

# Complementary Use Case Scenario Representations based on Domain Vocabularies

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

Warsaw University of Technology,  
Warsaw, Poland

{smialek, bojarsj1, nowakoww, ambrozia, straszat}@iem.pw.edu.pl

**Abstract.** Use cases are commonly used as notation for capturing functional requirements through scenarios. The problem is that there is no universal notation for use case contents which is capable of accommodating all the needs of software project participants. Business analysts and stakeholders need understandability and informality, while for architects and designers, precision and unambiguity are the most crucial features. In this paper we propose a metamodel and concrete syntax for three complementary representations of use case scenarios. These representations present the same information, but put emphasis on different aspects of it thus accommodating for different readers. This metamodel utilises the idea of separation of requirements as such from their representations as well as the idea of clear distinction between description of the system's behaviour and of the problem domain.

**Key words:** use cases, requirements, scenarios, activity diagrams, interaction diagrams

## 1 Introduction

In a typical software development project, several roles with sparse background have to maintain and read the requirements specification. For business people, requirements need to be understandable. Designers necessitate precision and unambiguity. Unfortunately, most notations for requirements do not offer both of these characteristics. Notations that are capable of being handled by machines for transforming into design are usually hard to read by “ordinary people”. Natural language notations are human readable, but leave too much space for interpretation and lack rigour needed by technical people. The ideal notation should allow for getting as diverse group of people as possible better involved in the process of eliciting, modelling, communicating and agreeing requirements, thus improving the quality of the resulting system. This issue has been widely discussed in the literature (see [1, 2, 10] for example).

It seems that an ideal notation should be a model – one that is understandable for the users (diagrammatic or in simple structured language) and precise enough to be handled by developers or even machines.

A commonly used notation for requirements in the modelling world are use cases. Unfortunately, there are numerous problems with use case notation – mainly with their representations. Use cases and relationships between them, as defined in UML [11] have quite vague semantics (see [4] for a discussion). This results in multitude of notations for their contents (see [6] for a survey) and is source for confusion and misunderstandings (as explained in [15]). Lack of clear separation of use case scenarios from problem domain description causes inconsistencies in requirements specifications (see [13, 16]).

Thus, in this paper we propose a notation that would unify sparse approaches to use case representation and allow for comprehension by different participants in a software project. This notation consists of three separate but complementary scenario-based representations of use cases. At the basis of the notation lies the idea of separation of requirements as such from their representations as well as the idea of clear distinction between description of the behaviour and the domain vocabulary. In the following sections, concrete and abstract syntax as well as semantics of the notation are explained.

## 2 Use case scenarios based on domain vocabulary

Before we present the various use case scenario representations, we shall start with providing a definition of a use case that seems most suitable for our purpose. From among tens of definitions which can be found in the literature, we shall use the one that is most representative and does not relate to any concrete notation. Such a definition of a use case was introduced by Cockburn in [3]. According to this definition a use case 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.”

Now, we need a definition for a scenario as a use case component. Again, referring to [3], a scenario 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.”

Above definitions are the basis on which we have designed three complementary use case scenario representations suitable for people having various roles in a software project and thus looking at a use case from different points of view. While designing these representations we took into account two important issues. First, we had to resolve the problem of precise control flow semantics for use cases. It can be argued that the semantics of “include” and “extend” relationships in UML is disadvantageous ([14], [9]). Thus, we substituted UML

relationships with “invoke” relationship. Its semantics and modified use case metamodel have been described in details in [16]. In the following section we summarise briefly this solution as it influences scenario representations introduced in this paper. The second important issue that applies to our approach is separation of use case scenario representations describing the system’s behaviour from the description of problem domain. This allows to eliminate many inconsistencies from requirements specifications ([16, 17]). In-depth research in above mentioned areas is being carried on as a part of the ReDSeeDS project ([www.redseeds.eu](http://www.redseeds.eu)). Below we present some results of this work concerning use case representations.

## 2.1 Redefined use case and types of scenario sentences

As defined above, use case is a set of scenarios with the same goal. There must be at least one scenario that reaches the goal (basic scenario). There can be also any number of alternate scenarios which end either with success or failure in reaching the goal. Every scenario is a sequence of actions forming a dialogue between the primary actor and the system. Every such action can be expressed by a single sentence in a simple SVO grammar <sup>1</sup> (see [5] for an original idea). In addition to action sentences, we need to introduce two additional sentence types: condition and control sentences. They are used in a scenario to express the flow of control between alternative scenarios of the same use case as well as between scenarios of different use cases (see [8, 18]).

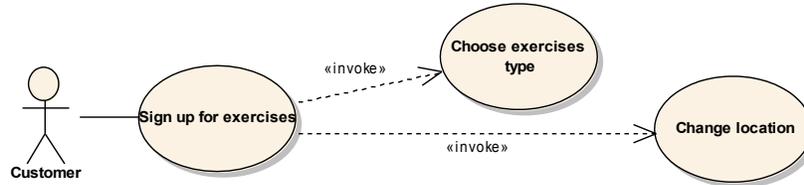


Fig. 1. Simple use case model with “invoke” relationships

Introducing control sentences needs prior redefinition of vague control flow semantics for use cases presented in UML. Figure 1 shows a simple use case model where one use case invokes two another use cases. In this model the “invoke” relationship 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 actions of the base use case scenario. There are two types of invocation: conditional and unconditional. A use case is invoked conditionally when explicitly requested

<sup>1</sup> A sentence in SVO grammar consists of a subject, a verb and an object. Optionally it can have a second indirect object.

by an actor or under a certain condition on the system state. A use case is invoked unconditionally every time the scenario of the base use case containing appropriate invocation sentence is performed. The type of the invocation, the name of a use case to be invoked and the exact point of invocation is defined by a special kind of control sentence in the invoking use case scenario.

Now we will describe the semantics of specific scenario sentences that is common for all use case representations. Figure 2 shows a fragment of the meta-

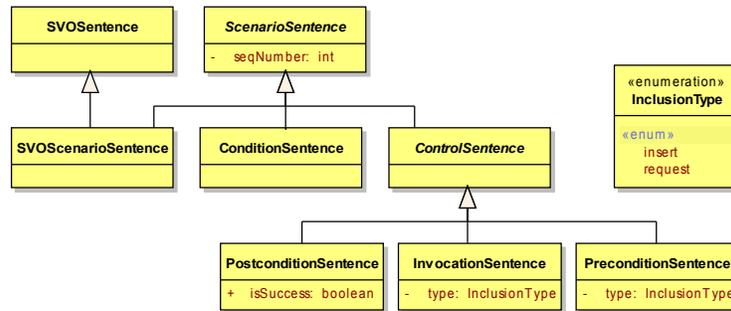


Fig. 2. Metamodel for scenario sentences

model that deals with scenario sentences. There are three main types of sentences that can be used in a scenario: *SVOScenarioSentence*, *ConditionSentence* and *ControlSentence*. All these sentences are subtypes of an abstract *ScenarioSentence* which has a sequence number that defines the sentences position in the scenario. *SVOScenarioSentence* describes a single scenario step (an action) in the form of a sentence in the SVO grammar – it derives the whole syntax from *SVOSentence* described in details in the following section. A scenario step represented by this sentence can be performed by an actor or by the system.

As every action can be performed under certain condition, we need to introduce *ConditionSentence* which is a special kind of *ScenarioSentence* that controls the flow of scenario execution. *ConditionSentence* is a point of conditional control flow: the following scenario step can be executed only when the condition expressed by the sentence is met, otherwise a sentence from alternative scenario is executed.

Another type of a scenario sentence we need is a *ControlSentence*. It is a type of *ScenarioSentence* which controls the flow of scenario execution in situations when the control enters or leaves a use case. This abstract class is a superclass for concrete classes: *PreconditionSentence*, *PostconditionSentence* and *InvocationSentence*. *PreconditionSentence* is an initial sentence of every use case scenario indicating where the flow of control of every use case scenario starts. There are two types of *PreconditionSentences*: *insert* and *request* defined in *InclusionType* enumeration. *PreconditionSentence* of type *request* is always performed when the actor triggers a use case directly or requests invoking a use case (see *InvocationSentence*) from another use case scenario through initial actor action

(first `SVOScenarioSentence` in the scenario). When a 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 scenario.

`PostconditionSentence` is a final sentence of every scenario. It indicates if the goal of a use case has been reached or not.

`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 executing its scenario sentences every time the flow of control reaches the point of invocation in invoking use case scenario. `request` means that the actor or the system explicitly requests invoking another use case – it depends either on the actor’s decision or system state (a certain condition is met) whether scenario sentences of invoked use case will be executed 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. `InvocationSentences` are related to appropriate `PreconditionSentences` in invoked use case’s scenario.

## 2.2 Sentences in SVO grammar with separated domain vocabulary

Sentences in the SVO grammar, as mentioned above, are used in scenarios to express actions performed by the actor or by the system. Experience shows that this simple grammar is sufficient to express interactions in a scenario, eg. “Customer submits sign-up for exercises” or “System signs up customer for exercises”. This grammar combines informality with necessary precision (see [15]). The biggest strength of this grammar, is that SVO sentences only allow for describing the behaviour – no interleaving of domain element definitions are allowed. Such a separation makes requirements specifications unambiguous and consistent.

Considering the above, we need means for creating a separate specification of the domain and the way to link notions used in sentences with their definitions in the domain vocabulary. This issue has been resolved in the Requirements Specification Language (RSL) which has been developed recently as a part of the ReDSeeDS project (see [7]). Below we will explain how the separation is done by presenting a simplified metamodel.

A domain element from the domain vocabulary is usually a noun along with its definition in the context of the problem domain. A noun can be preceded by a modifier which can change the meaning of the noun, e.g. “registered user”. In addition to nouns, the domain vocabulary can also contain verbs. Verbs do not have their own definitions - they are related to nouns as their meaning depends on the context of a noun. Verbs are treated as behavioural features of related nouns. For example, “choose exercise” has a different meaning than “choose time from time schedule”.

In order to use such constructs in SVO sentences, we introduced the concept of phrases. Figure 3 shows an `SVOSentence` composed of one `Subject` and one `Predicate`. These two classes are kind of hyperlinks that can be embedded in

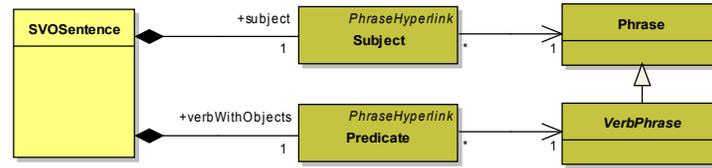


Fig. 3. The structure of SVOSentences

SVOSentences linking it with phrases which are part of domain specification. Phrases are building blocks of SVOSentence and depending on the type, they are composed of nouns (in the role of objects), modifiers, verbs and prepositions (see Figure 4 for phrases metamodel).

Every Phrase consists of at least one noun, optionally preceded by a modifier. Phrases occurs in SVOSentences in the role of 'subject's. In the role of a 'verbWithObjects' there can be used a ComplexVerbPhrase or a SimpleVerbPhrase. SimpleVerbPhrase extends Phrase by adding a verb which precedes the noun (with an optional modifier). This makes it possible to express constructs like "shows registered customers". ComplexVerbPhrase contains a preposition which links the SimpleVerbPhrase pointed by the phrase with a contained noun (again, with an optional modifier). This type of phrase can express constructs like "adds registered customers to customer list", where "registered customer" is a direct objects and "customer list" is an indirect object.

All phrases that refer to the same noun are grouped within a domain element, where the noun is the element's name. Every phrase grouped in a domain element has its own definition. Elements together with their relationships form the domain specification, which should be partially created during the problem domain analysis. While writing scenarios, the writer should be able to browse through the domain specification to search appropriate domain elements and their phrases and insert them directly into scenario sentences. The writer should also be able to add new elements to the domain specification as needed. Such an approach significantly improves the quality of the final requirements specification.

### 2.3 Introducing complementary scenarios representations

Now, as we have precisely defined use case scenarios and all necessary types of scenario sentences, we can introduce complementary representations of use case scenarios:

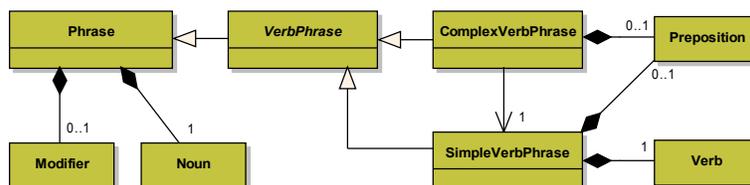
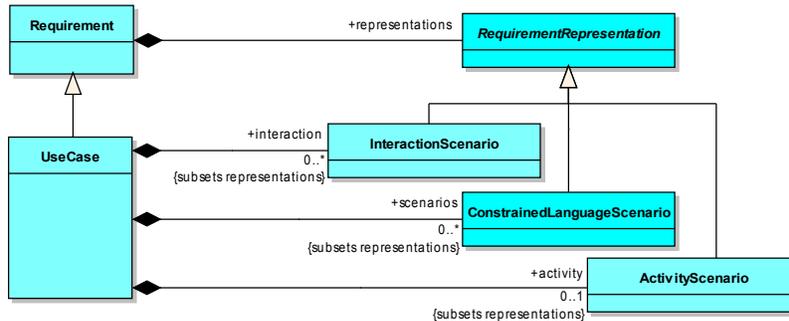


Fig. 4. Phrases metamodel

- constrained language representation,
- activity representation,
- interaction representation.

Each of these representations is capable of expressing exactly the same scenarios but it puts emphasis on some aspects of interactions while suppressing some others, thus making it usable for diverse groups of people having different roles in the software project. Due to the precise metamodel, every representation is directly transformable one into another.

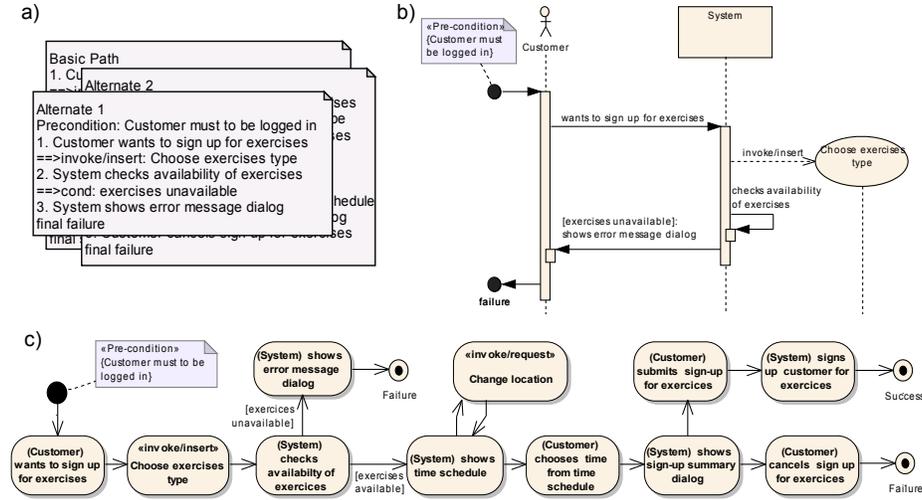
The constrained language representation is a purely textual representation, where scenarios are written as sequences of numbered sentences in the SVO grammar, interlaced with condition and control sentences. A single scenario represents a single story without alternative paths. This representation is most readable for “ordinary people” like users or stakeholders who are usually reluctant to any technical notation. Some people, usually analysts, prefer precise structure for use case scenarios in a graphical form. Activity representation shows all scenarios of a single use case (main path and all alternative paths) as one activity diagram. This precisely reflects the flow of control in a use case as a single unit of functional requirement. Interaction representation presents a single scenario as a sequence of messages send between the system, the actors and other use cases in the form of a sequence diagram. This representation clearly reflects temporal interaction of the actors with the system as well as actions performed by the system in response to the actors’ interaction. It seems to be most suitable for designers as it prepares them for transformation into design level interaction diagrams.



**Fig. 5.** Three complementary representations of use case scenarios

The metamodel in Figure 5 shows metaclasses representing the three introduced representations of use cases. `UseCase` derives from the `Requirement` metaclass, which can have any number of `RequirementRepresentations` (abstract metaclass). Such a separation of requirements and their representations gives us the possibility of representing requirements in different forms depending on the

current needs. For example, draft requirements can be represented with a natural language description while detailed requirements can be represented in more formal way, e.g. a constrained language description with relation to the domain vocabulary. Representations of a use case are concrete subclasses of Requirement-Representation and they subset representations pointed to from a Requirement.



**Fig. 6.** Example of concrete syntax for all complementary representations of the same scenario: a) constrained language b) interaction c) activity

Figure 6 shows examples of concrete syntax for these three representations. Figure 6a presents one of scenarios of a use case in constrained language. In Figure 6b, the interaction representation of the same scenario is shown. We can see that all sentences from the constrained language scenario have their equivalents in the interaction scenario in the form of messages between lifelines. The activity scenario representation in Figure 6c shows all scenarios of the use case in one diagram. Scenario sentences are presented as activity nodes or edges (in case of condition sentences). For comparison of concrete syntax for particular scenario sentence types for all three representations please refer to Table 1.

Figure 7 shows details of SVO sentence's subjects and predicates. In constrained language representation, both subject and predicate, while using an appropriate tool, could be shown as hyperlinks leading to their descriptions in the domain vocabulary. In activity representation predicate is delineated as an activity action's name, while subject is represented as an activity partition – subject name is placed in parenthesis above the action name. In interaction representation subject of a sentence is depicted as a lifeline (see Customer's and System's lifeline in the example). Predicate of every SVO sentence, in turn, is depicted as a message with its name above. It starts from the subject's lifeline

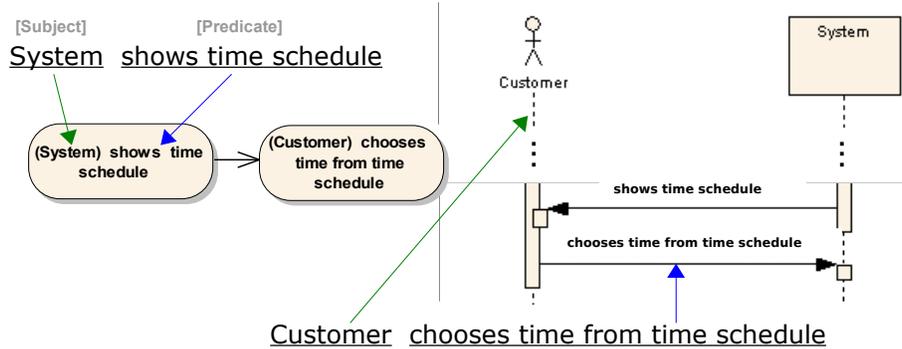


Fig. 7. Concrete syntax for SVO subject/predicate

and goes either to another subject’s lifeline or comes back to the same lifeline it starts from.

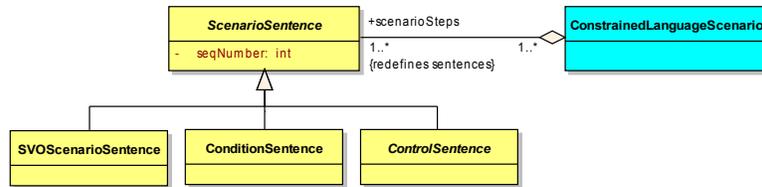
Condition and control scenario sentences have also different concrete syntax in different representations (see Table 1). A condition sentence in textual representation is a text expressing the condition preceded with “ $\Rightarrow$ cond:” prefix. In activity and interaction representation it is shown as a text in square brackets attached to a control flow edge in activity diagram or to a message in sequence diagram. Sentences of invoke type are presented in textual representation as invoking use case name preceded with a prefix denoting the type of invocation: “ $\Rightarrow$ invoke/request:” or “ $\Rightarrow$ invoke/insert:”. In activity representation, this type of sentences is depicted as an action node with the name of invoking use case inside and an appropriate stereotype: “invoke/request” or “invoke/insert”. In case of insertion, the incoming control edge comes from preceding action and the outgoing control edge goes to the action representing consecutive scenario sentence. In case of request, the outgoing control edge goes back to the action the incoming edge came from. In interaction representation, both invoke sentences are presented as “create” message from the actor’s or system’s lifeline to the invoking use case’s lifeline. The type of invocation is shown as the message’s text. Precondition and postcondition sentences are presented in textual representation as a text with “precondition:” or “postcondition:” prefix. In two another representations, sentences of these types are shown as constraints attached to start/final node in case of activity or lost/found message in case of interaction representation. In all representations, a postcondition sentence contains also “success” or “failure” text, indicating whether the goal of a use case has been reached or not.

### 3 Metamodel of complementary scenarios representations

In the following sections we introduce the metamodel defining the abstract syntax for three complementary representations of a use case scenarios. This metamodel is expressed by the means of MOF ([12]). It refers to elements defined in

Sentence	Textual	Activity	Interaction
SVO	System shows time schedule		
condition	$\Rightarrow$ cond: exercises available		
invoke/insert	$\Rightarrow$ invoke/insert: Change location		
invoke/request	$\Rightarrow$ invoke/request: Change location		
precondition	precondition: Customer must to be logged in		
postcondition	final: success postcondition: Customer is registered for chosen exercises		

**Table 1.** Examples of concrete syntax for particular scenario sentences in different use case representations



**Fig. 8.** Constrained language scenario representation

UML 2.0 Superstructure ([11]), mainly from BasicActivities and BasicInteractions packages. Due to the scope of this paper, we present only the most essential constructs from our metamodel. Some high-level elements of this metamodel have already been introduced in previous sections. For more details, please refer to the language specification ([7]).

### 3.1 Constrained language scenario representation

Constrained language scenario representation can be treated as a basis representation. Its syntax, both concrete and abstract, is the least complex in comparison with two remaining representations which extend the basic syntax mainly by specialising from UML elements.

As shown in Figure 8, `ConstrainedLanguageScenario` is composed of `ScenarioSentences` as its `scenarioSteps`. `ScenarioSentence` is an abstract metaclass that defines `seqNumber` which is an order number of a sentence in a scenario. This general metaclass is a base for subclasses representing scenario sentences of specific types like `SVOScenarioSentence`, `ConditionSentence` and `ControlSentence`. The abstract syntax as well as the semantics of these metaclasses has been described in section 2.1. Examples of the concrete syntax for metaclasses forming constrained language representation as well as other representations are shown in Figure 6 and in Table 1.

### 3.2 Activity scenario representation

Activity scenario representation utilises UML activity diagrams to present use case scenarios. In this way, it allows for showing all possible scenario paths of a single use case in one picture. In order to utilise activity diagrams for this purpose, elements of the metamodel for this representation specialise from appropriate UML elements defined in `BasicActivities` and `IntermediateActivities` packages (see [11]).

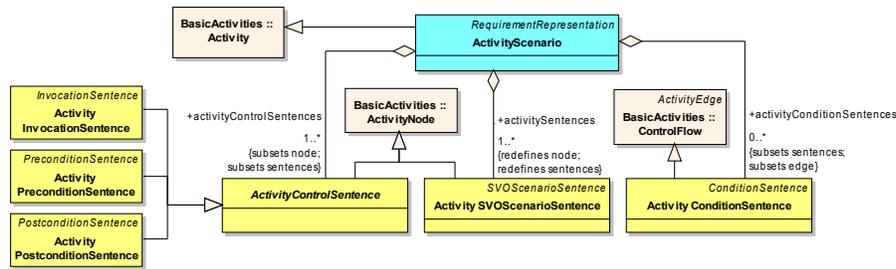
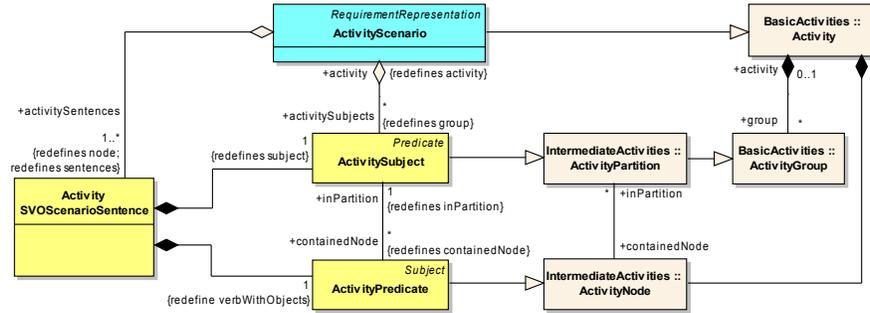


Fig. 9. Activity scenario representation

`ActivityScenario` extends `Activity` and it can contain three types of scenario sentences. Metaclasses representing these sentences combine syntax and semantics derived from sentences metaclasses from constrained language representation and from appropriate UML metaclasses (Figure 9).

An `ActivitySVOScenarioSentence` represents a single scenario action in the form of `ActivityNode` which it derives from. It means that these actions can have incoming and outgoing `ControlFlows` showing possible execution sequences. `ActivitySVOScenarioSentence` also derives from `SVOScenarioSentence` what means that the action it represents is described by SVO sentence (see Figure 10). `ActivitySVOScenarioSentence` redefines `subject` and `verbWithObjects` derived from `SVOsentence` with `ActivitySubject` and `ActivityPredicate` respectively. Redefined `ActivitySubject` derives from UML's `ActivityPartition` which is a kind of activity group for identifying actions that have some characteristic in common. In case of scenarios, it identifies whether an action is performed by an actor or by the

system. In the diagram, subject is represented as a text in parentheses, placed inside action above its name (see Figure 7).



**Fig. 10.** Realisation of Subject and Predicate in SVO sentence in activity representation

An `ActivityControlSentence` represents `ControlSentence` in `ActivityScenario` in the form of `ActivityNode` which is its superclass. It has three concrete subclasses: `ActivityPreconditionSentence`, `ActivityPostconditionSentence` and `ActivityInvocationSentence`. Each of these three subclasses corresponds to the appropriate `ControlSentences` subclass.

An `ActivityPreconditionSentence` shows the starting point of a scenario on an activity diagram. `ActivityPreconditionSentence` can have the precondition of a scenario attached as a constraint. It has semantics similar to `PreconditionSentence`. Additionally it has the semantics of UML's `InitialNode`.

The end point of a scenario on activity diagram is represented by an `ActivityPostconditionSentence`. It shows, if a scenario ends with success or failure. It can also have the postcondition of a scenario attached as a constraint. It has semantics similar to `PostconditionSentence` and additionally it derives the semantics from UML's `FinalNode`.

In activity representation, points where another use cases scenarios are invoked are shown as `ActivityInvocationSentence`. `ActivityInvocationSentence` is a subtype of `ActivityControlSentence` and `InvocationSentence`. In addition to that, it indirectly specialises from `ActivityNode` which makes it possible to be presented on activity diagrams.

### 3.3 Interaction scenario representation

Interaction representation of use case scenarios aims at emphasising the temporal sequence of interaction messages exchanged between an actor and the system as described in section 2.3. To achieve this, interaction representation utilises UML's sequence diagrams to represent scenarios. This means that elements of the metamodel for this representation specialise from appropriate UML elements defined in `BasicInteractions` packages (see [11]).

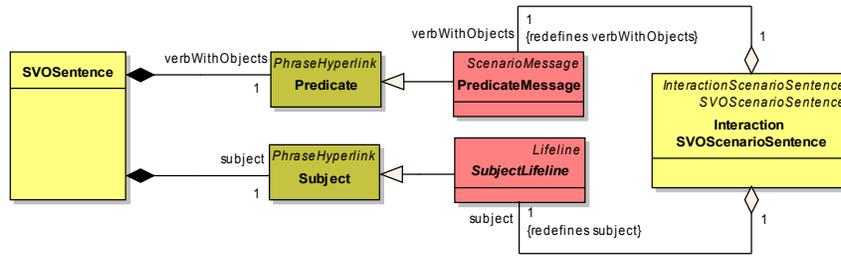


Fig. 11. Interaction SVO Scenario Sentence

Figure 11 presents a fragment of the metamodel, handling InteractionSVO-ScenarioSentence. It derives from SVOScenarioSentence and is composed of SubjectLifeline and PredicateMessage. Both derive from Subject and Predicate respectively and redefine subject and verbWithObjects. Realisation of scenario sentences is based on UML’s Interaction model. Interaction has a set of Messages connected to Lifelines by MessagesEnds (see Figure 12). Every message is depicted as an arrow pointing from one lifeline to another or to the same lifeline.

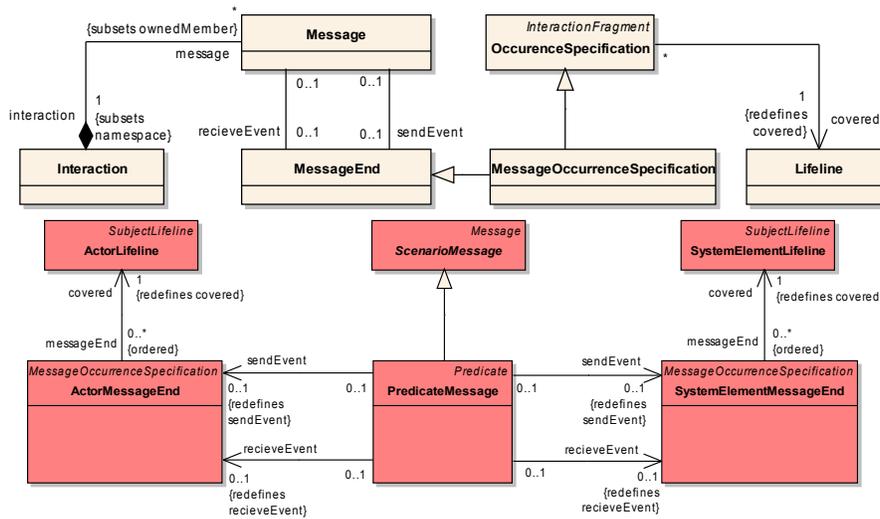


Fig. 12. Realisation of Predicate in interaction representation

Subject of InteractionSVOScenarioSentence acts as a lifeline in a sequence diagram. SubjectLifeline derives from UML’s interaction lifeline. This metaclass is abstract and it is the base class for two concrete subclasses: ActorLifeline and SystemElementLifeline. The first metaclass represents subjects of sentences performed by use case’s Actor. The latter, corresponds to subjects of sentences

performed by `SystemElement`. Both are kind of `DomainElement` (see Figure 13 for more details).

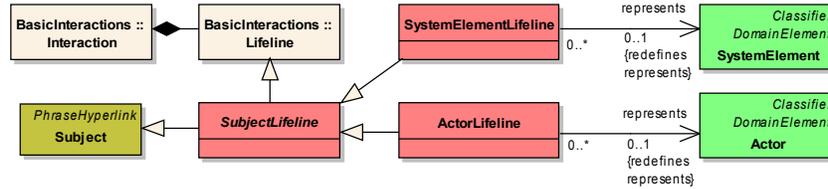


Fig. 13. Realisation of SVO Subject in interaction representation

Predicate of `InteractionSVOScenarioSentence` acts as a message. It indirectly derives from `Interaction`'s `Message`. Predicate message can be connected to `SystemElementLifeline` or actor's lifeline (through respective `MessagesEnds`). As it is shown in Figure 12, Predicate has `sendEvent` and `receiveEvent` redefined with `SystemElementMessageEnd` and `ActorMessageEnd`. Those two message ends are covered by `ActorLifeline` and `SystemLifeline`.

## 4 Conclusions and future work

Presented notations for use case scenarios as part of the requirements specification language, has already been validated by students during software engineering courses at the Carlos III University of Madrid and Warsaw University of Technology. The results of case studies prepared by students showed that proposed approach makes capturing and specifying of system requirements easier. Students had no problems with understanding and discussing the resulting models. Created requirements specifications appear to be much more consistent and precise than specifications written in natural language. They also appear to be a good basis for further development process. The next step to prove usability of the language is validation in real-life projects by the industrial partners involved in the ReDSeeDS project. This validation is currently in progress.

Full utilisation of capabilities of proposed language calls for a tool support. A tool supporting a simplified concept behind the RSL was already developed (see [16]). Development of a tool covering the whole RSL language is in the scope of the ReDSeeDS project.

**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 Sistemas. Initial interaction representation metamodel

was proposed by University of Koblenz-Landau team (J. Ebert, V. Riediger, D. Bildhauer, H. Schwarz).

## References

1. Ian Alexander. A taxonomy of stakeholders, human roles in system development. *International Journal of Technology and Human Interaction*, 1,1:23–59, 2005.
2. Ian Alexander and Neil Maiden. *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*. John Wiley, 2004.
3. Alistair Cockburn. Structuring use cases with goals. *Journal of Object-Oriented Programming*, 5(10):56–62, 1997.
4. Gonzalo Genova, Juan Llorens, Pierre Metz, Ruben Prieto-Diaz, and Hernan Astudillo. Open issues in industrial use case modeling. *Lecture Notes in Computer Science*, 3297:52–61, 2005.
5. Ian M Graham. Task scripts, use cases and scenarios in object-oriented analysis. *Object-Oriented Systems*, 3(3):123–142, 1996.
6. R R Hurlbut. A survey of approaches for describing and formalizing use cases. Technical Report XPT-TR-97-03, Expertech Ltd., 1997.
7. 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](http://www.redseeds.eu).
8. P Metz, J O’Brien, and W Weber. Specifying use case interaction: Types of alternative courses. *Journal of Object Technology*, 2(2):111–131, March-April 2003.
9. Pierre Metz, John O’Brien, and Wolfgang Weber. Against use case interleaving. *Lecture Notes in Computer Science*, 2185:472–486, 2001.
10. Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE - Future of SE Track*, pages 35–46, 2000.
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. Som S S. Beyond scenarios: Generating state models from use cases. In *Scenarios and state machines: models, algorithms and tools - ICSE 2002 Workshop*, Orlando, Florida, 2002.
14. 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.
15. Michał Śmiałek. Accommodating informality with necessary precision in use case scenarios. *Journal of Object Technology*, 4(6):59–67, August 2005.
16. 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.
17. Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, and Tomasz Straszak. Writing coherent user stories with tool support. *Lecture Notes in Computer Science*, 3556:247–250, 2005.
18. K G van den Berg and A J H Simons. Control flow semantics of use cases in UML. *Information and Software Technology*, 41(10):651–659., 1999.