Computing and Informatics, Vol. 34, 2015, 877–910

USE CASE SPECIFICATION USING THE SILABREQ DOMAIN SPECIFIC LANGUAGE

Dušan Savić, Siniša Vlajić, Saša Lazarević
Ilija Antović, Vojislav Stanojević, Miloš Milić

Faculty of Organizational Sciences
University in Belgrade
Jove Ilica 154
11000 Belgrade, Serbia
e-mail: {dules, vlajic, slazar, ilijaa, vojkans, mmilic}@fon.bg.ac.rs

Alberto Rodrigues da Silva

INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
e-mail: alberto.silva@tecnico.ulisboa.pt

Abstract. The software requirements engineering process is a part of a software development process and one of the key processes in software development. The elicitation, analysis, specification and validation of software requirements occur during the requirements engineering process. Use cases are used as a technique for functional system specification. Different notations can be used for a use case specification. In this paper, we present SilabReq Domain Specific Language (SilabReq DSL) for use case specification. On the one hand, we develop this language to describe the use cases in clear and precise way through the meta-model, and on the other hand to specify the use cases to be readable and understandable for all stakeholders in the software development project. This allows us to develop different transformations to get the structure and the behavior of the system from defined use cases. In this paper apart from the SilabReq DSL, we present some of these transformations.

Keywords: Software requirements, use case specification, model transformation, UML, domain specific language

1 INTRODUCTION

The development of information systems is a complex and social process that involves many interactions among different stakeholders. To make this process successful, it is necessary to understand the system requirements and document them in a suitable manner. There are different definitions of requirements, namely:

1. a property that must be exhibited in order to solve some real-world problem [17];
2. needs and constraints placed on a software product that contribute to the solution of some real-world problem [20] or descriptions of the services provided by the system and its operational constraints [35].

Still according to the IEEE definition, a requirement is

1. a condition of capability needed by a user to solve a problem; or
2. a condition or a capability that must be met or possessed by a system to satisfy a contract, standard specification or other formally imposed document [16].

Requirements may be presented in different ways. Herman and Svetinovic [8] make a distinction between requirements and their presentation. They define ontology for the representation of requirements and emphasize two forms of requirement representation as follows: a descriptive requirements presentation and a model based presentation of the requirements. Software requirements are a set of functions that the system should provide for users of the system. Depending on the level of abstraction, we can distinguish between user and system requirements. User requirements are the requirements of high-level abstractions representing functions the system should provide. On the other hand, system requirements are a detailed specification of these functions. Furthermore, both user and system requirements can be functional and nonfunctional. Functional requirements define the required system functions, while non-functional requirements define all other requirements, which are primarily related to quality requirements like usability, reliability and others [35].

The result of the requirements development specification process is a clear and precise specification of the system to be implemented. Since use cases are used as a technique to specify the desired function of the system, the problem of the use case specification and the use case notation are main problems that we consider in this paper. According to the UML, "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 [27]". However, UML does not define the specification of these actions. The focus of this paper is a more detailed specification of these actions.

Model-Driven Development (MDD) is a software development paradigm that emphasizes the importance of models during the entire software development process [24]. The aim of MDD is to use models throughout the software development process at different levels of abstraction. Therefore, models are used not only to document some part of a system; but models are first-citizen in software development. MDD process usually starts by developing a requirements model, which is defined by describing user's needs in a computational independent way. Then, this model can be refined into one or more models that describe the system without considering technological aspects. Finally, these models are either refined into design models that describe the system by using concepts of a specific technology and are then translated into a code; or are directly derived to a code if they contain enough information to implement the software system in a precise and complete way [45].

However, despite the importance of requirements engineering as a key success factor for software development projects [42, 43, 44, 29], there is still a lack of MDD "method" that would cover the full development lifecycle, from the requirements engineering level to the code generation level or writing level [23]. The integration of use cases within the MDD process [25] requires a rigorous definition of the use case specification, particularly description of sequences of actions step, pre- and post-conditions, and relationships between use case models and domain models [4].

In this paper, we propose the *SilabReq* language to specify use cases. *SilabReq* is a domain specific language (DSL) that allows to specify user and system actions in a clear and precise way. This language uses a text specific syntax. By defining the use case actions in a clear and precise way, the use case model is described more formally. Additionally, by applying transformations from the source use case model, it is possible to get different models that can describe the structure of the system, or the system behavior as a set of functions that the system should provide. The paper is organized as follows. Section 2 presents the related works. Section 3 describes *SilabReq DSL*, namely its concrete and abstract syntax. Section 4 explains different system models that can be created from the *SilabReq DSL* use case model. The models are obtained as a result of the use case model transformation which is described by *SilabReq DSL*, in particular models describing the system boundary, or its structures and the behavior of the system, as well as models which can be used for visual monitoring of the use cases execution. The usage of the *SilabReq DSL* is shown in some use cases in the system of the electronic office.

2 RELATED WORK

Requirements are mostly documented using natural languages as structured paragraphs of text. However, natural language requirements specification tends to be ambiguous, unclear, and inconsistent [35]. In fact, it is difficult clearly define data, function and behavior perspectives of these requirements because they overlap. On the other hand, documenting requirements using semi-formal models requires a specific modeling language (such as UML, SysML, ReqIF, ANDORA) for each particular perspective.

The specification of requirements is a difficult task because different stakeholders with different technical knowledge read the requirements. People prefer to use textual specification of requirements, but these representations are not suitable for automatic transformation or for reusing. We need a structured language for requirements specification that should be understandable by most stakeholders but also that should be precise enough to enable automatic transformations. That means this language should be defined by meta-model or grammars in order to enable automatic or semi-automatic processing.

UML has become a standard language for modeling software systems and many people have used it for requirements specification. However, some authors have argued that UML has some deficiencies as a semiformal requirements specification language [7]. For example:

1. UML specification defines a use case as a sequence of actions, but the specification of actions is not clearly defined, and consequently, different notations are used to describe these actions; or
2. UML uses different types of diagrams (e.g. interaction diagram or activity diagram) to describe the interactions within a use case. On the other hand, different authors such as Rolland [31], Cockburn [6], and Li [22] have also proposed textual descriptions of use cases.

Different forms or templates to specify use cases defined by different authors usually contain common elements that can be found in almost all templates. In practice, two templates emerged as the most commonly used:

1. Cockburn's use case template [6], and
2. Rational Unified Process use case template [18].

Smiałek suggests a different notation for the use case description [40]. He suggests that we need to have different notations for different stakeholders and he relates certain notation of use cases with user's role in software development process. He emphasizes that the ideal notation for the use cases description should be "rich" because different stakeholders have different views on the use case. Smiałek suggests several views of the use case, namely:

1. user's point of view,
2. analyst's point of view,
3. designer's point of view,
4. user interfaces designer's point of view and
5. tester's point of view.

Smiałek proposes five different notations for the use cases description that are based on structured text, interaction diagrams and activity diagrams. He also defines a meta-model for structured textual representation of use cases that is based on simple grammatical sentences. These sentences are in the form of "subject-verb-direct object", but can also appear in the form of "subject verb-indirect object" [40].

Furthermore, formalisms based on formal specification languages, such as Petri nets [19] or Z [36] have been used for the use cases specification. The main blemish of formal notations is that they are very difficult to be understood by non-technical stakeholders.

There are requirements specification languages (RSL) that use the natural language in a controlled way. RSL [37] is a semiformal natural language that employs use case for specifying requirements. RSL has been developed as a part of the ReDSeeDS project [38]. ReDSeeDS approach covers a complete chain of model-driven development: from requirements to coding [39]. Olek et al. [30] developed language called ScreenSpec that can be used to quickly specify screens during the requirements elicitation phase.

Some [41] defines the abstract syntax of a textual presentation of use cases. Some emphasizes that certain elements (formalisms) of the UML languages such as actions or activities are formally defined through a meta-model. He defines a meta-model to describe the interaction between system users and the system. His work was inspired by a variety of guidelines how to write use cases and various defined forms (templates) [5, 11].

The goal of ProjectIT [32, 34] is to provide a complete software development workbench with the support for a project management, requirements engineering, analysis, design and code generation activities. ProjectIT-Requirements is the component of the ProjectIT architecture that deals with requirements issues. The main goal of the ProjectIT-Requirements is to develop a model for the definition and documentation of requirements which, by raising their specification rigor, facilitates the reuse and fastens the integration with development environments driven by models. When different types of requirements are taken into account, this project uses software requirements, that can be more easily "converted" into software design models by MDD approaches [12, 33].

RSLingo [13] is a linguistic approach for improving the quality of requirements specification based on two languages and a mapping between them. The first language is the RSL-PL (Pattern Language), an extensible language for defining linguistic patterns dealing with information extraction from requirements written in the natural language [10]; and the second one is RSL-IL (Intermediate Language), a formal language with a fixed set of constructs for representing and conveying RE-specific concerns [14].

UML tools like IBM Rational Software Architect or EclipseUML do not support the description of the internal structure of use cases. On the other hand, tools focusing on textual use cases descriptions like CaseComplete leave out the use case constructs defined by the UML specification. NaUTiluS [9] presents an extensible, Eclipse-based toolkit, which offers integrated UML use case modeling support, as well as editing capabilities for their textual descriptions. NaUTiluS consists of a set of plug-ins that are embedded in the ViPER platform.

The subject of the research presented in this paper is a specification of the functional systems requirements described by the use cases. Since the use cases are defined as a sequence of actions between one or more system users and systems, the subject of this paper is how to specify these actions. Use cases contain one main scenario and zero or more alternative scenarios while each scenario contains one or more use case actions. We divide these actions in two categories:

1. actions performed by the users and
2. actions performed by system and formally described by using SilabReq Domain Specific Language.

Both categories contain different types of actions. In the category where the user performs actions, we identified actions types such as:

1.1 Actor Prepare Data to execute System Operation (APDExecuteSO) and

1.2 Actor Calls System to execute System Operation (ACSExecuteSO).

On the other hand, in the category in which actions are performed by the system, we identified two action types such as:

2.1 System executes System Operation (SExecuteSO) and

2.2 System replies and returns the Result of the System Operation execution (SR-ExecutionSO).

These actions are specified in requirements specification document using SilabReq DSL. SilabReq DSL has been developed using Xtext framework [35]. By defining the use case actions in a clear and precise way, the use case model is described more formally. In this way, by applying different transformations of the use case model, it is possible to get different models that can describe system boundary, the structure of the system or system behavior as a function that the system should provide.

3 LANGUAGE FOR USE CASE SPECIFICATION

This section is an overview of the *SilabReq DSL*, namely of its history, concrete and abstract syntax.

3.1 About Silab Initiative

Silab initiative was initiated in the Software Engineering Laboratory at the Faculty of Organizational Sciences, University of Belgrade, in 2007. The main goal of this project was to enable automated analysis and processing of software requirements in order to achieve automatic generation of different parts of a software system.

Initiative *Silab* was divided in *SilabReq* and *SilabUI* projects that were developed separately. *SilabReq* project considered the formalization of user requirements and their transformations into different UML models in order to facilitate the analyses process and to assure the validity and consistency of software requirements. *SilabReq* language is the main part of this project. On the other hand, *SilabUI* project considered impacts of the particular elements of software requirements and data models on resulting user interfaces in order to develop a software tool that enables automatic generation of user interfaces based on the use case specification and the domain model.

When both subprojects reached the desired level of quality, they were integrated in a way that some results of *SilabReq* project can be used as an input for *SilabUI* project. As a proof of concept, *Silab* initiative has been used for the Kostmod 4.0 project, which was implemented for the needs of the Royal Norwegian Ministry of Defense [3].

The *SilabReq* project includes the following components (Figure 1): *SilabReq Language*, *SilabReq Transformation* and *SilabReq Visualization*.

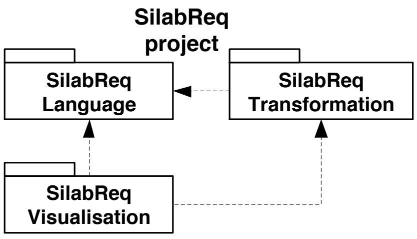

Figure 1. Main components of the SilabReq project

The *SilabReq* language is a controlled natural language for the specification use cases. The *SilabReq* transformation is responsible for transforming software requirements into different models. Currently, we have developed transformations that transform a *SilabReq* model into an appropriate UML model. These transformations are:

1. *SilabReqConceptModel* transformation generates domain model (T1),
2. *SilabReqSystemOperation* transformation generates system operations,
3. *SilabReq* transformation generates UML use case model (T3),
4. *SilabReq-Sequence* generates UML sequence model,
5. *SilabReq-StateMachine* generates UML state-machine model (T2) and,
6. *SilabReq-Activity* generates UML activity model (T4).

Figure 2 presents some of these transformations.

All these transformations are defined through Kermeta language for meta-modeling. Kermeta is a model-oriented language where the meta-model is fully compatible with OMG Essential Meta-Object Facility (EMOF) meta-model [28] and Ecore meta-model, and is part of the Eclipse Modeling Framework (Eclipse Modeling Framework EMF) [2].

The *SilabReq visualization* component is responsible for visual presentation of the specified software requirements. Therefore, we can present *SilabReq* use cases specification through UML use case, UML sequence, UML activity or UML state-machine diagram.

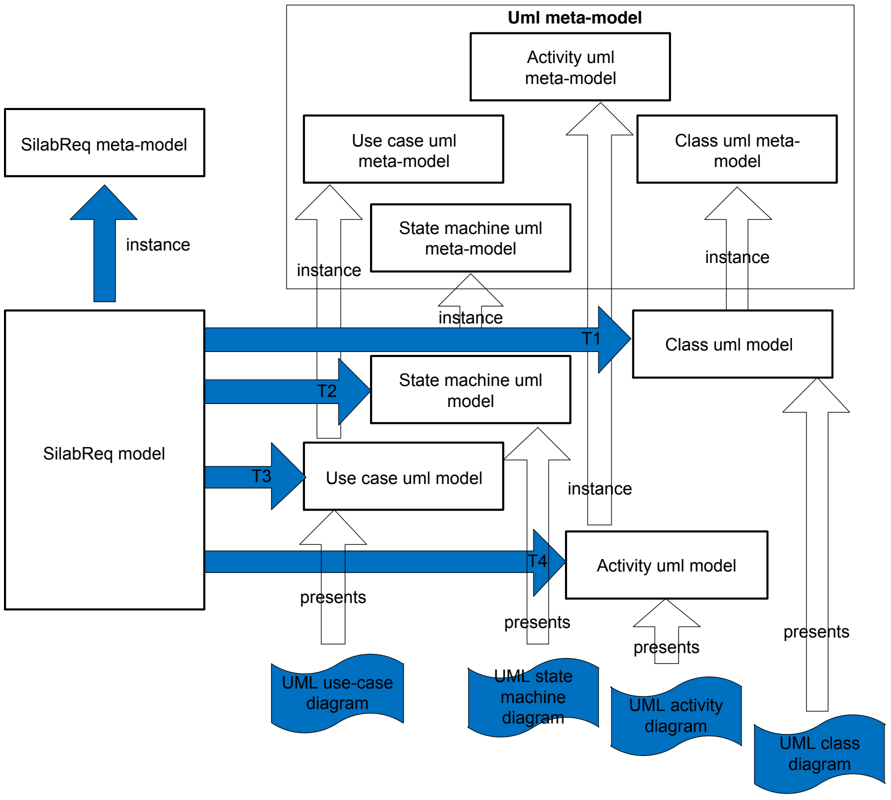

Figure 2. SilabReq transformations

3.2 Defining the SilabReq Language

Kleppe defines the language as "a set of rules according to which the linguistic utterances of L are structured, optionally combined with a description of the intended meaning of the linguistic utterances" [21]. In accordance to the Kleppe definition, when defining a programming language it is necessary to define the rules that are used to create a different structure of the language (language syntax), while defining the meaning of the terms (semantics of the language) is optional. Defining the syntax of the language involves defining concrete and abstract syntax of the language. The abstract syntax of the language describes the concepts that appear in the language and their relationships regardless of a manner of their presentation. On the other hand, the concrete syntax provides a representation of the language concepts defined by using abstract syntax, which allows us to use them to create a user-friendly language expression.

Different approaches for languages definition use different formalisms to define abstract and concrete syntax of these languages. Fondement and Baar [15] used the formalism meta-model for definition of both abstract and concrete syntax. The concrete syntax can be described by a separate meta-model, while the connection between the elements of the meta-model of concrete syntax and the elements of the meta-model of abstract syntax is realized through the model transformation. For example, XText framework uses meta-model and grammar formalisms. This framework uses the BNF grammar for describing the concrete syntax of languages. On the other hand, based on the BNF grammar, the XText framework creates a meta-model that describes the abstract syntax of the language.

SilabReq DSL was developed with XText framework, which is a framework for the development of both domain-specific and programming languages that use the text concrete syntax. This framework is based on openArchitectureWare generator framework, the Eclipse Modeling Framework and ANTLR (Another Tool for Language Recognition parser generator) [1]. *SilabReq* definition starts with definition of the context free grammar of the language that is described using the extended Backus Naur form.

The remainder of this section presents both the abstract and the concrete syntax of *SilabReq* language. The abstract syntax is described by the formalism meta-model, while the concrete syntax is described through XText grammar.

3.3 Defining the Syntax of SilabReq Language

A use case model includes the use cases, actors of the system and the relationship between them. *SilabReq* language extends the use case model with a concept that we call the *Concept* (see Figure 3).

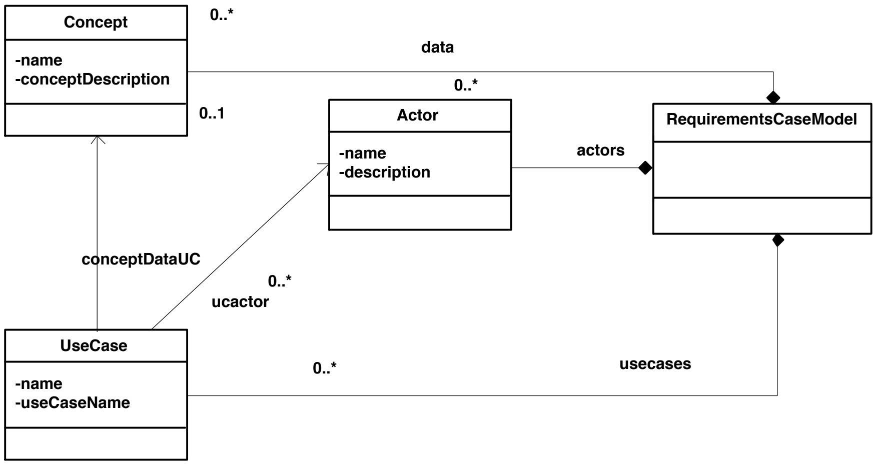

Figure 3. Use case model in SilabReq DSL

The *Concept* presents business entity and it was introduced to find objects and/or their properties (attributes). The founded objects are candidates for the domain (conceptual) class.

Furthermore, the *Concept* is introduced to specify the data that the user enters on one hand, and to specify the data that the system shows to user, on the other hand.

Nakatani et al. [26] say that most use cases that are identified in business system, are primarily related to the CRUD operations that are executed over the domain objects. According to their study, and based on our experience in developing software systems, we noticed that each use case is connected to a single entity or business entity, over which the observed (current) use case is executed.

Therefore, we created the following grammar defining the use case model using the *RequirementsUseCaseModel* rule, the user over the *Actor* rules, the concept over the *Concept* rule and the use case through *UseCase* rule. Below is the grammar of the *RequirementsUseCaseModel* rule.

```
RequirementsUseCaseModel:
        (actors+=Actor)+
        (data+=Concept)*
        (usecases+=UseCase)+;
Actor:
        "Actor:" name=ID description=STRING? ;
Concept:
        "DataConcept:" name=ID conceptDescription=STRING?;
```

In *SilabReq DSL* a use case is defined by using the scenario that consists of one or more blocks of actions (Figure 4).

The use case scenario is defined through *UseCaseFlow* rule, while the block of actions in the scenario is defined by *CompleteActionBlock* rule. The use case scenario consists of one or more blocks of actions. Each block of actions contains:

1. Actions performed by the user of the system, which are defined by the *UserActionBlock* rule and,
2. Actions performed by the system, as defined by *SystemActionBlock* rule.

UseCase rule is used to define the use cases. The use case definition begins by specifying a unique identifier and the use case name. Both attributes are required. After that, the use case definition continues by stating:

- Users of the system (*ucactor*) who participate in use case and
- *Concept* (*conceptDataUC*) in which a use case is executed.

The number of users who use a certain use case can be one or more. The user of the system and the *Concept* are defined in the use case model, so we just make a choice and establish the links (cross references) to a defined object (concrete user or concrete concept). The use case definition is finished with the definition of the use case scenarios over *UseCaseFlow* rule.

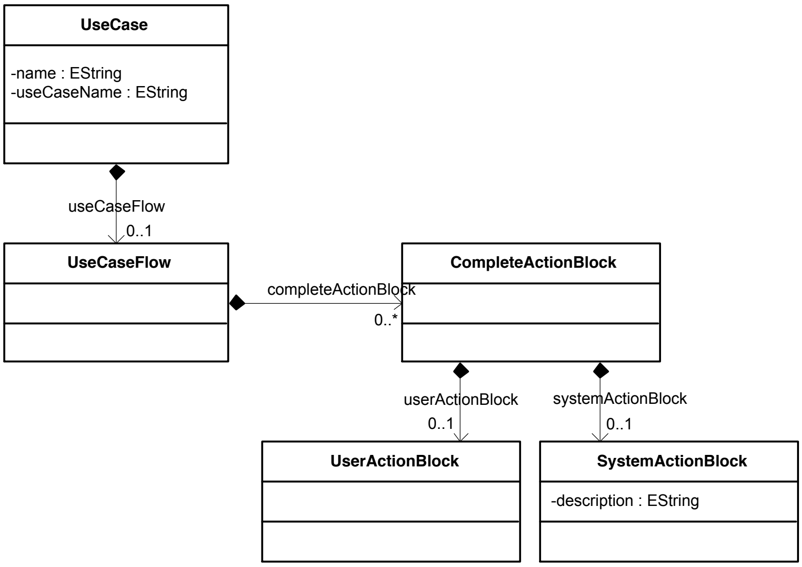

Figure 4. UseCaseFlow with CompleteActionBlock

The *CompleteActionBlock* rule is used to define a block of actions to execute by the user of the system described by *UserActionBlock* rules and actions performed by the system, described by *SystemActionBlock* rule. This block of actions is seen as a whole and each block of actions must contain an action performed by the user of the system and actions performed by the system.

```
CompleteActionBlock:
        (userActionBlock=UserActionBlock)
        (systemActionBlock=SystemActionBlock);
```

3.3.1 The User Actions

The user actions are described by *UserActionStepType* rule. This rule is an abstract parser rule that is implemented through:

- *DirectiveStep* rule that we use to describe UML include relationship (described using *Include* rule) and UML extend relationship (described using *ExtensionPoint* rules) between the use cases, or
- *UserActionStep* rules that we use to describe the actions performed by the user as we have defined using the *ActionUserActionStep* rule, and actions that we use to describe the control structure defined by *ControlUserActionStep* rule.

The Figure 5 describes meta-model of the action that are executed by the user.

The user immediately executes two types of actions: actions that the user uses to prepare data for execution of the system operations, which is defined by *APUSOActionStep*

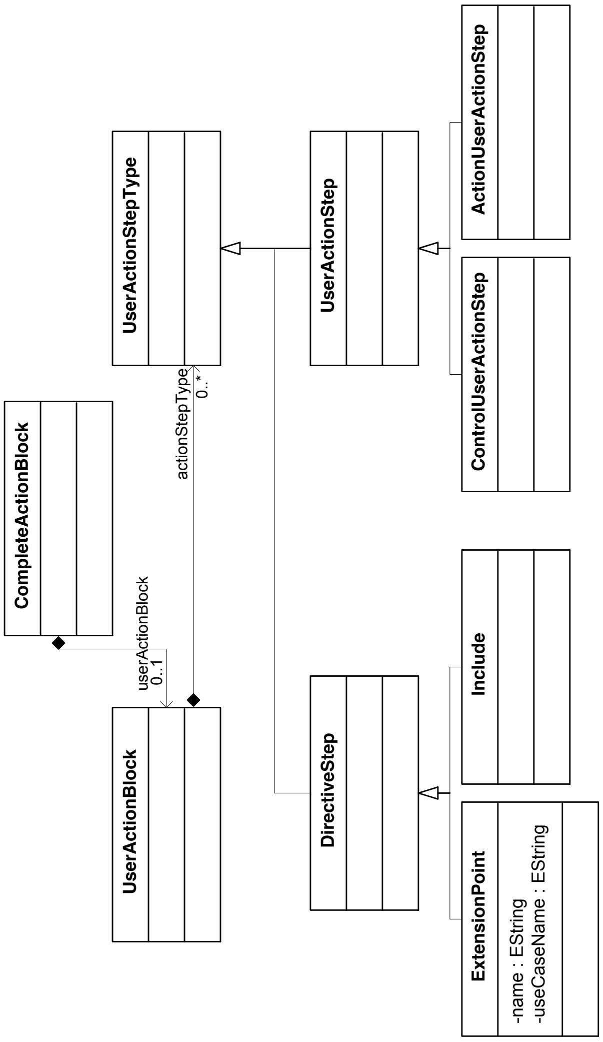

Figure 5. Meta-model of the user actions

rule, and the actions which the user uses to call system to perform the system operations, defined by the *APSOActionStep* rule. The actions that the user performs can be executed iteratively, so we have defined *UserIterateActionStep* rule to describe it. Also, the actions that the user performs can be executed under certain conditions for which we have defined *UserIFActionStep* rule. The Figure 6 shows the meta-model of these actions.

When we define the action that the user uses to prepare data for executing the system operations (*APUSOActionStep*), we use Data rules to describe these data. The Figure 7 describes the meta-model for the description of these data.

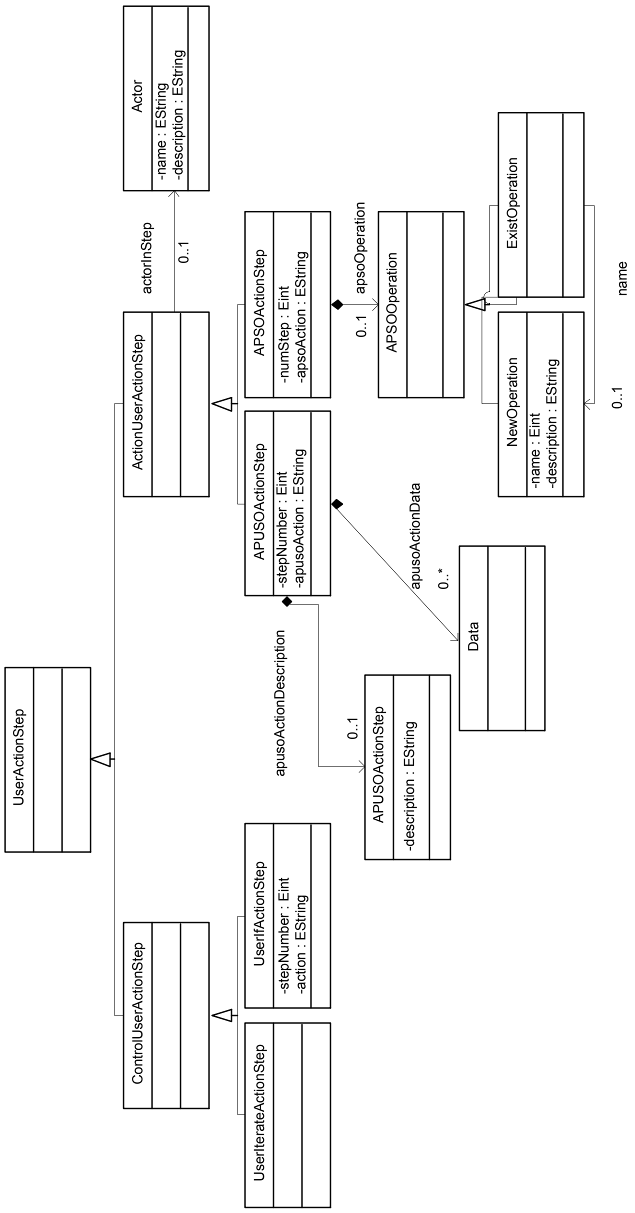

Figure 6. Meta-model of action that user executes immediately

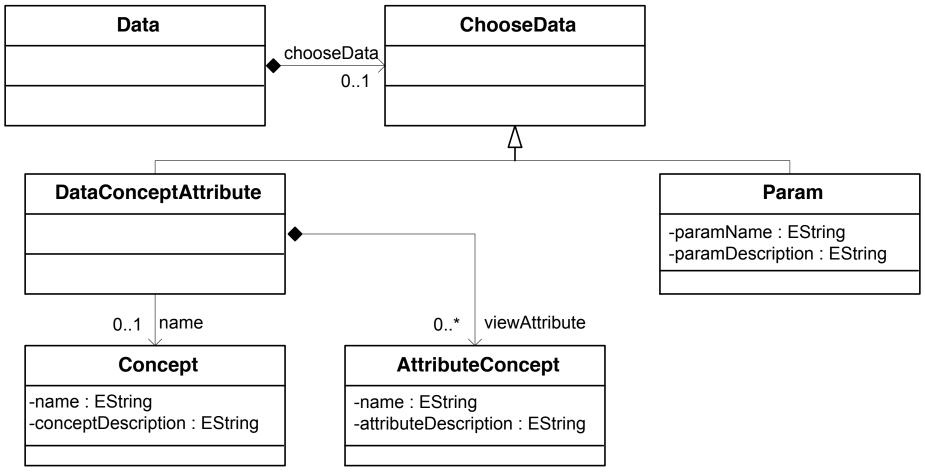

Figure 7. Meta-model of Data rules

The grammar of the *SilabReq DSL* contains two rules used to define the description of the different types of data that the user enters. The *DataConceptAttribute* rule describes the data entered by the user, which are candidates for the domain concepts, and domain attributes of the concept. The *Param* rule specifies other data entered by the user, which are required for execution of the system operations.

When we define the action that the user uses to call the system to execute the system operation that is defined by *APSOActionStep* rule, the user specifies the operation that the system should execute (name and description of the operation). The Figure 8 describes the meta-model of this action.

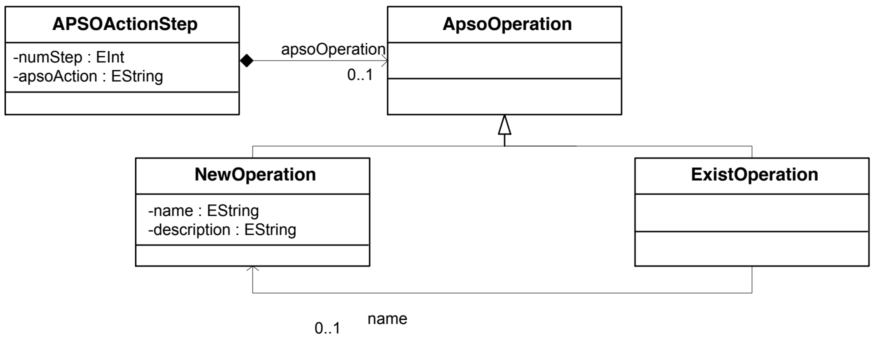

Figure 8. Meta-model of the system operation

3.3.2 The System Actions

The use case actions that the user performs are defined through *SystemActionBlock* rule. The system executes three types of actions. The first type of action is used to validate data that the system accepts from the user. When the system executes the data validation, it is necessary to define the data for the validation and the validation method. The validation method is defined by a descriptive attribute that describes how the system validates the data. Another type of action is used to define the operations that the system executes, it is defined by *APSOOperation* rule. The third type of action represents the system response to the user and it is defined by *IAResponse* rule. Corresponding grammar is presented bellow.

```
SystemActionBlock:
"SYSTEM ACTIONS:"
        "VALIDATE: "
                (rules=[Concept] "rule" description=STRING )*
        "EXECUTE: "
                systemOperation=APSOOperation
        "RESPONSE:"
                (ia=IAResponse)
"END SYSTEM ACTIONS";
```

Furthermore, the *Concept* is introduced to specify the data that the user enters on the one hand, and to specify the data that the system shows to user, on the other hand.

There are two different possibilities when the system executes the system operations:

- system operation is executed successfully and the system shows the answer to the user
- system operation generates an exception and the system shows an error to user.

The *SilabReq DSL* uses the *SuccessfulResponse* rule for a successful answer to the user, while it uses the *ErrorResponse* rule for an exception answer to system. When the system defines the response to the user, it returns a message. A message can contain only the text, only the required data or the text with the required data. The Figure 9 shows the meta-model for description of the response generated by the system after the execution of the system operations. In addition, the Figure 9 shows the grammar for these rules.

The *ErrorResponse* rule defines the response to the system when the execution of a system operation generates an exception. The Figure 10 presents the meta-model of *ErrorResponse* rule with an appropriate grammar.

The answer that the system returns to the system in the case of an unsuccessful execution of the system operation is defined by using a message (*messageDescription*) and the action that follows (*errorAction*). The action that follows can break the execution of the use case scenarios (interrupt scenario) or can be referenced by the previously defined action (*goto step*).

3.4 Models Based on the Use Case Specification

Model-Driven Engineering (MDE) is a software development approach based on models. Using models in software development requires a formal and complete definition of these

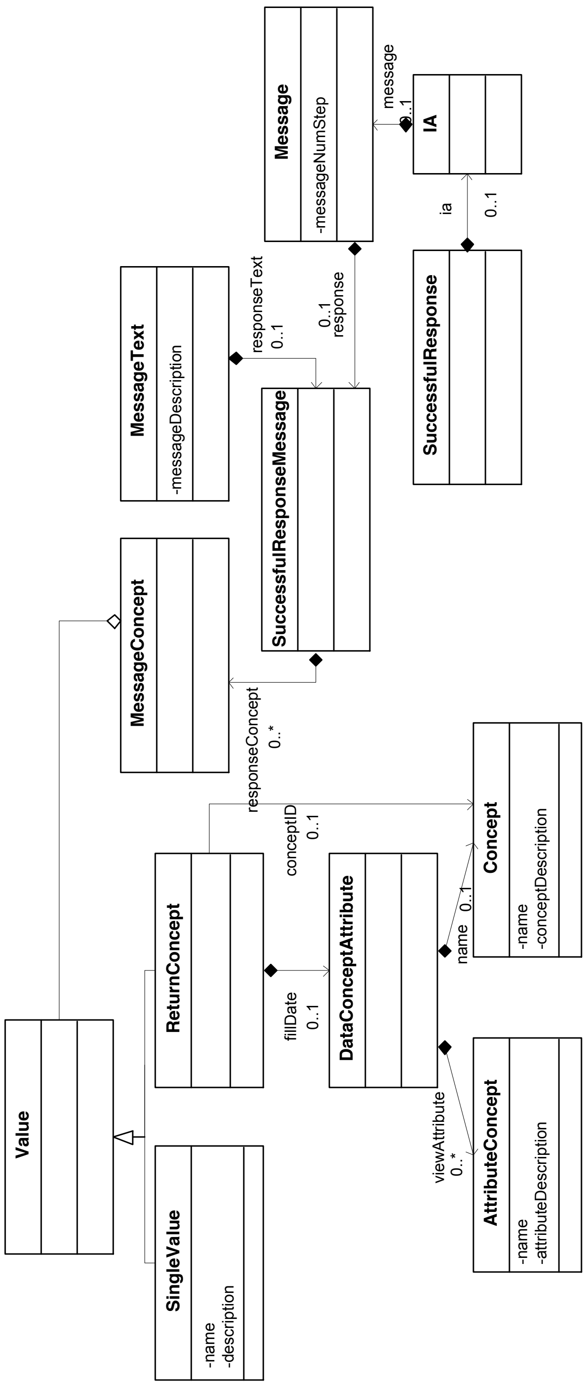

Figure 9. Meta-model of system successful response

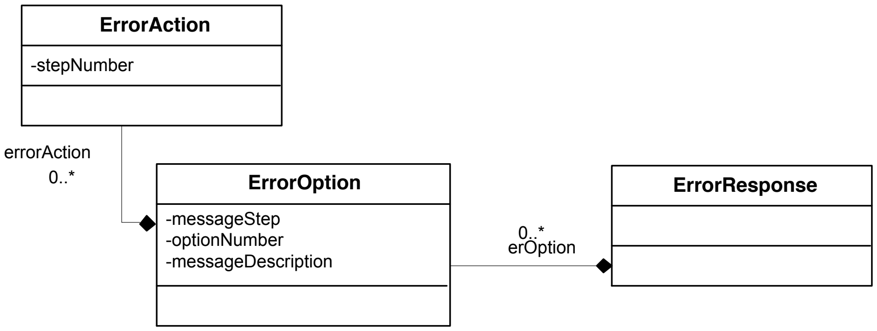

Figure 10. Meta-model of system error response

models. The model definition is done through a meta-model. Therefore, the meta-model is a model of the model and defines the rules for creating models. Meta-models can be defined by meta-model languages such as MOF, EMOF, Ecore. Furthermore, model transformations are one of the key mechanisms in MDE. The model can be executed through its transformation into the code of a programming language (code generation), or by creating an interpreter (model interpretation). There are different approaches in model transformation. We can roughly identify five categories for the model transformation such as:

- model transformation using general-purpose programming languages (e.g. Java)
- model transformation with tools for transformation (e.g. Graph Transformations, or XSLT)
- model transformation with tools for transformation models (e.g. OMG QVT, ATL, MTL)
- model transformation with tools for meta-modeling (e.g. Kermeta, Metacase or Xactium).

In this paper we present a set of transformations that we use to create models that show the system boundary (UML use case model), the structure (UML class model) and behavior (UML state machine model) of the system. These transformations are defined in the Kermeta language [2]. Kermeta is a model-oriented language, which meta-model is fully compatible with Essential OMG Meta-Object Facility (EMOF) meta-model and Ecore meta-model, both supported by the Eclipse Modeling Framework.

User requirements are usually defined by a natural language in some form of text. The reason for defining user requirements in the form of text comes from the fact that the natural language is understandable by all stakeholders in the software development and does not require additional technical knowledge. On the other hand, the system requirements are described more formally. One way to present system requirements is through a set of models. Most often, these models are describing graphical representations of the observed problems, namely by the business processes, the flow of documents or data, the architecture of the system, and describe the context of system.

Therefore, as these models are mostly visual and graphical, they become more understandable in relation to a natural language because of ambiguity that may cause misunderstanding of the observed problems. In a software development process, the use of models is desirable in all its phases. It is almost impossible to imagine the phase analysis of the system without using graphical models. As the analysis phase defines the system specifications, the use of the models is extremely important at this stage in order to explicitly describe a system. For example, in the analysis phase it is very important to determine the boundaries of the system and determine what the system is and what its environment is. The context diagram that is commonly used in structural analysis is a model that shows the context of the system. In this paper, the system and the system context are represented through the UML use case diagrams.

The conceptual model can describe the logical structure of a software system. The conceptual model contains classes and associations between these conceptual classes. In this paper, we identified the conceptual classes based on the use case specification. We identified conceptual classes as well as their attributes and associations that exist between them.

3.4.1 The System Context

One of the aims of the software requirements engineering is to define the functional requirements of the system. The system functional requirements are described using the use cases. The use cases can be visually presented through the UML use case diagram. The use case diagram does not show the details of the use cases themselves, but clearly describes the boundary of the system, defines the system actors and shows the relationship which exists between users of the system and the use cases, and relationships between the use cases. Therefore, the use case diagram provides an excellent conceptual overview of the system and the system boundaries.

We defined the *SilabReqUC* transformation that transforms *SilabReq* use case model (specified textually) into UML use case model (visually represented through UML use case diagrams). The Figure 11 shows this transformation.

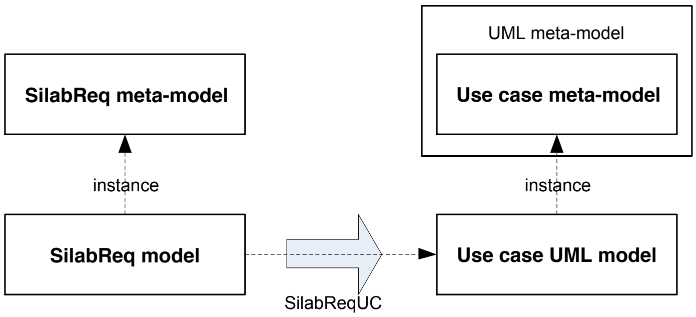

Figure 11. SilabReqUC transformation of use case model

The key elements of this transformation will be explained through several use cases such as *Register new book for certification (UC1)* and *Search books for certification (UC2)* in the electronic office system. The explanation begins with a specification of both use cases in *SilabReq language*, followed by an explanation of the transformation and presentation of results of the transformation.

The counter worker (actor) executes the use case UC1. This use case (UC1) is extended by UC2, in the case when user first wants to review existing books to be certified, and after that enters a new book for certification. The user enters the following data for the book certification:

- the date when the book is opened
- title of the book
- the book identification.

The counter worker executes the use case UC2. The counter worker first enters the criterion for searching, and then as a result gets a list of books that match the given criteria.

The use case specification of these use cases is given below.

```
UC: UC_RegisterNewBook "Register book for certification" {
        Actors: CounterWorker
        Concept: CertificationBook
        USE CASE FLOW:
        USER ACTIONS:
                extension point: "Searching existing books"
                                UC_SearchBook
                apuso:<1> CounterWorker chooses {Worker}
                apuso:<2> CounterWorker puts
                                {CertificationBook.dateOpened .code
                                .name .valid}
                apso: <3> CounterWorker calls system to
                                [op_saveBook "register new book
                                for certification"]
        SYSTEM ACTIONS:
        VALIDATE:
        EXECUTE:
                op_saveBook
        RESPONSE:
        SUCCESSFUL
                <4> shows message: <"The book is saved">
        EXCEPTION
        Start option:
                <4.1> error message: "The book is not saved"
                action: interrupt use case flow
        End option
        END SYSTEM ACTIONS
        }
```

Use case specification: Register new book for certification

```
UC: UC_SearchBooks "Search book for certification" {
        Actors: CounterWorker
        Concept: CertificationBook
        USE CASE FLOW:
        USER ACTIONS:
                apuso: <1> CounterWorker puts {$ "criteria for searching"}
                apuso: <2> CounterWorker puts {$ "values" "enters values
                                for searching"}
                apso:  <3> CounterWorker calls system to [op_findBook
                                "find book with given criteria"]
        SYSTEM ACTIONS:
        VALIDATE:
        EXECUTE:
                op_findBook
        RESPONSE:
        SUCCESSFUL
        <4> shows message: <"Books with given criteria"> return:
                        [CertificationBook] fill CertificationBook.Name
        EXCEPTION
        END SYSTEM ACTIONS
}
```

Use case specification: Searching books for certification

During the transformation of the use case specification into the use case model, the *SilabReqUC* transformation:

- for each use case, generates UML package (*uml::Package*) with the same name as the name of the use case
- for each use case, generates UML use case (*uml::UseCase*) whose name is identical to the name of a defined use case. The generated UML use case is inside the generated package with the same name
- generates "association" (*uml::Association*) between users and use case
- generates "extends" relationships (*uml::Extend*) between the two use cases when the use case specification of one use case contains defined extension points
- generates "include" relationships (*uml::Include*) between the two use cases when the use case specification of one use case includes the other.

So, the *printUseCaseIncludeExtend* method into *SilabReqUC* transformation is responsible for searching of the elements of the use case specification which we used to create "extend" and "include" relationships. The method is given below:

```
operation printUseCaseIncludeExtend(theUseCase : reqDSL::UseCase) :
                Void is do
var uc:uml::UseCase
uc:=umlModel.putInModelUseCase(theUseCase.name,theUseCase.name)
        theUseCase.useCaseFlow.completeActionBlock.each{
                completeActionBlock|var userActionBlock:reqDSL::
                                UserActionBlock
                userActionBlock:=completeActionBlock.userActionBlock
                userActionBlock.actionStepType.each{actionStepType|
                        if actionStepType.isInstanceOf(reqDSL::
                                        ExtensionPoint) then
                        var useCase:uml::UseCase
                        useCase:=umlModel.getUseCaseRef(uc.name,uc.name)
                        var extPoint:reqDSL::ExtensionPoint
                        extPoint:=actionStepType.asType(reqDSL::
                                        ExtensionPoint)
                                var useCaseWhichExtend:uml::UseCase
                                useCaseWhichExtend:=umlModel.getUseCaseRef
                                        (extPoint.extpoint.name,
                                        extPoint.extpoint.name)
                                        umlModel.putInModelExtendUseCase
                                        (uc.name,useCase,
                                        useCaseWhichExtend)
                end
                }
        }
end
```

Based on the use case specifications of these use cases (UC1 and UC2), *SilabReqUC* transformation creates an UML use case model that is visually displayed through UML use case diagrams. The figure below shows the UML use case diagram for these two use cases (Figure 12).

3.5 Structure and Behavior of the System

3.5.1 The SilabReqSpecUCConceptModel Transformation

The analysis model that describes the logical structure and behavior of software systems can be created from the use case specification. We have developed the *SilabReqSpecUC-ConceptModel* transformation that is responsible for creating a model that describes the logical structure of a software system, and the *SilabReqSpecUCSystemOperation* transformation which is responsible for creating a model that describes the behavior of the system. Both transformations are created in Kermeta language for meta-modeling.

The conceptual model describes the structure of a software system and can be visually represented through the UML class diagrams. The UML class diagram is a visual presentation of the elements contained in the UML class model. Therefore, *SilabReqSpecUC-ConceptModel* transformation is the transformation of the corresponding elements of the *SilabReq* use case model into UML class models.

The *SilabReq DSL* consists of the expressions that are formed based on concepts appearing in the meta-model of *SilabReq* language. Therefore, it can be said that expressions are model concepts defined in *SilabReq* meta-model. In this section, we present the key concepts (classes) of *SilabReq* meta-models that are used to generate the UML class model.

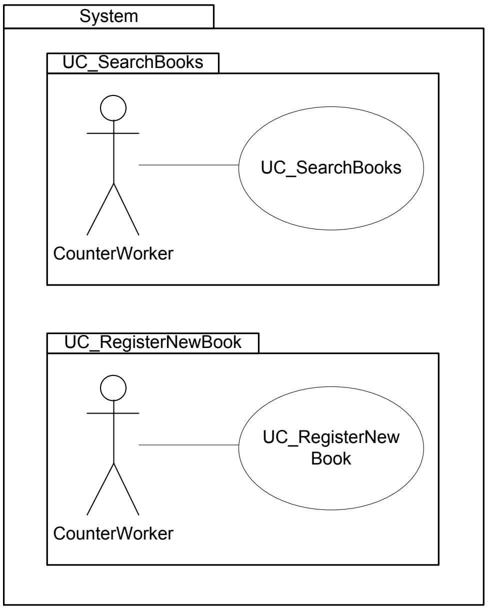

Figure 12. UML use case diagrams for UC1 and UC2

The UML class diagram below shows meta-classes of *SilabReq DSL* that contain semantics that allow us to identify conceptual classes and their attributes in the system (Figure 13).

In the transformation process, we have used the *Concept* meta-class to identify domain classes, while we use the meta-class *AttributeConcept* to identify appropriate attributes of domain classes.

For example, we have defined *printActionUserActionStep* method within the SilabReqSpecUCConceptModel transformation that searches required elements in the use case specification that contains semantics for identifying structure of the system. If this method finds these elements in the use case specifications of this action, then we created a relationship between the identified concept and the use case main concept (each use case is related with one main concept).

```
operation printActionUserActionStep(theActionUserActionStep : reqDSL::
                ActionUserActionStep) : Void is do
        if (theActionUserActionStep.isInstanceOf(reqDSL::
                        APUSOActionStep)) then
        var pom:reqDSL::APUSOActionStep
        pom:=theActionUserActionStep.asType(reqDSL::APUSOActionStep)
        pom.apusoActionData.each{d|
                var data:reqDSL::Data
                var cd : reqDSL::ChooseData
                data:=d
                cd:=data.chooseData
                if cd.isInstanceOf(reqDSL::DataConceptAttribute)then
                var cda:reqDSL::DataConceptAttribute
                cda:=cd.asType(reqDSL::DataConceptAttribute)
                cda.viewAttribute.each{att|
                        cModel.putInModelClassAttribute("DomenConcept",
                        cda.name.name,att.name)
        }
        var cc:reqDSL::Concept
        cc:=cda.name
        var cl:uml::Class
        cl:=cModel.getClassRef("DomenConcept",cc.name)
        var clOn:uml::Class
        clOn:=cModel.getClassRef("DomenConcept",ucMainConcept.name)
        if (cl.name!=clOn.name) then
                cModel.createRelationship(cl,clOn)
        end
        end
}
end
```

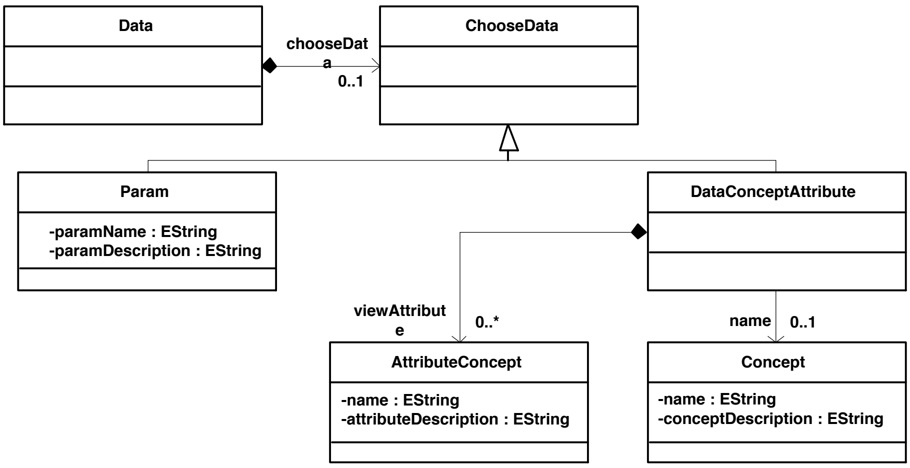

Figure 13. The classes of SilabReq DSL that contain semantic for conceptual model

Based on the use case specification of the use cases (UC1 and UC2), *SilabReqSpecUC-ConceptModel* transformation creates the UML conceptual model that shows the identified class of the system. This conceptual model class is presented through the UML class diagram in the Figure 14.

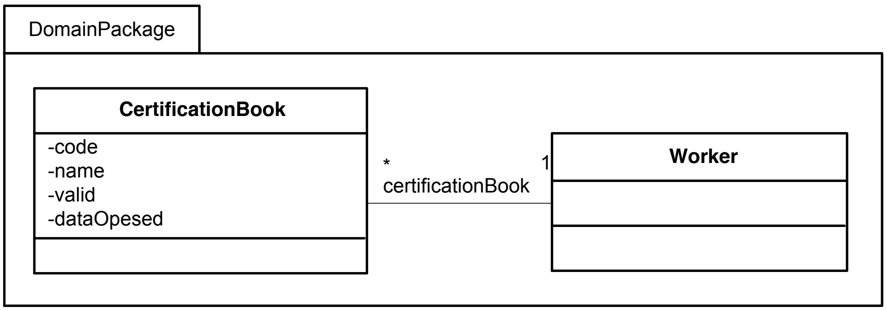

Figure 14. Conceptual model for Register new book for verification and Searching books for verification use cases

3.5.2 SilabReqSpecUCSystemOperation Transformation

Use cases describe how users interact with the system, and therefore use cases describe what the system does, not how it works. The system is viewed as a black box. During the interaction, the user sends a request for a certain requirements to system expecting to get a specific response. When the user sends a request to the system, he/she calls the system to perform a system operation.

SilabReq language contains the definition of an action, which the user uses to call the system to perform the system operation. Based on these actions, but also on the action that the system uses to execute the system operation, and the action by which the system returns the result of the execution of the system operation, it is possible to get specification of the system operations, in other words functions that the system should provide.

SilabReq language uses the instance of meta-class *APSOOperation* or classes *NewOperation* and *ExistOperation* to define the system operation. In the use case, *SilabReq* language allows:

- Creating the new system operation if it does not already exits, or
- Referencing to the existing system operation.

Therefore, the meta-model contains two classes for the specification of the system operation. The class *NewOperation* is used to specify new systems operations, and the class *ExistOperation* is used to refer to an existing system operation.

The result of the execution of the system operation is defined by meta-classes that are presented using class diagram in the Figure 15.

We use the *IAResponse* class to specify the system response for the execution of the system operations. The result of executing system operations are specified by the *MessageConcept* class. Based on the presented class diagram it can be concluded that the result of the execution of the system operation is a value for the customer (meta-class *Value*). This value for the user may be a message on the successful execution of system functions (meta-class *SingleValue*) or may be a result in the form of one or more domain objects that are displayed to the user (meta-class *ReturnConcept*). Based on the results that the system shows to the user, the specification of the system operations can

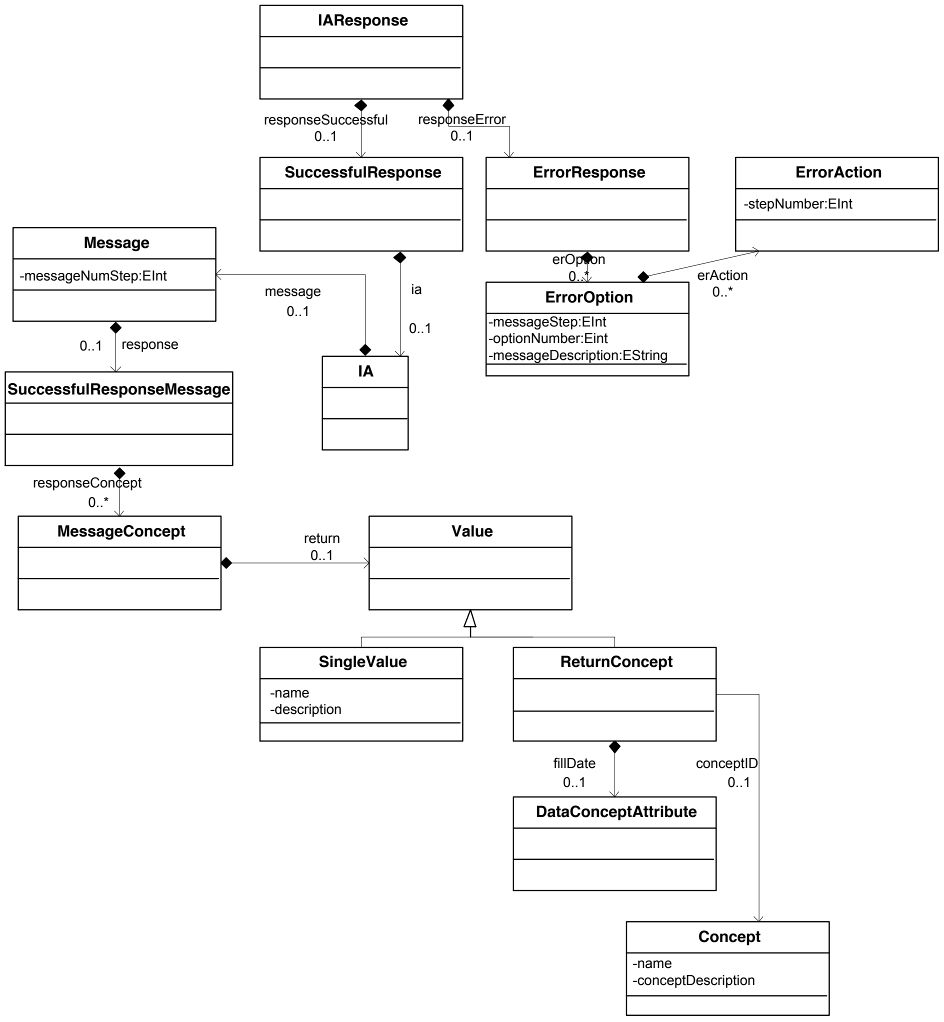

Figure 15. The meta-classes that describe execution of the system operation

be enriched semantically by specifying the type that the system operation returns. UML class diagrams can visually present discovered system operations as in the Figure 16.

3.6 Visualization of Use Case Execution

In practice, the models can be used in two ways:

- by transforming the models into source code in some programming language (e.g. Java or C#), or
- by a direct execution of models through an appropriate interpreter.

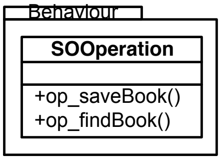

Figure 16. Identified system operations

We have developed the *SilabReqUCExecute* interpreter (which is created in Kermeta environment) to execute the use cases described in *SilabReq DSL*. We have developed the *SilabReqUCStateMachine* transformation, which accepts *SilabReq DSL* model as an input and transforms it into the UML state machine model. After that, the *SilabReqUCExecute* interpreter executes an appropriate use case. The Figure 17 describes these steps.

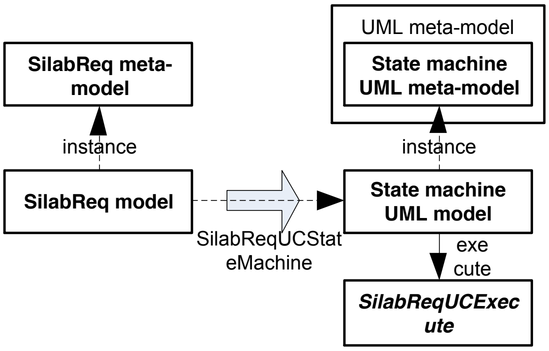

Figure 17. SilabReqUCExecute interpreter

We have identified different available states in the use case execution:

- state “Preparing data for system operation execution”
- state “Execution of the system operation”
- state “Showing the result of the system operation execution”
- “Successfully finished execution of a use case”
- “Interrupted execution of a use case”.

Figure 18 describes the state machine with the states and actions that make a use case move from one to another state.

From the initial state (start) the use case can move into one of two possible states:

- state “Preparing data for system operation execution” (APUSO) or
- state “Execution of the system operation” (SO).

From “Preparing data for system operation execution” state the use case can turn into “Execution of the system operation” state, while from “Execution of the system operation” use case can turn into “Showing result of the system operation execution” (IA) state.

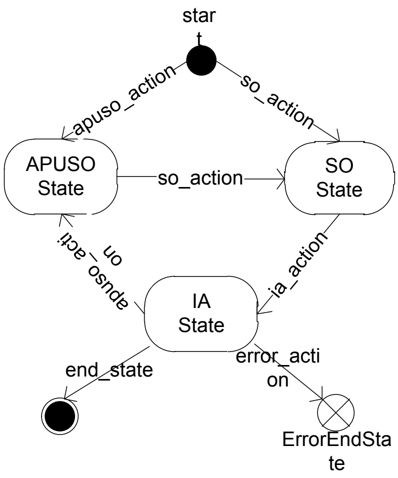

Figure 18. UML state machine diagram for use case states

From "Showing result of for system operation execution" state, the use case can turn into one of several possible states:

- "Preparing data for system operation execution" state
- "Execution of the system operation" state
- "Successfully finished execution of the use case" state or
- "Interrupted execution of the use case" state if the use case interrupts.

The *SilabReqUCStateMachine* transformation transforms the use case model into a model of the state machine as follows:

- For each use case, it generates the UML package (*uml::Package*) and the UML state machine (*uml::StateMachine*). The package name and the name of the machine state is identical to the name of the use case
- For each *UserActionBlock* that exists within a defined use case it generates:
  - one state of the *APUSOState* type (*uml::State*) if there is at least one action of APUSO type within a block of actions
  - as many transitions (*uml::Transition*) as APUSO actions types
  - one action (*uml::Transition*) which of APSO
- For each *SystemActionBlock* it generates:
  - a state of *SOState* type, its name has a prefix SO and an extension that is identical to the name of the system operation which the system executes
  - a state of *IAState* state, its name has the prefix IA and an extension of the same name as the name of the executed system operation

  - an action that the use case passes from an *SOState* state to an *IAState* state. The name of this action has an IA prefix and extension that is identical to the name of the executed system operation
  - as many states of *ErrorEndState* as possible – there are alternative scenarios that interrupt the execution of the use case

- Per each use case, one state of *EndState* type (*uml::State; kind:exitPoint*)
- For each include and/or extend directive it creates a separate state machine (submachine, *uml::StateMachine*)
- For each action of *UserIFActionStep* type it creates a pseudo-state of (*UML::State; kind:choice*) type
- For each action of *UserIterateActionStep* type it creates an action that executes iterative in the current state of the current use case

Figure 19 describes a UML state machine for the use case *Search books for certification.*

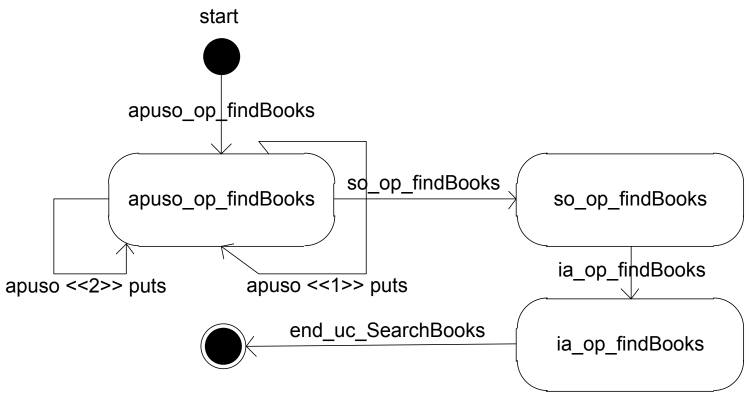

Figure 19. UML state machine diagram for Search books for verification use case

When an interpreter executes the use case, it leads the user through the use case execution. In each state, the interpreter asks user to choose one of the available actions, after which the use case moves to the next state. Therefore, by selecting one of the available actions the user moves to the use case execution and validates it.

4 CONCLUSION

Use cases as a technique for specification of the functional requirements of the system are used in many different software development methods. Adopting use cases in the software development methods, based on models, requires that use cases must be defined formally especially pre-condition, post-condition and actions.

In this paper, we propose the *SilabReq* language as a more formal way to specify use cases. This language can be used for both user's actions specification and system's actions specification. Therefore, it can be used to specify actions to:

- determine the conceptual (domain) model of the system (for which we have created the *SilabReqUCConceptModel* transformation)
- discover the functions of the system (for which we have created the *SilabReqUCSystemOperation* transformation)
- visually display boundaries of the system over UML use case diagrams (foe which we have created the *SilabReqUC* transformation)
- visually display case scenario execution using UML state machines (for which we have created the *SilabReqUC* transformation)
- monitor the execution of use cases (through developed the *SilabReqUCExecute* interpreter).

Figure 20 presents graphically the goals that can be achieved through the specification of the use cases using *SilabReq* languages.

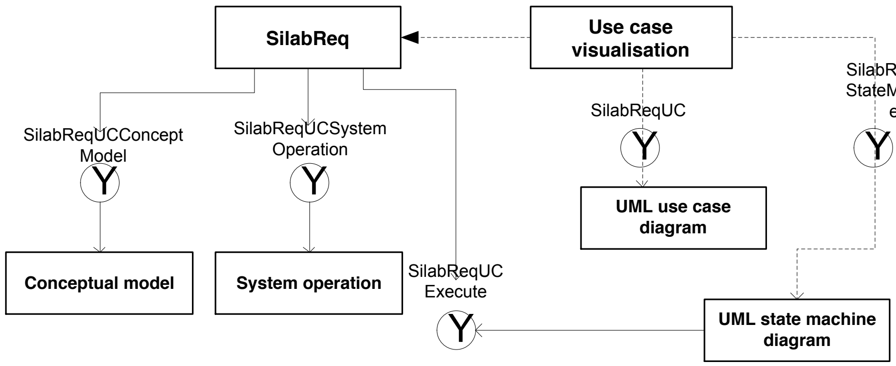

Figure 20. SilabReq DSL and defined transformation

REFERENCES

[1] ANother Tool for Language Recognition web site. Available on: `http://www.antlr.org`.

[2] Eclipse Modeling Framework web site. Available on: `http://www.eclipse.org/modeling/emf`.

[3] FFI Kostmod 4.0 Report. Available on: `http://rapporter.ffi.no`.

[4] Astudillo, H.—Génova, G.—Smiałek, M.—Morillo, J. L.—Metz, P.—Prieto-Díaz, R.: Use Cases in Model-Driven Software Engineering. Proceedings of MoDELS Satellite Events, 2005, pp. 272–279.

[5] Boettger, K.—Schwitter, R.—Mollá, D.—Richards, D.: Towards Reconciling Use Cases via Controlled Language and Graphical Models. Web Knowledge Management and Decision Support, Springer Verlag, Heidelberg, LNCS, 2003, Vol. 2543, pp. 115–128.

[6] Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, New York 2000.

[7] Glinz, M.: Problems and Deficiencies of UML as a Requirements Specification Language. Proceedings of the 10th IEEE International Workshop on Software Specification and Design, 2000.

[8] Herman, K.—Svetinovic, D.: On Confusion between Requirements and Their Representation. Requirements Engineering, Springer-Verlag, 2010.

[9] Hoffmann, V.—Lichter, H.—Nyssen, A.—Walter, A.: Towards the Integration of UML – and Textual Use Case Modeling. Journal of Object Technology, Vol. 8, 2009, No. 3, pp. 85–100.

[10] Ferreira, D.—Silva, A. R.: RSL-PL: A Linguistic Pattern Language for Documenting Software Requirements. Proceedings of Third International Workshop on Requirements Patterns (RePa '13), in the 21st IEEE International Requirements Engineering Conference (RE 2013), July 2013, IEEE Computer Society.

[11] Fantechi, A.—Gnesi, S.—Lami, G.—Maccari, A.: Application of Linguistic Techniques for Use Case Analysis. Requirements Engineering Journal, Vol. 8, 2003, No. 3, pp. 161–170.

[12] Ferreira, D. A.—Silva, A. R.: A Controlled Natural Language Approach for Integrating Requirements and Model-Driven Engineering. ICSEA 2009, pp. 518–523.

[13] Ferreira, D. A.—Silva, A. R.: RSLingo: An Information Extraction Approach Toward Formal Requirements Specifications. MoDRE 2012, pp. 39–48.

[14] Ferreira, D.—Silva, A. R.: RSL-IL: An Interlingua for Formally Documenting Requirements. Third IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE), in the 21st IEEE International Requirements Engineering Conference (RE 2013), July 2013, IEEE Computer Society.

[15] Fondement, F.—Baar, T.: Making Metamodels Aware of Concrete Syntax. In Hartman, A., Kreische, D. (Eds.): ECMDA-FA, Springer, LNCS, Vol. 3748, 2005, pp. 190–204.

[16] I Standard Glossary of Software Engineering Terminology. 1990, IEEE Std 610.12-1990.

[17] IEEE Computer Society Professional Practices Committee. 2004, SWEBOK®: Guide to the Software Engineering Body of Knowledge. The Institute of Electrical and Electronics Engineers, Inc. 2004.

[18] Jacobson, I.—Booch, G.—Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, New York, 1998.

[19] Jorgensen, J. B.—Bossen, C.: Executable Use Cases: Requirements for a Pervasive Health Care System. IEEE Software, Vol. 21, 2004, No. 2, pp. 34–41.

[20] Kotonya, G.—Sommerville, I.: Requirements Engineering Processes and Techniques. John Wiley and Sons, 2000.

[21] Kleppe, A. G.: A Language Description is More Than a Metamodel. Fourth International Workshop on Software Language Engineering, Nashville, USA, 2007.

[22] Li, L.: Translating Use Cases to Sequence Diagrams. Proceedings of Fifteenth IEEE International Conference on Automated Software Engineering, Grenoble, France, 2000, pp. 293–296.

[23] Loniewski, G.—Insfran, E.—Abrahão, S.: A Systematic Review of the Use of Requirements Engineering Techniques in Model-Driven Development. In: Petriu, D., Rouquette, N., Haugen, Ø. (Eds.): Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II (MODELS '10). Springer, 2010, pp. 213–227.

[24] Mellor, S. J.—Clark, A. N.—Futagami, T.: Model-Driven Development. IEEE Software, Vol. 20, 2003, pp. 14–18.

[25] Nguyen, P.—Chun, R.: Model Driven Development with Interactive Use Cases and UML Models. Software Engineering Research and Practice, 2006, pp. 534–540.

[26] Nakatani, T.—Urai, T.—Ohmura, S.—Tamai, T.: A Requirements Description Metamodel for Use Cases. Eighth Asia-Pacific Software Engineering Conference (APSEC '01), 2001, pp. 251–258.

[27] OMG (August 2011) UML Superstructure Specification. v2.4.1. OMG Formal Document 2011-08-06.

[28] OMG (August 2011) UML Infrastructure Specification. v2.4.1. OMG Formal Document 2011-08-05.

[29] Alchimowicz, B.—Jurkiewicz, J.—Ochodek, M.—Nawrocki, J.: Building Benchmarks for Use Cases. Computing and Informatics, Vol. 29, 2010, No. 1, pp. 27–44.

[30] Olek, L.—Ochodek, M.—Nawrocki, J.: Enhancing Use Cases with Screen Designs. A Comparison of Two Approaches. Computing and Informatics, Vol. 29, 2010, No. 1, pp. 3–25.

[31] Rolland, C.—Achour, C. B.: Guiding the Construction of Textual Use Case Specifications. Data and Knowledge Engineering, Vol. 25, 1998, No. 1-2, pp. 125–160.

[32] Silva, A.—Videira, C.—Saraiva, J.—Ferreira, D.—Silva, R.: The ProjectIT-Studio, an Integrated Environment for the Development of Information Systems. Proceedings of the 2nd International Conference of Innovative Views of .NET Technologies (IVNET '06), Sociedade Brasileira de Computação and Microsoft 2006, pp. 85–103.

[33] Silva, A. R.: Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model. Computer Languages, Systems and Structures, Vol. 43, 2015, pp. 139–155.

[34] Silva, A. R.—Saraiva, J.—Ferreira, D.—Silva, R.—Videira, C.: Integration of RE and MDE Paradigms: TheProjectIT Approach and Tools. IET Software: On the Interplay of .NET and Contemporary Development Techniques, 2007.

[35] Sommerville, I.: Software Engineering. Eight edition. Addison Wesley Longman Publishing Co. Inc., Boston 2006.

[36] Spivey, J. M.: The Z Notation: A Reference Manual. Prentice Hall 1992.

[37] Smiałek, M.—Bojarski, J.—Nowakowski, W.—Straszak, T.: Scenario Construction Tool Based on Extended UML Metamodel. LNCS, Vol. 3713, 2005, pp. 414–429.

[38] Smiałek, M.—Straszak, T.: Facilitating Transition from Requirements to Code with the ReDSeeDS Tool. RE 2012, pp. 321–322.

[39] Smiałek, M.—Nowakowski, W.—Jarzebowski, N.—Ambroziewicz, A.: From Use Cases and Their Relationships to Code. MoDRE 2012, pp. 9–18.

[40] Smiałek, M.: Accommodating Informality with Necessary Precision in Use Case Scenarios. Journal of Object Technology, 2005.

[41] Somè, S. S.: A Meta-Model for Textual Use Case Description. Journal of Object Technology, Zurich 2009.

[42] The Standish Group: CHAOS Summary. The Standish Group International, Inc., 2006.

[43] The Standish Group: CHAOS Summary. The Standish Group International, Inc., 2009.

[44] The Standish Group: CHAOS Summary. The Standish Group International, Inc., 2011.

[45] Valderas, P.—Pelechano, V.: A Survey of Requirements Specification in Model-Driven Development of Web Applications. TWEB, Vol. 5, 2011, No. 2, pp. 10.

Dušan Savić received the Magister degree in information system and technologies from the Faculty of Organization Sciences, University of Belgrade, in 2010. He is currently postgraduate student and a teaching assistant at the Faculty of Organizational Sciences at the Software Engineering Department. He has interests in the following areas: modeling and meta-modeling, model driven engineering, requirement engineering, software development, software design, domain specific languages, automation of user interface development. He has lectured undergraduate and graduate level courses in his area. He is the author or co-author of several publications for national and international conferences or workshops and scientific journal.

Siniša Vlajić is Associate Professor of software engineering at University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems. He has lectured undergraduate and graduate level courses: introduction to programming, introduction to information system, software design, software patterns, programming methodology and Java programming language. He wrote many books, scripts and publications about C++, Java, software design, software patterns, database and information systems. His main research interests include software process, software design, software maintenance, software pattern formalization and programming methodology. He is one of the founders of the Laboratory and Department of the Software Engineering at Faculty of Organizational Sciences.

Saša Lazaravić is Assistant Professor of software engineering at University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems. He has lectured undergraduate and graduate level courses: introduction to programming, software design, software construction, software testing, software quality. His main research interests include software process, software design, software testing, software quality and programming on .Net platform.

Ilija Antović defended his Magister thesis in the software engineering in 2010. Currently he is working at Software Engineering Department, Faculty of Organizational Sciences, University of Belgrade and writing his Ph.D. thesis. His research interests include automation of user interface development, modeling and meta-modeling, model driven engineering, requirement engineering, software patterns, code generation. He lectures at undergraduate and graduate level courses in his area. He is the author or co-author of several publications for national and international conferences and scientific journals.

Vojislav Stanojević is Teaching Assistant of software engineering at University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems. He has lectured undergraduate and graduate level courses: introduction to programming, introduction to information system, software design, software patterns, programming methodology and Java programming language. He wrote publications about Java, software design, software patterns, application frameworks and domain specific languages. His main research interests include software design, application frameworks, business rules, domain specific languages.

Miloš Milić is Teaching Assistant at Faculty of Organizational Sciences, University of Belgrade, Serbia. He has lectured undergraduate and graduate level courses: introduction to programming, software design, software patterns and Java programming language. His research interests include software quality, software design and software testing. He holds B.Sc. in information systems and M.Sc. in software engineering. He is a Ph.D. student at University of Belgrade.

Alberto Rodrigues da Silva is Associate Professor at the Instituto Superior Técnico (Universidade de Lisboa), a senior researcher at INESC-ID, and a partner of the SIQuant company. He has research interests in the following areas: information systems, modeling and metamodeling, model driven engineering, requirement engineering, and social computing. He is the author of 5 technical books and over 200 peer-reviewed scientific papers. He is a member of the ACM, PMI and the Portuguese Society of Chartered Engineers.