realised by any kind of network that is using web protocols.

Version	Nature	Date	Page
V1.00	R	2014-02-10	65 of 74

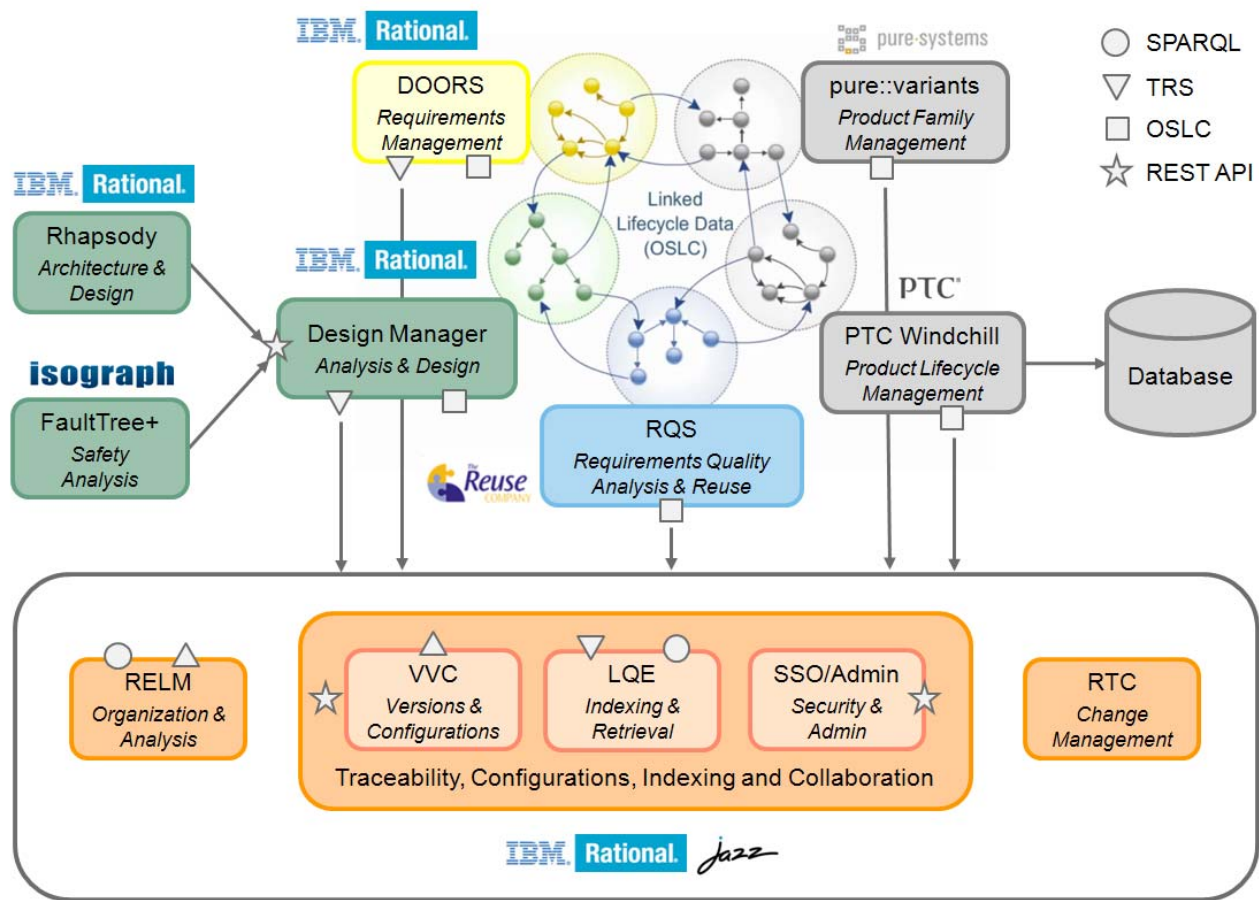


Figure 5-1 Envisaged SEE

6 Terms, Abbreviations and Definitions

6.1 Abbreviations

API	Application Programming Interface
CCC	Correctness, Consistency, Completeness
CI	Configuration Item
CM	Configuration Management
CO	[Dissemination Level]
CVL	Common Variability Language
EM	Engineering Method
EMF	Eclipse Modeling Framework
FHA	Functional Hazard Analysis
FT	Fault Tree
FTA	Fault Tree Analysis
GMF	Graphical Modeling Framework
IOS	Interoperability Specification
IW	Innovation Works
kM	knowledgeMANAGER
KPI	Key Performance Indicator
LQE	Lifecycle Query Engine
MAM	Method Authoring Method
MBD	Model-based Development
MBSE	Model-based Systems Engineering
MSE	Mission Support Equipment
N/A	Not applicable
NL	Natural Language

NLP	Natural Language Processing
OSLC	Open Services for Lifecycle Collaboration
OVM	Orthogonal Variability Model
PA	Process Activity
PLM	Product Lifecycle Management
RAT	Requirements Authoring Tool
RELM	Rational Engineering Lifecycle Manager
REST	Representational State Transfer
RM	Requirements Management
RPE	Rational Publishing Engine
RQA	Requirements Quality Analyzer
RQS	Requirements Quality Suite
RTC	Rational Team Concert
SC	State Chart
SD	Sequence Diagram
SE	Systems Engineering
SEE	Systems Engineering Environment
SPARQL	SPARQL Protocol and RDF Query Language
SPEM	Software & Systems Process Engineering Meta-Model
SSO	Single Sign-On
SW	Software
SysML	Systems Modelling Language
TRS	Tracked Resource Set
UCD	Use Case Diagram
UMA	Unified Method Architecture
UMF	Unified Method Framework

US	User Story
VP	Variation Point
VVC	Versions, Variants, and Configurations
WP	Work Package

Table 6-1 Abbreviations

6.2 Glossary

Artefact	An artefact is any type of object within the engineering development environment that can be referenced to or is a configuration item of its own. Examples are requirements, models, model elements and files.
Baseline	A Baseline is an approved and released set of artefacts having an association with the system or a dedicated configuration item. A baseline is managed by the configuration management and represents a reliable and consistent basis for subsequent design and development activities to which changes are addressed.
Boilerplates	Requirements boilerplates can be thought of as semi-complete requirements which are parameterized to suit a particular context. The parameters in a boilerplate generally refer to different attributes with respect to a given system (e.g. the system itself, stakeholders involved, and functions of the system, the objects and events involved in the system, performance characteristics, or units of measurement).
Configuration	<p>The configuration is the configuration status of a single item, which status may change independently from other items.</p> <p>On the lowest levels we have:</p> <ul style="list-style-type: none"> • source code files having a dedicated version • HW modules having a modification state • documents having an issue <p>Higher integrated configuration items and finally the system itself are an arrangement of configuration items on a lower level, each having its own configuration state. The configuration of such an integrated item consists of a listing of all these lower level items and their configuration state. This configuration identifies also the selected options, i.e. the variant.</p>
Configuration Item	A Configuration Item (CI) is any work product within the SEE designated for separate configuration management. A Configuration Item could be the complete

	system model or sub parts of it (packages), a requirements module, an analysis model, a simulation or mathematical model or a document. Single requirements and model elements are not treated as a configuration item as the granularity would be too fine and the consistency of a set of such elements is too complicated to ensure.
Configuration Management	Configuration Management (CM) is a process for establishing and maintaining consistency of a product's performance, functional and physical attributes with its requirements, design and operational information throughout its life. CM comprises following disciplines: <ul style="list-style-type: none"> • Configuration Identification • Configuration Control • Configuration Status Accounting • Configuration Verification and Audit
Configuration Status	Configuration Status is a report on the configuration baselines associated with each configuration item and all departures from the baseline, limitations and problems during design and production.
Controlled Vocabulary	Controlled vocabularies to organize knowledge for subsequent retrieval. They are used in indexing schemes, thesauri, taxonomies and other forms of knowledge organization schemes (source: Wikipedia). In CRYSTAL we use controlled vocabularies to build the ontology supporting requirements formalization.
Feature	Features are end-user visible characteristics of a system. Typically, a feature is coarser than a requirement. Features are a convenient way to characterize a function in terms that are understandable to various stakeholders.
Ingoing Link	An ingoing link is, from the view of a link target, the link from a link source.
Language Defects	Language defects are syntactical issues in the formulation of natural language requirements. Examples are <ul style="list-style-type: none"> • Omitting imperative shall (modal) • Using passive voice instead of active voice
Link	A link is defined as the relation between a link source artefact (the origination of the link) and the link target artefact (the destination of the link)
Link type	The type of a link defines the characteristic of the relation. Links are grouped regarding their syntax and semantic, for example «refine», «verify» and «satisfy» are common link types. The SEE is capable to discriminate links according to their types, i.e. to filter links for one or more dedicated types.
Model Element	A model element is any object in a model that can be referenced. For example in a SysML model we have blocks, operations, transitions, states, events, data items, diagrams, views, packages.
Option	An Option defines exactly one possible resolution of a variation point.

Product Family	A product family (also called product line) is a set of systems and products sharing a common, managed set of functions with several features, that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of building blocks in a prescribed way.
Reference Technology Platform	A cross-domain standardised platform that provides meta-models, methods, and tools for safety-relevant hard real-time system development.
Requirement	A formalised statement identifying a capability, functionality, a physical characteristic or a quality that must be met or possessed by a system or system component to satisfy a contract, standard, a specification or other formally imposed documents. A requirement may be developed at any point in the product lifecycle by any number of stakeholders.
Requirement quality	Requirements shall fulfil quality characteristics such as CCC (complete, correct, and consistent) and SMART (specific, measurable, achievable, relevant, and traceable).
Requirements group	A set of requirements that serve the same role in the engineering process. Good examples are the requirements allocated to a particular System Element, the system requirements, or the assurance requirements featuring means of compliance and test objectives.
Stakeholder	An individual or organization having a right, share, claim, or interest in a system or its characteristics that meet their needs and expectations. NOTE: Stakeholders include, but are not limited to end users, end user organizations, supporters, developers, producers, trainers, maintainers, disposers, acquirers, customers, operators, supplier organizations, creditors, and regulatory bodies.
Stakeholder Needs	Expectations stakeholders have about the characteristics of the system that may not necessarily be clear, consistent or even achievable.
Suspect Link	Linked objects are marked as having suspect links if the object that they link to has changed.
System Engineering Environment	The System Engineering Environment (SEE) is the tool environment for the system engineer including tools for different purposes and concepts. The tools are collaborating to automate tasks within the environment and to establish common tool independent methods. A focus is on Model Based System Engineering with a system design model as the common link between all the different tasks and models.
Systems Engineering	An interdisciplinary approach and means to enable the realization of successful systems. This approach starts with the definition of stakeholder needs, the identification of product functionality and the intended validation very early in the lifecycle. Systems engineering considers both the business and the technical needs of all stakeholders with the goal of providing a quality product that meets

	the user needs.
Traceability	The ability to identify the relationship between various artefacts of the development process, i.e. the lineage of requirements, the relationship between a design decision and the affected requirements and design features, the assignments of requirements to design features, the relationship of test results to the original source of requirement. Bi-directional traceability is required to permit top-down impact analysis and down-top traceability analysis.
Transformation Protocol	<p>A transformation protocol documents the variability resolution during the creation of a product variant:</p> <ul style="list-style-type: none"> • Common and selected features • System requirements variation points and selected options • Identified system requirements • System model variation points and selected options • Identified system model elements <p>For each creation of a product variant a transformation protocol shall be created.</p>
User Story	A User Story describes a typical action pattern or work flow within an industrial domain. The user stories are used to describe general processes that are too high level to derive development requirements directly out of it.
Validation	<p>Confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled.</p> <p>NOTE Validation in a system life cycle context is the set of activities ensuring and gaining confidence that a system is able to accomplish its intended use, goals, and objectives. The right system has been built.</p>
Variant	A variant selects options of a variation point (e.g. a red car, a green car) or a product with several options selected for its variation points making it different to other products based on the same specifications.
Variation Point	<p>A variation point is a representation of a subject or attribute that can vary (e.g. colour of a car). Options are linked to the variation point and show the range of the variability (e.g. colour red, green blue; for some reasons other colours are not available). A Variant identifies a single option of a variation point (e.g. a red car, a green car).</p> <p>The specification of Variation Point definitions shall include:</p> <ul style="list-style-type: none"> • Description WHAT shall vary • Identification of the possible options in order to define range of variation • Definition of the binding time when the options are implemented into the product (compile/link time, integration time, installation time, operation time) • Rationale why this variation is required or how it will pay off • Stakeholder requesting for the variation • Visibility of the variation point (internal or external) shall be defined.
Verification	Confirmation, through the provision of objective evidence, that specified requirements have been fulfilled.



	NOTE: Verification in a system life cycle context is a set of activities that compares a product of the system life cycle against the required characteristics for that product. This may include, but is not limited to, specified requirements, design description and the system itself. The system has been built right.
--	--

Table 6-2 Terms

7 References

If no other remarks are given the last valid issue of the mentioned document applies.

[Binder 2013]	Binder, I.: Automatisierte Dokumentengenerierung in der modellbasierten Systementwicklung bei Cassidian, Student Thesis, Technische Akademie Konstanz, 2013
[Binder 2014]	Binder, I.: Advanced Traceability in a Model-based Systems Engineering Environment, Project Thesis, Technische Akademie Konstanz, 2014
[CESAR CCC]	Allain, G. <i>et al.</i> : Completeness/Consistency/Correctness, CESAR deliverable D_SP2_R3.3_M3_Vol4, 2011
[CESAR RSL]	Mitschke, A. <i>et al.</i> : Definition and exemplification of RSL and RMM, CESAR deliverable D_SP2_R2.1_M3, 2011
[CRYSTAL DOW]	Critical System Engineering Acceleration (CRYSTAL) Joint Undertaking (JU) under grant agreement № 332830, Annex I - "Description of Work", 2013
[CRYSTAL D203.011]	Bogusch, R. <i>et al.</i> : MSE Report – V1, CRYSTAL deliverable D203.011, 2014
[CRYSTAL D607.021]	Fuentes, J.: Requirements Quality Analyzer, CRYSTAL deliverable D607.021, 2013
[CRYSTAL D607.031]	Fuentes, J.: Requirements Authoring Tool, CRYSTAL deliverable D607.031, 2013
[CRYSTAL D607.041]	Fuentes, J.: knowledgeMANAGER, CRYSTAL deliverable D607.041, 2013
[Hanser 2013]	Hanser, M.: Integrate QuantUM into a Lifecycle Collaboration Environment, Master Project, University of Konstanz, 2013
[INCOSE RWG]	Guide for Writing Requirements, Requirements Working Group, International Council on Systems Engineering, 2009
[ISO/IEC 29148:2011]	Systems and Software Engineering - Life Cycle Processes - Requirements Engineering
[OVM]	Lauenroth, K.; Pohl, K.: Principles of Variability. In: Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, 2005
[Stocker 2011]	Stocker, Th.: An Approach to Support Product Families in Requirements Management, Bachelor Thesis, DHBW Ravensburg, 2011

Table 7-1 References