Behavioural Requirements Language Definition

Defining the ReDSeeDS Languages

Deliverable D2.1, version 1.00, 30.01.2007

IST-2006-033596
ReDSeeDS
Requirements Driven
Software Development System
www.redseeds.eu

Infovide-Matrix S.A., Poland
Warsaw University of Technology, Poland
Hamburger Informatik Technologie Center e.V., Germany
University of Koblenz-Landau, Germany
University of Latvia, Latvia
Vienna University of Technology, Austria
Fraunhofer IESE, Germany
Algoritmu sistemos, UAB, Lithuania
Cybersoft IT Ltd., Turkey
PRO DV Software AG, Germany
Heriot-Watt University, United Kingdom

Behavioural Requirements Language Definition

Defining the ReDSeeDS Languages

Workpackage	WP2
Task	T2.1
Document number	D2.1
Document type	Deliverable
Title	Behavioural Requirements Language Definition
Subtitle	Defining the ReDSeeDS Languages
Author(s)	Hermann Kaindl, Michal Smialek, Davor Svetinovic, Albert Ambroziewicz, Jacek Bojarski, Mohamad Hani el Jamal, Wiktor Nowakowski, Tomasz Straszak, John Paul Brogan, Hannes Schwarz, Daniel Bildhauer, Jürgen Falb
Internal Reviewer(s)	Daniel Bildhauer, Rober Draber, Hermann Kaindl, Sevan Kavaldjian, Thorsten Krebs, Roman Popp, Michal Smialek, Radoslaw Ziembinski
Internal Acceptance	Project Board
Location	https://svn.redseeds.eu/svn/redseeds/1_DeliverablesSpace/WP2_Requirements_specification_language/D2.1.00/ReDSeeDS_D2.1_Behavioural_Requirements_Language_Definition.pdf
Version	1.00
Status	Final
Distribution	Public

30.01.2007

History of changes

Date	Ver.	Author(s)	Change description
27.12.2006	0.01	Hermann Kaindl (TUW)	Proposition of ToC
30.12.2006	0.02	Michal Smialek (WUT)	Modified ToC and example of contents
30.12.2006	0.03	Hermann Kaindl (TUW)	Modified ToC and task assignment
09.01.2007	0.04	Albert Ambroziewicz, Jacek Bojarski (WUT)	Added content for UseCaseRelationship package description
09.01.2007	0.05	Albert Ambroziewicz, Jacek Bojarski (WUT)	Added content for BasicRepresentations package description
09.01.2007	0.06	Tomasz Straszak (WUT)	Added content for RequirementsRepresentations package description
09.01.2007	0.07	Wiktor Nowakowski (WUT)	Added content for RequirementRelationships package description
09.01.2007	0.08	Tomasz Straszak (WUT)	Added examples for concrete syntax for NaturalLanguageRepresentations package description
09.01.2007	0.09	Albert Ambroziewicz, Jacek Bojarski (WUT)	Added content for SVOSentences package description
10.01.2007	0.10	Daniel Bildhauer (UKo)	Added content for Natural language package description
10.01.2007	0.11	Daniel Bildhauer (UKo)	Added content for RepresentationSentence package description
10.01.2007	0.12	Wiktor Nowakowski, Tomasz Straszak (WUT)	Added content for ScenarioSentences package description
10.01.2007	0.13	John Paul Brogan (HWU)	Added content for Document Scope
11.01.2007	0.14	Hannes Schwarz (UKo)	Added content for ConstrainedLanguageRepresentation package description
11.01.2007	0.15	Hannes Schwarz (UKo)	Added content for Requirement Representation Sentences overview

Date	Ver.	Author(s)	Change description
11.01.2007	0.16	Daniel Bildhauer (UKo)	Added content for InteractionScenario package description and some overviews
11.01.2007	0.17	John Paul Brogan (HWU)	Added content for Related work and relations to other documents
11.01.2007	0.18	Albert Ambroziewicz, Jacek Bojarski (WUT)	Added content for "Approach to language definition and notation conventions" section
11.01.2007	0.19	Tomasz Straszak (WUT)	Added content for "Usage guidelines" section
11.01.2007	0.20	John Paul Brogan (HWU)	Added content for Requirements representations Overview
12.01.2007	0.21	Albert Ambroziewicz, Jacek Bojarski (WUT)	Added content for Activity representations package description
12.01.2007	0.22	Albert Ambroziewicz, Jacek Bojarski (WUT)	Added content for Use case representations package description
12.01.2007	0.23	Wiktor Nowakowski, Tomasz Straszak (WUT)	Updated content for ScenarioSentences package description
12.01.2007	0.23	Wiktor Nowakowski (WUT)	Updated content for RequirementRelationships package description
13.01.2007	0.24	Michał Śmiałek (WUT)	Added complete overview for Chapter 6 - Requirements, edited 6.2 Requirements specifications, corrected and updated certain Figures and descriptions in Chapter 6, added overview Figures for Chapters 7 and 8
15.01.2007	0.25	John Paul Brogan (HWU)	Added content update for chapter 7.2.1 Requirements representations Basic representations Overview
16.01.2007	0.26	John Paul Brogan (HWU)	Added content update for chapter 7.3.1 Requirements representations Overview
16.01.2007	0.27	Hermann Kaindl (TUW)	Added executive summary
17.01.2007	0.28	Davor Svetinovic (TUW)	Added chapters 2, 3, 4, and 5
18.01.2007	0.29	Daniel Bildhauer (UKo)	Added example for communication diagram in interaction scenario
18.01.2007	0.30	Hannes Schwarz (UKo)	Updated and added content to Requirement Representation Sentences overview
18.01.2007	0.31	Mohamad Eljamal (TUW)	Added example

Date	Ver.	Author(s)	Change description
18.01.2007	0.32	Hermann Kaindl (TUW)	Added conclusion
19.01.2007	0.33	Daniel Bildhauer (UKo)	Split metamodel for interaction diagram, minor changes
20.01.2007	0.34	Michał Śmiałek (WUT)	Changes to Chapter 1 made, small corrections and finalisations elsewhere
21.01.2007	0.35	Hermann Kaindl (TUW)	Added references
23.01.2007	0.36	John Paul Brogan (HWU)	Updated/Corrected English Spelling and Grammar for Chapters 1-4
24.01.2007	0.37	Daniel Bildhauer (UKo)	Updated layout of some figures
24.01.2007	0.38	John Paul Brogan (HWU)	Updated/Corrected English Spelling and Grammar for Chapters 5-9
26.01.2007	0.39	Wiktor Nowakowski (WUT)	Small corrections in Chapters 6-8
26.01.2007	0.40	Albert Ambroziewicz (WUT)	Small corrections in Chapters 6&7
26.01.2007	0.41	Albert Ambroziewicz, Tomasz Straszak (WUT)	Added descriptions for Sec.7.3 and updated appropriate diagrams
28.01.2007	0.42	Michał Śmiałek (WUT)	Small corrections made
29.01.2007	0.43	Hermann Kaindl (TUW)	Clean-up
30.01.2007	1.00	Hermann Kaindl (TUW)	Finalisation

Summary

Function and behaviour are mostly confused with one another or mixed up in requirements engineering. In addition, when specifying or handling requirements, it is usually not understood that actually *representations* of requirements are dealt with rather than requirements. In particular, indexing for reuse has to be done with concrete representations. While requirements are traditionally simply described in practice, mostly in natural language or some subset of it, requirements *modelling* as an approach to having accurate requirements capture, is being more and more introduced and utilised within software engineering methodologies. However, the conceptual differences between requirements capture approaches are not understood well.

The behavioural part of our requirements specification language distinguishes between Functional and Behavioural Requirements. While the former specify the required effects of some system, the latter specify required behaviour across the system border, in the form of Envisionary Scenarios. Functional Requirements are further specialised into Functional Requirements on Composite System and Functional Requirements on System to be built. The former are fulfilled by an Envisionary Scenario, while the functions of the latter will make its execution possible. Related Envisionary Scenarios together make up a Use Case.

We distinguish strictly between requirements and *representations* of requirements. Strictly speaking, only the latter can actually be reused. Requirements representations can be *descriptive* or *model-based*, and our RRSL language makes this distinction explicit. The former describe the needs of certain requirements, while the latter represent models of the system to be built. A requirement is then to build a system like the one modelled.

This deliverable contains the behavioural part of the requirements specification language, i.e., all the parts of the ReDSeeDS meta-model and other descriptions dealing with what happens over time across the system border. The language is capable of describing the dynamics of dialogue between the users (human or machine) of the system and that system precisely yet comprehensibly. This deliverable first gives a conceptual overview of this behavioural requirements specification language. In a second part, it provides a comprehensive language reference including concrete syntax.

Table of contents

List of figures

Chapter 1

Scope, conventions and guidelines

1.1 Document scope

This document provides a conceptual overview, and defines syntax and semantics for the ReDSeeDS Behavioural Requirements Language (BRL). This definition is required to aid the construction of accurate requirements specifications in the form of descriptive or model-based representations.

The conceptual overview of the BRL explains the approach taken to allow for describing functional and behavioural requirements as such and how functional and behavioural requirements can be represented in the language. This document then presents the BRL Reference which covers definitions for Requirements, Requirements Representations and Requirement Representation Sentences. This reference explains the syntax of the language in its abstract form (using a meta-model) and in its concrete form (using concrete examples of language usage). The semantics of all the language constructs is also defined.

The definitions for Requirements describe language constructs that allow for depicting individual requirements as such. This explains how to structure requirements into full requirements specifications. It also defines possible relationships between requirements. The reference for Requirements Representations defines all the representations of requirements possible to be expressed in BRL. These include textual, descriptive representations in natural or constrained language and schematic, model-based representations mostly derived from UML. Finally, the Representation Sentences define the smallest "building blocks" of BRL, ie. sentences. These sentences allow and usually necessitate for extensive use of hyperlinks to the domain vocabulary (as described in the Structural Requirements Language Definition, document D2.2). Apart from natural language sentences, several types of controlled, structured language sentences are defined. These are mostly based on the Subject-Verb-Objects SVO(O) grammar.

1.2 Approach to language definition and notation conventions

1.2.1 Meta-modelling

The Behavioural Requirements Language (BRL) is defined using a meta-model. The meta-model is a model of models, where a model of a system is a description or specification of that system and its environment for some known task. A model is often presented as a combination of drawings and text. The text may be in a modelling language or in a natural language (adapted from [MM03]).

The meta-model can be treated as a definition of a language in which models can be expressed properly. Meta-model sets well-formedness rules for models. A model has to comply with the meta-model of the language it uses. For instance, a UML model [Obj05b] has to comply with the UML meta-model.

Meta-models and models define two levels of meta-modelling. In fact we can have four levels: M0 - model instance level, M1 - model level, M2 - meta-model level, M3 - meta-meta-model level. The model instance level contains all the objects (real time instances or real world objects) of classifiers (classes) included in the model level. The meta-meta-model level defines the language to represent a meta-model (a meta-modelling language). In defining the whole Requirements Specification Language (RSL) we use MOF[1] [Obj03a] as a meta-modelling language. From the perspective of MOF, UML and RSL both can be viewed as user models based on MOF as language specification. From the perspective of RSL, requirements specification is a model of requirements expressed by the RSL.

The most common role of a meta-model is to define the semantics for how language tokens from a language specification can be used. As an example, consider figure 1.1, where the meta-classes Classifier, Generalisation and Class are defined as part of the UML meta-model. These are instantiated in a user model in such a way that the classes Car, Truck and Vehicle are all instances of the meta-class Class. The generalisation between Car and Vehicle (or Truck and Vehicle) classes is an instance of the Generalisation meta-class (based on [Obj05a]).

[1]MOF is similar to UML but it is reduced to simplified class diagrams with embedded OCL [Obj03b] constraint expressions (expressions in curly brackets "{}"). These special class diagrams have their semantics defined for language construction.

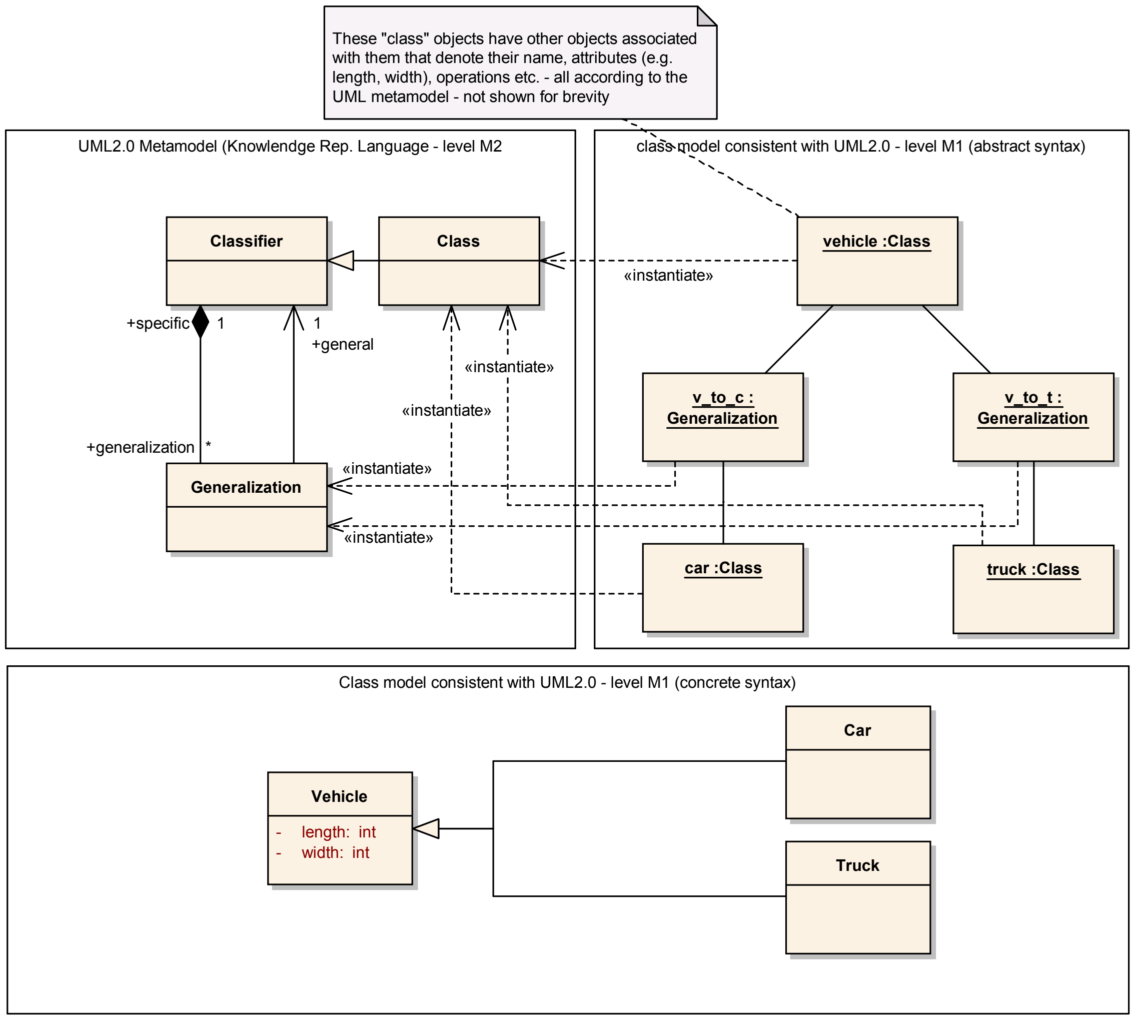

These "class" objects have other objects associated with them that denote their name, attributes (e.g. length, width), operations etc. - all according to the UML metamodel - not shown for brevity
UML2.0 Metamodel (Knowlendge Rep. Language - level M2
class model consistent with UML2.0 - level M1 (abstract syntax)
Classifier
Class
vehicle :Class
«instantiate»
+specific
1
1
+general
+generalization
*
Generalization
v_to_c : Generalization
v_to_t : Generalization
car :Class
truck :Class
Class model consistent with UML2.0 - level M1 (concrete syntax)
Vehicle
- length: int
- width: int
Car
Truck

Figure 1.1: UML meta-modelling example

1.2.2 Defining languages using meta-modelling

In languages defined with MOF we define tokens of a language, their relationships and meaning. Every token has to be described in terms of its syntax (abstract, concrete) and semantics.

The abstract syntax defines the tokens of the language and their relationships and integrity constraints available in the language. Relationships and constraints determine a set of correct sentences that can be created in the language (its grammar). Note that abstract syntax should be independent from graphical or textual representation of the language elements it is defining. The abstract syntax can be perceived as the static semantics of the language. In the RSL, abstract syntax is expressed by MOF diagrams and natural language descriptions.

The concrete syntax is a description of specific notation used in representing a language's elements. In other words it is a mapping from notation to the abstract syntax. If an element of the meta-model is marked as abstract it does not have any concrete syntax (because it cannot be instantiated). In the RSL, its concrete syntax is expressed by natural language descriptions and illustrated with examples of the language's usage. The figure 1.2 is an example of a definition of the abstract and concrete syntax for the RSL.

The semantics of meta-model elements expresses the meaning of properly formulated constructs of a language (according to its abstract syntax). In the RSL, its semantics are represented by natural language descriptions.

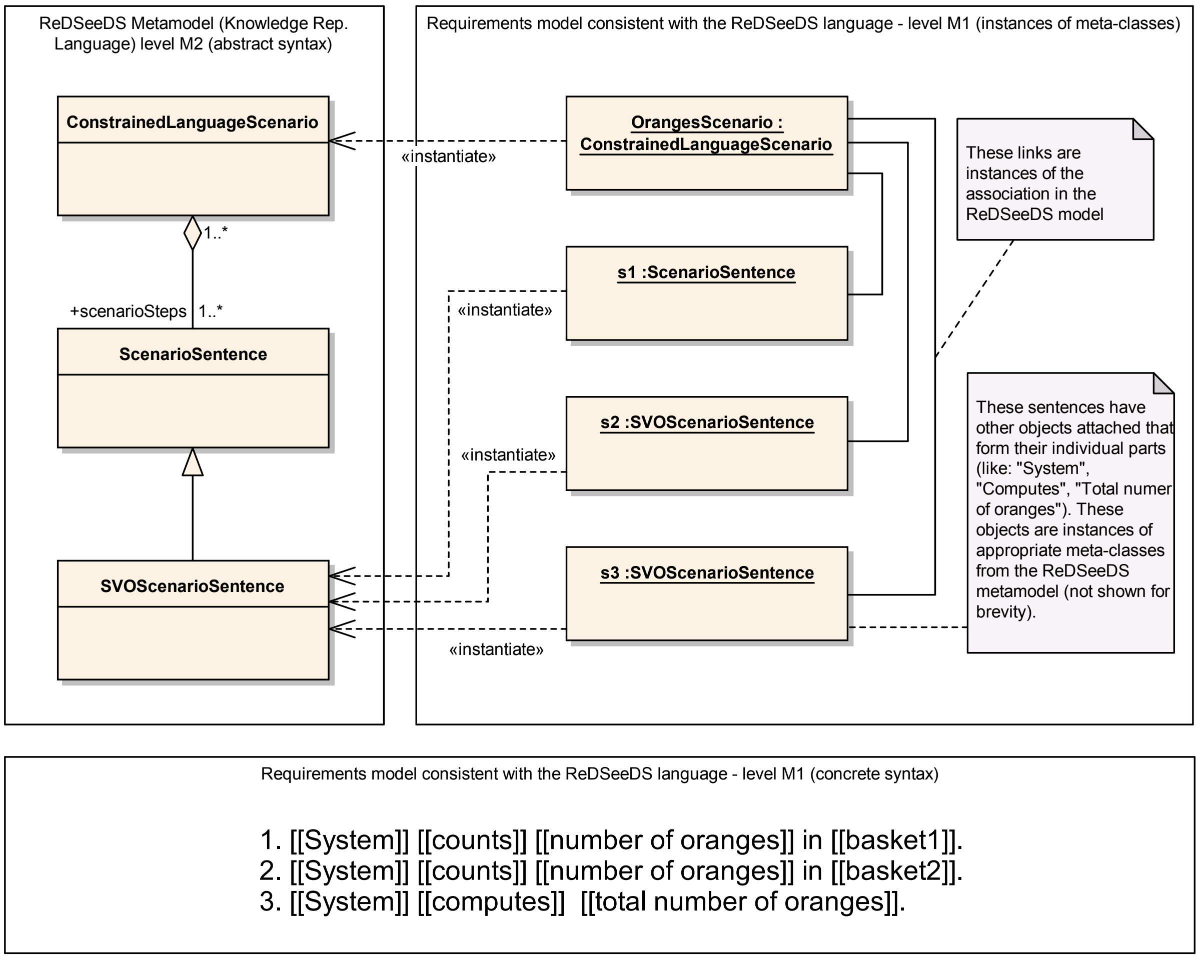

ReDSeeDS Metamodel (Knowledge Rep. Language) level M2 (abstract syntax)
ConstrainedLanguageScenario
1..*
+scenarioSteps
1..*
ScenarioSentence
SVOScenarioSentence
Requirements model consistent with the ReDSeeDS language - level M1 (instances of meta-classes)
OrangesScenario : ConstrainedLanguageScenario
«instantiate»
s1 :ScenarioSentence
«instantiate»
s2 :SVOScenarioSentence
«instantiate»
s3 :SVOScenarioSentence
«instantiate»
These links are instances of the association in the ReDSeeDS model
These sentences have other objects attached that form their individual parts (like: "System", "Computes", "Total numer of oranges"). These objects are instances of appropriate meta-classes from the ReDSeeDS metamodel (not shown for brevity).
Requirements model consistent with the ReDSeeDS language - level M1 (concrete syntax)
1. [[System]] [[counts]] [[number of oranges]] in [[basket1]].
2. [[System]] [[counts]] [[number of oranges]] in [[basket2]].
3. [[System]] [[computes]] [[total number of oranges]].

Figure 1.2: ReDSeeDS meta-modelling example

The RSL is not an extension to UML, though we use certain UML packages (for those parts of the RSL that derive from UML). Those packages were merged into the RSL definition (using «merge» or «include») from "UML Superstructure" [Obj05b] packages. This merger is done on the package level. Inside a package that is part of the RSL's definition, meta-classes from the merged UML package are directly specialised.

1.2.3 Structure of the language reference

Part II of this document contains the BRL definition. It has been divided into sections according to the logical structure of packages and subpackages of the BRL.

The BRL is divided into three main packages:

- Requirements (requirements as such with their relationships)
- Requirements Representations (definitions of individual requirements in various notations)
- Requirement Representation Sentences (basic "building blocks", ie. sentences that form the representations as above)

Each of theses packages is described briefly with an overview section (including a package diagram), which is followed by description of its subpackages. Every subpackage is presented in an overview explaining general ideas behind a package, a meta-model diagram for this package and two sections which describe the abstract syntax with semantics of language constructs and the concrete syntax.

1.2.4 Notation conventions

Lowest level package descriptions use the following notation conventions:

- sans-serif font is used for names of classes, attributes and associations, e.g. Requirement
- if a class name is used in description of package other than the one it is included in, it is preceded with package name and a double colon ("::"), e.g. RequirementsSpecifications::Requirement
- ***bold/italics font*** is used for emphasized text, e.g. ***Abstract syntax***

Class colours used on the diagrams indicate membership of the packages. Introduction of colours is intended to enhance readability of diagrams which contain classes from different packages (e.g. blue colour denotes that classes are from Requirement packages, yellow are from RequirementRepresentation package and green are from DomainElement package).

1.3 Structure of this Document

Part I gives a conceptual overview of the Behavioural Requirements Language. It contains the chapters 2 to 5. Chapter 2 introduces the requirements language and the representation language and chapter 3 deals with functional and behavioural requirements and presents their conceptual model. The next chapter 4 then outlines possible representations of the requirements without going too deeply into detail. In the last chapter 5 in part I, the newly introduced concepts are discussed.

Part II defines the metamodel of the RSL's behavioural part, again dealing with the subject of requirements itself and different possibilities for requirement representation. It is divided into four chapters, each of them dealing with a part of the metamodel. Every chapter has a short overview, defines abstract syntax and semantics and then gives a short example of the concrete syntax, using the Fitness Club case study as a running example. Chapter 6 defines the part of the metamodel containing the requirements themselves and their arrangement. Chapter 7 explains the part of the metamodel that deals with different kinds of representations, especially textual representations and schematic representations, which can be displayed as UML-like models. Chapter 8 then defines the grammar for the semi-formal textual representation.

The final chapter 9 sums up the document and draws the conclusion of the previous parts.

1.4 Usage guidelines

The ReDSeeDS Behavioural Requirements Language (BRL) definition should be used as a book that guides the reader through the structure, syntax and semantics definitions of the BRL, as part of the complete ReDSeeDS Requirements Specification Language. It should be used mainly by creators of appropriate software CASE reusabilty tools that would allow handling of the language by the end users (analysts, etc.) to express behaviour of the system under development. It can be used by advanced end users of the language as a reference for the language's syntax and semantics. Examples of BRL elements' concrete syntax have illustrative character and should be treated only as support in understanding of an element's occurrence.

Users of the BRL Specification are expected to know the basics of metamodelling and MOF (Meta Object Facility) specification [Obj06]. Knowledge of UML ([Obj05b] and [Obj05a]) could be helpful as some elements of BRL are extensions, constraints or redefinitions of UML elements.

Part I

Conceptual Overview of the Behavioural Requirements Language

Chapter 2

Introduction

The ReDSeeDS Requirements Specification Language (meta-)model consists of 3 parts, which are linked to an extra model of the Reuse Domain:

1. Requirements Language
2. Requirements Representation Language
3. Application Domain Language

The primary reason for separation of the overall language model into several parts is the separation of concerns. In particular, the main separation is between Requirements Language and Requirements Representation Language. This is a crucial innovation, which is important since we are not reusing requirements themselves but rather requirements representations.

The separation of Requirements Language and Requirements Representation Language allows separation and simplification of the Reuse Domain. Avoiding the separation between the former two would force integration of the latter and mixing of the different concerns and lead to higher complexity of the overall language specification. In addition, requirements are not reusable directly, and this fact should be reflected in the language specification.

It is also important that the representation of the Application Domain is distinct from the Requirements Representation, while they will be linked, of course. The requirements may not be understood without the links to the Application Domain, but the content of the application domain is not requirements.

Hyperlinks between Application Domain and Requirement Representation are a crucial element of our language, and greatly facilitate keeping the specification coherent.

Chapter 3

Functional and Behavioural Requirements

Requirements engineering (RE) is the essential activity in assuring that one builds computer-based systems that will satisfy stakeholders' goals. As the need for a systematic method of requirements elicitation and specification first became obvious for very large and complex systems, most RE research focused on discovery and development of requirements techniques and artefacts that are tailored to support the development of these very large systems in those environments with relatively large amounts of resources. Developers of a small system, on the other hand, traditionally used an *ad hoc* approach to RE due to the system's small size and the developers' unsystematic approach to development. The importance of systematic RE increases dramatically, even for small systems, with the introduction of product-line approaches, customisable software, etc. So, over time, we have gone from *ad hoc* approaches to requirements management to more formal ways of capturing and managing requirements. This trend is what led to our project and the goal of building systems based on requirements reuse. That is, if we are systematically capturing and managing requirements for one project then we should be able to reuse some of these requirements on other similar projects.

The other important impact on RE techniques is due to the nature of the development of large systems. Traditionally, the typical CBS is developed in-house, where developers work with relatively stable domains, are responsible for the development of the system from scratch, and have relatively stable production teams and a large amount of resources. This way of development has led to the dominance of the requirements specification that focuses mainly on product-level requirements such as features [Lau02] and subsequently on low-level requirements and design. Designs of different systems have already been successfully reused either through reuse such as using design pattern or through larger scale reuse such as using reference architectures for the specification of the new systems. The reuse of these design elements implied the reuse of some of the background requirements that led to these designs, but there was no intentional and systematic reuse of the requirements as such. The main contribution of this project will be in pushing the reuse effort even further, i.e., beyond design reuse — all the way to systematic requirement reuse. The primary target, for facilitating reuse are product-level and low-level requirements.

Product-level and low-level requirements are very well studied and widely applied in industrial settings, but the main difficulty is in ensuring that they fulfil essential business goals. Product-level requirements form a set of features that, combined, are used to achieve the organisation's business goals. The success of the overall goal depends on every single one of the features and on the particularities of their interactions. The problem of achieving goals is exacerbated as a result of their frequent changes over time, which cause a chain reaction of changes in product-level and low-level requirements.

Lauesen has observed that product-level and low-level requirements management is straightforward and changes to them are relatively easy to deal with in practice [Lau02]. Developers can usually sense when these requirements are not correct and do not fit with each other. This ability usually does not work at the higher levels of abstractions, and it is the responsibility of the business analyst to deal with these higher level requirements. In an occasional case, it is not even possible to estimate how changes in the product-level requirements effect overall goals until the changes are implemented [Lau02]. Therefore, besides attempting to reuse product-level and low-level requirements, our project is also dealing with the reuse of higher-level requirements in order to ensure the traceability and the fulfilment of all the requirements at the later design stages for the system that is built through the requirements reuse.

Requirements are specified either directly or indirectly for many different purposes and as part of many different engineering activities. One example classification is [Lau02]:

1. *Business-level requirements specification* — Business-level requirements are most often specified indirectly as part of business reengineering activities. The most common concepts that appear at this level are business goals, processes, resources, and rules. For example, for an elevator system, a business-level requirement is: "The elevator shall transport passengers and goods from any floor to any other floor."

2. *Domain-level requirements specification* — Domain-level requirements, as mentioned previously, are most often indirectly specified in the traditional requirements specifications [DvLF93]. Newer, more systematic versions of domain-level RE have received a lot of attention recently [BPG$^+$01, CKM01, MCL$^+$01]. Most explicit domain-level requirements are captured and specified for domains, which are becoming increasingly complex and difficult to adequately support by systems [CKM02, GPS01, GMP01, MC00]. The

most common concepts that appear at this specification level are user goals, user tasks, domain input, and domain output. A more recent trend is the incorporation of agent-based analysis as part of domain modelling [MKG02, KGM02, MKC01, GPM$^+$01]. For an elevator system, an example domain-level requirement is: "The elevator shall be accessible from each floor."

3. *Product-level requirements specification* — Product-level requirement specifications are the most common type of requirement specifications. There is an extensive body of knowledge about them, and most previous research focused on improving the different techniques used to elicit, specify, and validate this type of requirement. The common artefacts and concepts that occur as parts of product-level specifications are features, use cases, functional lists, data input, data output, etc. For an elevator system, an example product-level requirement is: "The elevator shall accept elevator calls only while stationary."

4. *Design-level requirements specification* — Design-level requirements specification are the requirements that directly constrain the design of a system. Much effort has been invested into its standardization through the Unified Modelling Language (UML) [Lar04, Fow04]. UML artefacts and underlying techniques present the most common types of concepts and techniques used to capture requirements at this level. This level acts as a transition phase between product-level specification and code-level requirement specifications. For an elevator system, an example design-level requirement is: "A queue data structure shall be used to store the data for elevator calls."

5. *Code-level requirements specification* — Code-level requirements are usually specified as part of the programming activity and describe details of low-level algorithm and data structures. This type of specification is that with which most programmers are familiar, as it is inseparable from coding. Code-level requirements focus mainly on implementation-related issues and constraints and are probably the best understood form of requirements specification. For an elevator system, an example code-level requirement is: "Due to the timing constraints, function calls to retrieve elevator call data shall be implemented in the C programming language rather than in the Python programming language."

This work and our requirements language is applicable and can be used at all these requirement and requirement specification levels in order to ensure full reuse and proper development of the new system.

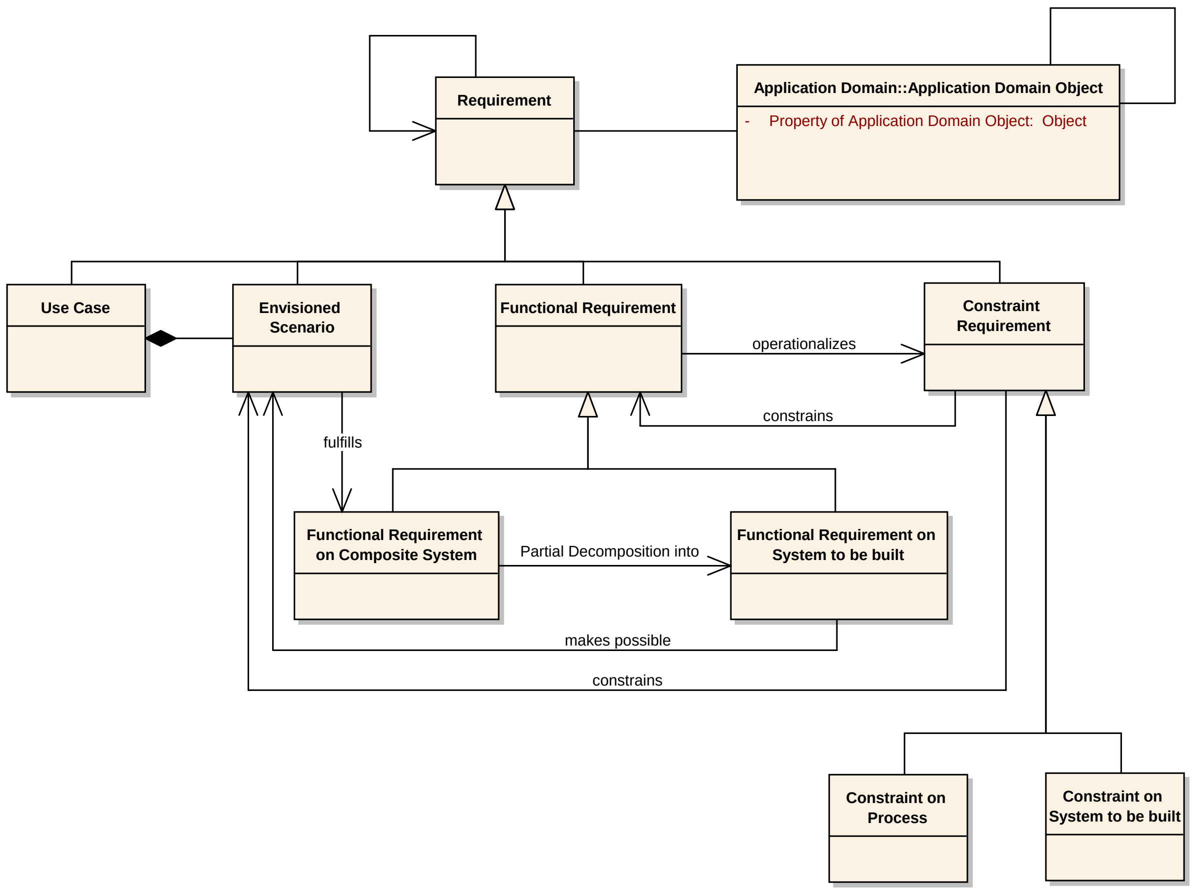

Requirement
Application Domain::Application Domain Object
- Property of Application Domain Object: Object
Use Case
Envisioned Scenario
Functional Requirement
Constraint Requirement
operationalizes
constrains
fulfills
Functional Requirement on Composite System
Partial Decomposition into
Functional Requirement on System to be built
makes possible
constrains
Constraint on Process
Constraint on System to be built

Figure 3.1: Conceptual Requirements Model

3.1 Requirements Model Overview

Our conceptual Requirements Model is presented in Figure 3.1. It shows how we conceptualize the domain of requirements, influenced by decades of research and practice of requirements engineering and especially [Kai97, Kai00, EK02, Kai05]. This conceptual model shows what models of concrete requirements should look like in our applications. This is already in the spirit of a metamodel, but the formal metamodel of our requirements specification language is given below.

The main entity in the Requirements Language is Requirement. Requirement can be decomposed into a number of other requirements or aggregated to composites, thus the granularity is flexible. There are four specialisations of requirements: Use Case, Envisioned Scenario, Functional Requirement, and Constraint Requirement:

- A Use Case consists of a number of Envisioned Scenarios that belong together in terms of use. E.g., the Use Case for getting cash money has several scenarios of how this is actually envisioned to be achieved.
- Envisioned Scenarios are the means of achieving high-level functions given as Functional Requirements on Composite System (e.g., Cash Withdrawal). The composite system includes the system to be built (e.g., an ATM) and possibly other systems (e.g., a bank system), including human users (e.g., bank customers).
- Functional Requirements are the generalisation of Functional Requirements on Composite System and Functional Requirement on System to be built. The latter are functions needed (e.g., Customer Identification or Cash Provision) that will make possible the enactment of Envisioned Scenarios once available. In effect, Functional Requirements on Composite System are partially decomposed into Functional Requirements on System to be built.
- Functional Requirements are tightly related to Constraint Requirements. Constraint Requirements are often operationalized by Functional Requirements. E.g., a security requirement will be operationalized by (required) functions for accepting a password, etc. At the same time, Constraint Requirements constrain other Functional Requirements. E.g., only solutions for Cash Provision are acceptable (in the overall solution space) that are reliable. There are two specialisations of Constraint Requirements: Constraint on Process and Constraint on System to be built. The former involve, for instance, development method or tools, the latter, for instance, security or reliability.

3.2 Requirements Model Details

Requirement

IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology defines requirement as:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition or capability as in (1) or (2).

As discussed previously, a requirement can exist at multiple abstraction levels. It is common to decompose higher-level requirements to lower-level requirements forming some kind of requirement decomposition tree. Each requirement is typically related to a number of other requirements. In addition, each requirement can be represented in multiple views. As such, we have a number of dimensions that constrain and make it challenging to capture a requirement properly with all its relationships.

This difficulty was obvious even during our work. It was hard to boil down and to keep in mind the clear definition of what a requirement is. This was particularly difficult when discussing different types of requirements. Any deviation from the common definition resulted in a ripple effect of conflicts with other terms and definitions of other language constructs. This is why we insisted on strictly following and not modifying the standard definition of what a requirement is in the early stages our work. Nevertheless, at the end we had to adapt and limit this definition as discussed in the Discussion section.

Use Case

The official definition for use cases in our project is:

> "A collection of possible scenarios between the system under discussion and external actors, characterized by the goal the primary actor has toward the system's declared responsibilities, showing how the primary actor's goal might be delivered or might fail." [Coc97]

Envisioned Scenario

The official definition for a scenario in our project is:

> "A sequence of interactions happening under certain conditions, to achieve the primary actor's goal, and having a particular result with respect to that goal. The interactions start from the triggering action and continue until the goal is delivered or abandoned, and the system completes whatever responsibilities it has with respect to the interaction." [Coc97]

The difference between scenario and envisioned scenario is rather a significant. We use the term “envisioned” in order to emphasise that the captured scenario is not-yet being performed as described, i.e., the system that will support this scenario is not yet built.

Although subtle, the difference that the term “envisioned” brings in is the realisation that the scenario that we have written at the time the requirements are specified may not necessarily be exactly the same as at the time the system that will support it is finished. Keeping this in mind helps developers and stakeholders realise that the scenarios and use cases that unify them are never really fully complete until the system is delivered; and even after the delivery of the system it makes it easier to modify use cases and scenarios with even less resistance on anyone’s behalf.

Again, the same as with use cases, our language supports writing scenarios using the language of different levels of formality; from informal natural language to constrained versions such as SVO(O). Of course, a more formal representation is to be preferred for any kind of processing by machine, in our case for finding similar cases facilitating reuse.

Functional Requirement

Taking into account the previously mentioned definition of a requirement, one can define a functional requirement as:

1. A capability needed by a user to solve a problem or achieve an objective.
2. A capability that must be met by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a capability as in (1) or (2).

Compared to the previous definition, the definition of a functional requirement limits the requirements definition to the usage of the term capability.

Functional requirements typically represent the majority of the requirements. A major issue to resolve with any functional requirement is who is responsible for fulfilling it, i.e., what is the system that is performing the activity or activities that will satisfy the respective functional requirement.

This issue often comes up when a functional requirement is discussed in relationship with a use case or scenario. Is a use case or scenario a functional requirement? Since fulfilment of a use case typically involves activities performed by both actors and the system to be built, we cannot say that a use case is a functional requirement on the system to be built. On the other hand, if we take into consideration that actors together with a system to be built from another, composite, system, one can claim that use cases and scenarios are requirements on that composite system. That is, the name of the use case or scenario can be thought of as a functional requirement on a composite system, and the actual steps that the system to be built has to perform in order to satisfy the functional requirement on the composite system can be thought of either as functional requirements on the system to be built or as activities needed to fulfil those requirements, depending on the perspective taken.

Therefore, in our model we have two different kinds of functional requirements: functional requirement on composite system and functional requirement on system to be built.

Functional Requirement on Composite System

Functional Requirement on Composite System is a functional requirement that is supposed to be fulfilled by the system composed of system to be built and its actors. This type of requirement is primarily fulfilled through the envisioned scenarios.

Functional Requirement on System To Be Built

Functional Requirement on System To Be Built is a functional requirement that is supposed to be fulfilled by the system that is being specified. This type of requirement primarily appears as the steps in envisioned scenarios, i.e., makes envisioned scenarios possible.

3.3 Why No Goals?

Goal-driven requirements engineering is an important area of requirements engineering. In particular, several researchers proposed goal-driven RE [DvLF93, BI96, Kai95, MCY99, Kai00, vLL00] as a promising technique for dealing with domain-level requirements for large systems. Goal-driven RE focuses on ensuring that software actually fulfils business needs and requirements. This focus has been achieved by shifting from considering *what* a system should do to considering *why* the CBS should provide particular functionality. In other words, *requirements rationale* is the main focus.

Although most of the original goal-driven RE techniques concern domain-level requirements, for example, through analysis of *personal* and *system* goals, the main idea of goal-driven RE techniques has been to enhance certain traditional requirements techniques such as use cases [Coc00]. Nevertheless, although goal-driven RE techniques are extensively studied, goal-driven RE remains an immature area. This immaturity is apparent from the many different definitions of the word "goal" [DvLF93, Ant96, MCY99, Kai00]. The common pattern to all these definitions is that goals capture the *intention, i.e., objective,* and the *target state* for the entity under analysis and at the entity's own abstraction level. For example, in the case of an elevator system, a goal for the elevator system is to *deliver passengers to the requested floor*. This goal captures the *intention* of *delivering passengers* and also the *target state* of *arriving at the requested floor*. This particular goal captures the rationale for the elevator's responsibility for carrying passengers from a floor to another.

An interesting point to note is that depending on the abstraction level from which one is observing a system and the goal decomposition, a goal may or may not become a functional requirement. For example, for an elevator system, the next level of the goal decomposition might include goals such as *move elevator cab*, *stop elevator cab*, *pick up a passenger*, and so on. Now, if we start working at this abstraction level, the higher-level goal of *delivering passengers to the requested floor* becomes a functional requirement for the lower-level goals such as *moving elevator cab*. An advantage of this goal hierarchy is that it provides traceability when moving from one abstraction level to another and from one goal decomposition level to another.

Overall, one can see that the goals can be ultimately represented or decomposed as regular requirements. In fact, one can even claim that goals are neither necessary nor sufficient for the specification of a system. As such, in our project we have decided not to include them as part of the requirements model. This is not to be interpreted as if goals are not useful for the specification of a system. Quite contrary, but we had to make this decision in the early stages of specifying our language in order to make it:

- manageable,
- useful to those who are not using goals as part of their requirements work,

Nevertheless, it will be possible to extend the language and add goals in the future revisions of the language if deemed necessary or desirable.

Chapter 4

Representation of Functional and Behavioural Requirements

In this section we discuss different requirements specification techniques. Prior to this discussion, it is important to emphasise different aspects of a system to be built from a RE perspective. The four main aspects of each system from the RE perspective are processes, data, architecture, and interfaces. Requirements specification techniques focus on modelling one of these four main aspects. Nevertheless, in many articles in the requirements literature, this division is represented slightly differently.

4.1 Requirements Representation Model Overview

The Requirements Representation Model is presented in Figure 4.1. The main entity in the Requirements Representation Language is Requirement Representation, which is used to represent a Requirement entity from the Requirements Language. The Requirement Representation Language supports representing requirements in two distinct ways:

1. Through the use of Descriptive Requirement Representation, i.e., through Natural Language Requirement Statements and/or Constrained Language Requirement Statements, and
2. Through the use of Model-Based Requirement Representation, in particular UML-Based Requirement Representation.

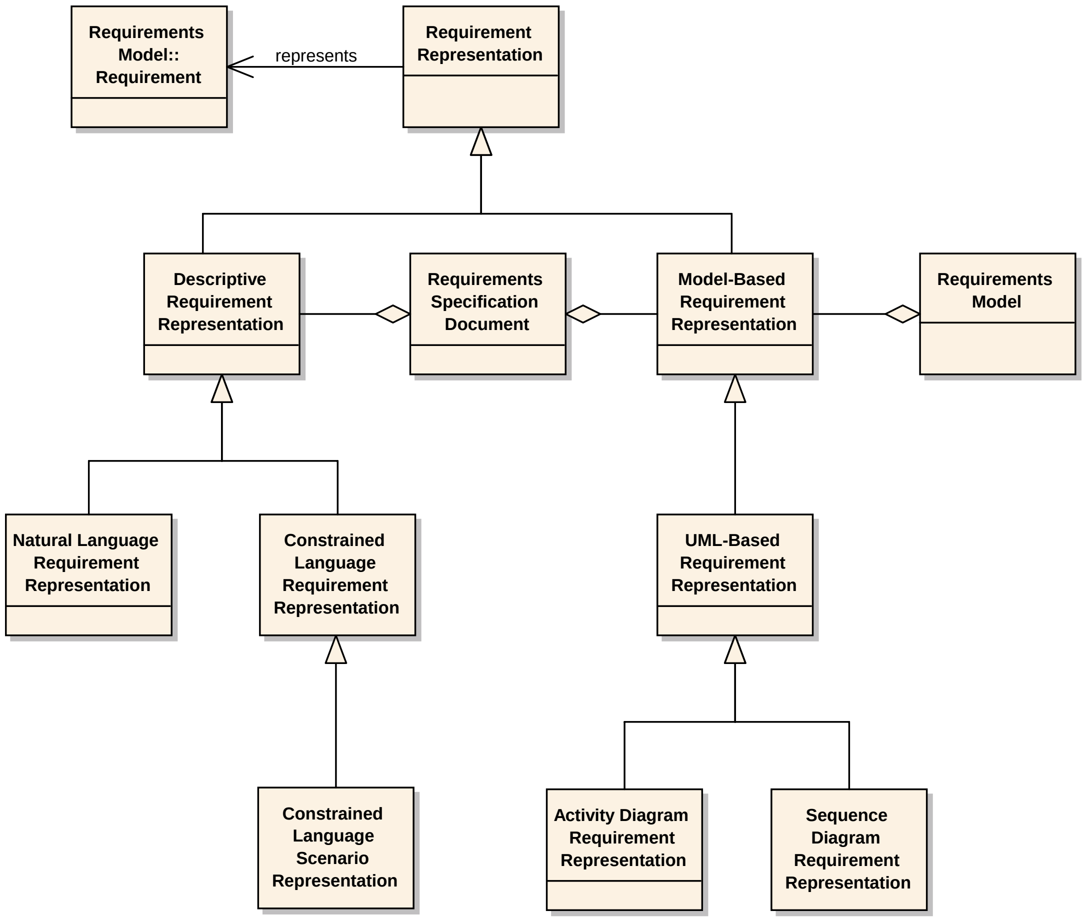

Requirements Model:: Requirement
represents
Requirement Representation
Descriptive Requirement Representation
Requirements Specification Document
Model-Based Requirement Representation
Requirements Model
Natural Language Requirement Representation
Constrained Language Requirement Representation
UML-Based Requirement Representation
Constrained Language Scenario Representation
Activity Diagram Requirement Representation
Sequence Diagram Requirement Representation

Figure 4.1: Requirements Representation Model

The mapping from the Requirements Domain to the Requirements Representation Language is not clear cut. For example, a simple Functional Requirement can be captured through the use of a Natural Language Requirement Statement, while a Use Case can be captured through the use of both Constrained Language Requirement Statements and UML-Based Requirement Representations complementing each other.

A self-contained set of Model-Based Requirement Representations makes up a Requirements Model. A Requirements Specification Document contains instances of Descriptive Requirement Representation or Model-Based Requirement Representation.

4.2 Requirements Representation Model Details

Requirement Representation

Requirement Representation is the requirement as specified using a requirements specification language.

Descriptive Requirement Representation

Descriptive Requirement Representation is the requirement as specified using a descriptive specification language, e.g., natural language, SVO(O), etc.

Natural Language Requirement Representation

Natural Language Requirement Representation is the requirement as specified using a natural language, e.g., English, Turkish, Bulgarian, etc. Note, that we technically include also hypertext links into natural-language text, see below. The use of hypertext for representing requirements was already proposed long time ago, see [Kai93, Kai96], but it was not yet defined as precisely as below in a metamodel.

Constrained Language Requirement Representation

Constrained Language Requirement Representation is the requirement as specified using a controlled/constrained natural language, e.g., SVO(O), Attempto Controlled English (ACE), etc.

Constrained Language Scenario Representation

Constrained Language Scenario Representation is the envisionary scenario specified using a constrained language.

Model-Based Requirement Representation

Model-Based Requirement Representation is the requirement as specified using a modelling language, e.g., UML, etc.

UML-Based Requirement Representation

UML-Based Requirement Representation is the requirement as specified in RSL in parts that are based on UML.

Activity Diagram Requirement Representation

Activity Diagram Requirement Representation is the requirement as specified using UML-Based activity diagrams specialised for the requirements specification purposes.

Interaction Diagram Requirement Representation

Interaction Diagram Requirement Representation is the requirement as specified using UML-Based sequence or communication diagrams specialised for the requirements specification purposes.

Chapter 5

Discussion

Going back to the original IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology definition of requirement that we used for this project:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition or capability as in (1) or (2).

we can see in (3) that it includes the documented *representation* of the requirement. During this project we realised that the documented representation of the requirement should not be considered to be the requirement itself, but a representation. There are several reasons:

- A requirement is something that exists even if it is never documented, i.e., represented.
- A requirement can have multiple representations.
- A requirement can be represented at different abstraction levels.
- Each representation of a requirement is usually not complete from each possible perspective.

The first issue tackles the fact that requirements and their representations belong to two different dimensions. Similar to real world entities and the OOA representations, an orange in the real world is different than an orange represented using UML. Besides the fact that it is very hard to elicit all requirements and represent them properly, even when they are elicited, their representations are not necessarily the right ones.

The second issue is that each requirement can be represented in multiple ways and some can be represented in certain ways that others cannot. For example requirements concerning different sound types and ranges are hard to capture using UML sequence diagrams.

The third issue is that requirements can be represented at different abstraction levels leaving some part of the actual requirement out. By doing this any single requirement representation abstraction level represents no full requirement. On the other hand, requirements themselves are harder to decompose into multiple abstraction levels.

Finally, taking into account that each requirement can be represented in a number of different ways, and for each way at different abstraction levels, it is obvious that almost any requirements representation cannot be considered as complete representation of the requirement and as such different than the requirement itself.

This distinction between requirements and their representations is evident throughout our language. As such, it removes the confusion between requirements themselves and their representations that commonly exists in the requirements engineering community. The removal of this problem is one of the prerequisites for successful capture of the requirements, proper traceability, and quality insurance involved in checking that requirement specification is consistent and complete. This is one of the major contributions of our language and this project.

Part II

Language Reference

Chapter 6

Requirements

6.1 Overview

The Requirements part defines all the RSL constructs that pertain to Requirements as such and relationships between them. This part of the language defines only the top level elements which do not show details of individual representations of Requirements. Language users will typically use elements from this part to present requirements as such (in diagrams and project trees) contained in the requirements specifications they create. These diagrams or trees will generally consist of icons denoting individual Requirements (including UseCases) and lines denoting appropriate RequirementRelationships (including relationships for UseCases).

The specification in this part contains three packages, as shown in Figure 6.1.

- The RequirementsSpecifications package contains all the general constructs. These constructs allow for expressing whole requirements specifications, groups of logically related requirements (their packages) and individual requirements. It «import»s from the UML :: Kernel package to allow for specializing the syntax and semantics of UML Packages. It also «merge»s the UML :: UseCases package as it redefines the UML's UseCase class.
- The RequirementsRelationships package generally introduces the possibility to relate individual requirements. These relationships are very general, but their semantics can be constrained by applying stereotypes (as in UML). In this way, conceptual relationships between Requirements as specified in Chapter 4 can be expressed. This package also imports from UML :: Kernel in order to reuse the syntax and semantics of DirectedRelationships.

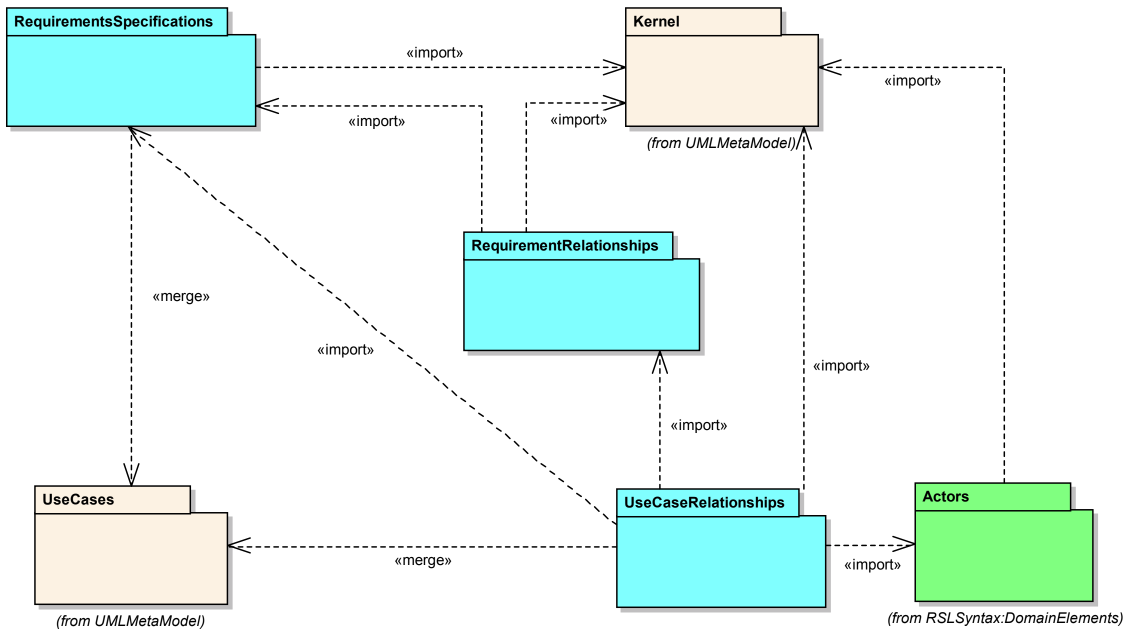


Figure 6.1: Overview of packages inside the Requirements part of RSL

- The UseCaseRelationships package is a modification of the UML :: UseCases package with which it «merges» to redefine use case relationship classes. It also relates UseCases with DomainElements :: Actors :: Actors, which is make such relationships explicit as opposed to what is defined in UML.

Individual classes in the above packages can be traced from the conceptual model described in Chapter 3. These traces are shown in Figure 6.2. The main class in this part is the Requirement class. This class allows for expressing requirements as such without going into details of the requirement's representation. Its source is the Requirements Model :: Requirement class from the conceptual model. Different specialisations of Requirement in the conceptual model are expressed through ReqTypes «enumeration». This enumeration defines values for the 'type' of Requirement. UseCase class is introduced due to a specific syntax for relationships between UseCases. This class directly traces from the Requirements Model :: UseCase class from the conceptual model. In addition, RequirementRelationship class allows for connecting Requirements with different relations defined in the conceptual model: 'constrains', 'operationalises', 'fulfills' and 'partial decomposition into'. Other relationships in this part include relationships that pertain to the UseCase class and are not directly described in the conceptual model. However, these relationships directly specialise and modify appropriate relationships from the UML model.

All the elements contained in the Requirements part can be shown on Requirements Diagrams or Use Case Diagrams. Requirements Diagrams show Requirement icons with relevant RequirementRelationships between them. Use Case diagrams show UseCases with DomainElements ::

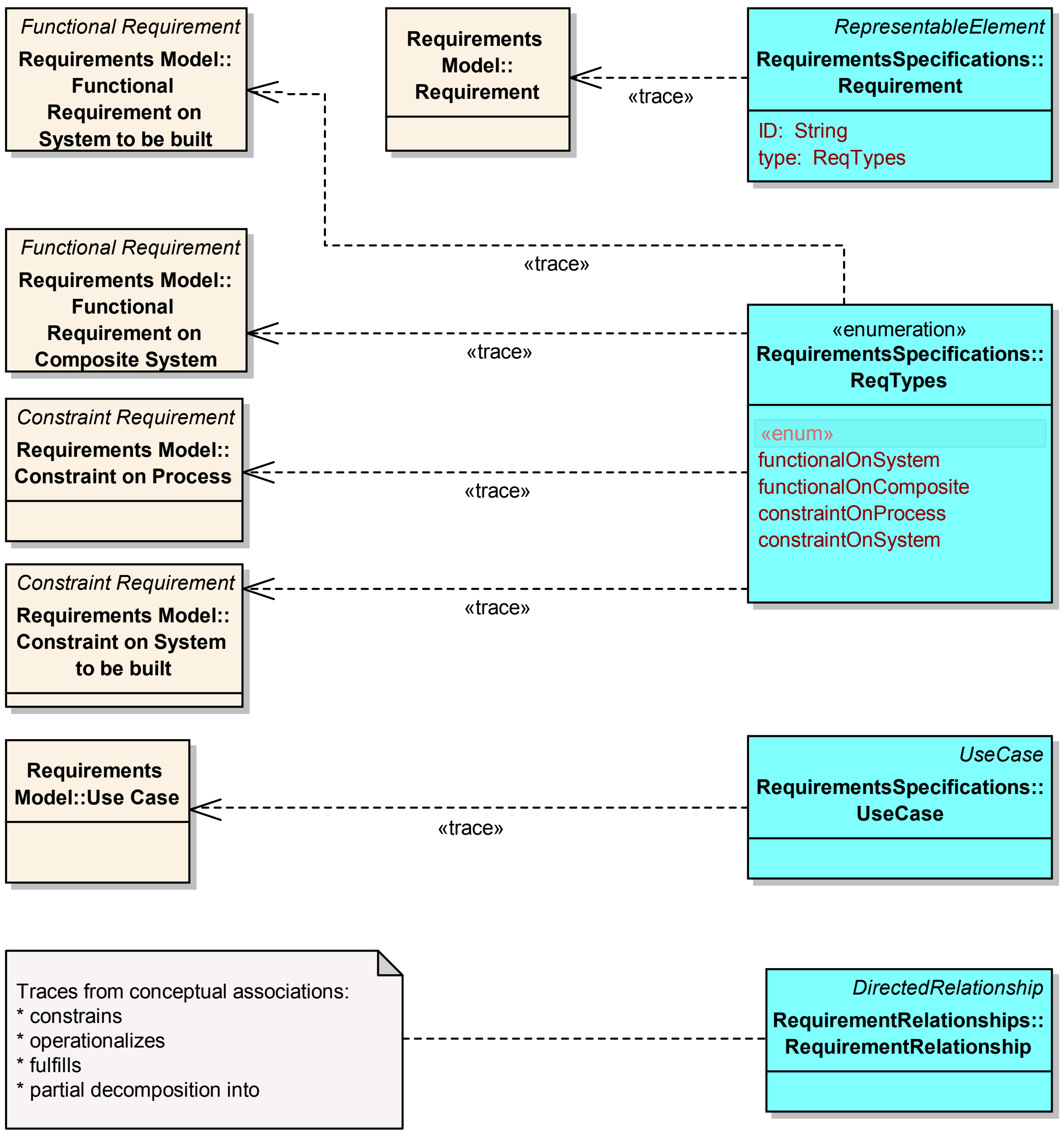

Functional Requirement
Requirements Model:: Functional Requirement on System to be built
Requirements Model:: Requirement
RepresentableElement
RequirementsSpecifications:: Requirement
ID: String
type: ReqTypes
«trace»
Functional Requirement
Requirements Model:: Functional Requirement on Composite System
«enumeration»
RequirementsSpecifications:: ReqTypes
«enum»
functionalOnSystem
functionalOnComposite
constraintOnProcess
constraintOnSystem
Constraint Requirement
Requirements Model:: Constraint on Process
Constraint Requirement
Requirements Model:: Constraint on System to be built
Requirements Model::Use Case
UseCase
RequirementsSpecifications:: UseCase
Traces from conceptual associations:
* constrains
* operationalizes
* fulfills
* partial decomposition into
DirectedRelationship
RequirementRelationships:: RequirementRelationship

Figure 6.2: Main classes in the Requirements part with traces from the conceptual model

Actors :: Actors, also with appropriate relationships between them. These diagrams do not show the details of individual elements. These details can be shown using diagrams or text as defined in the RequirementRepresentations part.

Requirements can be also logically grouped into larger containers - RequirementPackages. These containers can be shown on Package Diagrams as derived from UML. Containment of Requirements in RequirementPackages can be shown in Project Trees. These trees show containment hierarchies of RequirementsSpecifications with RequirementsPackages and Requirements.

6.2 Requirements specifications

6.2.1 Overview

This package describes the general structure of requirements specifications. This structure is similar to the structure of Models in UML. So, by analogy, we have the RequirementsSpecification class that defines the top level element holding a complete specification of requirements for a specific system. Every such specification has to have a DomainElements :: DomainVocabulary and can be divided into many RequirementsPackages. RequirementsPackages can be nested and contain Requirements (including UseCases). Different types of Requirements are distinguished by ReqTypes «enumeraion».

Requirements are presented on Requirements Diagrams as simple rectangle icons with their 'name', 'ID' and 'type' appropriately expressed. This notation is modified for UseCases by substituting rectangles with ovals (for consistency with UML)on Use Case Diagrams. RequirementsSpecifications and RequirementsPackages can be presented on Package Diagrams that have their syntax derived from UML Package Diagrams. Requirements and UseCases can be placed in Project Trees under appropriate Packages and a RequirementSpecification. These trees are presented as any browser tree with appropriate small icons expressing all the above elements.

6.2.2 Abstract syntax and semantics

Abstract syntax for the RequirementsSpecifications package is described in Figure 6.3.

Requirement

Semantics. Requirement is understood as a placeholder for one or more BasicRepresentations :: RequirementRepresentations. It is treated as a concise way to symbolize this representation. Type of a Requirement is restricted by ReqTypes enumeration.
Abstract syntax. Requirement is a kind of ElementRepresentations :: RepresentableElement Requirement has a 'name' which is a BasicRepresentations :: HyperlinkedSentence. It is detailed with one or more 'representations' in the form of BasicRepresentations :: RequirementRepresentation. Requirements can be grouped into RequirementsSpecifications :: Require-

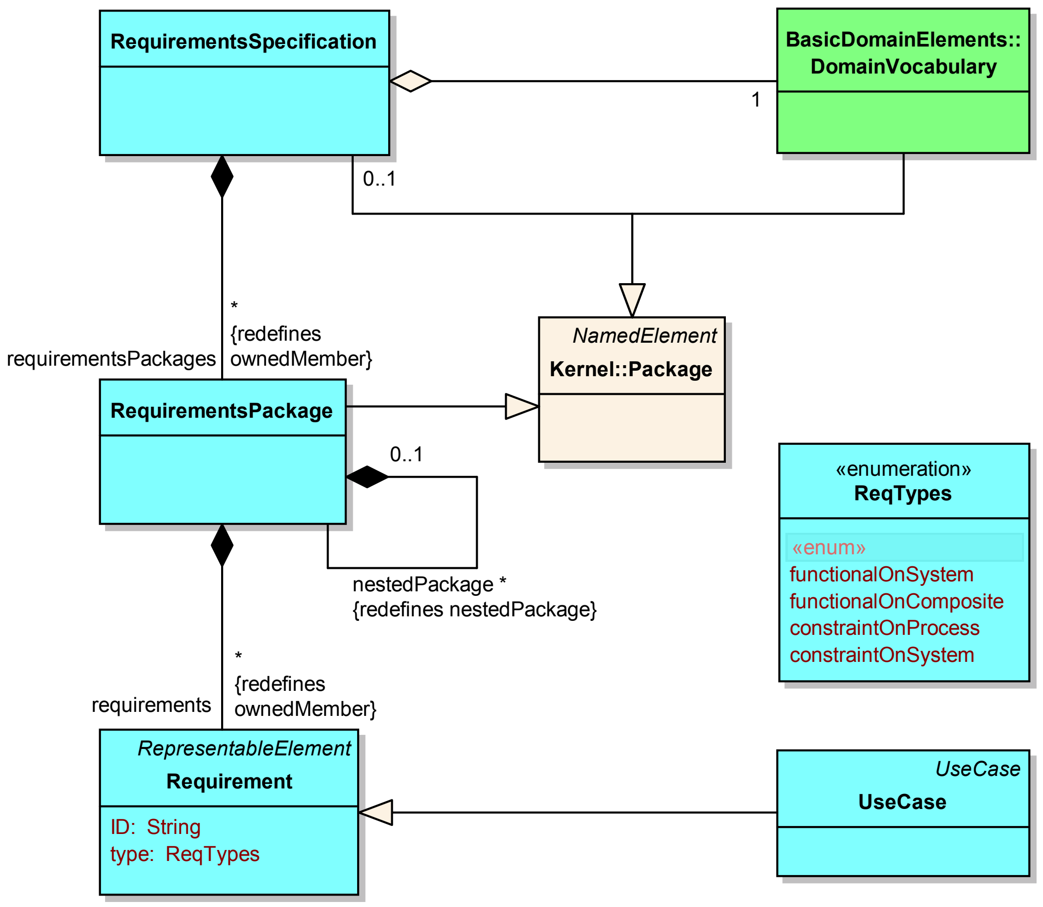


Figure 6.3: Requirements specifications

mentsPackages. Requirement has an attribute type with value range determined by ReqTypes enumeration.

ReqTypes

Semantics. ReqTypes is is an enumeration type that defines literals to determine the kind of Requirements in a RequirementSpecification. ReqTypes specifies the following types of requirements:

- functionalRequirementOnSystem – a functional requirement of a software system
- functionalRequirementOnComposite – a functional requirement of a part of the system (for instance a component)
- constraintOnProcess – a requirement constraining development process of the software system

- constraintOnSystem – a non-functional requirement of the software system

Abstract syntax. ReqTypes is an enumeration of the following literal values:

- functionalRequirementOnSystem
- functionalRequirementOnComposite
- constraintOnProcess
- constraintOnSystem

RequirementsPackage

Semantics. RequirementsPackage is a type of UML Package, i.e. a structure that groups elements and constitutes a container for these elements. It can contain only Requirements and their specialisations.
Abstract syntax. RequirementsPackage is a specialisation of UML :: Kernel :: Package ([Obj05b]). It redefines ownedMember. Owned members for the RequirementsPackage must be Requirements. It also redefines nestedPackage, which can only be another RequirementsPackage. Every RequirementsPackage can be part of a RequirementsSpecification.

RequirementsSpecification

Semantics. RequirementsSpecification is a type of UML Package, i.e. a structure that groups elements and constitutes a container for these elements. It can contain all elements of a requirements specification for a given project - Requirements grouped in RequirementsPackages and BasicDomainElements :: DomainElements grouped in BasicDomainElements :: DomainVocabulary. RequirementsSpecification is a root package for the whole requirements specification. It can be treated as equivalent of Model from UML.
Abstract syntax. RequirementsSpecification is a specialisation of UML :: Kernel :: Package ([Obj05b]). It can contain many RequirementsPackages and one DomainVocabulary.

UseCase

Semantics. UseCase has the same meaning as described in the UML Superstructure: "A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system." ([Obj05b]) This definition is analogous to one specified in section 3.2 of Chapter 4. In accordance with the classification in Chapter 4, this semantics is extended by stating that UseCase is a special kind of Requirement. It is also a placeholder for its Representations which might be of various kinds, as specified in the Representations and UseCaseRepresentations packages. These representations describe the behaviour (set of actions) for the UseCase.
Abstract syntax. UseCase is a specialization of UML :: UseCases :: UseCase and Requirement. Instances of UseCase can be related with each other by UseCaseRelationships :: InvocationRelationships. This relationship redefines UML's extend and include. UseCase can contain several UseCaseRelationship :: Participation relationships and can be pointed to by UseCaseRelationship :: Usage relationships. These relationships relate it with Actors :: Actors. It can contain InteractionRepresentations :: InteractionScenarios, ConstrainedLanguageRepresentations :: ConstrainedLanguageScenarios and an ActivityRepresentations :: ActivityScenario as its representations.

6.2.3 Concrete syntax and examples

Requirement. It is depicted as a rectangle with two additional vertical lines on its left. Requirement's 'ID' is written in the top left corner of the box. Requirement's 'name' is written inside the rectangle centred horizontally and vertically. See Figure 6.4 for illustration of this, and Figure 6.12 for an example of usage of these icons in a Requirement Diagram.

Optionally Requirement can have its type shown in form of text indicating this type surrounded by double angle brackets ("« »", an "angle quote").

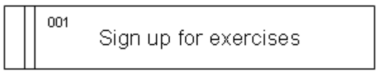


Figure 6.4: Requirement example

ReqTypes. It can be represented by the following values:

- for functionalRequirementOnSystem – "functional"
- for functionalRequirementOnComposite – "functional"

- for constraintOnProcess – "constraint"
- for constraintOnSystem – "constraint"

RequirementsPackage. Concrete syntax is almost the same as for Kernel :: Package, described in UML Superstructure (in [Obj05b], paragraph 7.3.37, page 104): "A package is shown as a large rectangle with a small rectangle (a 'tab') attached to the left side of the top of the large rectangle. The members of the package may be shown within the large rectangle. Members may also be shown by branching lines to member elements, drawn outside the package. A plus sign (+) within a circle is drawn at the end attached to the namespace (package). If the members of the package are not shown within the large rectangle, then the name of the package should be placed within the large rectangle. If the members of the package are shown within the large rectangle, then the name of the package should be placed within the tab. In addition to the above Kernel :: Package description, RequirementsPackage has two vertical lines to the left of the "rectangle with a tab" icon. It can also be presented in a tree structure. See Figures 6.5, 6.6 for examples of concrete syntax in a Package Diagram and in a Project Tree structure, respectively.

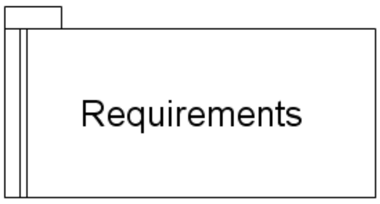


Figure 6.5: RequirementsPackage example

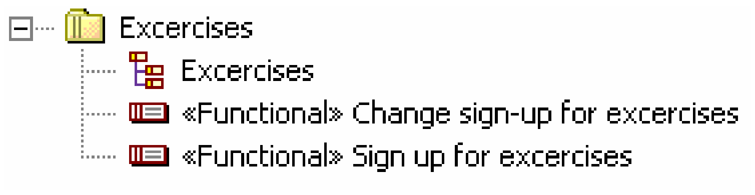

Excercises
Excercises
«Functional» Change sign-up for excercises
«Functional» Sign up for excercises

Figure 6.6: RequirementsPackage tree example

RequirementsSpecification. Concrete syntax is almost the same as for Kernel :: Package, described in UML Superstructure (in [Obj05b]); for this description see concrete syntax for RequirementsPackage. In addition to concrete syntax for plain Packages, RequirementsSpecification has one thick vertical line on its left. It can also be presented in a Project Tree structure with a minimized icon. See Figures 6.7, 6.8 for illustration of RequirementSpecification icon on a Package Diagram and in a Project Tree.

UseCase. Concrete syntax is an extension of concrete syntax for UML :: UseCases :: UseCase, as described in UML Superstructure ([Obj05b], paragraph 16.3.6, page 579). "A use case is shown as an ellipse, either containing the name of the use case or with the name of the use

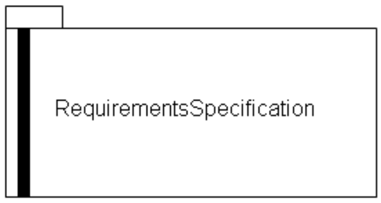


Figure 6.7: RequirementsSpecification example

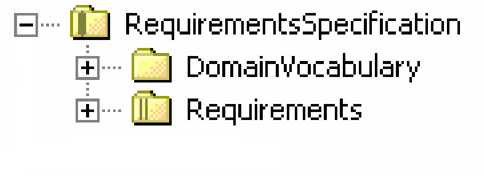


Figure 6.8: RequirementsSpecification tree example

case placed below the ellipse.” As for any Requirement, every UseCase icon can present the ‘ID’ (see concrete syntax for Requirement). Additionally, UseCase can be presented with a minimised icon on a tree structure. See Figures 6.9, 6.10 for illustration of the above on a Use Case Diagram and Project Tree structure, respectively. See also 6.14 for example of usage of UseCases on Use Case Diagrams.

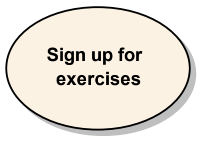


Figure 6.9: UseCase example

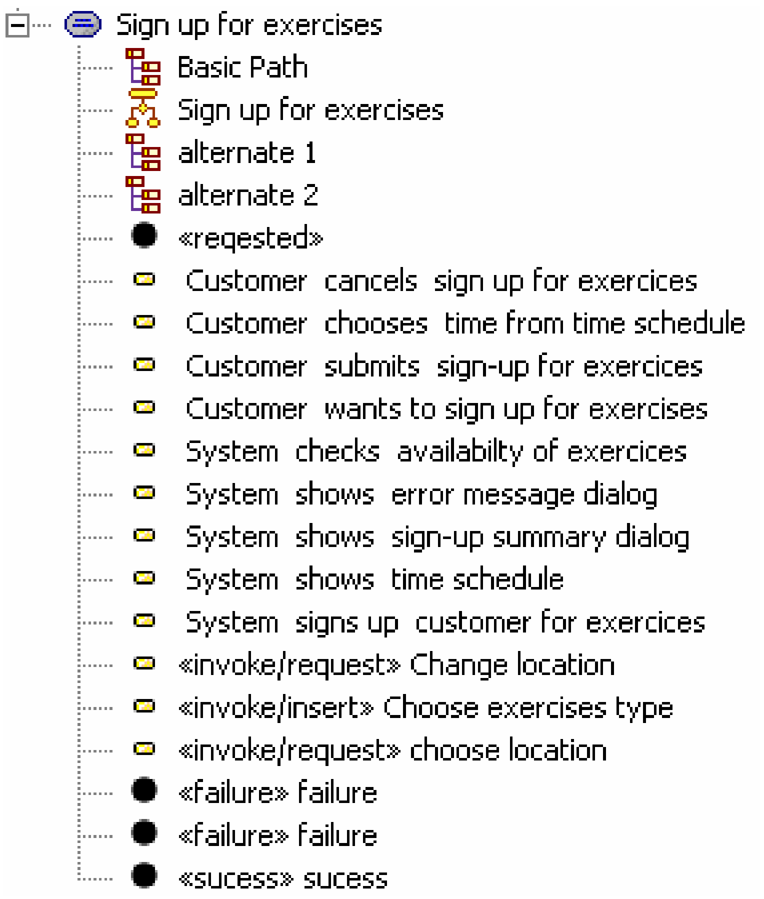
6.3 Requirement relationships

6.3.1 Overview

This package describes relationships between requirements. It is assumed that these relationships are extensible through stereotyping mechanisms. Specific standard stereotypes, that derive from the conceptual model are defined.

6.3.2 Abstract syntax and semantics

Abstract syntax for the RequirementRelationships package is described in Figure 6.11.



Sign up for exercises
- Basic Path
- Sign up for exercises
- alternate 1
- alternate 2
- «reqested»
- Customer cancels sign up for exercices
- Customer chooses time from time schedule
- Customer submits sign-up for exercices
- Customer wants to sign up for exercises
- System checks availabilty of exercices
- System shows error message dialog
- System shows sign-up summary dialog
- System shows time schedule
- System signs up customer for exercices
- «invoke/request» Change location
- «invoke/insert» Choose exercises type
- «invoke/request» choose location
- «failure» failure
- «failure» failure
- «sucess» sucess

Figure 6.10: UseCase tree example

RequirementRelationship

Semantics. RequirementRelationship denotes a relationship between two requirements. The type of a relationship (e.g. similarity, conflict) is specified by a stereotype defined in an appropriate Profile.

Abstract syntax. RequirementRelationship is a kind of DirectedRelationship from the UML :: Kernel package [Obj05b]. RequirementRelationship is a component of RequirementsSpecifications :: Requirement (source of the relationship) and it points to another RequirementsSpecifications :: Requirement (target of the relationship). Source of the relationship should be different than its target – RequirementsSpecifications :: Requirement cannot be associated with itself.

6.3.3 Concrete syntax and examples

RequirementRelationship is drawn as a dashed line connecting two RequirementsSpecifications :: Requirements. An open arrowhead may be drawn on the end of the line indicating the target of the relationship. The line is labeled with an appropriate stereotype determining the

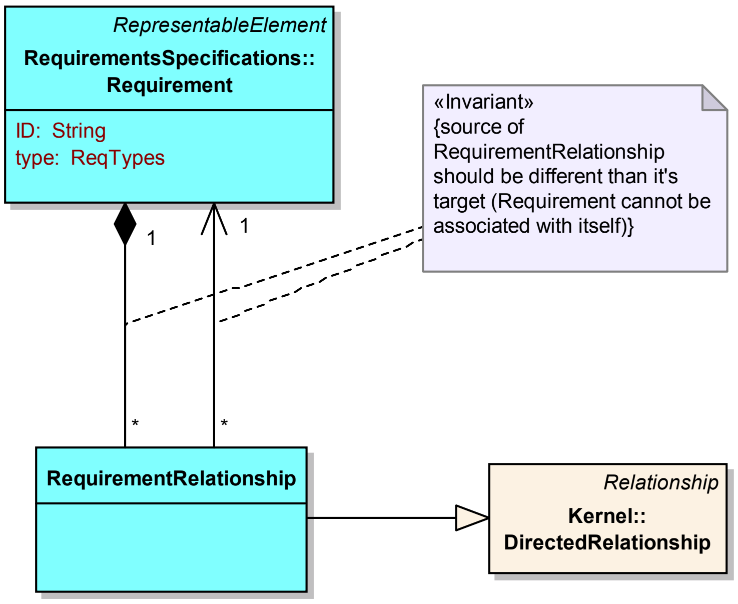

RepresentableElement
RequirementsSpecifications::
Requirement
ID: String
type: ReqTypes
1
1
*
*
«Invariant»
{source of RequirementRelationship should be different than it's target (Requirement cannot be associated with itself)}
RequirementRelationship
Relationship
Kernel::
DirectedRelationship

Figure 6.11: Requirement relationships

type of a relationship (see Figure 6.12). The line may consist of many orthogonal or oblique segments.

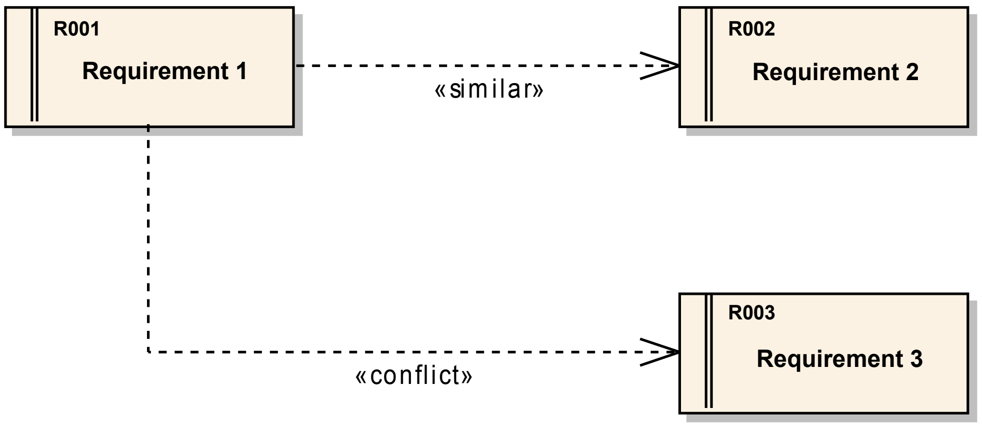


Figure 6.12: Requirements and requirement relationships concrete syntax example

6.4 Use case relationships

6.4.1 Overview

This package describes relations between UseCases and Classifiers (mainly Actors) or between two UseCases. The UseCaseRelationships package redefines parts of the UseCases package from the current UML specification [Obj05b].

6.4.2 Abstract syntax and semantics

Abstract syntax for the UseCaseRelationships package is described in Figure 6.13.

InvocationRelationship

Semantics. InvocationRelationship substitutes «include» and «extend» relationships from UML and unifies their disadvantageous semantics ([Sim99], [MOW01]). InvocationRelationship denotes that another use case (actually, one of its scenarios) can be invoked from within currently performed use case. After performing one of the final actions in the invoked use case, the flow of control returns to the invoking use case right after the point of invocation to perform the remaining part of the base use case. There are two types of invocation: a use case can be invoked conditionally – only when requested by an actor, or unconditionally – every time the appropriate scenario of the base use case is performed. The type of the invocation, the name of a use case to be invoked and the exact point of invocation in the invoking use case scenario is defined by a special kind of scenario sentence (see ScenarioSentences :: InvocationSentences in Chapter 8.4).
Abstract syntax. InvocationRelationship is a kind of DirectedRelationship from the UML :: Kernel package [Obj05b]. It redefines the Include and Extend relationships from the current UML meta-model ([Obj05b], p.570). It is part of an 'invoking' RequirementSpecifications :: UseCase pointing to an 'invoked' UseCase. It also points to an ScenarioSentences :: InvocationSentence, which must be contained in a RequirementRepresentation of the 'invoked' UseCase.

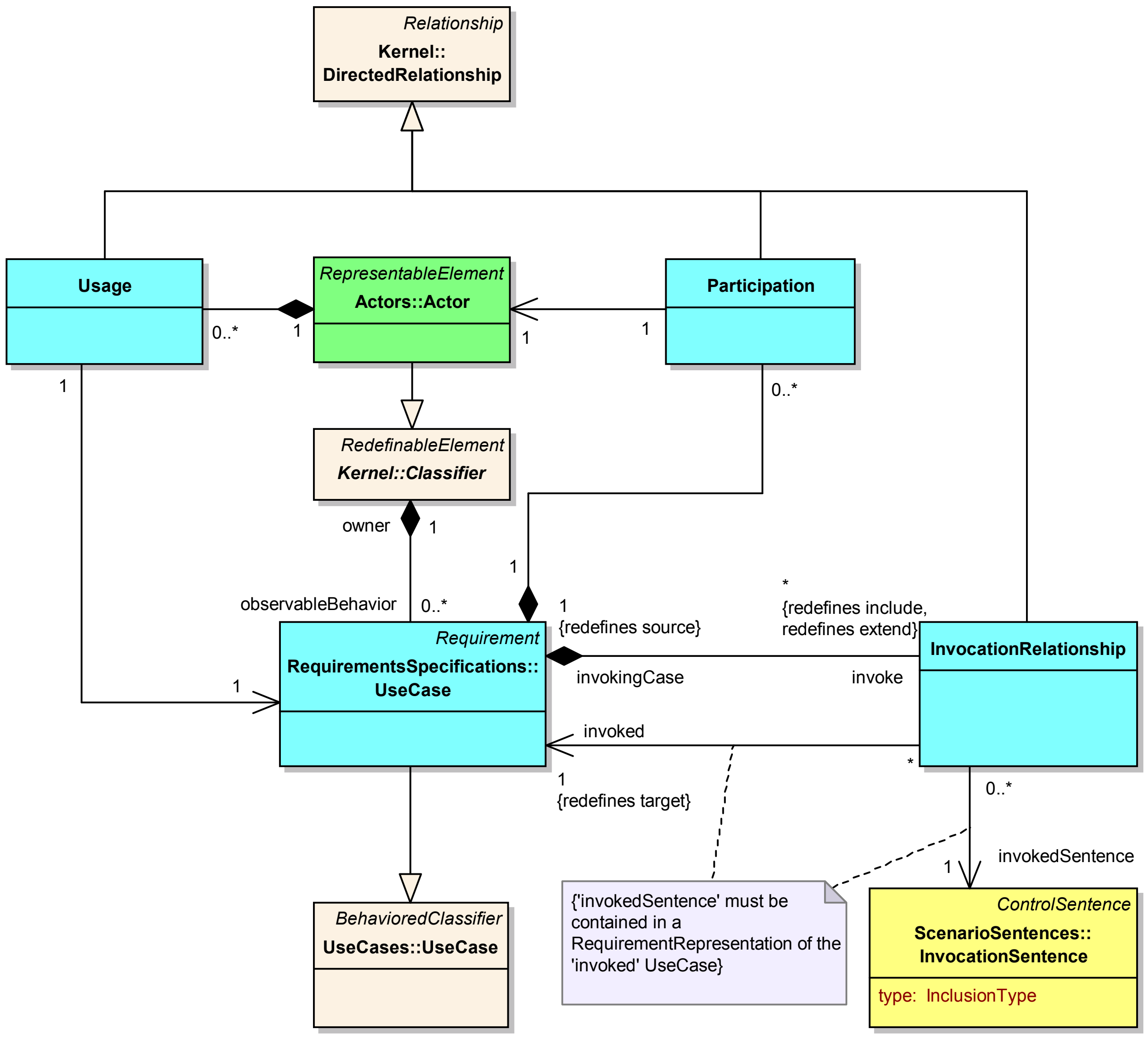


Figure 6.13: Use case relationships

Usage

Semantics. Usage indicates possibility for an Actors :: Actor to initiate a particular UseCase performance directly as a primary actor (the one that expects to reach the use case's goal).
Abstract syntax. Usage is a kind of DirectedRelationship from the UML :: Kernel package [Obj05b]. It is a component of an Actor and points to a RequirementSpecifications :: UseCase.

Participation

Semantics. Participation indicates possibility for an Actor to participate as a secondary actor in the execution of a particular UseCase.

Abstract syntax. Participation is a kind of DirectedRelationship from the UML :: Kernel package [Obj05b]. It is a component of a RequirementSpecifications :: UseCase and points to an Actor.

6.4.3 Concrete syntax and examples

InvocationRelationship. It can be shown similarly to a UML Dependency relationship between RequirementSpecifications :: UseCases with an «invoke» stereotype and an open arrowhead denoting navigability on the end of the 'invoked' RequirementSpecifications :: UseCase (see Figure 6.14).

Usage. It's concrete syntax is a solid line between an Actor and a RequirementSpecifications :: UseCase and an arrowhead on the side of the UseCase (see Figure 6.14). This arrow can be appended with a «use» UML-like stereotype.

Participation's concrete syntax is a solid line between a RequirementSpecifications :: UseCase and an Actors :: Actor and an arrowhead on the side of the Actor (see Figure 6.14). This arrow can be appended with a «participate» UML-like stereotype.

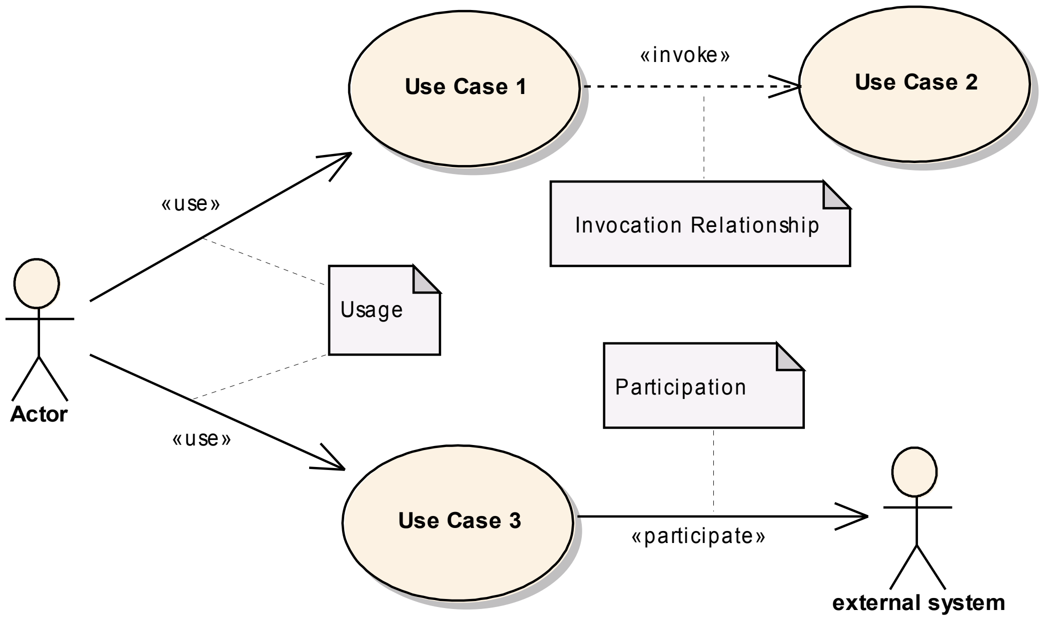


Figure 6.14: Use case relationships concrete syntax example

Chapter 7

Requirements representations

7.1 Overview

In the RequirementsRepresentations part we describe how our language defines requirement representations and present differences between various requirement representations. As stated in Part I, requirements can have descriptive representations (natural or constrained language) or schematic representations (model-based). Language users will typically use elements from this part to describe the contents of individual requirements using the notations chosen from those available in the language. Requirement representations are tightly bound to the appropriate requirements they represents. Particular representation depends highly on the requirement type, as specified in the Requirements part.

The specification in this part contains five packages, as shown in Figure 7.1.

- The BasicRepresentations package contains all the general constructs needed to polymorphically represent differing requirement representations. It «import»s from the RequirementsSpecifications package to relate representations with appropriate requirements defied there. It also «import»s the Phrases package and the Terms package. This allows for using a uniform vocabulary of domain notions which makes all the representations within a requirements specification - coherent. In this manner the differing requirement representations can be specified as being specialisations of elements from the BasicRepresentations package. As such the BasicRepresentations package gives access to the RequirementsSpecifications package, the Terms package and the Phrases package for any kind of requirement representation being utilised within a requirements specification. The

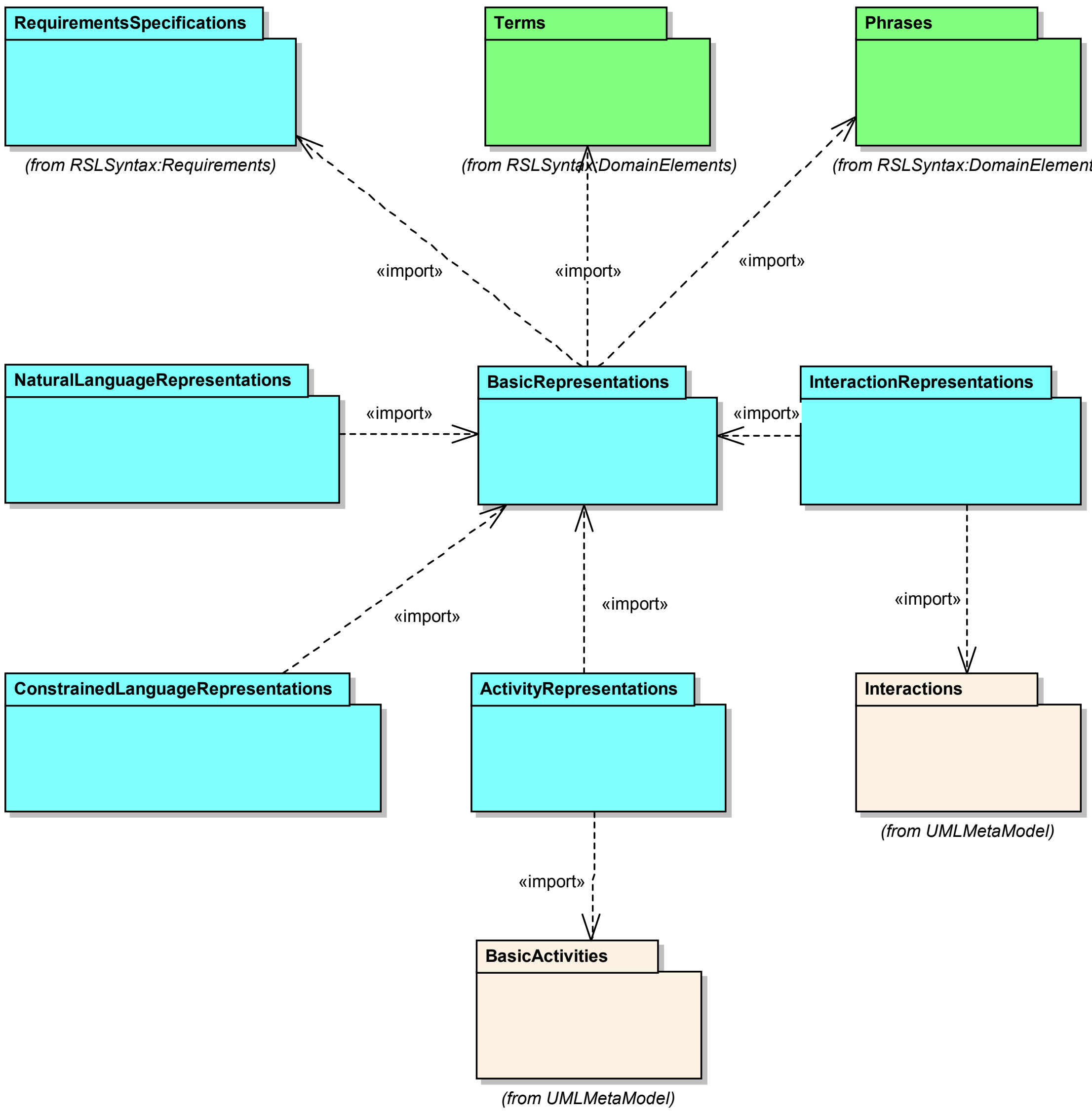

RequirementsSpecifications
(from RSLSyntax:Requirements)
Terms
(from RSLSyntax:DomainElements)
Phrases
(from RSLSyntax:DomainElements)
«import»
«import»
«import»
NaturalLanguageRepresentations
«import»
BasicRepresentations
«import»
InteractionRepresentations
«import»
«import»
«import»
ConstrainedLanguageRepresentations
ActivityRepresentations
Interactions
(from UMLMetaModel)
«import»
BasicActivities
(from UMLMetaModel)

Figure 7.1: Overview of packages inside the RequirementRepresentations part of RSL

BasicRepresentations package describes a general way of representing Requirements: every Requirement is a component for it's RequirementRepresentations, which contain sentences (specialisations of HyperlinkedSentence).

- The Natural LanguageRepresentations package contains all the constructs needed to express requirements in natural language. It «import»s from the BasicRepresentations package. In this manner, natural language representations are specialisations of elements in BasicRepresentations. As such the Natural LanguageRepresentations package gives access to the Basic Representations package, the Requirements Specifications package, the Terms package and the Phrases package for any variant of natural language requirement representation being utilised within a requirements specification.

- The ConstrainedLanguageRepresentations package contains all the constructs needed to express requirements in constrained language. It «import»s from the BasicRepresentations package. In this manner constrained language representations can be specified as being specialisations of elements in BasicRepresentations. As such the ConstrainedLanguageRepresentations package gives access to the Basic Representations package, the Requirements Specifications package, the Terms package and the Phrases package for any variant of constrained language requirement representation being utilised within a requirements specification.

- The ActivityRepresentations package contains all the constructs needed to express requirements with diagrams that specialise from UML activity diagrams. It «import»s from the BasicRepresentations package and the UML :: BasicActivities package. In this manner activity requirement representations are specialisations of UML Activity and elements from BasicRepresentations. As such, the ActivityRepresentations package gives access to the BasicRepresentations package, the RequirementsSpecifications package, the Terms package, the Phrases package and the BasicActivities package for any variant of activity diagram based requirement representation being utilised within a requirements specification.

- The InteractionRepresentations package contains all the constructs needed to represent UML 2.0 interaction diagram based requirement representations. It «import»s from the BasicRepresentations package and the UML :: Interactions package. In this manner interaction requirement representations are specialisations of UML Interaction and elements from the BasicRepresentations package. As such the InteractionRepresentations package gives access to the BasicRepresentations package, the Requirements Specifications package, the Terms package, the Phrases package and the Interactions package for any variant of interaction diagram based requirement representation being utilised within a requirements specification.

Individual classes in the above packages can be mapped from[1] conceptual model described in Chapter 4. These mappings are shown in Figure 7.2. The most general class in this part is the RequirementRepresentation class. This class allows for expressing details of requirement representations. Its source is the Requirements Model :: RequirementRepresentation class from the conceptual model. Different specialisations of RequirementRepresentation also trace from relevant classes of the conceptual model, and particularly, the representation hierarchy as shown in Figure 4.1.

Representations found in this package can be expressed through diagrams or in text. Diagrams include Activity Diagrams where ActivityRepresentations can be shown and Sequence Diagrams

[1]«mappedFrom» specifies a relationship between RSL model elements that represent corresponding ideas in the conceptual model.

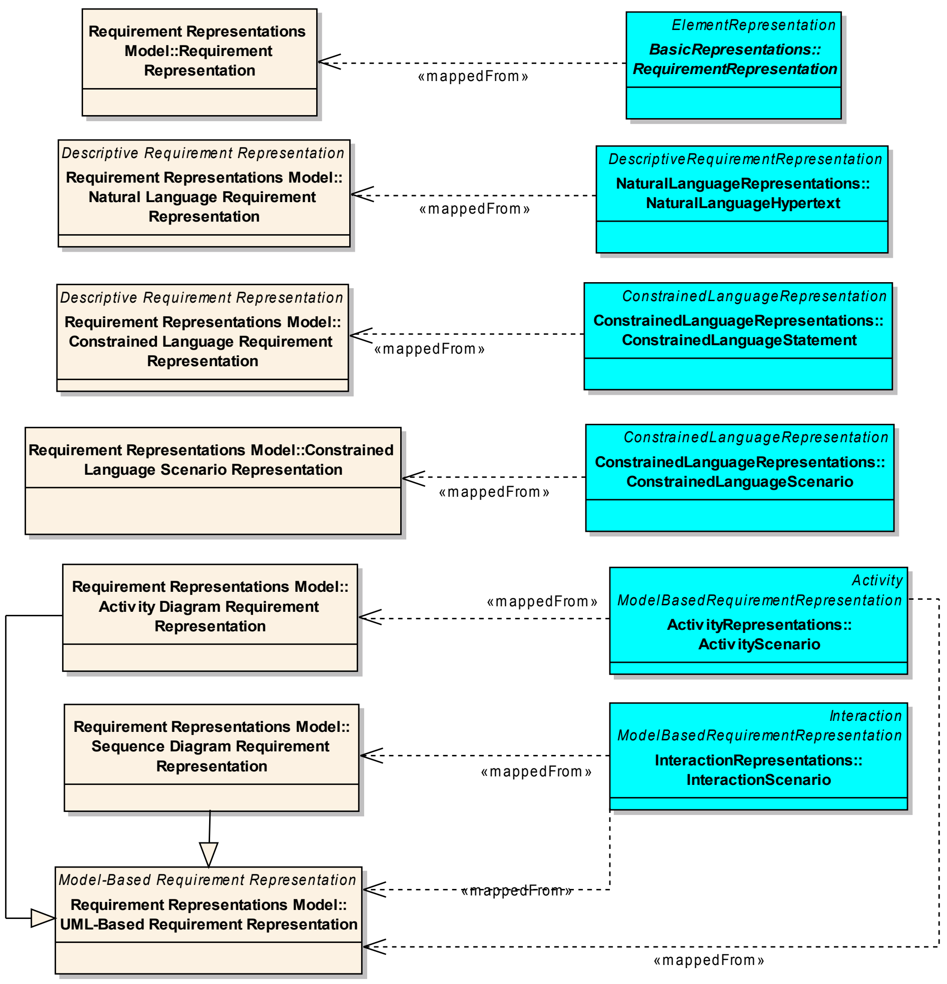

Requirement Representations Model::Requirement Representation
ElementRepresentation
BasicRepresentations::RequirementRepresentation
«mappedFrom»
Descriptive Requirement Representation
Requirement Representations Model::Natural Language Requirement Representation
DescriptiveRequirementRepresentation
NaturalLanguageRepresentations::NaturalLanguageHypertext
«mappedFrom»
Descriptive Requirement Representation
Requirement Representations Model::Constrained Language Requirement Representation
ConstrainedLanguageRepresentation
ConstrainedLanguageRepresentations::ConstrainedLanguageStatement
«mappedFrom»
Requirement Representations Model::Constrained Language Scenario Representation
ConstrainedLanguageRepresentation
ConstrainedLanguageRepresentations::ConstrainedLanguageScenario
«mappedFrom»
Requirement Representations Model::Activity Diagram Requirement Representation
Activity
ModelBasedRequirementRepresentation
ActivityRepresentations::ActivityScenario
«mappedFrom»
Requirement Representations Model::Sequence Diagram Requirement Representation
Interaction
ModelBasedRequirementRepresentation
InteractionRepresentations::InteractionScenario
«mappedFrom»
Model-Based Requirement Representation
Requirement Representations Model::UML-Based Requirement Representation
«mappedFrom»
«mappedFrom»

Figure 7.2: Main classes in the RequirementRepresentations part with mappings to the conceptual model

where InteractionRepresentations can be shown. Concrete syntax of these diagrams derives from the syntax of appropriate UML diagrams. Concrete syntax for textual representations is composed of “source” and “preview” syntax. The first variant allows to represent various elements of representation in purely textual way. The second variant uses also font variations (underlining, bolding, etc.).

7.2 Basic representations

7.2.1 Overview

The BasicRepresentations package contains the most general and abstract constructs of the representation language. On this structure, all the concrete representations are built. Generally, every RequirementRepresentation is part of an appropriate Requirements :: Requirement. It is kind of an ElementRepresentation which contains several 'sentence's in the form of HyperlinkedSentences. Such HyperlinkedSentences can also form 'name's for RepresentableElements (where Requirements are one of them). HyperlinkedSentences may contain Hyperlinks which point to Terms :: Terms or Phrases :: Phrases to be found in the BasicDomainElements :: DomainVocabulary.

7.2.2 Abstract syntax and semantics

Abstract syntax for the BasicRepresentations package is described in Figure 7.3.

RequirementRepresentation

Semantics. Defines the content of a RequirementsSpecifications :: Requirement which should, according to IEEE definition, generally constitute a condition or capability needed by a user to solve a problem or achieve an objective. It also contains a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. This content depends on the concrete representation type that specialises RequirementRepresentation.
Abstract syntax. It is part of every RequirementsSpecifications :: Requirement and forms its 'representations'. It consists of one or more 'sentences' in the form of HyperlinkedSentences derived from ElementRepresentations :: ElementRepresentation. This class is abstract and has several concrete specialisations. This is a kind of BasicRepresentations :: RepresentableElement and it redefines representations for RequirementsSpecifications :: Requirement.

HyperlinkedSentence

Semantics. A single sentence that forms part of a requirement's representation or its name. This sentence might (or might not) reference elements of the domain vocabulary through hyperlinks. The sentence might contain only text or can also be expressed with certain graphical elements.
Abstract syntax. HyperlinkedSentences can be Requirement's 'name' or part of a RequirementRepresentation as its 'sentence'. HyperlinkedSentences contain Hyperlinks.

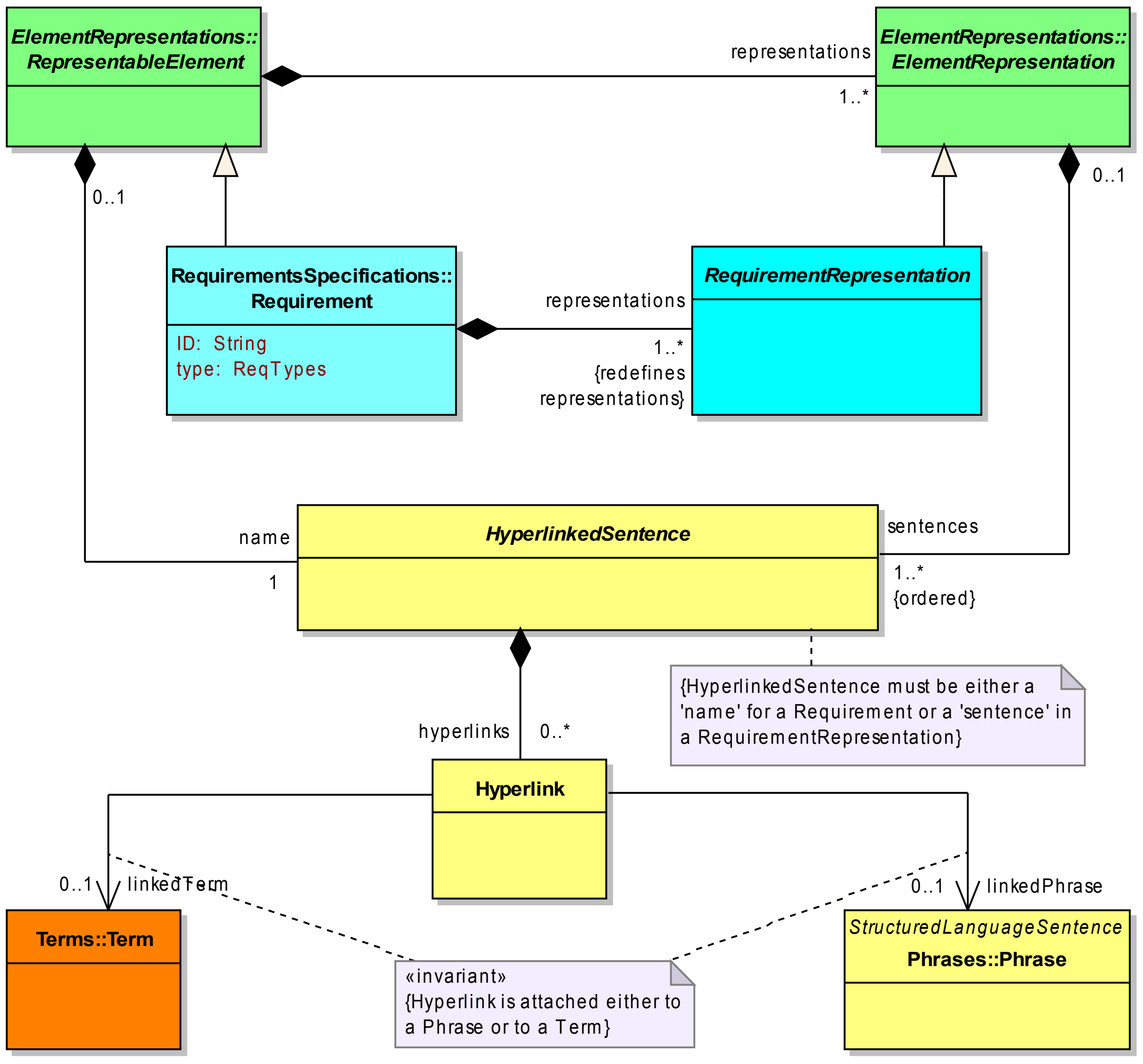

ElementRepresentations::RepresentableElement
representations
1..*
ElementRepresentations::ElementRepresentation
0..1
0..1
RequirementsSpecifications::Requirement
ID: String
type: ReqTypes
representations
1..*
{redefines representations}
RequirementRepresentation
name
1
HyperlinkedSentence
sentences
1..*
{ordered}
{HyperlinkedSentence must be either a 'name' for a Requirement or a 'sentence' in a RequirementRepresentation}
hyperlinks
0..*
Hyperlink
0..1
linkedTerm
0..1
linkedPhrase
Terms::Term
«invariant»
{Hyperlink is attached either to a Phrase or to a Term}
StructuredLanguageSentence
Phrases::Phrase

Figure 7.3: Basic representations

Hyperlink

Semantics. Hyperlink expresses a reference from a requirement representation to an element of the domain vocabulary. By using Hyperlinks, domain vocabulary elements can be used in the content of requirement representations without copying their names, but by pointing to their definitions.

Abstract syntax. Hyperlink can reference either a single Terms :: Term or a single Phrases :: Phrase. It can be part of a HyperlinkedSentence.

7.2.3 Concrete syntax and examples

RequirementRepresentation. As an abstract meta-class, this meta-model element has no concrete syntax. It can be formulated in any of representations of meta-classes that derive from it.

HyperlinkedSentence. As an abstract meta-class, this meta-model element has no concrete syntax. It can be formulated in any of representations of meta-classes that derive from it.

Hyperlink. Its concrete syntax depends on the context in which Hyperlink is presented to the user. It can be represented in the form of a purely textual "source", or in a "preview" form of underlined wiki-like links. In source, Hyperlink consists of a double pair of square brackets ("[[]]") surrounding the hyperlinked text. In preview, Hyperlink is represented as coloured and underlined text (see Figure 7.4).

Source:	Preview:
[[customer]]	<u>customer</u>

Figure 7.4: Hyperlink concrete syntax example

7.3 Requirement representations

7.3.1 Overview

The RequirementRepresentations package describes a way to represent generic requirements for legitimate requirements specifications as specified under traditional software engineering practices. Requirements under a generic context of software development can be represented as RequirementRepresentation (kind of ElementRepresentation) which is a base class for all generic requirement representation types and which is extended by the hierarchical structure of sub-classes as shown in Figure 7.5.

This Figure shows a simple hierarchy of requirements representations that are allowed by the current language. These all specialise the RequirementRepresentation as described in the BasicRepresentations package.

Requirements can be presented in a textual form (DescriptiveRequirementRepresentation). NaturalLanguageRepresentations allow requirements to be represented as NaturalLanguageHyper-

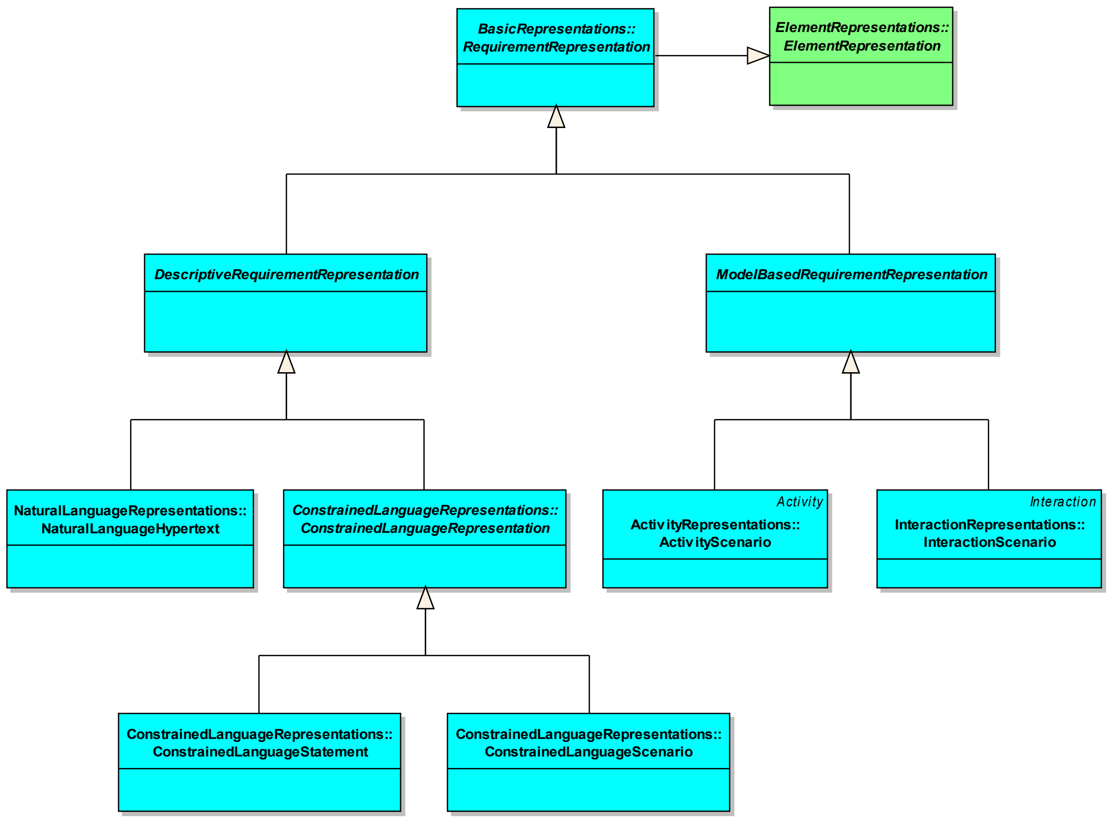

BasicRepresentations::
RequirementRepresentation
ElementRepresentations::
ElementRepresentation
DescriptiveRequirementRepresentation
ModelBasedRequirementRepresentation
NaturalLanguageRepresentations::
NaturalLanguageHypertext
ConstrainedLanguageRepresentations::
ConstrainedLanguageRepresentation
Activity
ActivityRepresentations::
ActivityScenario
Interaction
InteractionRepresentations::
InteractionScenario
ConstrainedLanguageRepresentations::
ConstrainedLanguageStatement
ConstrainedLanguageRepresentations::
ConstrainedLanguageScenario

Figure 7.5: Requirement representations

text. ConstrainedLanguageRepresentations allow requirements to be represented as either a ConstrainedLanguageStatement or a ConstrainedLanguageScenario.

Another representation of requirements is a model-based form (ModelBasedRequirementRepresentation). ActivityRepresentations allow a requirement to be represented as an ActivityScenario. InteractionRepresentations allow a requirement to be represented as an InteractionScenario.

7.3.2 Abstract syntax and semantics

Abstract syntax for the RequirementRepresentations package is described in Figure 7.5.

DescriptiveRequirementRepresentation

Semantics. This meta-class allows for representing requirements in textual way in form of free text or structured text.

Abstract syntax. It is a kind of BasicRepresentations :: RequirementRepresentation. DescriptiveRequirementRepresentation is an abstract class.

ModelBasedRequirementRepresentation

Semantics. This meta-class allows for representing requirements in schematic form.
Abstract syntax. It is a kind of BasicRepresentations :: RequirementRepresentation. ModelBasedRequirementRepresentation is an abstract class.

7.3.3 Concrete syntax and examples

DescriptiveRequirementRepresentation. As an abstract meta-class, this meta-model element has no concrete syntax. However any of meta-classes that specialise from it may have capital letter "D" in their concrete syntaxes, indicating their descriptive character.

ModelBasedRequirementRepresentation. As an abstract meta-class, this meta-model element has no concrete syntax. However any of meta-classes that specialise from it may have capital letter "M" in their concrete syntaxes, indicating their model-based character.

7.4 Natural language representations

7.4.1 Overview

The Natural language representations package describes a way to represent requirements in plain natural language without any formal structure. Sentences in natural language may contain Hyperlinks to build a connection to the domain knowledge in the vocabulary.

7.4.2 Abstract syntax and semantics

The diagram in Figure 7.6 shows the abstract syntax of the NaturalLanguageRepresentations package. The following subsections will describe the classes in this diagram.

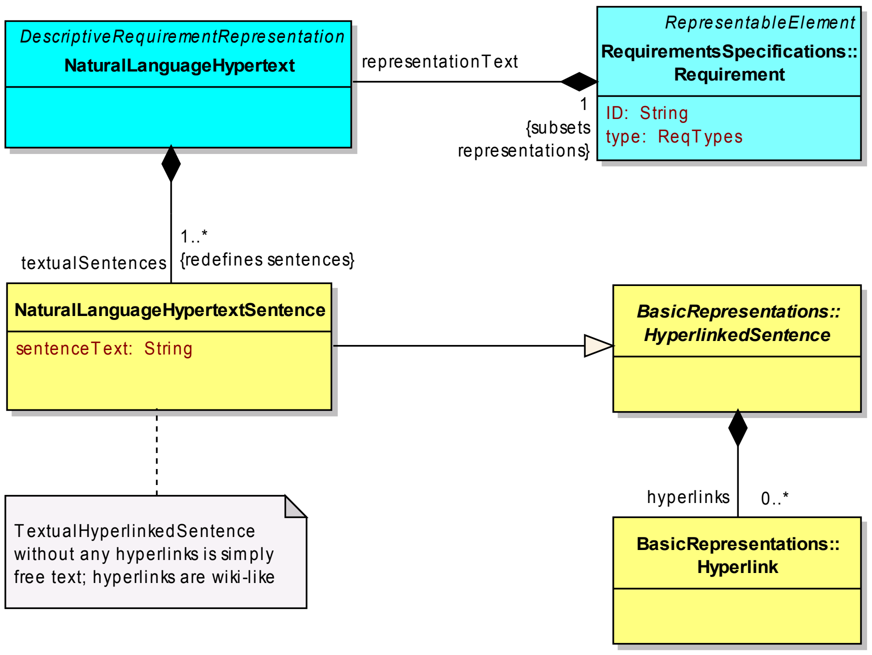

DescriptiveRequirementRepresentation
NaturalLanguageHypertext
representationText
1
{subsets representations}
RepresentableElement
RequirementsSpecifications::Requirement
ID: String
type: ReqTypes
textualSentences
1..*
{redefines sentences}
NaturalLanguageHypertextSentence
sentenceText: String
BasicRepresentations::HyperlinkedSentence
hyperlinks
0..*
BasicRepresentations::Hyperlink
TextualHyperlinkedSentence without any hyperlinks is simply free text; hyperlinks are wiki-like

Figure 7.6: Natural language representations

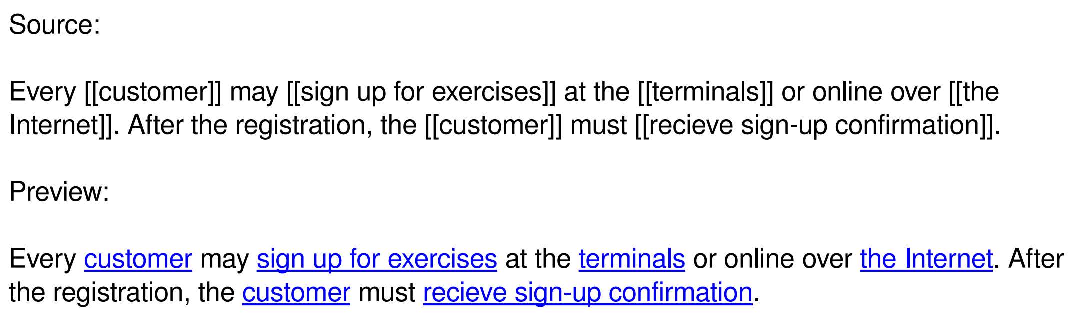
NaturalLanguageHypertext

Semantics. A NaturalLanguageHypertext is the simplest possible representation of a single requirement. The text is written in natural language.
Abstract syntax. A NaturalLanguageHypertext is derived from RequirementRepresentations :: DescriptiveRequirementRepresentation. If it represents a Requirement, its role is representationText. The text consists of one or more NaturalLanguageHypertextSentences.

NaturalLanguageHypertextSentence

Semantics. A NaturalLanguageHypertextSentence is used in a natural language description of a requirement. Using wiki-like hyperlinks in the sentence, a connection to the domain knowledge in the vocabulary is possible. If the sentence does not contain any Hyperlink, it is simply free text.
Abstract syntax. A NaturalLanguageHypertextSentence is part of a NaturalLanguageHypertext, its role is textualSentence. Since NaturalLanguageHypertextSentence is derived from HyperlinkedSentence, it may contain zero or more Hyperlinks. Each of those wiki-like hyperlinks links to a Term or Phrase in the vocabulary.

7.4.3 Concrete syntax and examples



Source:

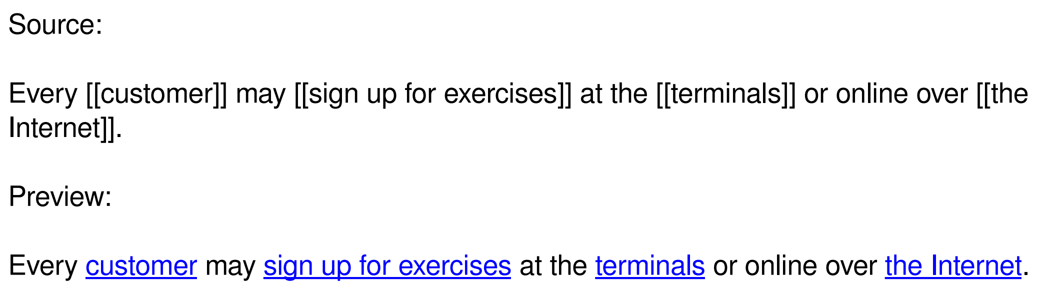
Every [[customer]] may [[sign up for exercises]] at the [[terminals]] or online over [[the Internet]]. After the registration, the [[customer]] must [[recieve sign-up confirmation]].

Preview:

Every customer may sign up for exercises at the terminals or online over the Internet. After the registration, the customer must recieve sign-up confirmation.

Figure 7.7: NaturalLanguageHypertext example

NaturalLanguageHypertext. Figure 7.7 shows an example for the concrete syntax of NaturalLanguageHypertext. The text is composed of several NaturalLanguageHypertextSentences which contain zero or more hyperlinks. The upper sentence shown in the example is the syntax as the requirements engineer will write it down, the lower sentence shows the presentation in the requirements document.



Source:

Every [[customer]] may [[sign up for exercises]] at the [[terminals]] or online over [[the Internet]].

Preview:

Every customer may sign up for exercises at the terminals or online over the Internet.

Figure 7.8: Example for NaturalLanguageHypertextSentence

TextualHypertextSentence. Figure 7.8 shows an example for the concrete syntax of NaturalLanguageHypertextSentence. The second line shows the source of such an sentence a requirements engineer would write. The brackets indicate the hyperlinks, as used in common wiki engines, and this sourcecode will later be displayed as shown in the last line of Figure 7.8. The hyperlinks are highlighted and refer to the phrases “customer”, “sign up for exercises”, “terminal”, and “the internet” in the vocabulary.

7.5 Constrained language representations

7.5.1 Overview

The package ConstrainedLanguageRepresentations allows for representing requirements by using constrained language, i.e. a subset of natural language whose sentences are limited to a certain structure. Refer to chapter 8 and document *D2.2 – Structural Requirements Language Definition* for details on this structure.

7.5.2 Abstract syntax and semantics

Figure 7.9 shows the three classes contained in the package ConstrainedLanguageRepresentations: ConstrainedLanguageRepresentation, ConstrainedLanguageStatement and ConstrainedLanguageScenario.

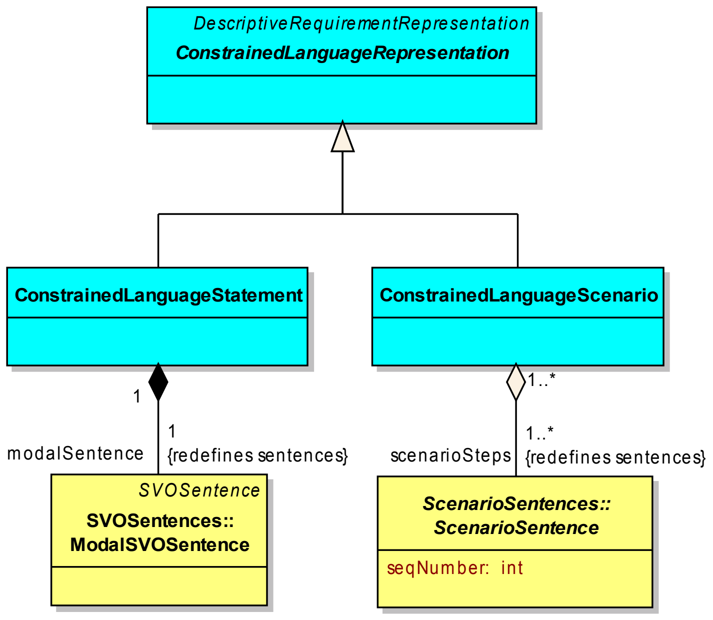


Figure 7.9: ConstrainedLanguageRepresentations

ConstrainedLanguageRepresentation

Semantics. ConstrainedLanguageRepresentation constitutes the description of a requirement by one or more sentences in a constrained language.
Abstract syntax. ConstrainedLanguageRepresentation is a kind of RequirementRepresentations :: DescriptiveRequirementRepresentation.

ConstrainedLanguageStatement

Semantics. This class represents a requirement by a single sentence in a constrained language. This sentence has to contain a modal verb expressing the liability of the requirement. The sentence may contain hyperlinks to phrases or terms in the vocabulary.
Abstract syntax. ConstrainedLanguageStatement is a kind of ConstrainedLanguageRepresentation. It is composed of exactly one single SVOSentences :: ModalSVOSentence in the role of 'modalSentence'.

ConstrainedLanguageScenario

Semantics. ConstrainedLanguageScenario represents a requirement as a scenario and is part of a use case. This scenario consists of a sequence of sentences in constrained language constituting its individual steps.
Abstract syntax. ConstrainedLanguageScenario is a kind of ConstrainedLanguageRepresentation. It is composed of one or more RepresentationSentences :: ScenarioSentences taking the role of 'scenarioSteps'. ConstrainedLanguageScenario is part of a RequirementsSpecifications :: UseCase.

7.5.3 Concrete syntax and examples

ConstrainedLanguageRepresentation. As an abstract meta-class, ConstrainedLanguageRepresentation does not have a concrete syntax. It can be formulated in any of the representations of its subclasses.

ConstrainedLanguageStatement. Figure 7.10 shows an example of the concrete syntax of a ConstrainedLanguageStatement. The *Source* part shows the statement as it is entered by the requirements engineer. The words enclosed in double square brackets denote a hyperlink. The *Preview* part below depicts the statement's presentation in the requirements document. Hyperlinks appear coloured and underlined.

ConstrainedLanguageScenario. An example for a ConstrainedLanguageScenario can be taken from Figure 7.11. The left hand side, the *Source* side, displays the sequence of ScenarioSentences as it is entered by the requirements engineer. The words enclosed in double square brackets denote a hyperlink. On the right hand side, the result in the requirements document is shown. Hyperlinks appear coloured and underlined.

Source:

[[Fitness Club]] should [[provide]] [[bracelets]].

Preview:

Fitness Club should provide bracelets.

Figure 7.10: Example of a ConstrainedLanguageStatement

Source	Preview
1. pre: [[Customer]] is not [[registered]].	1. pre: Customer is not registered.
2. [[Customer]] [[submits]] [[personal information]].	2. Customer submits personal information.
3. ==> cond: [[Receptionist]] is [[logged in]].	3. →cond: Receptionist is logged in.
4. [[Receptionist]] [[registers]] [[customer]].	4. Receptionist registers customer.
5. [[Receptionist]] [[verifies]] [[personal information]].	5. Receptionist verifies personal information.
6. [[Receptionist]] [[issues]] [[customer card]].	6. Receptionist issues customer card.
7. [[Receptionist]] [[prints]] [[personal information]].	7. Receptionist prints personal information.
8. [[Receptionist]] [[gives]] [[customer card]] to [[customer]].	8. Receptionist gives customer card to customer.
9. post: [[Customer]] is [[registered]].	9. post: Customer is registered.

Figure 7.11: Example of a ConstrainedLanguageScenario

The above example also includes ScenarioSentences :: ControlSentences (see lines one and nine) and ScenarioSentence :: ConditionSentence (line three). They are described in more detail in section 8.4.

7.6 Activity representations

7.6.1 Overview

Activity representations package describes ActivityScenario as an alternative way of representing UseCase scenarios. Such representation emphasises flow of control between scenarios within a UseCase in the form of an Activity.

7.6.2 Abstract syntax and semantics

Abstract syntax for this package is shown in Figure 7.12.

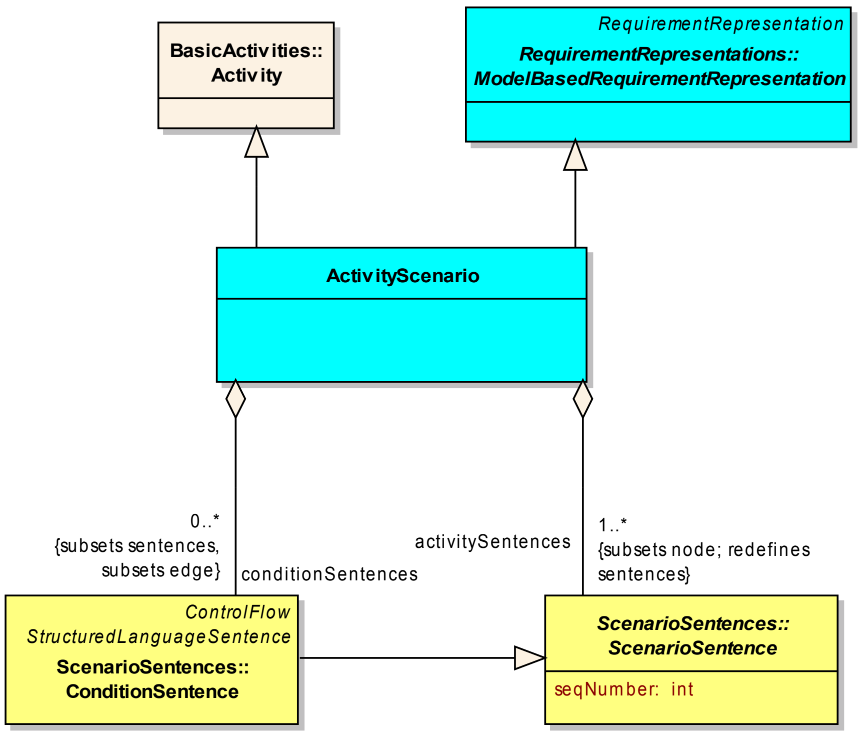


Figure 7.12: Activity representations

ActivityScenario

Semantics. An ActivityScenario is an alternative to ConstrainedLanguageScenario as a way of representing a UseCase's content. In this representation, UseCase scenarios are represented in the form activities. Beside showing the sequence of ScenarioSentence (a scenario), it also represents in a graphical way the flow of control between different scenarios within one UseCase.
Abstract syntax. An ActivityScenario is a kind of RequirementsRepresentations :: ModelBasedRequirementRepresentation. It also specialises BasicActivities :: Activity out of the UML2.0 superstructure. ActivityScenario contains zero or more ConditionSentences's which subsets 'edge' from Activity superclass and one or more activitySentences (kind of ScenarioSentence) which subset 'node'. Both classes redefine and subset sentences from the BasicRepresentations :: RequirementRepresentation superclass.

7.6.3 Concrete syntax and examples

ActivityScenario. Figure 7.13 shows an example of the concrete syntax of an ActivityScenario. The notation for an activity is a combination of the notations of the nodes and edges it contains (just like the notation of UML's BasicActivities :: Activity). For more details please refer to section 8.4.

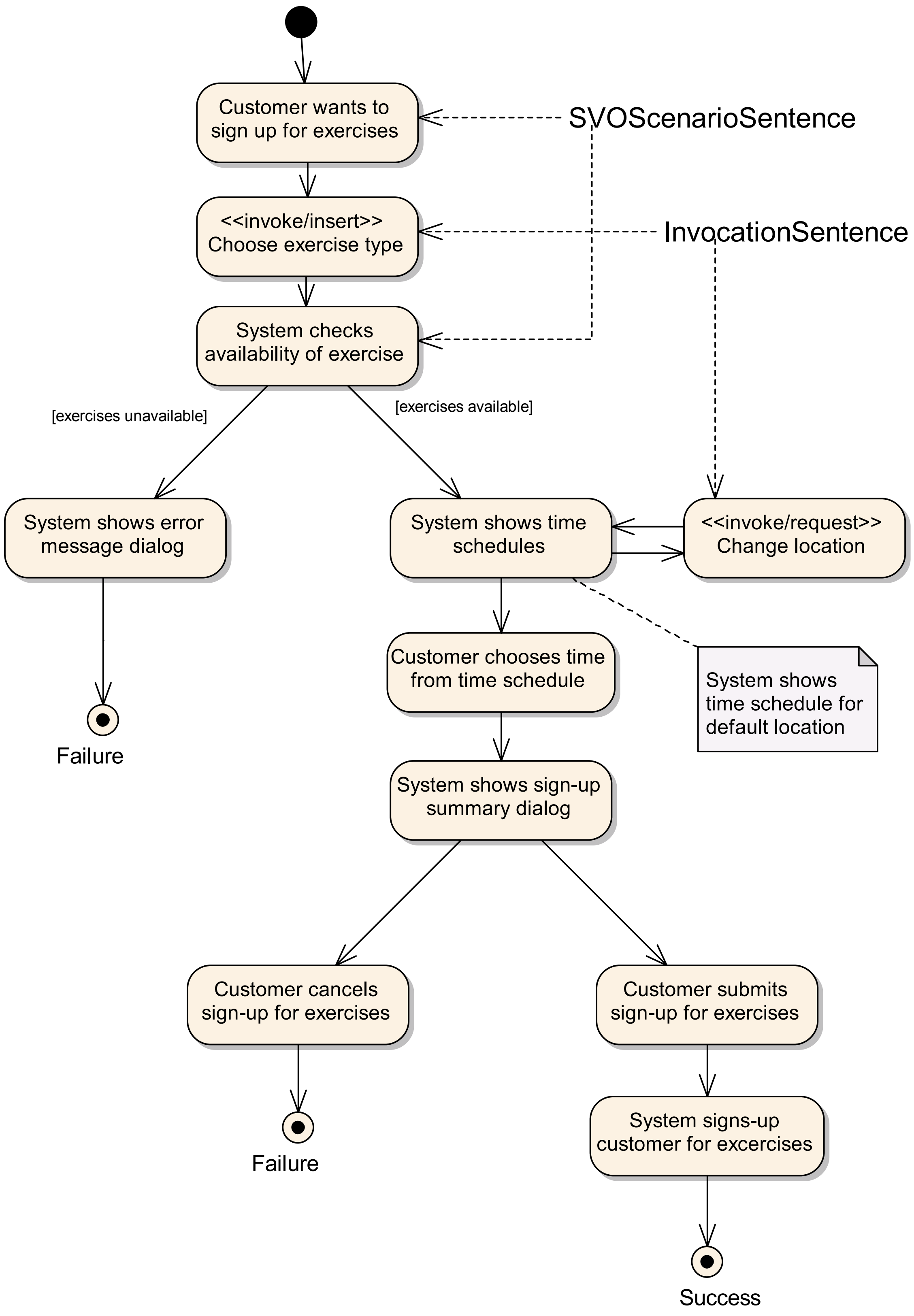

Customer wants to sign up for exercises
SVOScenarioSentence
<<invoke/insert>> Choose exercise type
InvocationSentence
System checks availability of exercise
[exercises unavailable]
[exercises available]
System shows error message dialog
System shows time schedules
<<invoke/request>> Change location
Customer chooses time from time schedule
System shows time schedule for default location
Failure
System shows sign-up summary dialog
Customer cancels sign-up for exercises
Customer submits sign-up for exercises
System signs-up customer for excercises
Failure
Success

Figure 7.13: ActivityScenario example

7.7 Interaction representations

7.7.1 Overview

This section introduces the language elements used to describe scenarios with interaction diagrams. The InteractionScenario is a representation for scenarios based upon the UML 2.0 interaction diagram.

7.7.2 Abstract syntax and semantics

The diagrams in Figure 7.14 and 7.15 describe the abstract syntax of the interaction diagrams that can be used to describe requirements in ReDSeeDS. The following subsections explain the several classes displayed in this diagrams.

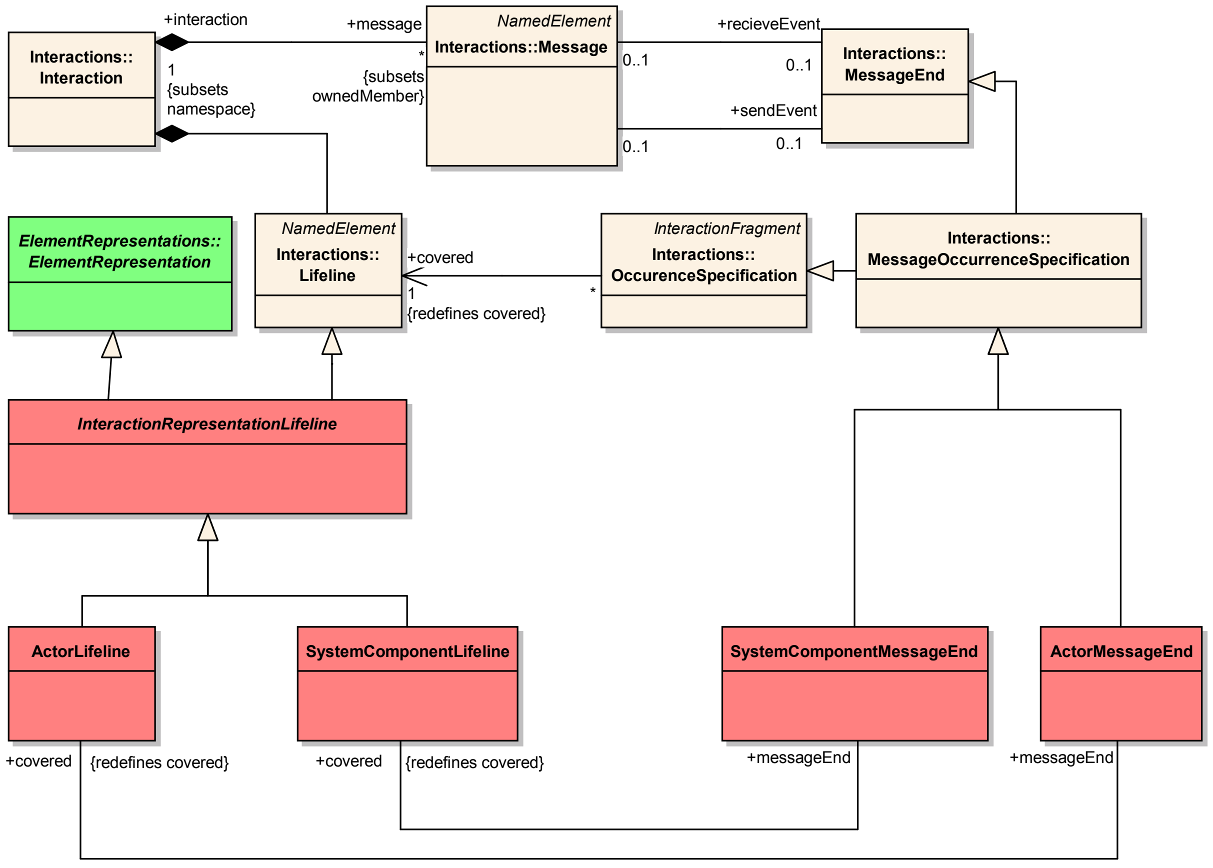


Figure 7.14: Interaction representation, relation to UML superstructure

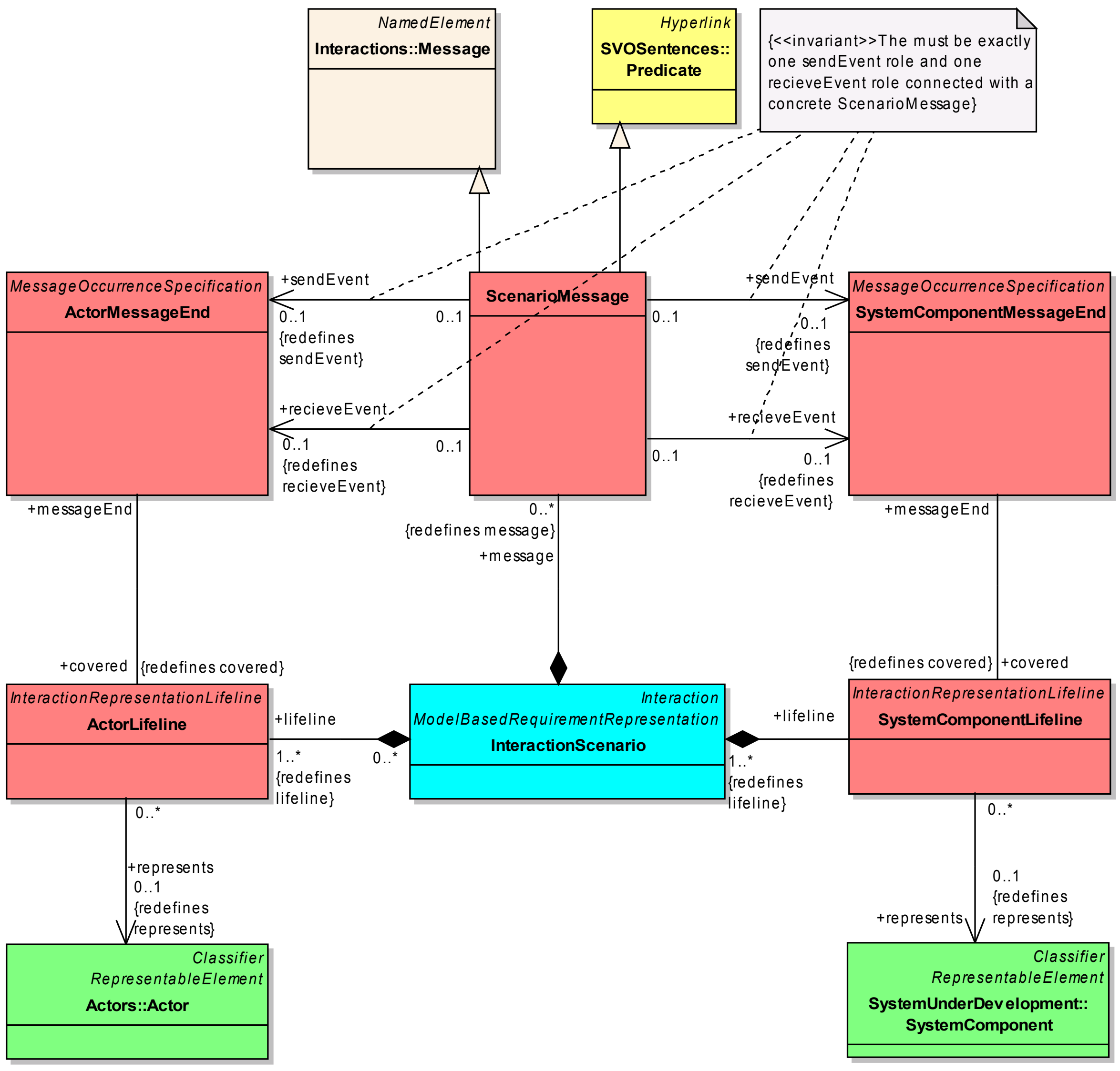


Figure 7.15: Interaction representation, lifelines and messages

InteractionScenario

Semantics. An InteractionScenario is one possible way to describe a scenario in a use case that represents a requirement. It contains at least one actor and at least one component of the system under development. Those are modeled as lifelines. Messages between different lifelines may contain hyperlinks that refer to phrases and terms in the vocabulary.

Abstract syntax. The base classes of InteractionScenario are the classes Interaction out of the UML2.0 superstructure and ModelBasedRequirementRepresentations. While interaction may contain Lifelines, a InteractionScenario may contain only ActorLifelines and SystemComponentLifelines. Since a InteractionScenario is used to describe scenarios in use cases, which consist of interactions between actors and system components, at least one ActorLifeline and and least one SystemComponentLifeline must be present. The example in section 7.7.3 shows why it is not sufficient to restrict the number of possible SystemComponentLifelines to one. The aggregations to these lifelines redefine the original aggregation between Interaction and Lifeline. In the same way the aggregation between Interaction and Message from the UML superstructure is redefined through the aggregation between InteractionScenario and ScenarioMessage, because an InteractionScenario may not contain any kind of Messages but only ScenarioMessages.

InteractionRepresentationLifeline

Semantics. Lifelines in an InteractionScenario are always InteractionScenarioLifelines. They represent actors or components of the system under development. Communication between lifelines can be modeled with ScenarioMessages.
Abstract syntax. An InteractionRepresentationLifeline is the abstract base class for ActorLifeline and SystemComponentLifeline. It is derived from the class Lifeline out of the UML2.0 superstructure and the class ElementRepresentation out of the package ElementRepresentations.

ActorLifeline

Semantics. Every actor who participates in a scenario described by a InteractionScenario is represented by an ActorLifeline. Since the scenario models an interaction between an actor and parts of the system, every ActorLifeline can have some outgoing and incoming messages.
Abstract syntax. The base class of ActorLifeline is the abstract class InteractionRepresentationLifeline. An ActorLifeline may cover zero or more ActorMessageEnds, this association redefines the inherited association between Lifeline and MessageEnd. Every ActorLifeline belongs to a InteractionScenario, as the aggregation between these two classes indicates.

SystemComponentLifeline

Semantics. While ActorLifelines represent actors who participate in a scenario described by an InteractionScenario, a SystemComponentLifeline represents a component of the system under development which participates in a scenario. Depending on the level of granularity the requirements engineer chose, the whole system can be modeled as one single component. Also communications between different system components or between a system component and an actor are represented as outgoing and incoming messages.

Abstract syntax. SystemComponentLifeline is derived from the abstract class InteractionRepresentationLifeline. It may cover SystemComponentMessageEnds, the association describing this fact redefines the inherited association between Lifeline and MessageEnd in almost the same manner as it is for ActorLifeline.

ActorMessageEnd

Semantics. A ActorMessageEnd models the connection between a ScenarioMessage and a ActorLifeline. Every ScenarioMessage may contain zero to two ActorMessageEnds, depending on what kind of communication is modelled with that message.
Abstract syntax. The base class of ActorMessageEnd is the class MessageEnd out of the UML2.0 Superstructure. From this class the associations to Message are inherited, but because ActorMessageEnds may only be endpoints of ScenarioMessages, the associations with the roles sendEvent and recieveEvent are redefined. The constraint which holds for ActorMessageEnd, that also affects SystemComponentMessage, is described later in context of the class ScenarioMessage. The last class associated to ActorMessageEnd is ActorLifeline, the association between these two classes is inherited from Message and Lifeline, but since ActorLifelines should be the only lifelines that may cover an ActorMessageEnd the association is redefined in the metamodel.

SystemComponentMessageEnd

Semantics. Analogous to ActorMessageEnd, SystemComponentMessageEnd models the connection between a ScenarioMessage and a SystemComponentLifeline. In contrast to ActorMessageEnd, it is also possible to have two SystemComponentMessageEnds at one ScenarioMessage, because it should be possible to model communications between different system components in a scenario description. An example for this is shown in section 7.7.3.
Abstract syntax. The base class of SystemComponentMessageEnd is the class MessageEnd out of the UML2.0 superstructure. Again analogous to ActorMessageEnd, the associations to Message are inherited from this class. Since SystemComponentMessageEnds may only be endpoints of ScenarioMessages, the associations with the roles sendEvent and recieveEvent are redefined here also. The same holds for the association to SystemComponentLifeline, these associations are inherited from MessageEnd but must be redefined so that SystemComponentMessageEnds may be covered only by SystemComponentLifelines.

ScenarioMessage

Semantics. A ScenarioMessage represents an interaction between an actors and components of the system under development. It may contain a hyperlink which refers to elements in the vocabulary. As common in UML2.0, ScenarioMessages are verbal phrases.
Abstract syntax. A ScenarioMessage is derived from the class Predicate and from the class Message out of the UML 2.0 metamodel. Every ScenarioMessage belongs to a InteractionScenario, as the redefined aggregation with rolename message indicates.
ScenarioMessage has associations to SystemComponentMessageEnd and ActorMessageEnd, these are described in detail at the class descriptions of SystemComponentMessageEnd and ActorMessageEnd. To ensure that ScenarioMessage is associated to exactly one message end with rolename sendEvent and exactly one with rolename recieveEvent a constraint is added.

7.7.3 Concrete syntax and examples

The Figures 7.16 and 7.17 describe the concrete syntax of the interaction diagram that can be used to describe requirements in the RSL. The first Figure shows a sequence diagram as one possible form of interaction diagram, the second Figure shows the other possible form, the communication diagram. The following sections explain the concrete syntax of the different classes mentioned in the section above.

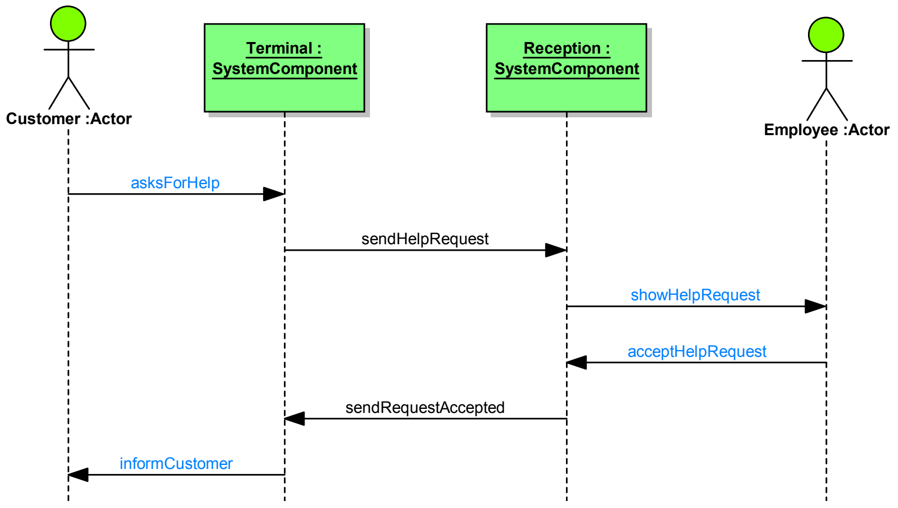


Figure 7.16: Interaction representation with sequence diagram

InteractionScenario. Both diagrams shown in the Figures 7.16 and 7.17 show the same InteractionScenario. Concrete syntax of specific elements of InteractionScenario are described below.

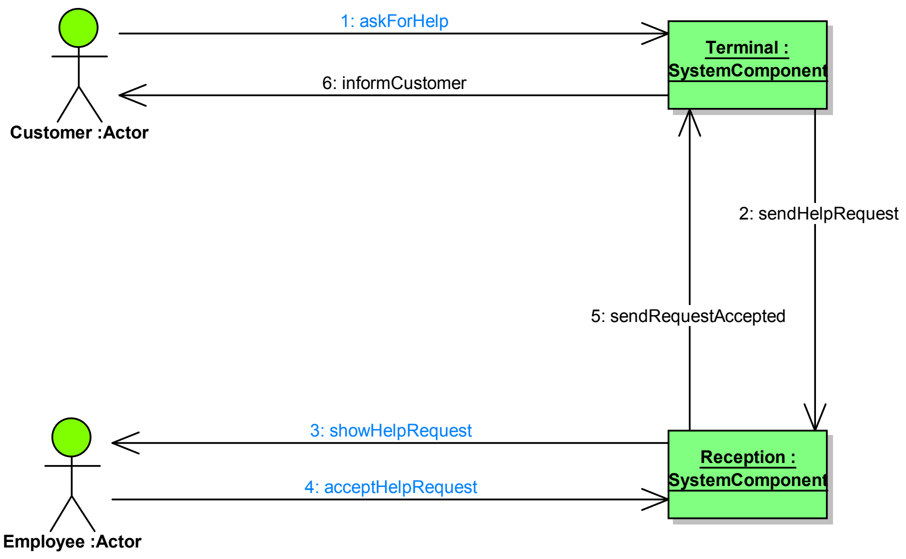


Figure 7.17: Interaction representation with communication diagram

InteractionRepresentationLifeline. Since this class is abstract, it has no concrete syntax.

ActorLifeline. The outer left and the outer right lifelines in Figure 7.16 are examples for ActorLifelines. As common in UML, they are modelled as dotted lines. Each of them starts at an Actor, which is represented as it is common in UML sequence and use case diagrams. In Figure 7.17 the lifelines are not explicitly modelled. Here the actors itself takes over the role of their lifeline.

SystemComponentLifeline. The two lines in the sequence diagram's centre are SystemComponentLifelines. They start at an instance of SystemComponent and are modelled as dotted lines. Again, in the communication diagram the lifelines are not explicitly modelled.

ActorMessageEnd. The connections between the ScenarioMessages and the ActorLifelines are ActorMessageEnds. As it is common in UML sequence diagrams, the message whose end is modelled just starts or ends at the appropriate lifeline. If the ActorMessageEnd has the role of sendEvent, no additional graphical element is needed, if it has the role of receiveEvent the connection to the lifeline is modelled as a black arrow, as it is common in UML. By analogy to this, the messages in the communication diagram are represented as it is common in UML.

SystemComponentMessageEnd. The graphical representation of SystemComponentMessageEnd is analogous to ActorMessageEnd, just it connects not to ActorLifelines but to SystemComponentLifelines.

ScenarioMessage. All messages in the diagrams are instances of ScenarioMessage. They are modelled as black lines between two different InteractionRepresentationLifelines. The messages in the centre of Figure 7.16, which connect the SystemComponents Terminal and Reception, are ScenarioMessages but do not contain a Hyperlink. In this diagram, all messages between an Actor and a SystemComponent contain a Hyperlink, as the blue font colour indicated. The whole blue part in the message name is the Hyperlink, it refers to the appropriate Phrase in the vocabulary. The points where a ScenarioMessage and a InteractionRepresentationLifeline meet are ActorMessageEnds or SystemComponentMessageEnds, their notation is explained at the description of the appropriate classes.

The distribution of Hyperlinks in this example is not representative, there is no general restriction for a message between an Actor and a SystemComponent to contain a Hyperlink neither a restriction for other messages to contain no Hyperlinks.

7.8 Use case representations

7.8.1 Overview

This package does not contain any meta-classes. However, it introduces several meta-associations defining the structure of representations of content for RequirementsSpecifications :: UseCases.

RequirementsSpecifications :: UseCase meta-class is a kind of RequirementsSpecifications :: Requirement and also inherits from UML's UseCase :: UseCase. It's content can be expressed through three different perspectives:

- ConstraintLanguageRepresentation :: ConstraintLanguageScenario – textual representation of UseCase's scenarios
- ActivityRepresentation :: ActivityScenario – adds graphical representation of control flow between different scenarios of a single UseCase
- InteractionRepresentation :: InteractionScenario – emphasises the aspect of interaction between a system and its users by showing a sequence of messages sent between them

7.8.2 Abstract syntax and semantics

The abstract syntax in this package is presented in Figure 7.18. UseCase is a special kind of Requirement that can have its content represented through three RequirementRepresentations.

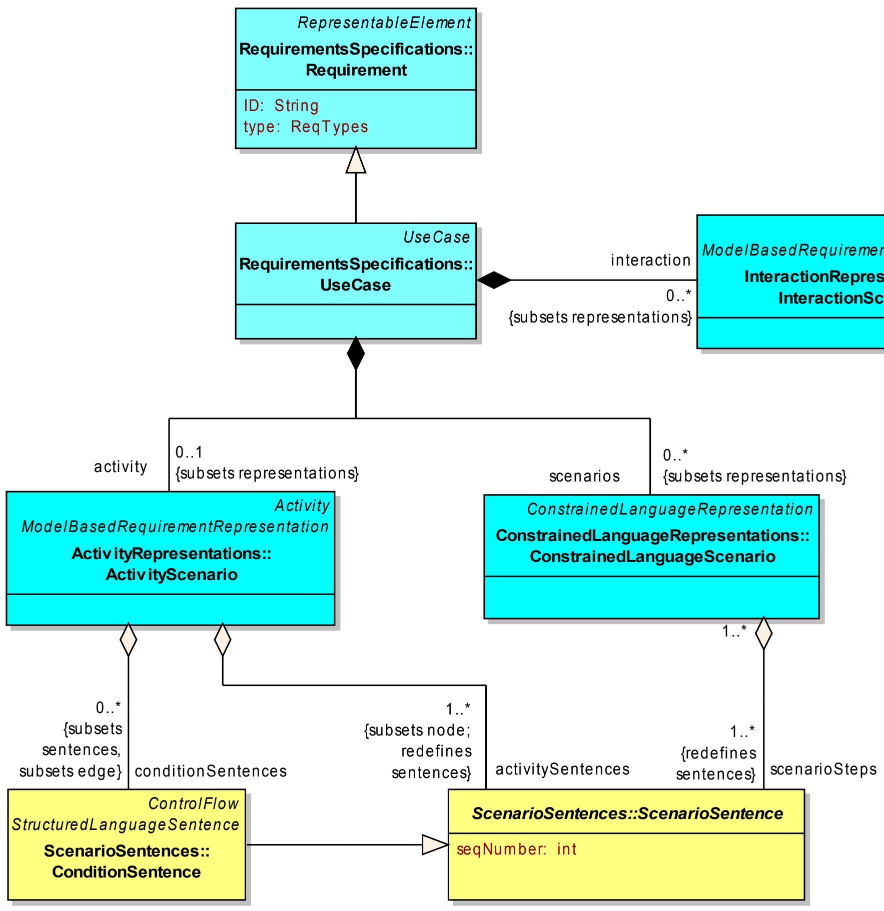
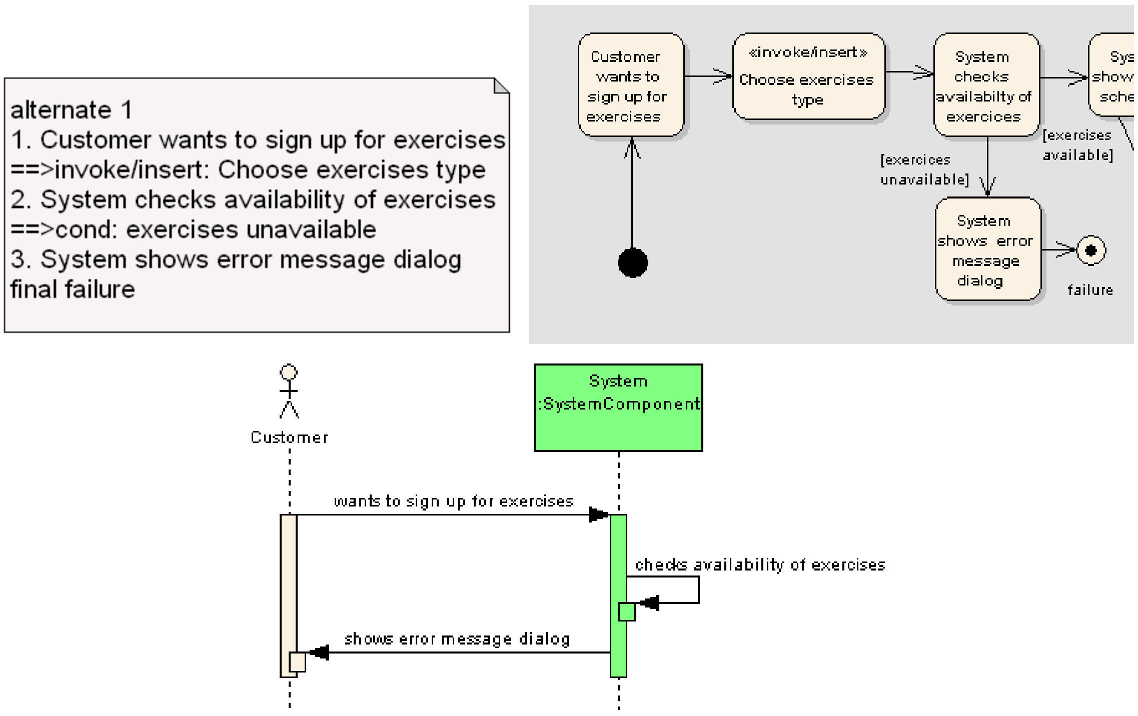

RepresentableElement
RequirementsSpecifications::Requirement
ID: String
type: ReqTypes
UseCase
RequirementsSpecifications::UseCase
interaction
0..*
{subsets representations}
Interaction
ModelBasedRequirementRepresentation
InteractionRepresentations::InteractionScenario
activity
0..1
{subsets representations}
scenarios
0..*
{subsets representations}
Activity
ModelBasedRequirementRepresentation
ActivityRepresentations::ActivityScenario
ConstrainedLanguageRepresentation
ConstrainedLanguageRepresentations::ConstrainedLanguageScenario
1..*
0..*
{subsets sentences, subsets edge}
conditionSentences
1..*
{subsets node; redefines sentences}
activitySentences
1..*
{redefines sentences}
scenarioSteps
ControlFlow
StructuredLanguageSentence
ScenarioSentences::ConditionSentence
ScenarioSentences::ScenarioSentence
seqNumber: int

Figure 7.18: UseCase representations

Two of them are ModelBasedRequirementRepresentations and one of them is a ConstrainedLanguageRequirementRepresentation. All the three representations of the UseCase content (ConstrainedLanguageScenario,ActivityScenario and InteractionScenario) are described in detail in sections 7.5, 7.6 and 7.7 respectively. It has to be noted that ActivityScenarios and ConstrainedLanguageScenarios share the same ScenarioSentences including ConditionSentences.

7.8.3 Concrete syntax and examples

Figure 7.19 shows three different representations of content of a UseCase. This diagram compares alternative notations for the contents of UseCases. Details of concrete syntax for the three alternative notations is given in sections 7.5, 7.6 and 7.7.


alternate 1
1. Customer wants to sign up for exercises
==>invoke/insert: Choose exercises type
2. System checks availability of exercises
==>cond: exercises unavailable
3. System shows error message dialog
final failure

Customer wants to sign up for exercises
«invoke/insert» Choose exercises type
System checks availabilty of exercices
Sys show sche
[exercises available]
[exercices unavailable]
System shows error message dialog
failure
Customer
System :SystemComponent
wants to sign up for exercises
checks availability of exercises
shows error message dialog

Figure 7.19: The same scenario in three different representations: ConstrainedLanguageScenario, ActivityScenario and InteractionScenario

Chapter 8

Requirement Representation Sentences

8.1 Overview

This chapter covers the different types of sentences in constrained language which can be used to represent requirements. Figure 8.1 gives an overview of the three packages (marked in yellow) inside this part of the Requirements Specification Language.

- The RepresentationSentences package only contains the abstract class StructuredLanguageSentence representing a single sentence in constrained language. In order to distinguish between different types of sentences, this class is further specialized in the other two packages described below.
- Inside the SVOSentences[1] package, there exist constructs for representing sentences and their breakdown into more fine-granular elements, such as Subject or Predicate.
- The contents of the ScenarioSentences package which imports SVOSentences are used for describing a sentence of a scenario. They differ from an "ordinary" SVOSentence by containing a sequence number denoting their position inside the scenario. The subtypes of a ScenarioSentence allow for describing a single scenario step as well as for expressing the control flow of a scenario's execution.

[1]*SVO* stands for *subject – verb – object*. The identifier refers to the SVO(O) (subject – verb – object – (object)) grammar used for the constrained language.

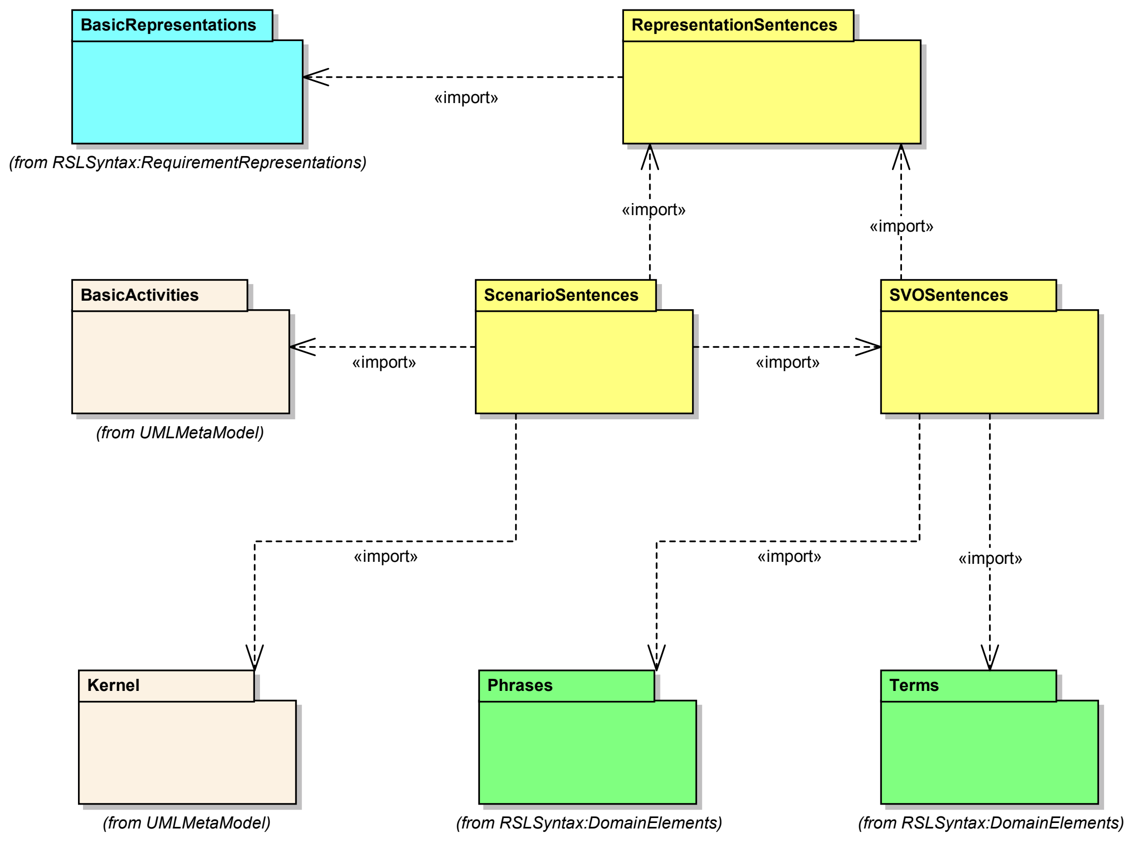


Figure 8.1: Overview of packages inside the RequirementRepresentationSentences part of RSL

8.2 Representation sentences

8.2.1 Overview

This section introduces sentences written in structured language which may be used to describe requirements.

8.2.2 Abstract syntax and semantics

Figure 8.2 shows the abstract syntax of the RepresentationSentences package. Specific meta-classes are described in the sections below.

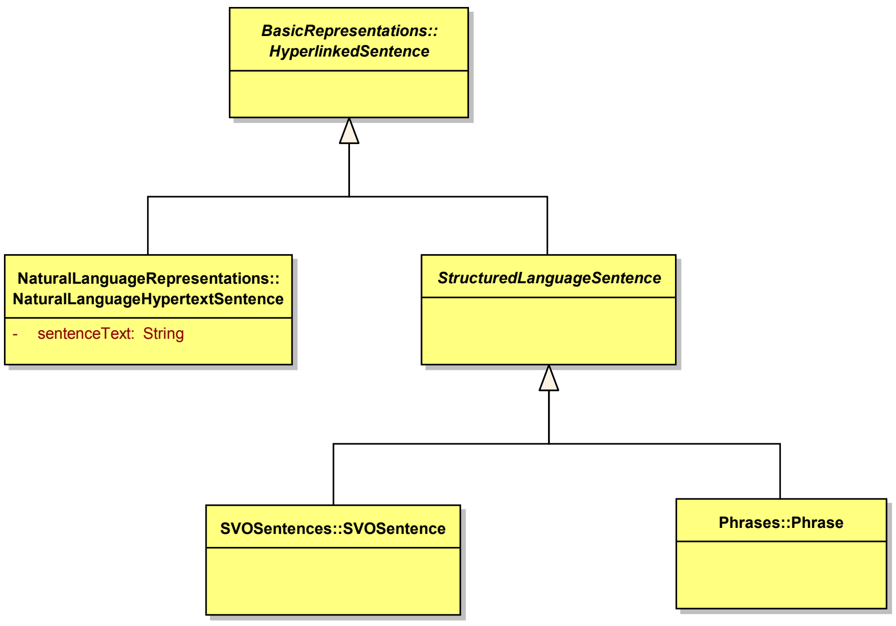

BasicRepresentations::
HyperlinkedSentence
NaturalLanguageRepresentations::
NaturalLanguageHypertextSentence
- sentenceText: String
StructuredLanguageSentence
SVOSentences::SVOSentence
Phrases::Phrase

Figure 8.2: Representation Sentences

StructuredLanguageSentence

Semantics. Structured language is a natural language which is structured by some restrictions. Every type of structured language sentence which is used in the RSL meta-model is a specialisation of this class. A more detailed explanation of the different kinds of structured language sentences used in the RSL can be found in the sections below. In addition to its specific structure, the StructuredLanguageSentence contains zero or more Hyperlinks.
Abstract syntax. The StructuredLanguageSentence is an abstract base class for all other sentences that use structured language. These are SVOSentence (see section 8.3) and Phrase, which are stored in the vocabulary. StructuredLanguageSentene itself is derived from HypertextSentence, so it may contain Hyperlinks.

8.2.3 Concrete syntax and examples

StructuredLanguageSentence. Since StructuredLanguageSentence is an abstract meta-class, there is no concrete syntax for this specific class.

8.3 SVO sentences

8.3.1 Overview

This package describes meta-models for two kinds of simple grammar sentences used for expressing individual requirements.

8.3.2 Abstract syntax and semantics

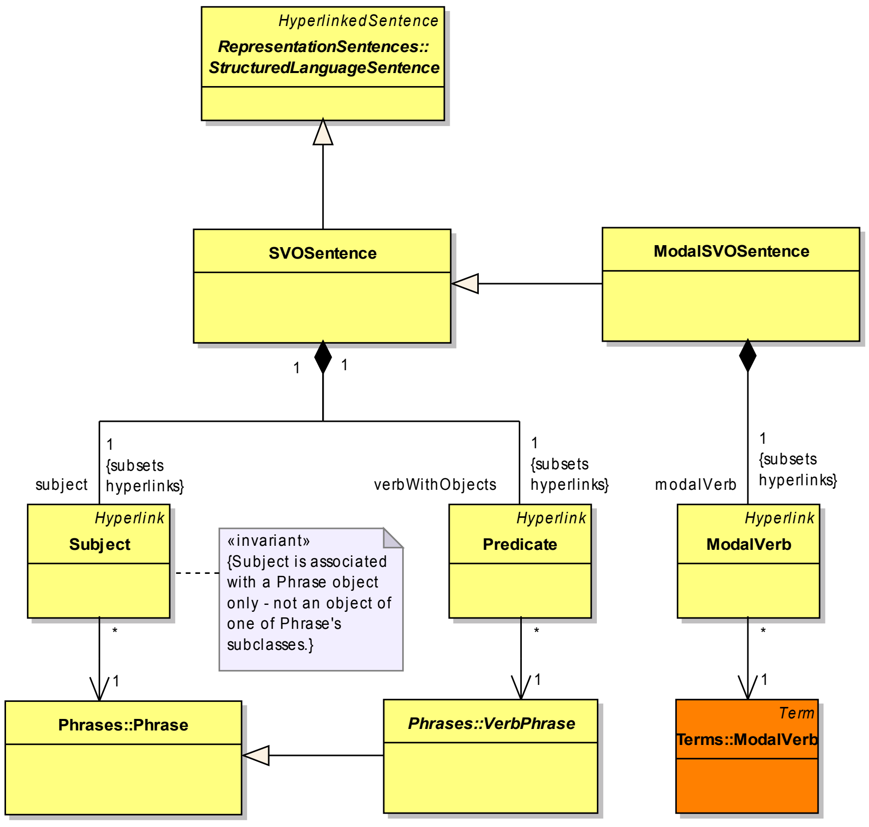


Figure 8.3: SVO Sentences

Figure 8.3 shows the part of the RSL meta-model which deals with the content of the SVOSentences package. The meta-classes in this package are described in detail below.

SVOSentence

Semantics. Represents a sentence in simple SVO(O) [2] grammar, where VO(O) part is represented by Predicate pointing to a Phrases :: VerbPhrase.
Abstract syntax. SVOSentence is a kind of RepresentationSentences :: StructuredLanguage-Sentence. It has one Subject and one Predicate (in the role of a verb with objects). Both are Hyperlinks.

ModalSVOSentence

Semantics. ModalSVOSentence is a SVOSentence extended with ModalVerb allowing to express: 1) priority of the described activity, 2) modality of the described activity, 3) obligation or possibility of the subject to perform an action (described by a Predicate)
Abstract syntax. It is a kind of SVOSentence with an additional Hyperlink being a ModalVerb.

Subject

Semantics. Subject denotes the part of an SVOSentence being its subject from the point of view of natural language grammar. Subject points to a Phrases :: Phrase that is associated with an Actors :: Actor or SystemRepresentations :: SystemUnderDevelopment (see document D2.2). This element can perform an action described by the predicate of the SVOSentence.
Abstract syntax. Subject is kind of BasicRepresentations :: Hyperlink that in a context of an SVOSentence subsets the 'hyperlink' being part of a BasicRepresentations :: HyperlinkedSentence. It is thus part of an SVOSentence and points to a Phrases :: Phrase. The Phrase that is associated with the Subject cannot be one of Phrase's subclasses. The Phrases :: Phrase has to belong to an Actors :: Actor or SystemReprsentations :: SystemUnderDevelopment.

Predicate

Semantics. Predicate hyperlinks an action performed by a Subject and all the words governed by this action's Phrases :: VerbPhrase or modifying it in a given SVOSentence.
Abstract syntax. Predicate is kind of BasicRepresentations :: Hyperlink that in a context of SVOSentence subsets 'hyperlink' being part of BasicRepresentations :: HyperlinkedSentence.

[2] Subject – Verb – Object – (Object)

It is thus part of an SVOSentence and points to a Phrases :: VerbPhrase. The VerbPhrase that is associated with the Predicate must be either Phrases :: SimpleVerbPhrase or Phrases :: ComplexVerbPhrase.

ModalVerb

Semantics. ModalVerb is an additional element of ModalSVOSentence. It allows for expressing modality, priority, obligation and/or possibility of action performed by the Subject of the sentence.
Abstract syntax. ModalVerb is kind of BasicRepresentations :: Hyperlink that in a context of ModalSVOSentence subsets 'hyperlink' being part of BasicRepresentations :: HyperlinkedSentence. It points to a Words :: ModalVerb (see document D2.2).

8.3.3 Concrete syntax and examples

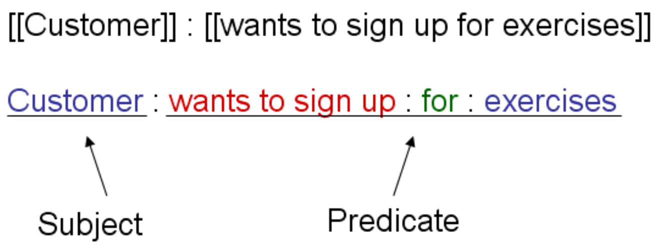

[[Customer]] : [[wants to sign up for exercises]]
Customer : wants to sign up : for : exercises
Subject
Predicate

Figure 8.4: SVOSentence concrete syntax example

SVOSentence. Its concrete syntax depends on the context in which the particular SVOSentence is presented to the user. It can be represented in a source form or preview form, where hyperlinks are presented as in a Wiki. In the source form, SVOSentence consists of a hyperlink to a Phrases :: Phrase (the Subject), a colon (":") and a hyperlink to a Phrases :: VerbPhrase (the Predicate). In the preview form, the SVOSentence is represented as a set of coloured hyperlinks separated with colons (see Figure 8.4).

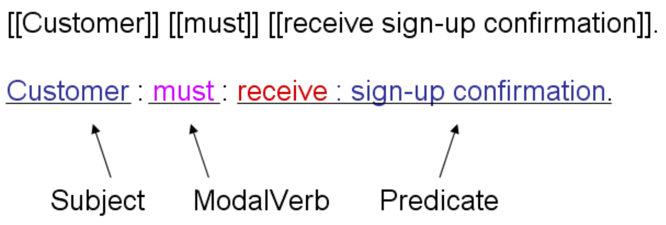

[[Customer]] [[must]] [[receive sign-up confirmation]].
Customer : must : receive : sign-up confirmation.
Subject
ModalVerb
Predicate

Figure 8.5: ModalSVOSentence concrete syntax example

ModalSVOSentence. Its concrete syntax is analogous to the SVOSentence's concrete syntax, with addition of a ModalVerb between a Subject and a Predicate (see Figure 8.5).

Subject. Predicate. ModalVerb. Their concrete syntax is not changed in respect to that of the BasicRepresentations :: Hyperlink meta-class.

8.4 Scenario sentences

8.4.1 Overview

This package describes scenario sentences. It contains SVOScenarioSentence as a sentence in the SVO grammar and ControlFlowSentence determining the flow of control in a scenario. ControlFlowSentence has three specialised concrete classes: InvocationSentence, Precondition-Sentence, PostconditionSentence.

8.4.2 Abstract syntax and semantics

Figure 8.6 shows part of the RSL meta-model which deals with the content of the ScenarioSentences package. The classes in this package are described in detail below.

ScenarioSentence

Semantics. A ScenarioSentence is a sentence which can be used in a scenario. To use sentence types which do not specialise from the ScenarioSentence in a scenario description is not possible since the sentences in a scenario description must have an order. Since the sentences in a scenario description may have different purposes, the ScenarioSentence is just the base for several more specialised sentence types.
Abstract syntax. ScenarioSentence is an abstract class and is a base for all the scenario sentences. It includes an attribute called seqNumber and type int. This attribute defines the sentence's position in the scenario description. ScenarioSentences form scenarioSteps of ConstrainedLanguageRepresentations :: ConstrainedLanguageScenarios. They are also activity-Sentences of ActivityRepresentations :: ActivityScenarios which are alternative forms of representing ConstrainedLanguageRepresentations :: ConstrainedLanguageScenarios. Scenar-

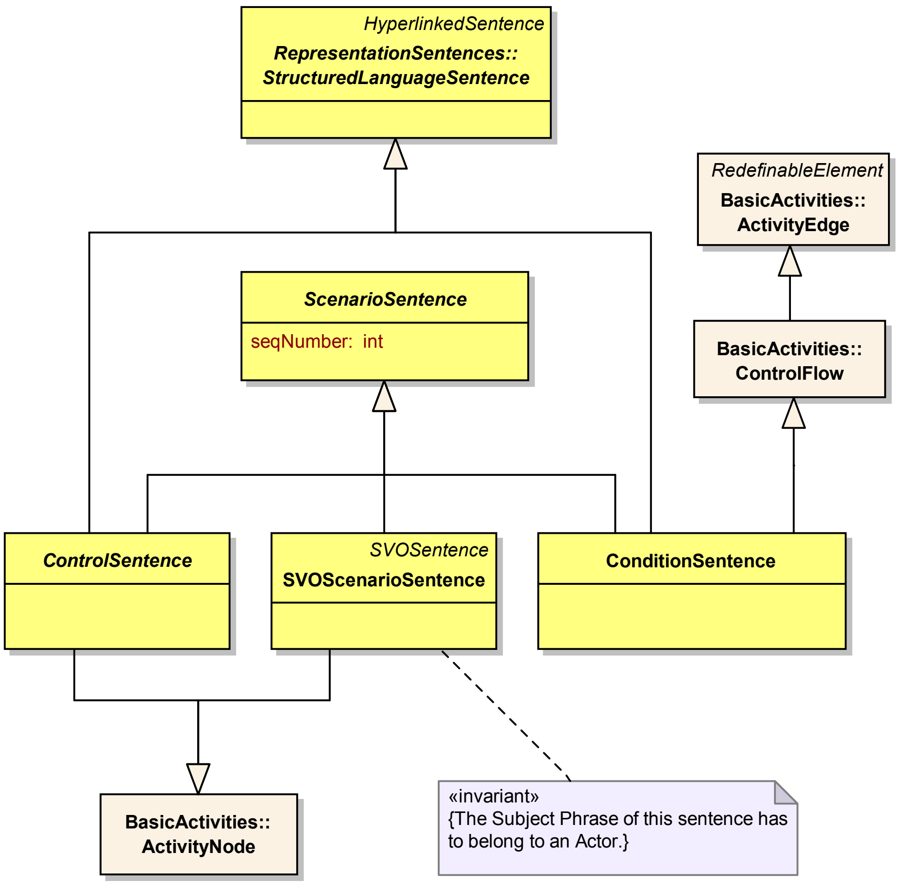


Figure 8.6: Scenario Sentences

ioSentence's subclasses are SVOScenarioSentence, ControlSentence and ConditionSentence, which are described in detail in the sections below.

SVOScenarioSentence

Semantics. SVOScenarioSentence describes a single scenario step (an action) in the form of a sentence in the SVO(O) grammar. This action can be performed by an actor or by the system.
Abstract syntax. SVOScenarioSentence is a kind of RepresentationSentences :: ScenarioSentence and has the whole syntax of SVOSentence :: SVOSentence. It also derives from BasicActivities :: ActivityNode. Because the action described in SVOScenarioSentence can be performed only by an actor or by the system, there is a constraint that the Phrases :: Phrase associated with this sentence as a subject (see SVOSentences :: SVOSentence) has to belong to an Actors :: Actor or SystemRepresentations :: SystemUnderDevelopmet.

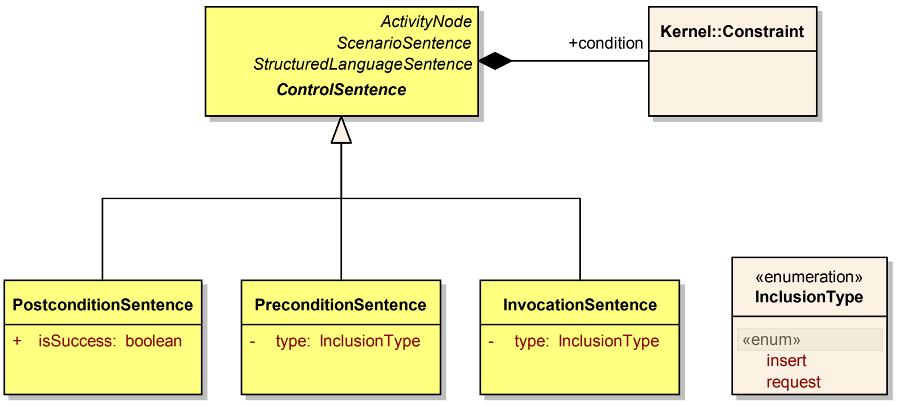


Figure 8.7: Control Sentences

ConditionSentence

Semantics. ConditionSentence is a special kind of scenario sentence that controls the flow of scenario execution. It is a point of conditional control flow: the following scenario step can be executed only when the condition expressed by the ConditionSentence is true.
Abstract syntax. ConditionSentence is a kind of RepresentationSentences :: ScenarioSentence It also derives from BasicActivities :: ControlFlow and RepresentationSentences :: Structured-LanguageSentence. In ActivityRepresentations :: ActivityScenario it subsets 'edge'.

ControlSentence

Semantics. ControlSentence is a general type of scenario sentences that control the flow of scenario execution. Depending on the concrete kind of ControlSentence, the flow of execution can be initiated, stopped or moved to another use case. ControlSentence can have an associated condition which must be met in order to perform this sentence.
Abstract syntax. ControlSentence is a kind of RepresentationSentences :: ScenarioSentence. It also derives from BasicActivities :: ActivityNode and RepresentationSentences :: Structured-LanguageSentence. ControlSentences can have an associated Kernel :: Constraint. This abstract class is a generalisation of concrete classes: InvocationSentence, PreconditionSentence and PostconditionSentence.

InvocationSentence

Semantics. InvocationSentence denotes the invocation of another use case scenario from within the currently performed use case scenario. There are two types of InvocationSentence: insert

and request. Insert means that the system invokes another use case by inserting its scenario sentences. Request means that the Actor requests invoking another UseCaseRelationships :: UseCase – it depends on the actor decision whether scenario sentences of invoked use case will be inserted or not. After performing all scenario steps of the invoked use case, the flow of execution returns to the invoking use case scenario to execute the remaining sentences. InvocationSentence is semantically related to PreconditionSentence (see below).
Abstract syntax. PreconditionSentence is a kind of ControlFlowSentence. It has the 'type' attribute determining the type of InvocationSentence, which can have one of the values enumerated in InclusionType.

PreconditionSentence

Semantics. PreconditionSentence is an initial sentence of every use case scenario. It indicates where the flow of control of every use case scenario starts. There are two types of InvocationSentence: insert and request. PreconditionSentence of type request is always performed when the actor triggers a use case directly or requests invoking a use case (see InvocationSentence above) from another use case scenario through initial actor action (first SVO(O) sentence in the scenario). When use case is invoked by inserting its scenario into the flow of invoking use case, the initial action is omitted. In this case PreconditionSentence of type insert is performed. PreconditionSentence may contain an associated condition which must be fulfilled before executing the sentence.
Abstract syntax. PreconditionSentence is a kind of ControlFlowSentence. It has the 'type' attribute determining the type of PreconditionSentence, which can have one of the values enumerated in InclusionType.

PostconditionSentence

Semantics. PostconditionSentence is a final sentence of every use case scenario. It indicates if the goal of a use case has been reached or not. It may contain an associated condition which must be fulfilled before executing the sentence.
Abstract syntax. PostconditionSentence is a kind of ControlFlowSentence. Its isSuccess attribute can have value 'true' or 'false'.

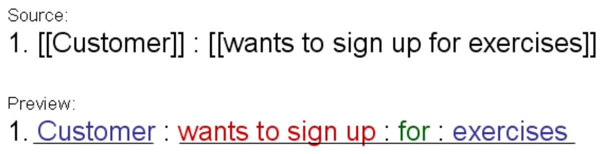
InclusionType

Semantics. InclusionType specifies the type of InvocationSentence and PreconditionSentence scenario sentences.
Abstract syntax. InclusionType is an enumerator which defines values: insert and request.

8.4.3 Concrete syntax and examples

ScenarioSentence. As an abstract meta-class, this meta-model element has no concrete syntax. It can be formulated in any of the representations of meta-classes that specialise from it.

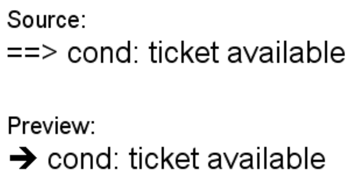
ControlSentence. As an abstract meta-class, this meta-model element has no concrete syntax. It can be formulated in any of the representations of meta-classes that specialise from it.


Source:
1. [[Customer]] : [[wants to sign up for exercises]]

Preview:
1. Customer : wants to sign up : for : exercises

Figure 8.8: SVOScenarioSentence example

SVOScenarioSentence. This class has two kinds of concrete syntax. It depends on the context where the SVOScenarioSentence is presented to the user. In scenario view, the SVOScenarioSentence is an ordered list of words. It has structure similar to SVOSenteces :: SVOSentence. In addition to SVOsentence, SVOScenarioSentence has its sequence number in a scenario placed at its front. See Figure 8.8 for an example. In activity diagram view, SVOScenarioSentence is presented as an Activities :: Action (([Obj05b], paragraph 12.3.2, page 303): "Actions are notated as round-cornered rectangles. The name of the action or other description of it may appear in the symbol." In our case, the 'name' is a SVOScenarioSentence without Hyperlinks. See Figure 7.13 for an example.

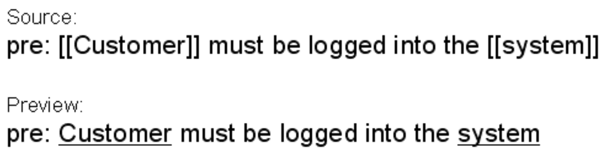

Source:
==> cond: ticket available

Preview:
➔ cond: ticket available

Figure 8.9: ControlFlowSentence example

ConditionSentence. Concrete syntax of ConditionSentence depends on the context where it is presented to the user. In scenario view it contains a 'condition' (instance of Kernel :: Constraint). It is presented with a special sign '==>', key word 'cond:' and a set of words forming a

NaturalLanguageHypertextSentence. See Figure 8.9 for an example. In activity diagram view ConditionSentence is presented as an arrowed line connecting two ordered SVOScenarioSentences. In addition, it contains a NaturalLanguageHypertextSentence in square brackets put near the arrow. See Figure 7.13 for an example.

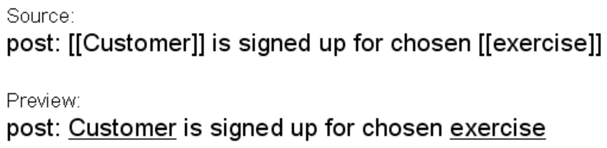


Source:
pre: [[Customer]] must be logged into the [[system]]

Preview:
pre: Customer must be logged into the system

Figure 8.10: PreconditionSentence example

PreconditionSentence. This class has two kinds of concrete syntax. It depends on context where the PreconditionSentence is presented to the user. In scenario view it is notated by a keyword 'pre:' and a set of words forming a NaturalLanguageHypertextSentence. PreconditionSentence can occur only before the first sentence in the scenario. See Figure 8.10 for an example In activity diagram view this element is shown as a note attached to the Activities :: InitialNode. See Figure 7.13 for an example.



Source:
post: [[Customer]] is signed up for chosen [[exercise]]

Preview:
post: Customer is signed up for chosen exercise

Figure 8.11: PostconditionSentence example

PostconditionSentence. PostconditionSentence's concrete syntax depends on the context where it is presented to the user. In scenario view it is notated by a keyword 'post:' and a set of words forming a NaturalLanguageHypertextSentence. PostconditionSentence can occur only after the last sentence in the scenario. See Figure 8.11 for an example. In activity diagram view this element is shown as a note attached to the Activities :: FinalNode. See Figure 7.13 for and example.

InvocationSentence. Concrete syntax of InvocationSentence depends on the context where it is presented to the user. In scenario view it is notated by a special sign '==>', one of two keywords: 'invoke/request:' or'invoke/insert:' and a set of words forming a NaturalLanguageHypertextSentence (this constitutes the name of invoked use case). See Figure 8.12 for an example. In activity diagram view, InvocationSentence is presented as a round-cornered rectangle with stereotype 'invoke/request' or 'invoke/insert'. The name of the InvocationSentence appears in the symbol. InvocationSentence is connected with SVOScenarioSentences with two arrowed lines. See Figure 7.13 for an example.

InclusionType. Concrete syntax of this element is one of two expressions: 'invoke/request:', 'invoke/insert:'. See Figures 8.12, 7.13.

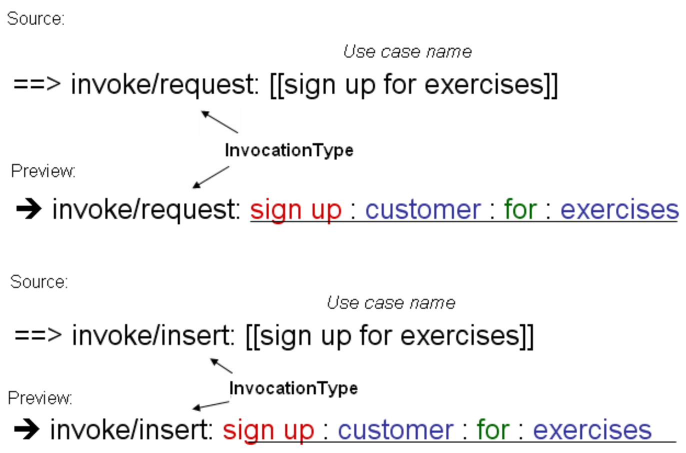

Source:
Use case name
==> invoke/request: [[sign up for exercises]]
InvocationType
Preview:
➔ invoke/request: sign up : customer : for : exercises
Source:
Use case name
==> invoke/insert: [[sign up for exercises]]
InvocationType
Preview:
➔ invoke/insert: sign up : customer : for : exercises

Figure 8.12: InvocationSentence example

Chapter 9

Conclusion

This deliverable presents a novel requirements specification language, more precisely its behavioural part. This language is special because of its clear distinction between Functional and Behavioural Requirements as well as its precise definition of their relationships. Its conceptual definition is new in its clear distinction between requirements and *representations* of requirements. This distinction is important for the use of this language as a basis for reuse based on requirements, since only representations can actually be reused. This language is also unique through its explicit distinction between *descriptive* and *model-based* requirements representations. Our language is the first requirements specification language intimately integrated with UML and defined using the same metamodelling approach as used for UML itself (using MOF). This deliverable also presents and explains the behavioural part of this language definition, from abstract down to concrete syntax.

Bibliography

[Ant96] Annie I. Antón. Goal-based requirements analysis. In *ICRE '96: Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96)*, page 136, Washington, DC, USA, 1996. IEEE Computer Society.

[BI96] Barry Boehm and Hoh In. Identifying quality-requirement conflicts. *IEEE Software*, 13(2):25–35, 1996.

[BPG$^+$01] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Modeling early requirements in tropos: A transformation based approach. In *AOSE*, pages 151–168, 2001.

[CKM01] Jaelson Castro, Manuel Kolp, and John Mylopoulos. A requirements-driven development methodology. *Lecture Notes in Computer Science*, 2068, 2001.

[CKM02] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: The tropos project. *To Appear in Information Systems, Elsevier, Amsterdam, The Netherlands*, 2002.

[Coc97] Alistair Cockburn. Structuring use cases with goals. *Journal of Object-Oriented Programming*, 5(10):56–62, 1997.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. In *6IWSSD: Selected Papers of the Sixth International Workshop on Software Specification and Design*, pages 3–50, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V.

[EK02] Gerald Ebner and Hermann Kaindl. Tracing all around in reengineering. *IEEE Software*, 19(3):70–77, 2002.

[Fow04] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, Massachusetts, third edition, 2004.

[GMP01] Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The tropos software development methodology. *Technical Report No. 0111-20, ITC - IRST. Submitted to AAMAS '02. A Knowledge Level Software Engineering 15*, 2001.

[GPM^{+}01] Paolo Giorgini, Anna Perini, John Mylopoulos, Fausto Giunchiglia, and Paolo Bresciani. Agent-oriented software development: A case study. In *Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering (SEKE01)*, 2001.

[GPS01] Fausto Giunchiglia, Anna Perini, and Fabrizio Sannicolo. Knowledge level software engineering. *In Springer Verlag, Editor, In Proceedings of ATAL 2001, Seattle, USA. Also IRST TR 011222, Istituto Trentino Di Cultura, Trento, Italy*, 2001.

[Kai93] H. Kaindl. The missing link in requirements engineering. *ACM Software Engineering Notes (SEN)*, 18(2):30–39, 1993.

[Kai95] H. Kaindl. An integration of scenarios with their purposes in task modeling. In *Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods, & Techniques (DIS '95)*, pages 227–235, Ann Arbor, MI, August 1995. ACM.

[Kai96] H. Kaindl. Using hypertext for semiformal representation in requirements engineering practice. *The New Review of Hypermedia and Multimedia*, 2:149–173, 1996.

[Kai97] H. Kaindl. A practical approach to combining requirements definition and object-oriented analysis. *Annals of Software Engineering*, 3:319–343, 1997.

[Kai00] H. Kaindl. A design process based on a model combining scenarios with goals and functions. *IEEE Transactions on Systems, Man, and Cybernetics (SMC) Part A*, 30(5):537–551, Sept. 2000.

[Kai05] Hermann Kaindl. A scenario-based approach for requirements engineering: Experience in a telecommunication software development project. *Systems Engineering*, 8(3):197–209, 2005.

[KGM02] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. A goal-based organizational perspective on multi-agent architectures. In *ATAL '01: Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, LNCS 2333, pages 128–140. Springer, 2002.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, Englewood Cliffs, NJ, second edition, 2004.

[Lau02] Søren Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley, Reading, MA, 2002.

[MC00] John Mylopoulos and Jaelson Castro. Tropos: A framework for requirements-driven software development. *In J. Brinkkemper and A. Solvberg, Editors, Information Systems Engineering: State of the Art and Research Themes. SpringerVerlag*, 2000.

[MCL$^+$01] John Mylopoulos, Lawrence Chung, Stephen Liao, Huaiqing Wang, and Eric Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, /2001.

[MCY99] John Mylopoulos, Lawrence Chung, and Eric Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, 1999.

[MKC01] John Mylopoulos, Manuel Kolp, and Jaelson Castro. UML for agent-oriented software development: The tropos proposal. In *UML 2001 - The Unified Modeling Language.Modeling Languages, Concepts, and Tools: Fourth International Conference*, LNCS 2185, pages 422–441. Springer, 2001.

[MKG02] John Mylopoulos, Manuel Kolp, and Paolo Giorgini. Agent-oriented software development. In *Methods and Applications of Artificial Intelligence: Second Hellenic Conference on AI, SETN*, LNCS 2308, pages 3–17. Springer, 2002.

[MM03] Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide Version 1.0.1, omg/03-06-01*. Object Management Group, 2003.

[MOW01] Pierre Metz, John O'Brien, and Wolfgang Weber. Against use case interleaving. *Lecture Notes in Computer Science*, 2185:472–486, 2001.

[Obj03a] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification, Final Adopted Specification, ptc/03-10-04*, 2003.

[Obj03b] Object Management Group. *OCL 2.0, Final Adopted Specification, ptc/03-10-14*, 2003.

[Obj05a] Object Management Group. *Unified Modeling Language: Infrastructure, version 2.0, formal/05-07-05*, 2005.

[Obj05b] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04*, 2005.

[Obj06] Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, 2006.

[Sim99] A J H Simons. Use cases considered harmful. In *Proceedings of the 29th Conference on Technology of Object-Oriented Languages and Systems-TOOLS Europe'99*, pages 194–203, Nancy, France, June 1999. IEEE Computer Society Press.

[vLL00] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, 2000.