Introducing a unified Requirements Specification Language

Michał Śmiałek, Albert Ambroziewicz, Jacek Bojarski,
Tomasz Straszak, and Wiktor Nowakowski

Warsaw University of Technology,
Warsaw, Poland
`{smialek,ambrozia,bojarsj1,straszat,nowakoww}@iem.pw.edu.pl`
`http://www.iem.pw.edu.pl/~smialek`

Abstract. Requirements are currently an undervalued "beast" in the software modelling world. All the model transformation approaches ignore requirements as not having the essential prerequisite of a precise metamodel. Even so, the approaches to requirements modelling are sparse and disable the possibility to construct certain uniform transformation and reuse mechanisms. In this paper we introduce a proposition of a unified language to specify requirements. This language is based on a metamodel that allows for unambiguous transformation from requirements to design. The language offers flexibility in the level of formality set on the requirements specifications. Using the most precise (constrained tongue and diagrammatic) elements of the language, certain level of traceability into design can be achieved. Currently being validated in practice, this language has already been evaluated in student projects. Some results of this evaluation are also presented.

Key words: requirements models, use cases, scenarios, domain models

1 Introduction

Requirements specification seems to be such element of the software development lifecycle that has the lowest support in model-driven development. The tendency is such that requirements have to be somehow "translated" by hand into more precise notations in order to be included into an automatic handling path. This is due to the fact that requirement representations lack an essential prerequisite of having a formally defined metamodel. In practice, requirements are usually written as paragraphs of natural language text, even though some notations allow for constructing "requirements models". These models however, operate on "requirements as such" level, without going into details of individual requirements. Modelling languages do not directly support requirements and specifically their detailed representations. In UML [11], the basic functional requirements units – use cases are defined in a very ambiguous manner (see [15] for a discussion, still valid for UML2). In SysML [18], the basic units of granularity are requirements themselves – their definitions being paragraphs of text. This is also the case for most widely used requirements management tools. Moreover, such models usually do not have precise links to the domain vocabulary. The vocabulary is often used inconsistently throughout the whole specification. This lack of common vocabulary also prevents from achieving any significant level of requirements reuse (see eg. [2] for a discussion on use case reuse). Typical scenarios (eg.: press button + show window + enter data + validate data + store data + show acknowledgement) get repeated over and over in many applications. Possibility of reuse of such standard scenarios that have unambiguously mapped design and code to it, could have tremendous value to software development organisations.

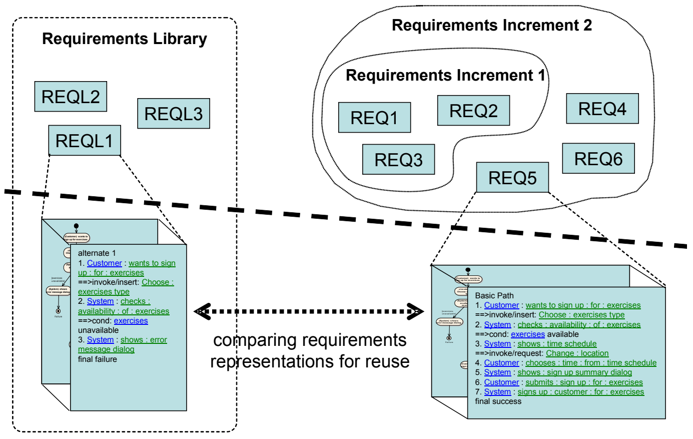
Unfortunately, multitude of approaches in various methodologies and ambiguity of natural language requirements lead to serious problems in translating them into design models and assuring traceability (including traceability between different projects). Thus, the current paper presents an approach to unify various tendencies in requirements modelling into a single coherent language. This language is supplied with a formally defined metamodel which allows for defining automatic transformations and comparison queries.

2 From natural language to models

Let us first analyse a typical path for requirements that leads from the initial user needs to design and then code. Most often, the initial vision for the prospective software system is written in natural language. So, typical notation for vision level requirements is simply paragraphs of text. Usually, these requirements combine functional and non-functional aspects of the prospective system. They are not structured in any way, as they reflect the clients' ideas on the system they envision to support their business.

Having the vision, we need to define the scope of the system. Most often, this is done by identifying certain "units of functionality" to be supported by the developed system. Such units of functionality can be simply paragraphs of text that describe some behaviour of the system. In more structured approaches, these paragraphs of text can be based on certain goals to be achieved for the user. This includes use cases as used in such methodologies as the Unified Process [9] or ICONIX [14], user stories as used in eXtreme Programming [1] or features as in Feature Driven Development [13]. Units of functionality are usually supported by non-functional requirements divided into groups (performance, usability, etc.) and a domain description (vocabulary of terms used throughout paragraphs of text).

Here we have to note limitations of having the above requirements represented in just the natural language. First limitation is caused by the fact that requirements as such are managed as wholes. This means that traces or mappings to or from requirements are done on the basis of "whole" requirements, not considering the requirement details (expressed through appropriate representations). Second limitation is based on inability to analyse the contents of requirements for comparison. Since the contents are paragraphs of text, the only



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

Basic Path
1. Customer : wants to sign up : for : exercises
==>invoke/insert: Choose : exercises type
2. System : checks : availability : of : exercises
==>cond: exercises available
3. System : shows : time schedule
==>invoke/request: Change : location
4. Customer : chooses : time : from : time schedule
5. System : shows : sign up summary dialog
6. Customer : submits : sign up : for : exercises
7. System : signs up : customer : for : exercises
final success

Fig. 1. Two levels of requirements management

way to compare requirements is through natural language processing which has serious limitations. This makes it practically impossible to compare different requirements specifications in order to achieve certain reuse of their contents (reuse of functionality descriptions, reuse of term descriptions and so on).

What can we do with "whole" requirements? These requirements can be traced one onto another and can be assigned certain attributes. We can also trace certain requirements into design. Such functionality is offered by major Requirements Management tool vendors (RequisitePro, DOORS, etc.). Finally, some level of reuse can be achieved by attaching certain "manifestation" information to the requirement artifacts (as in Reusable Asset Specification [10]). It can be noted that maintaining traces or manifestation information is quite laborious and has to be done manually. Moreover we can assure that requirements are met in design only on a coarse grained level (whole requirements).

What we really would like to do with requirements is to use their contents for certain tasks in a software development lifecycle. We would like to try to compare the contents of requirements in order to facilitate reuse. We would also be interested in performing some automatic operations (transforming) of the contents of several requirements to obtain at least some initial sketch of architectural design.

Here we reach the moment where a major question has to be asked. How to represent the contents of requirements? This is a difficult task, as requirements are read by many participants in a software development project: users, stakeholders, analysts, designers and so on (see [16]). People got used to textual, natural language representations of requirements. As stated above, such representations are not proper in automatic handling for transformations or reuse. We need some structured language which has a strict metamodel or grammar behind it. This language has to be designed with care as it has to be understandable by "ordinary people", and at the same time precise enough to allow for the tasks we mention above.

In this requirements specification language we need to clearly distinguish between requirements as such and their representations (see [16] for an insight to this). Requirements as such are just names with identifiers and attributes. Representations express the information contained within requirements. This distinction introduces two levels of abstraction. On one level we handle requirements as atomic units, on the other level we handle the details of their contents. "Requirements level" allows for coarse grained requirements management. "Requirement representations" level allows for automatic requirements processing (such as transformations and reuse). This is illustrated in Figure 1.

In order to achieve certain level of automation in requirements handling, as mentioned above, we need to constrain the ways requirements can be represented. This certainly should not mean limiting the analysts, but giving certain order to their products. For our purposes we are obviously interested in constrained language representations and schematic (diagrammatic) representations. In this paper we shall restrict ourselves to representations for functional requirements which originate in scenario approach. In the further description we shall concentrate on constrained language scenarios with equivalent diagrammatic notations: activity diagrams and sequence (interaction) diagrams. These constrained representations can be expressed in a formal way through a grammar. This grammar can be expressed as a MOF [12] metamodel, thus allowing to treat scenarios and their parts as elements in a model. Scenarios expressed as models can be used for automatic parsing which enables querying and transformations. The next section contains an overview of the grammar with concrete syntax for scenarios.

Apart from functional and non-functional requirements, the requirements specification usually contains some form of a glossary or vocabulary of the problem domain. In certain approaches, also class diagrams are used to express the domain model. Diagrammatic approach is very valuable for the means of representing the domain vocabulary, due to its visual impact and expressiveness. However, using class diagrams in requirements specifications most often leads to design-influenced discussions. Moreover, class diagrams tend to concentrate on the "nouns" in requirements (classes and their attributes) rather than the "verbs" (operations, which are treated as design elements). When trying to define the verbs, analysts tend to fall into analysing (or rather: designing) possible message paths between objects. In other cases, "verbs" are completely ignored in the vocabulary and their definitions are scattered throughout the functional requirements definitions.

Scattered definitions of terms is a common problem in a requirements specification. Often, a single term is used to denote several different notions (homonyms). For instance, the term "account" is used to denote a "user account" or "banking account" in the same specification leading to serious confusion. In other situations, synonyms are used to denote the same notions (compare: "system *calculates* **interest rate**", and "system *computes* **loan interest**"). This often leads to serious incoherence of the overall specification. Thus, we need a single domain vocabulary, where all the definitions are referenced consistently from inside of various requirement representations.

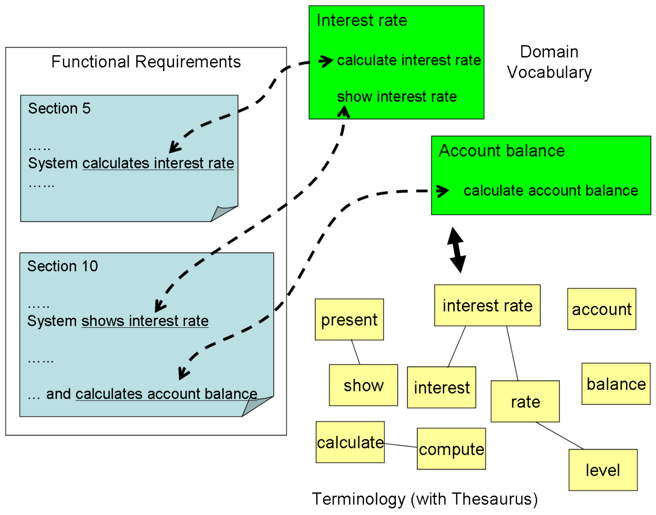


Fig. 2. Illustration of hyperlinking between requirements, domain elements and terms

On top of the above it should be noted that in fact, terms and their definitions are not what we want in a domain vocabulary. We need *notions* and *phrases*. A notion defines a specific domain element that can be named with different (perhaps synonymic) terms. Moreover, these notions can participate in many different phrases - most often containing certain verbs. For instance, a notion expressed with a term “interest rate” can be part of phrases: “calculate interest rate” or “show interest rate”. Of these phrases each has a specific meaning (thus also a definition) in the described problem domain.

It can be noted that a specific term (a noun or a verb) can be used in many phrases. For instance the phrase “calculate account balance” uses the same verb term (“calculate”) as one of the phrases above. It also has a noun term (“account balance”, composed of two terms: “account” and “balance”) that can take part in many verb phrases. A question arises, how to organise phrases in a vocabulary. It is quite obvious that the most natural way of grouping phrases would be by the nouns (and specifically not by the verbs). This is because verb phrases have their definitions always put in a context of specific notions expressed through nouns. This is not the case for verbs as their meaning is usually completely different for different associated notions.

Grouping of phrases leads us to the structure of domain models which could be expressed in diagrammatic form. Domain diagrams could contain vocabulary notions (domain elements) that are composed of several associated phrases containing the same notion. Example of this is shown in Figure 2, where two related domain elements (*Interest rate* and *Account balance*) with contained phrases are presented. Such organisation of the vocabulary greatly facilitates keeping coherence of functional requirements (eg. scenarios). The Figure shows two sections of a requirements specification where clear links to the vocabulary are introduced. It can be noted that the functional requirements themselves – in such approach

– do not contain any business domain information. They only contain sequences of events/interactions – pure functionality of the application. All the business domain information is placed in the vocabulary. It is only there that all the business notions (nouns) and business processing algorithms (verb phrases) are described (their definitions kept).

As we have noted above, all the notions and phrases contain various terms. Typical problem here is high variability in using sometimes synonymic terms in different requirements specifications. This prevents from reusing such specifications from project to project, due to different terminology used which makes comparison of specifications practically infeasible. However, having a coherent domain vocabulary, as presented above, might help. For this it would be important to use an external "global" terminology within this vocabulary. This terminology would contain terms (possibly with some globally recognised definitions) and links which denote synonymity or close semantic relationship between them (in other words: a thesaurus). Such terminologies are widely and publicly available (eg. WordNet, [4], wordnet.princeton.edu) and can be used for building domain vocabularies. This is illustrated in Figure 2 where an external terminology is shown. Terms from this terminology (taken explicitly from WordNet) have links that allow for comparing with specifications written with different terms (see eg. links between "calculate" and "compute" or "present" and "show").

With the above approach we substitute purely natural language specifications with structured language and diagrammatic representations according to a specific formal grammar. Functional and non-functional requirements are made coherent through a central vocabulary of domain notions with associated phrases. These phrases use a globally recognised terminology which allows for comparison and leverage the potential for their reuse (see [3] for a domain based reuse approach with use cases). It can be noted that such a complete specification resembles a wikipedia, where hyperlinks are extensively used, as introduced by Kaindl (see [7]). Hyperlinks are organised around domain diagrams which introduce diagrammatic view of a vocabulary.

In the following sections we shall present the syntax of the language and illustrate its usage. We will show that the proposed language is understandable by its users (both readers and writers) and at the same time precise enough to allow for easy transformation into design models.

3 Unifying syntax for requirements

In order to reach repeatability between requirements specifications we need a common syntax for individual elements of such specifications. This syntax is part of a language which we will call the Requirements Specification Language (RSL). The syntax is presented below mainly in its concrete form, while also some elements of the abstract syntax are given. The key concepts of RSL, like requirements, notions and requirements representations, can be expressed in graphical way.

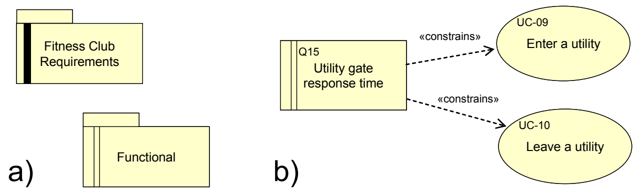


Fig. 3. RSL notation for specifications, packages (a) and requirements, use cases and relationships between requirements (example of one relationship type) (b) in diagrams

An RSL model, consistent with its syntax is called a RequirementsSpecification. Such specifications can be composed of RequirementsPackages. Within these packages, individual Requirements can be placed. Finally, every Requirement has its representations contained within it. RSL is mainly a visual language, thus appropriate elements written in this language can be shown in diagrams: Requirements diagrams, Domain diagrams, sequence and activity diagrams. Moreover, these elements can be shown as a hierarchy tree in a tool. Appropriate icons denoting packages are presented in Figure 3a. Distinguished from other Requirements (which can express any kind of requirement - functional, constraint, etc.) are UseCases, which specify "sets of actions performed by a system, which yield an observable results that is, typically, of values for one or more actors or other stakeholders of the system" [11]. UseCases are denoted with an oval (see Figure 3b) in order to make this notation consistent with a commonly used UML notation and different from other Requirements. It can be noted that requirements, besides their descriptive name, have a unique identifier placed in their top left corner. The above notation allows to depict individual requirements, what we still need is the way to express relationships between requirements and traces to design. In Figure 3b we can see also a notation for RequirementRelationships. This diagram shows relationships between a quality and two use case requirements. Also, other types of relationships are possible, however we shall omit them here for brevity.

The most important part of the RSL is that pertaining RequirementRepresentations. In general, all the representations are based on the concept of hyperlinks. Even the natural language representations allow for including hyperlinks to domain elements or phrases. An example of such a natural language representation of requirement from Figure 3b is presented below:

- `Whenever entering or leaving a [[n: utility]], the time between [[v: swiping n: card]] and [[v: opening n: utility gate]] should be smaller than 0.5 second.`
- Whenever entering of leaving a utility, the time between swiping : card and opening : utility gate should be smaller than 0.5 second.

In this example, PhraseHyperlinks to three phrases are used. Two of the phrases are simple noun Phrases, and one of them is a VerbPhrase. It can be noted that the RSL notation allows for a "source" version and a "preview" version of any textual requirement representation. In the source variant, every element of a PhraseHyperlink is denoted by a prefix: *n:* for a noun expression and *v:* for a verb expression. In the preview variant, phrase parts are separated with a colon "*:*". In the following examples we shall only show the source versions. Hyperlinks can also contain modifiers (usually adjectives) but we shall omit them here for brevity.

In contrast to NaturalLanguageRepresentations described above, ConstrainedLanguageRepresentations have a precise grammar defined. In this paper we shall concentrate on ScenarioSentences where SVOScenarioSentence is the most important of them. SVO(O) sentences [5], [6], [17] are simple sentences consisting of a subject and a predicate. This predicate is composed of a verb and and one or two objects (with an optional preposition). Examples of such sentences are given below:

- `[[n: FC System]] [[v: opens n: utility gate]]`
- `[[n: Customer]] [[v: swipes n: card]]`
- `[[n: Receptionist]] [[v: chooses n: bill p: from n: bill list]]`
- `[[n: FC System]] [[v: prints n: bill]]`

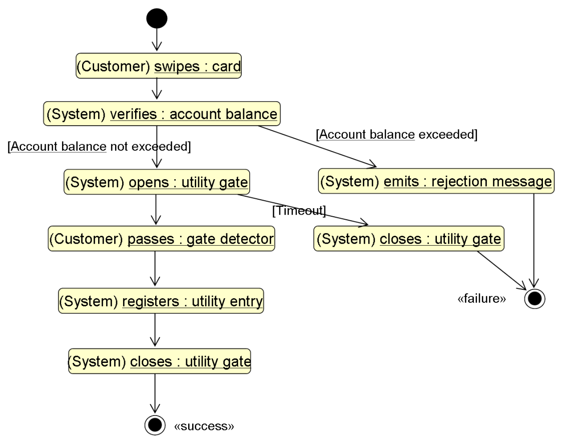
The syntax for SVO(O) sentences allows for only hyperlinks. The first Hyperlink relates to a domain element representing the sentence subject. This domain element can be the system to be built or an actor (representing someone or something interacting from the outside of the system to be built). In the example above, the system to be built is "*FC System*" (Fitness Club System), and the actors are "*Receptionist*" and "*Customer*". The second PhraseHyperlink represents the sentence predicate. It relates to a VerbPhrase contained in a specific notion in the domain vocabulary.

We can combine sequences of such SVO(O) sentences into ConstrainedLanguageScenarios. These scenarios show interactions between an actor (or actors) and the system to be built. In other words they present a dialogue between the user and the system. Two example scenarios are shown below:

1. `[[n: Customer]] [[v: swipes n: card]]`
2. `[[n: FC System]] [[v: verifies n: account balance]]`
3. `==> cond [[n: Account balance]] not exceeded`
4. `[[n: FC System]] [[v: opens n: utility gate]]`
5. `[[n: Customer]] [[v: passes n: gate detector]]`
6. `[[n: FC System]] [[v: registers n: utility entry]]`
7. `[[n: FC System]] [[v: closes n: utility gate]]`

and

1. `[[n: Customer]] [[v: swipes n: card]]`
2. `[[n: FC System]] [[v: verifies n: account balance]]`
3. `==> cond [[n: Account balance]] exceeded`
4. `[[n: FC System]] [[v: emits n: rejection message]]`



(Customer) swipes : card
(System) verifies : account balance
[Account balance not exceeded]
[Account balance exceeded]
(System) opens : utility gate
(System) emits : rejection message
[Timeout]
(Customer) passes : gate detector
(System) closes : utility gate
«failure»
(System) registers : utility entry
(System) closes : utility gate
«success»

Fig. 4. Activity scenario expressed as an activity diagram – consistent with the constrained language scenarios

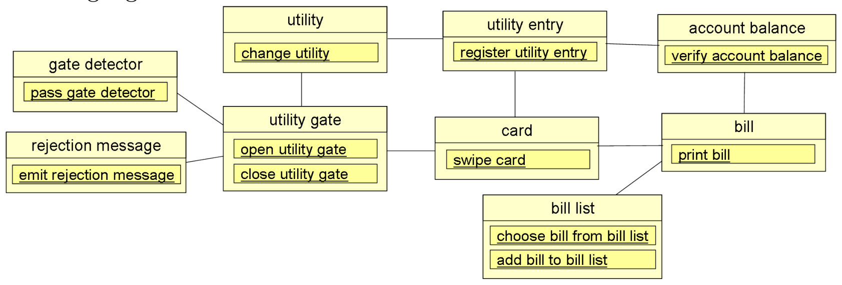


Fig. 5. Domain diagram showing notation for notions and phrases – targets for the hyperlinks

As it can be seen, the two scenarios complement each other and show two alternative paths of the same UseCase: "*Enter a utility*". They contain two conditions denoted by ==> *cond* which control the flow between these two paths. Both of the scenarios can be placed inside the "*Fitness Club Requirements*" specification (see Fig. 3) – within the *Functional* package under the appropriate use case. There we can also place another representation of the UseCase – in the form of an ActivityScenario. This representation is shown in Figure 4. The two representations contain almost exactly the same information. Additional notation for sentences in this diagram allows distinguishing parts of SVO sentences forming scenario steps. It can be noted that the activity diagram contains an additional path and distinguishes success and failure paths.

The domain model is the target of all the hyperlinks presented in the above examples. For a domain model we draw domain diagrams. These diagrams have certain elements in common with class diagrams, but they shift the paradigm from classes with operations to notions with phrases. Domain diagrams contain DomainElements (specifically: Notions) expressed as boxes. These domain elements (notions) can contain Phrases, where every phrase is a also a box. This is

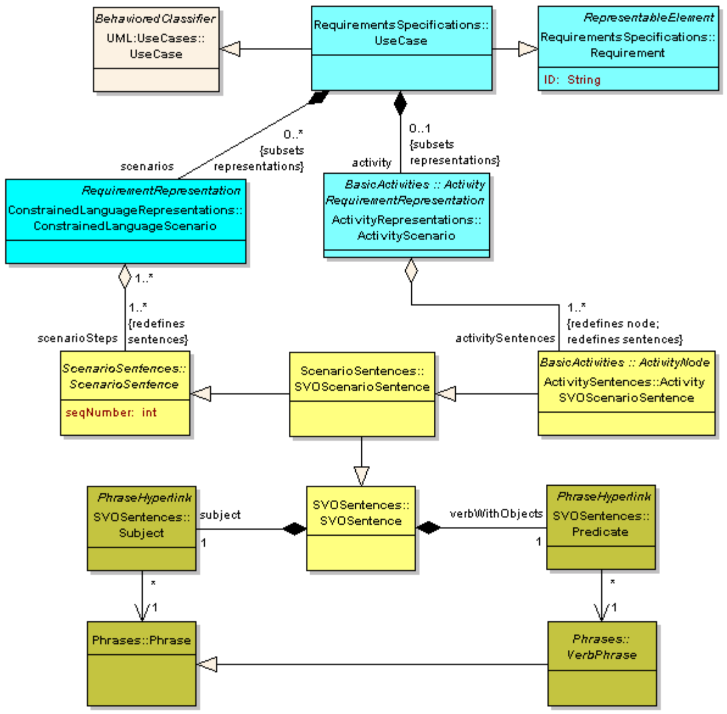

BehavioredClassifier
UML:UseCases::UseCase
RequirementsSpecifications::UseCase
RepresentableElement
RequirementsSpecifications::Requirement
ID: String
0..*
{subsets representations}
scenarios
activity
0..1
{subsets representations}
RequirementRepresentation
ConstrainedLanguageRepresentations::ConstrainedLanguageScenario
BasicActivities :: Activity
RequirementRepresentation
ActivityRepresentations::ActivityScenario
1..*
1..*
{redefines sentences}
scenarioSteps
activitySentences
1..*
{redefines node; redefines sentences}
ScenarioSentences::ScenarioSentence
seqNumber: int
ScenarioSentences::SVOScenarioSentence
BasicActivities :: ActivityNode
ActivitySentences::Activity SVOScenarioSentence
PhraseHyperlink
SVOSentences::Subject
subject
1
SVOSentences::SVOSentence
verbWithObjects
1
PhraseHyperlink
SVOSentences::Predicate
*
1
*
1
Phrases::Phrase
Phrases::VerbPhrase

Fig. 6. Summary of the RSL metamodel

shown in Figure 5. The diagram contains all the phrases hyperlinked within the presented example requirements.

Moreover, all the DomainElements are related with appropriate DomainElementRelationships. These relationships are created on the basis of hyperlinks in domain element descriptions. Let's take a description of the "*card*" notion: "... Groups all [[n: utility entries]]. Can be swiped through a [[n: utility gate]] ... for every card a [[n: bill]] can be issued ...". As it can be seen, the above hyperlinks lead to appropriate Notions related to the "*card*" notion as shown in Figure 5.

Behind the above presented concrete syntax of the RSL, a metamodel defining the language's abstract syntax resides. This metamodel is shown in a very brief overview in Figure 6. This Figure presents most of the elements described in this paper. We can see the Requirement metaclass which denotes the most important element of the language. Another basic element is the DomainElement (not shown in Figure). Both of these elements are RepresentatbleElements, ie. elements that have associated Representations. Such 'representations' are shown for a UseCase metaclass which is a specialisation of Requirement. It can be seen that use cases can have as their representations ConstrainedLanguageScenarios and ActivityScenarios. It should be noted that use cases are specialised from UML UseCases (which are BehavioredClassifiers) and activity scenarios are specialised from UML Activities.

Below, we can see some selected elements of the structure of the two use case representations. Constrained language scenarios contain ScenarioSentences as their 'scenarioSteps'. Activity scenarios generally contain 'nodes' which can be (among other) ActivitySVOScenarioSentences. Most of the sentences in scenarios are SVOSentences (there can be also eg. ConditionalSentences). Such sentences contain a 'subject' and 'verbWithObjects being its Predicate. These two sentence elements are Hyperlinks that point to a Phrase or a VerbPhrase respectively. Due to lack of space, a description of the domain model will be omitted. A very detailed metamodel of the language, together with concrete syntax and semantics description can be found in [8].

4 Summary and future work

We have presented a new Requirements Specification Language. The language allows for creating understandable and coherent specifications by even unexperienced analysts. Validation of user comprehension of the language by experienced developers and business people is currently under way. In order to validate the Requirements Specification Language in practice, several commercial companies are currently specifying requirements for real-life projects using RSL (see Acknowledgments). Complete results are expected by the end of July '07. Prior to these projects, several student projects were successfully conducted as part of software engineering courses at the Warsaw University of Technology and the University of Carlos III in Madrid. Moreover, current efforts in the ReDSeeDS project concentrate on creating transformations for the RSL (in MOLA). This will allow for validating usefulness of the language in a full software development lifecycle that uses a model-driven approach. Of course, the current plans also include building a dedicated tool that would allow for full support of the language grammar (especially the hyperlinking features). So far, a profile for an existing tool (Enterprise Architect from SparxSystems) has been created and used for creating diagrams as described in this paper. Also, a experimental tool for validation of writing scenarios sentences in SVO grammar has been implemented [17]. Future plans also include creating a grammar for statements allowing to specify non-functional requirements. Finally, a tool is planned that would allow for seeking and reusing full "software cases" that lead from requirement specifications written in RSL through transformations to design models. This way, a requirements-driven reuse could be achieved.

Acknowledgments. This work is partially funded by the EU: Requirements-driven Software Development System (ReDSeeDS) (contract no. IST-2006-33596 under 6FP). The project is coordinated by Infovide-Matrix, Poland with technical lead of Warsaw University of Technology and with University of Koblenz-Landau, Vienna University of Technology, Fraunhofer IESE, University of Latvia, HITeC e.V. c/o University of Hamburg, Heriot-Watt University, PRO DV, Cybersoft and Algoritmu Sistemos.

References

1. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
2. Robert Biddle, James Noble, and Ewan Tempero. Supporting reusable use cases. *Lecture Notes in Computer Science*, 2319:210–226, 2002.
3. M C Blok and J L Cybulski. Reusing UML specifications in a constrained application domain. In *Proceedings of 1998 Asia Pacific Software Engineering Conference*, pages 196–202, 1998.
4. Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
5. Ian M Graham. Task scripts, use cases and scenarios in object-oriented analysis. *Object-Oriented Systems*, 3(3):123–142, 1996.
6. Ian M Graham. *Object-Oriented Methods Principles & Practice*. Pearson Education, 2001.
7. Hermann Kaindl. Using hypertext for semiformal representation in requirements engineering practice. *The New Review of Hypermedia and Multimedia*, 2:149–173, 1996.
8. Hermann Kaindl, Michał Śmiałek, Davor Svetinovic, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Hannes Schwarz, Daniel Bildhauer, John P Brogan, Kizito Ssamula Mukasa, Katharina Wolter, and Thorsten Krebs. Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project, 2007. www.redseeds.eu.
9. Philippe Kruchten. *The Rational Unified Process: An Introduction, 3rd ed.* Addison Wesley, 2003.
10. Object Management Group. *Reusable Asset Specification: Final Adopted Specification, ptc/04-06-06*, 2004.
11. Object Management Group. *Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04*, 2005.
12. Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, 2006.
13. Stephen R Palmer and John M Felsing. *A Practical Guide to Feature-Driven Development*. Prentice Hall PTR, 2002.
14. D Rosenberg, M Stephens, and M Collins-Cope. *Agile Development with ICONIX Process: People, Process, and Pragmatism*. Apress, 2005.
15. 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.
16. Michał Śmiałek. Accommodating informality with necessary precision in use case scenarios. *Journal of Object Technology*, 4(6):59–67, August 2005.
17. Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, and Tomasz Straszak. Scenario construction tool based on extended UML metamodel. *Lecture Notes in Computer Science*, 3713:414–429, 2005.
18. SysML Partners. *Systems Modeling Language (SysML) Specification, version 0.9*, 2005.