VNU Journal of Science: Comp. Science & Com. Eng., Vol. 39, No. 1 (2023) 87–106

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

FRSL: A Domain Specific Language to Specify Functional Requirements

Duc-Hanh Dang*

VNU University of Engineering and Technology,
144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Received 05 May 2023;
Revised 05 June 2023; Accepted 09 June 2023

Abstract: In software development, obtaining a precise specification of the software system's functional requirements is crucial to ensure the software quality and enable automation in development. Use cases are an effective approach for capturing functional requirements. However, the use of ambiguous or vague language in use cases can result in imprecision. It is essential to ensure that use case specifications are clear, concise, and complete to avoid imprecision in requirements. This paper aims to develop a domain specific language called FRSL to precisely specify use cases and to provide a basis for transformations to generate software artifacts from the use case specification. We define a metamodel to capture the technical domain of use cases for FRSL's abstract syntax and provide a textual concrete syntax for this language. Additionally, we define a formal operational semantics for FRSL by characterizing the execution of a FRSL specification as sequences of system snapshot transitions. This formal semantics enables precise explanation of the meaning of use cases and their relationships and serves as a basis for transformations from the use case specification. We implement a tool support for this language and evaluate its expressiveness in comparison with current use case specification languages. This work brings out i) a DSL to specify use cases that is defined based on a formal semantics of use cases; and ii) a tool support realized as an Eclipse plugin for this DSL.

Keywords: Use Case, UML/OCL, Contract-Based Specification, Model Transformation, Domain Specific Language (DSML/DSL).

1. Introduction

In software development, use cases are an effective way to capture functional requirements of the system, ensuring that all stakeholders have a clear and consistent understanding of what the software system should do. Use cases are often specified in the form of narrative text,

*Corresponding author.
E-mail address: hanhdd@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.803

providing a detailed description of the steps involved in achieving a particular goal. Use cases can also be represented in a graphical form [1] for either a high-level overview of the system's functionality or a more detailed description of use case steps. Use cases can be formalized in several ways [2] in order to capture the system behavior from a certain perspective for a verification purpose. However, the use of ambiguous or vague language in the use case can lead to imprecision. To bring more automation in the software development, use cases should be precisely specified with a balance between i) the right level of detail to automatically generate artifacts from the use case specification; and ii) the understandability for stakeholders.

There are several approaches available in the literature to describe and formalize use cases, according to the surveys conducted in [2, 3]. Current work either introduces use case templates [4, 5] to enhance the writing, reading and reviewing of use cases, or provides patterns and anti-patterns together with guidelines for use case specification [6, 7]. Many authors propose using either UML diagrams [1] or formal languages such as Event-B [8] and graph transformation [9] or domain specific languages (DSLs) such as RUCM [10], RSL [11], SilabReq [12], and USL [13] to precisely specify use cases. However, current work in the literature tends to capture use cases from the developer's point of view, i.e., only using the concepts from the solution space, rather than from the problem domain, to express the specification. Use case relationships are either simplified as alternative flows or just ignored; The effect of use case actions is often not explicitly specified.

This paper introduces a DSL called FRSL to specify use cases, resulting in a precise specification of the system's functional requirements. Specifically, a FRSL specification would provide a general description of the use case as well the other detailed information such as use case relationships, scenarios, and snapshot patterns to express use case constraints. We define a precise semantics of use case by characterizing the execution of a FRSL specification as sequences of state transitions: Each current state is represented by an object model. This formal operational semantics allows us to precisely explain the meaning of use cases and the relationships *include* and *extend* between them. It also provides a basis for transformations to automatically generate software artifacts from a FRSL specification. We implement a support tool as an Eclipse plugin and then evaluate the expressiveness of FRSL compared with current use case specification languages. The use case specification language FRSL would help precisely specify the system's functional requirements and bring more automation in the software development.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 presents background concepts and motivates this work with a running example. Section 4 defines the FRSL's abstract and textual concrete syntax. Section 5 provides a formal semantics for it. A tool support is explained in Section 6. Section 7 evaluates our language. This paper is closed with conclusions and a discussion of future work.

2. Related Work

Current work in the literature for software requirements specification mainly focuses on functional requirements. Several techniques as surveyed in [14] are proposed to specify non-functional requirements: Non-functional requirements are represented as system properties, that need to be verified, based on logics such as first-order predicate logic or temporal logic. To specify functional requirements, current methods often represent them in the form of natural, structured or unstructured languages, or semi-formal languages like UML using diagrams such as Use case diagrams, ER diagrams, Interaction diagrams, or Statecharts. Several modeling languages such as SysML (Systems Modeling Language [15]) have been proposed to specify complex systems, at different levels of abstraction, and at the same time, provide traceability between artifacts such as requirements specification and test cases. The KAOS (Knowledge Acquisition in autOmated Specification language) was proposed in [16] to specify a goal-oriented requirement. Formal languages such as algebraic specifications (e.g., CASL [17]), object-oriented specification languages (e.g., Object-Z, Alloy, and Event-B [18]), and metamodeling-based languages (e.g., UML/OCL) have been employed to precisely specify functional requirements.

Use cases have been widely used to specify functional requirements. The surveys conducted in [2, 3] present the approaches available in the literature to describe and formalize use cases. Current approaches on use case specification can be divided in three groups. First, many use case templates [4, 5] have been introduced for helping their writing, reading and reviewing. The work in [19] proposed an intermediate use case template to ease the extraction of class diagrams from use case specifications. The authors aim to increase either the level of detail or the degree of formality in the use cases. In [7] a pattern language is proposed for use case specification. Similarly, the work in [6] focused on to the guidelines, suggestions and techniques provided to develop quality use case specification. The authors introduce anti-patterns together with a template to define them in order to improve quality in use case models.

In the second group, many efforts have been made to formalize use cases even further to bring more automation to the development process. They concentrate on developing domain-specific languages (DSL) that allow specification of textual use cases and semi-automated generation of software artifacts within a model-driven approach. They aim to obtain the balance between the right level of detail for the generation of artifacts and the understandability for stakeholders. The authors in [10] introduced RUCM (Restricted Use Case Modeling) as a restricted natural language (NL) for reducing imprecision and incompleteness in use case specifications. The DSL aims to capture all the necessary information required for the generation of analysis models. The approach avoids behavioral modeling (e.g., Activity and Sequence diagrams) by applying Natural Language Processing (NLP) to a more structured and analysable form of use cases. The work in [11] introduced RSL as a restricted NL for use case specification. To specify use cases in RSL the user needs to manually parse NL sentences by indicating their subjects, verbs, objects and predicates. The work proposes using the relation <<invoke>> between scenarios to replace the use case relationships <<include>> and <<extend>>. The work also developed transformations to obtain UML diagrams and Java programs from an RSL specification. The authors in [12] proposed a language for use case specification named SilabReq. They aim to generate from use case specifications domain models, the system operations list, the use case model and activity and state diagrams. The work tends to separate use case specifications into different layers of abstraction, each of which is suitable to each stakeholder, including end-users, requirement engineers, business analysts, designers, developers, and testers. In SilabReq a use case is defined as a set of scenarios. Each scenario consists of one or more blocks of actions, and each block contains actions performed by either the actor or the system. Similarly, the work in [20] proposes a DSL named LUCAM that allows specification of textual use cases and semi-automated generation of UML diagrams. The work in [21] proposed an extension of UML metamodel for use cases. They aim to incorporate new meta-concepts into UML for use case behavior specification. A tool named UCDesc is introduced to support this approach.

In the third group, many authors have attempted to introduce rigor into use case descriptions, as surveyed in [3]. The work in [8] introduces UC-B as a plug-in for the Rodin platform that supports the authoring and management of use case specifications with both informal and formal components. The use case behavior is formally defined based on Event-B's mathematical language. The UC-B automatically generates a corresponding Event-B model, taken as input for the Rodin verification tools in order to verify system level properties. Within the approach, each use case specification contains a contract and scenarios. The contract enables to specify constraints, i.e., pre-conditions, post-conditions and invariants that apply to the execution of the use case. The authors in [2, 22–24] suggest employing UML Sequence diagrams and Activity diagrams in order to represent the sequences of interactions described in the use case.

Like our previous work reported in [13, 25], this work employs OCL conditions to express action contracts. Current methods [26–31] in the literature are often based on NLP techniques to extract information from the use case specification. This work captures such an information using the pre- and postconditions of use case actions. The constraints are represented based on object-oriented paradigm. Thus, our use case specification (i.e., in FRSL) can be seen as an intermediate representation that would help narrow the gap between the requirements and other software artifacts. Unlike our previous work, use case constraints in a FRSL specification can be expressed with the concepts of not only the solution space, but also the problem domain, as explained in Section 4.1.4. Besides, our work provides a precise semantics of use cases. Thus, the true semantics of extension points and rejoin points, as discussed in [32], could be precisely defined.

3. Running Example and Background

This section reviews background concepts of use case modeling and object models that will be used in the remainder of the paper. We will illustrate the concepts using a Point of Sale (POS) software example [33].

3.1. Use Case Modeling

Use case was first introduced in [5] as a means to model the interaction between a software system and its environment. They have been widely used to capture the system's functional requirements from the user's perspective. A use case is defined as follows.

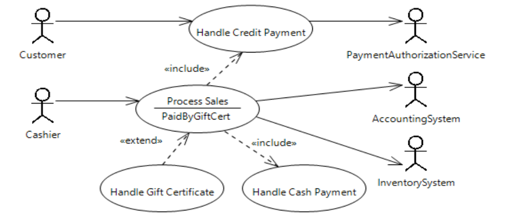
Use Case. A use case is "*the specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside objects to provide a service of value*" [34]. The outside objects here are referred to as actors.

Actor. An actor is "*a classifier for entities outside a subject that interact directly with the subject. An actor participates in a use case or coherent set of use cases to accomplish an overall purpose*" [34].

Example 1. Figure 1 shows on the right top a simplified use case model of the POS adapted from [33]. This example use case model is represented by a UML use case diagram together with textual use case descriptions. Considering the `Process Sales` use case, the actor `Cashier` takes part in this use case in order to meet her/his goals, i.e., to record the purchased items and collect payment. To help the primary actor `Cashier` achieve the main goal, the system needs to interact with other secondary actors including `AccountingSystem` and `CreditAuthorizationService`.

A use case is often specified using a description template [4], which includes two main parts, as illustrated in Fig. 1. The first part overviews the use case with fields about the "use case name", "actors", and "pre- and postcondition". The precondition (postcondition) needs to be fulfilled before performing (after finishing) the use case. The other part of the use case specification focuses on use case flows. A use case contains a *basic flow* and several *alternative flows*. The basic flow captures what normally happens for the use case. If the basic flow is unsuccessful

Use Case UC1: Process Sales
Brief Description: The use-case describes Cashier's process sale.
Primary Actor: Cashier
Secondary Actor: AccountingSystem; InventorySystem.
Preconditions: Cashier is identified and authenticated. Customer is ready to buy.
Postconditions: Sale is saved. Tax is correctly calculated.
Accounting and Inventory are updated. Commissions recorded. Receipts are generated.
Basic Flow:
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. The system creates a new sale and requires the cashier to enter items
4. Cashier enters item identifier.
5. System records sale line item and presents item description, price,
and running total with calculated tax.
(*) Cashier repeats steps 4-5 until indicates done.
6. System presents total with taxes calculated.
7. Cashier tells Customer the total and asks for payment.
8. Customer wants to pay the sale by cash. The use case HandleCashPayment is involked.
9. System logs completed sale and sends sale and payment information to
the external systems AccountingSystem (for accounting and commissions)
and InventorySystem (to update inventory)
10. Customer leaves with receipt and goods (if any).
Alternative Flows:
8a. Customer wants to pay the sale by credit payment.
8a.1. The use case HandleCreditPayment is involked.
Extension Points:
E1. PaidByGiftCert: at step 8; when Customer wants to pay with gift certificate.



Use Case UC3: Handle Credit Payment
Brief Description: The Customer wants to pay by a credit payment.
Primary Actor: Customer
Secondary Actor: PaymentAuthorizationService
Preconditions: Customer is ready to pay the sale by a credit payment.
Postconditions: Customer purchases. Cashier takes bill. Done the sale.
Basic Flow:
1. Customer enters the credit account information.
2. System sends payment authorization request to the external system
PaymentAuthorizationService, and requests payment approval.
3. System receives payment approval and signals approval to Cashier.
4. System records the credit payment, which includes the payment approval.
5. System presents credit payment signature input mechanism.
6. Cashier asks Customer for a credit payment signature.
Customer enters signature.

Use Case UC2: Handle Cash Payment
Brief Description: This use-case allows the customer to pay the sale by a cash payment.
Primary Actor: Cashier
Preconditions: Customer is ready to pay the sale by a cash payment.
Postconditions: The sale is paid by a cash payment.
Basic Flow:
1. Customer wants to pay the sale by cash.
2. Cashier enters the cash amount tendered.
3. System presents the balance due, and releases the cash drawer.
4. Cashier deposits cash tendered and returns balance in cash to Customer.

Use Case UC4: Handle Gift Certificate
Brief Description: This use case allows the customer to pay with certificate.
Primary Actor: Cashier
Preconditions: Customer wants to pay with gift certificate.
Postconditions: The sale is partly paid by the gift certificate.
Basic Flow:
1. Customer gives gift certificate to Cashier. Cashier enters gift certificate ID.
2. System displays the balance of the gift certificate.
3. Cashier enters an amount to pay part of the total payment.
4. System records the gift payment.

Figure 1. A simplified use case model and its textual description for the POS system (adapted from [33]).

due to any condition, the system would take an alternative flow. The alternative flow is then either an optional or exceptional behavior or any variation of the normal behavior. Each use case flow consists of actions performed either by the system or actors. An execution of such a sequence of actions specified in a use case is referred to as *an instance of a use case* or *a scenario*. A flow is often further structured into steps or subflows [5].

A use case can participate in the following relationships: *generalization*, *inclusion*, and *extension*. A *use case generalization* denotes "*the relationship between a general use case and a more specific use case that inherits and adds features to it*" [34]. In this work, we concentrate on the two other relationships.

Inclusion. A use case inclusion denotes "*the inclusion of the behavior sequence of the supplier use case into the interaction sequence of a client use case, under the control of the client use case at a location the client specifies in its description*" [34]. For example, as shown in Fig. 1, the client use case `ProcessSales` includes the supplier use case `HandleCreditPayment`. When the `ProcessSales` use case reaches the inclusion point at `Step8`, it begins executing the `HandleCreditPayment` use case until it is complete. Then it resumes executing the `ProcessSales` use case at `Step9`.

Extension. A use case exension is "*a relationship from an extension use case to a base (extended) use case, specifying how the behavior defined for the extension use case can be inserted into the behavior defined for the base use case*" [34]. The locations of the base use case at which the extension might be inserted are defined by an *extension point*: The extension use case is invoked only when the current scenario of the base use case reaches a point where the guard condition defined by the extension point evaluates to true. When the execution of the extension use case is complete, the flow returns to the original use case at the referenced point. For example, Fig. 1 shows the base use case `ProcessSales` extended by the extension one `HandleGiftCertificatePayment` at the extension point `E1`. The guard condition is true when the `Customer` pays by a gift certificate.

3.2. Object Model

An object model is an effective means to represent a problem domain. It allows defining concepts of the domain and relationships between them. The object model also states properties and constraints to sharpen the domain. An object model can be represented using a UML class diagram attached with OCL constraints [35]. Statements in a use case specification are often expressed based on the object model. An interpretation of an object model captures a current system state referred to as *a snapshot*. A snapshot is constituted by objects, links, and attribute values.

Example 2. Figure 2 shows an object model for the POS in the form of a class diagram. Figure 3 shows a snapshot in the form of an object diagram. It is an interpretation of the object model that is depicted in Fig. 2.

An object model in the form of a class diagram is often attached with OCL conditions, as explained in [35], for either properties or restrictions on the domain. The OCL is a formal language with the following characteristics. First, OCL expressions, which might be either object constraints or queries, do not have side effects. Second, the OCL is a typed language. Each valid (well-formed) OCL expression has a type, which is the type of the evaluated value of this expression. The type system of OCL includes basic types (e.g., `Integer, Real, String, and Boolean`), object types, message types, and collection types (e.g., `Collection(t), Set(t), Bag(t), and Sequence(t)`) to describe collections of values of type `t`. Third, OCL is often employed for different situations: i) to specify invariants, i.e., conditions that must be true for all instances of the class in all system states; ii) to express pre- and postconditions on operations; iii) to express guard conditions within a statechart; and iv) to query a given system state.

Example 3. The POS object model is restricted by the constraint "*For any `SalesLineItem` described by a `ProductDescription` and for any `Item` recorded by the `SalesLineItem`, we have the `ProductDescription` describes the `Item`, i.e., there is a link `Describes` between the `ProductDescription` and the `Item`*". This constraint is expressed in OCL as follows.

```
context SalesLineItem:
self.item->forAll(it:Item |
   it.prdtDesc = self.productDesc)
```

3.3. Research Question

A use case model within model-driven approaches often needs to be transformed into software artifacts such as analysis and design, implementation, and testing models. As an initial effort to achieve the goal, we need to tackle the following challenge: How can we obtain a precise specification of the use case, i.e., that covers the general description of the use case, use case constraints (pre- and postconditions and invariants), use case actions and scenarios, as well as relationships between use cases? This work concentrates on developing a domain-specific language to precisely specify use cases, resulting a basis to define transformations for generating software artifacts from the use case specification.

4. Specifying the FRSL Syntax

This section introduces a DSL called FRSL to specify use cases, resulting in a precise specification of functional requirements.

4.1. Abstract Syntax

We define a FRSL metamodel as shown in Fig. 4 for a technical domain of use cases. A FRSL model provides an overview description of use cases and captures the detailed information of use cases as follows: i) The domain model in the form of a UML/OCL class diagram; ii) The use case structure that is defined by use case relationships, inclusion and extension; iii) Use case scenarios; iv) Snapshot patterns to express the pre- and post condition of either use cases or steps; and v) The guard condition of either alternative flows or rejoining steps or use case extensions.

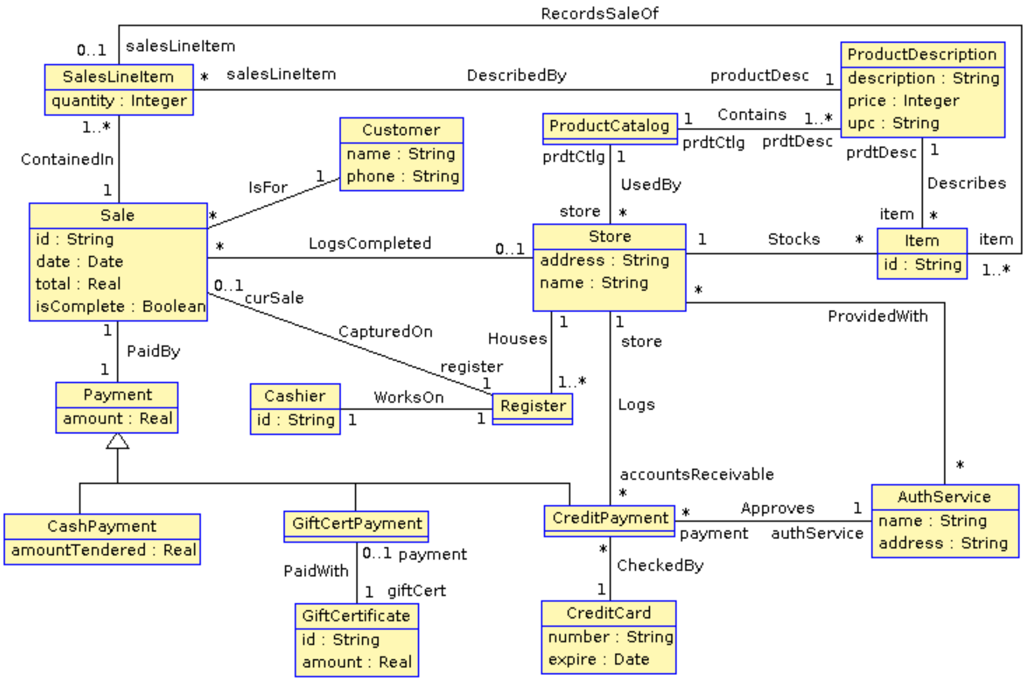


Figure 2. The POS object model represented in the form of a class diagram.

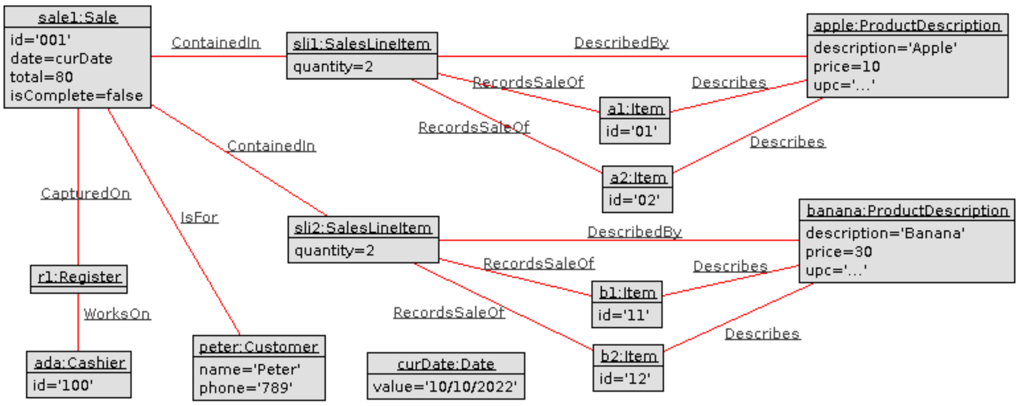


Figure 3. A snapshot of the POS represented by an object diagram.

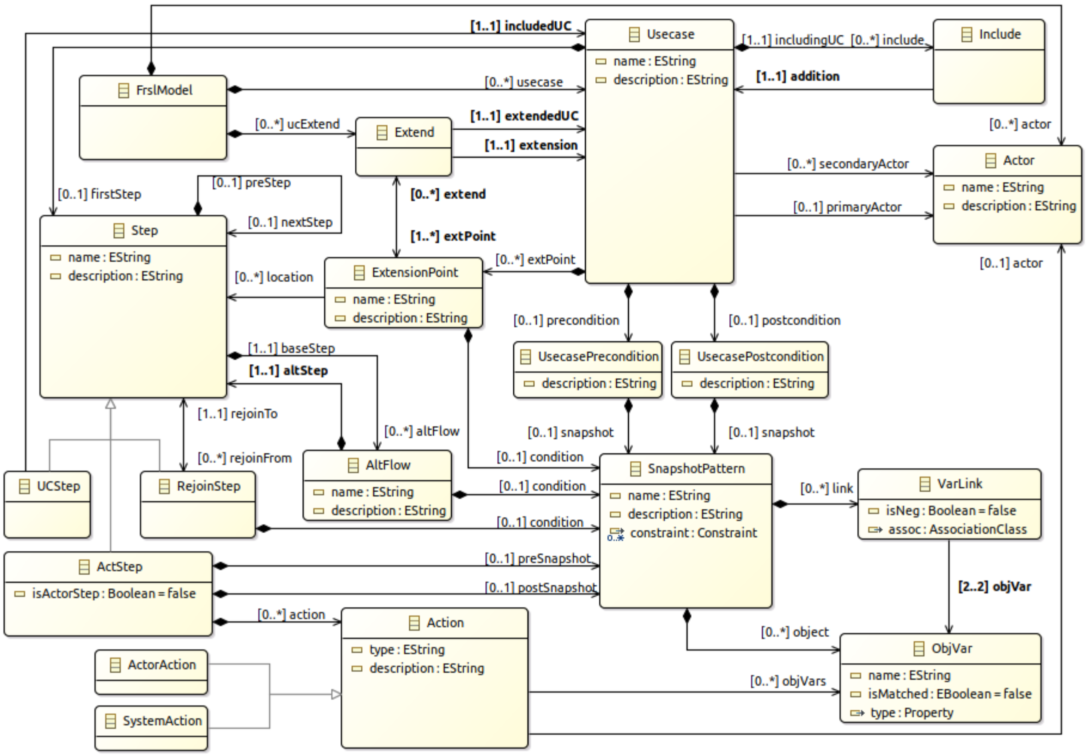


Figure 4. The FRSL metamodel.

4.1.1. Representing Domain Model

A FRSL specification represents a domain model in the form of a class diagram. Therefore, the UML meta-concepts for class diagram and OCL are embedded into the FRSL metamodel to represent the domain model.

4.1.2. Representing Use Case Structure

The FRSL meta-concepts to represent the use case structure mainly include the UML meta-concepts for use case diagrams [1]: `UseCase`, `Actor`, `Include`, `Extend`, and `ExtensionPoint`. A use case (`extendedUC`) might be extended by other use cases (`extension`). Such extensions occur when the guard condition given by a corresponding extension point is fulfilled.

4.1.3. Representing Use Case Scenarios

Use case scenarios are represented using the following FRSL meta-concepts: `Step`, `ActStep`, `UCStep`, `RejoinStep`, `AltFlow`, `Action`, `ActorAction`, and `SystemAction`. A step (`Step`) could be either an action step or a rejoining step or a use case step w.r.t. `ActStep`, `RejoinStep`, and `UCStep`. An `ActStep` step contains actor/system actions (`ActorAction`, `SystemAction`). A `RejoinStep` step aims to determine the next step of the current execution by evaluating a guard condition. The guard condition could be expressed by a snapshot pattern (`SnapshotPattern`). A `UCStep` step invokes another use case as an included use case (`addition`). The basic flow of a use case (`Usecase`) is defined by its property `firstStep` and the property `nextStep` of the first step (`Step`). Any alternative flow (`AltFlow`) is defined by the property `altFlow` of the base step (`Step`). The base step here is the first step of the alternative flow. The guard condition for an alternative flow could be also represented by a snapshot pattern.

4.1.4. Representing Snapshots

The remaining FRSML meta-concepts (`SnapshotPattern`, `ObjVar`, `VarLink`, and `Constraint`) represent conditions and use case states in the form of snapshot patterns.

`SnapshotPattern`. A snapshot pattern represents system states, referred to as system snapshots. A snapshot includes a set of objects and links between them. The snapshot needs to satisfy a given set of constraints. Specifically, a snapshot pattern consists of a set of object variables (`objVars`), variable links (`varLinks`), and constraints where:

- Each `objVar` represents an object of the current system snapshot;
- Each `varLink` represents a link between two objects defined by `objVars`. The link is instantiated by an association within the domain model;
- Each constraint is a restriction on the current system snapshot, expressed as an OCL condition on `objVars`.

`ObjVar`. An object variable of a snapshot pattern represents either an object of the *problem domain* or of the *software domain*. These domains correspond to the problem world and the machine solution as explained in [16]. The problem domain is part of reality defining the context of the system-to-be with two parts: i) the software-to-be as a component of the system-to-be, corresponding to the software domain; and ii) the so-called environment of the software-to-be, that consists of the remaining components of the system-to-be.

A current system state, that consists of objects, links, and constraints, can be seen as a combination of two snapshots, one belongs to the problem domain and the other belongs to the software domain. These two snapshots could be expressed by two object diagrams of the same class diagram. An object of the problem domain is often tracked by another object of the software domain. Such a tracking could be expressed by a link of an association `_Track`. The `_Track` is a reflexive association from the domain class `_DomainClass` to itself. Every domain class inherits the `_DomainClass`. For example, an object `_item:Item` of the problem domain, that represents a physical thing in reality and can not be directly monitored or controlled by the software, should be tracked by another object `item:Item` of the software domain. The software can only directly manipulate on this “image” object instead of the original one.

To explicitly specify that there is no link between two objects referred to by objVars, we need to employ so-called *negative links*, whose `isNeg` property is true. We also refer to `objVars` as *matched objects* if the `objVars` have just been updated in the current state, i.e., if the `objVar` is not a matched object, we have `objVar@pre = objVar`. A snapshot pattern represents system states that satisfy certain constraints. Therefore, we could employ a snapshot pattern as a condition expression to express i) the pre- or postcondition of a use case; and ii) the guard condition for either invoking an alternative flow or rejoining another step or extending a use case.

4.2. Concrete Syntax

This section presents a textual syntax for FRSL. A FRSL specification basically includes two parts: the domain model and the specification of use cases that includes snapshots, scenarios, and use case extension.

4.2.1. Specifying Snapshots

A snapshot pattern includes also so-called negative objects that existed in the previous system state and are destroyed in the current state. They are instantiated by the concept `NegObjVarCS` and stated by the syntax of this form `!<objVarName>;`. A negative link is specified by the form `!(<varLink>);`. Listing 1 shows a snapshot pattern in FRSL for the use case `ProcessSale`.

Listing 1. A FRSL snapshot pattern for the use case ProcessSales

```
store: Store;
sale: Sale;
pos: Register;
$payment: Payment;
(sale, pos): CapturedOn;
(sale, payment): PaidBy;
!(store, sale): LogsCompleted;
[sale.isComplete = false]
```

This pattern contains a matched object `payment`, marked by the '$'. To represent `objVars` as instances of the problem domain, its name should start with the character "_". We provide the following syntax to express there is no link between two objects: `!(store,sale):LogsCompleted`. The syntax `!object` is to state the object is destroyed. Thus, it is no need to explicitly specify in the post-snapshot the objects and links that are already appeared in the pre-snapshot.

4.2.2. Specifying Use Case Scenarios

A FRSL specification provides the information about primary and secondary actors, scenarios, and snapshot patterns to represent the pre- and postcondition of the use case as well as extension conditions. In this syntax the precondition (`preSnapshot`) and postcondition (`postSnapshot`) of the action step are represented as snapshot patterns.

Listing 2. A FRSL specification for system steps

```
sysStep step03
description = '3. The system creates a new sale and
     requires the cashier to enter items'
from
_sale: Sale;
_pos: Register;
_cashier: Cashier;
$pos: Register;
$cashier: Cashier;
$curDate: Date;
(cashier, pos): WorksOn;
(_pos, pos): _Tracks;
(_cashier, cashier): _Tracks;
to
sale: Sale;
(sale, pos): CapturedOn;
(_sale, sale): _Tracks;
[sale.oclIsUndefined() = false]
[sale.total = 0]
[sale.date = curDate]
actions
Cashier <- saleInfor: Sequence(OclAny) = Sequence{sale
     .id, sale.total};
Cashier <- cashierInfor: Sequence(OclAny) = Sequence{
     cashier.name};
end
```

Listing 2 shows a system step of the use case `ProcessSales`. This specification contains two system actions to display information to the actor `Cashier` via the object variables `saleInfor` and `cashierInfor`.

4.2.3. Specifying Use Case Extension

The locations characterized in the extension point refer to the steps where the extending use case could be invoked. The form to specify a location might be either `<stepName>` or `<stepName01>::<stepName02>` or `<stepName>::all`. The first form means the extension should occur at the step `<stepName>`. The second form means the extension point occurs at any step that belongs to part of the scenario starting at the step `<stepName01>` and reaching to the step `<stepName02>`. The third form means the extension point could occur at any step of the scenario starting from the step `<stepName>`.

Listing 3. A FRSL specification for use case extensions

```
extensionPoint PaidByGiftCert at {step08}
  description = 'It occurs as the Customer would pay
       with gift certificate'
when
  $_giftCert:GiftCertificate;
end
HandleGiftCertPayment extends ProcessSales at {
     PaidByGiftCert}
```

Listing 3 shows that the use case `HandleGiftCertPayment` extends the use case `ProcessSales` at `Step8`.

5. A Formal Semantics

This section aims to define a formal semantics of the FRSL. Specifically, we characterize the execution of a FRSL specification as sequences of state transitions. Each current state is represented as an object model, resulting in an operational semantics of the FRSL.

5.1. Preliminaries

An *object model* is often represented in the form of a UML class diagram together with OCL constraints. We could formally define object model as follows.

Definition 1 (Object Model)**.** An object model is the structure $\mathcal{M} = (CLASS, ATT_c, ASSOC,$ $associates, roles, multiplicities, \prec, constraints)$ where

1. $CLASS \subseteq \mathcal{N}$ is a set of names to represent classes, where $\mathcal{N} \subseteq \mathcal{A}^+$ is a non-empty set of names over alphabet $\mathcal{A}$. Each class $c \in CLASS$ induces a type $t_c \in T$ whose values refer to objects of the class.
2. ATT_c is the attributes of a class $c \in CLASS$, defined as a set of signatures $a : t_c \to t$, where the attribute name a is an element of $\mathcal{N}$, $t_c \in T$ is the type of class c, and $t \in T$ is the type of the attribute.
   - $associates : ASSOC \to CLASS^+$ maps each association name to a list of participating classes. The list has at least two elements.
   - $roles : ASSOC \to \mathcal{N}^+$ maps each association to a list of role names. Each class of the association is assigned with a unique role name.
   - $multiplicities : ASSOC \to \mathcal{P}(\mathbb{N}_0)^+$ maps each association to a list of multiplicities. Each class of the association is assigned with a multiplicity. The multiplicity is a non-empty set of natural numbers (an element of the power set $\mathcal{P}(\mathbb{N}_0)^+$) different from $\{0\}$.
3. $\prec$ is a partial order on $CLASS$ representing the generalization hierarchy of classes.
4. $constraints$ are OCL conditions formed by an OCL algebra, that is an extension of the object model structure, as explained in [36]. □

A *snapshot* as an interpretation of an object model is constituted by objects, links, and attribute values. A formal definition for snapshots is provided as follows.

Definition 2 (Snapshot)**.** A snapshot of an object model $\mathcal{M}$ is the structure $\sigma(\mathcal{M}) = (\sigma_{CLASS},$ $\sigma_{ATT}, \sigma_{ASSOC})$ such that:

1. For each $c \in CLASS$, the finite set $\sigma_{CLASS}(c)$ contains all objects of class $c \in CLASS$ existing in the snapshot: $\sigma_{CLASS}(c) \subset oid(c)$.
2. Functions σ_{ATT} assign attribute values for each object in the state. $\sigma_{ATT}(a) : CLASS(c) \to I(t)$ for each $a : t_c \to ATT^*_c$.
3. For each $as \in ASSOC$, there is a set of current links: $\sigma_{ASSOC}(as) \subset I_{ASSOC}(as)$. A link set must satisfy all multiplicity specifications: $\forall i \in \{1, ..., n\}, \forall l \in \sigma_{ASSOC}(as)$: $|\{l' | l' \in \sigma_{ASSOC}(as) \wedge (\pi_i(l') = \pi_i(l))\}| \in \pi_i(multiplicities(as))$

where

- $I(t)$ is the domain of each type $t \in T$.
- $oid(c)$ is the objects of each $c \in CLASS$. The set is often infinite. $I_{CLASS}(c) = oid(c) \cup \{oid(c') | c' \in CLASS \wedge c' \prec c\}$.
- ATT^*_c is the direct and inherited attributes of the class c: $ATT^*_c = ATT_c \cup_{c \prec c'} ATT_{c'}$.
- $I_{ASSOC}(as) = I_{CLASS}(c_{p1}) \times ... \times I_{CLASS}(c_n)$ interprets the association as, where $associations(as) = \langle c_{p1}, c_{p2}, ..., c_n \rangle$, $as \in ASSOC$, and $c_{p1}, c_{p2}, ..., c_n$ are the classes. Each $l_{as} \in I_{ASSOC}(as)$ is a link.
- $\pi_i(l)$ projects the i^{th} element of the list l. □

Example 4. Figure 2 shows an object model in the form of a class diagram. One of the snapshots of the object model is presented as in Fig. 3 in the form of an object diagram.

5.2. Use case meta-concepts

We define a *snapshot pattern* as a parameterized snapshot. Each action within a use case scenario, performed by either the system or the actor, is specified by pair of snapshot patterns as the pre- and postcondition of the action within a contract. This allows us to characterize each use case scenario as a sequence of snapshot patterns.

Definition 3 (Snapshot Pattern)**.** A *snapshot pattern* of an object model CD is a tuple

$\langle objVars, varLinks, conds \rangle$ defining a set of snapshots of the *CD*, where

- *objVars* are variables referring to objects of a snapshot of the *CD*;
- *varLinks* represent relationships between the *objects*;
- *conds* are OCL conditions within the context of the *CD*.

A snapshot of the snapshot set is so-called *matched by* the snapshot pattern. □

Definition 4 (Operators on Snapshot Patterns)**.** Let p, q be given as snapshot patterns of an object model *CD* where

- $|p|$ denotes a set of all snapshots each of which can be matched to p. Such a matching assigns the variables of p to the objects and links of the snapshot such that its *conds* are fulfilled;
- $|CD|$ denotes the set of all snapshots of the *CD*.

We define logical operators on snapshot patterns as follows.

- p is a *total snapshot pattern*, denoted by $\top$, if $|p|$ coincides with $|CD|$.
- p is an *empty snapshot pattern*, denoted by $\bot$, if $|p|$ is empty.
- $\neg p$ denotes a snapshot pattern that satisfies $|\neg p| = |CD| \setminus |p|$.
- $p \wedge q$ denotes a snapshot pattern that satisfies $|p \wedge q| = |p| \cap |q|$.
- $p \vee q$ denotes a snapshot pattern that satisfies $|p \vee q| = |p| \cup |q|$. □

Definition 5 (Refinement & Equivalence of Snapshot Patterns)**.** Let p and q be two snapshot patterns of an object model *CD*.

- We say *p refines q*, denoted by $p \preceq q$, iff $\forall s \in CD,\ s \models p \implies s \models q$.
- We say p is *equivalent to q*, denoted by $p \equiv q$, iff $p \preceq q \wedge q \preceq p$. □

Listing 4. Snapshot pattern $snap_L$

```
$sale_L$: Sale;
$reg_L$: Register;
($sale_L$, $reg_L$): CapturedOn;
[$sale_L.total>0$]
```

Listing 5. Snapshot pattern $snap_R$

```
$sale_R$: Sale;
$reg_R$: Register;
$sli_R$: SalesLineItem;
($sale_R$, $reg_R$): CapturedOn;
($sli_R$, $sale_R$): ContainedIn;
[$sale_R.total=80$]
```

Example 5. The snapshot pattern $snap_R$ as shown in Listing 5 would refine the snapshot pattern $snap_L$ as shown in Listing 4 because: i) Both snapshot patterns could be matched to the snapshot depicted in Fig .3; and ii) Any snapshot matched by $snap_R$ is also matched by $snap_L$. We could write $snap_R \preceq snap_L$.

Definition 6 (Problem and Software Domain Model)**.** A *software domain model SDM* of a software system within *a problem domain PDM* is an object model such that

- *PDM* is an object model to represent instances as individuals of the underlying domain and associations between them.
- any current system state represented by a snapshot of the *PDM* is also represented by a corresponding snapshot of the *SDM*;
- any object captured by a class cls_P of the *PDM* is also represented by a corresponding class cls_S of the *SDM*. Such a corresponding could be maintained by the one-one association $\_Track(cls_P, cls_S)$. Only the agent of the *PDM*, i.e., by actor actions, can directly control or monitor entities of the cls_P. The software, i.e., by system actions, can only indirectly access the entities via the corresponding ones of the cls_S as their "image".

A *unified domain model UDM* of the *PDM* and *SDM* is an object model that consists of all their classes, associations, and the tracking association. □

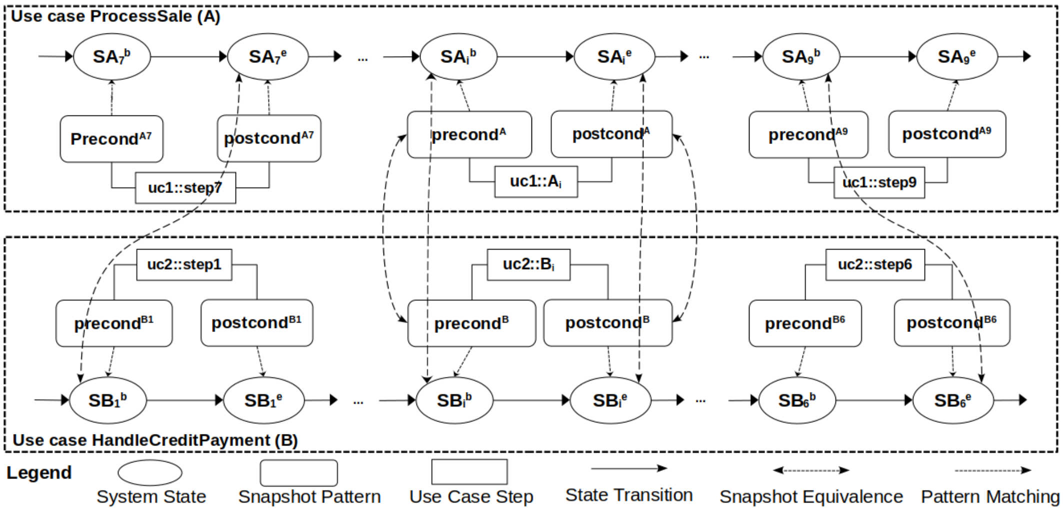


Figure 5. An illustration for use case inclusion.

Definition 7 (Use Case)**.** A *use case* of a system within a *SDM* w.r.t. *PDM* is a tuple $\langle \Sigma, \mathcal{S}, E, S, s_0, \rightarrow, F, \delta \rangle$ such that[1]

- $\Sigma = \Sigma_e \cup \Sigma_s \cup \{\varepsilon, \varepsilon', \varepsilon''\}$ where Σ_e denotes actor actions, Σ_s denotes system actions, and $\{\varepsilon, \varepsilon', \varepsilon''\}$ handle use case flows;
- $\mathcal{S}$ is all the snapshot patterns of the underlying unified domain model;
- $E = \left\langle e_p, E_s \right\rangle$ where e_p denotes an active object (so-called agent) of *PDM* as *primary actor* and E_s denotes agent objects as *secondary actors*;
- S is a finite non-empty subset of $\mathcal{S}$ to represent states;
- s_0 is an initial state, an element of the S;
- $\rightarrow \subset S \times \mathcal{S} \times \Sigma \times \mathcal{S} \times S$ is a transition relation, written $\alpha \xrightarrow{c_{p1}, a, c_{p2}} \beta$ for $\left\langle \alpha, c_{p1}, a, c_{p2}, \beta \right\rangle \in \rightarrow$, such that
  - if $a \in \Sigma_e$ (Σ_s) then the a is an actor (system) action, and both c_{p1} and c_{p2} are snapshot patterns of *PDM* (*SDM*). The c_{p1} and c_{p2} represents the pre- and postcondition of the a, respectively;
  - if $a = \varepsilon$ then $\alpha \equiv \beta \wedge c_{p1} \equiv c_{p2}$ and the c_{p1} is referred to as a constraint for the current state α;
  - if $a \in \{\varepsilon', \varepsilon''\}$ then $c_{p1} \equiv c_{p2}$ and the c_{p1} is referred to *a guard condition* of *an extension point* (in the case $a = \varepsilon'$) or *a rejoining point* (in the case $a = \varepsilon''$);
  - if $s_0 \xrightarrow{c_{p1}, \varepsilon, c_{p1}} s_0$ then the c_{p1} is referred to as *the precondition of the use case*;
  - if $s \xrightarrow{c_{p2}, \varepsilon, c_{p2}} s \wedge s \in F$ then c_{p2} is referred to as *the postcondition of the use case*;
- F is a subset of S containing final states.
- $\delta : \Sigma \rightarrow$ *Bool* is a function such that $\exists! n, \forall 0 \leq i < n, \exists! s_{i+1} : s_i \xrightarrow{a_i} s_{i+1} \wedge \delta(a_i) \wedge s_n \in F \wedge \forall a \in \Sigma \setminus \{a_0, ..., a_n\} : \neg\delta(a)$. This function is to check if an action belongs to the basic flow of the use case.

A *use case model* of a system consists of all the use cases of the system. □

Definition 8 (Scenario)**.** A scenario *sc* of a use case $uc = \langle \Sigma, \mathcal{S}, A, S, s_0, \rightarrow, \delta, F \rangle$ is a transition sequence ($s_0 \xrightarrow{c^1_{p1}, a_1, c^1_{p2}} s_1 \xrightarrow{c^2_{p1}, a_2, c^2_{p2}} \dots \xrightarrow{c^n_{p1}, a_n, c^n_{p2}} s_n$) such that $\{s_0, ..., s_n\} \cap F = \{s_n\}$.

[1]the 'E' in this definition means the 'Environment' of the software, i.e., the actor.

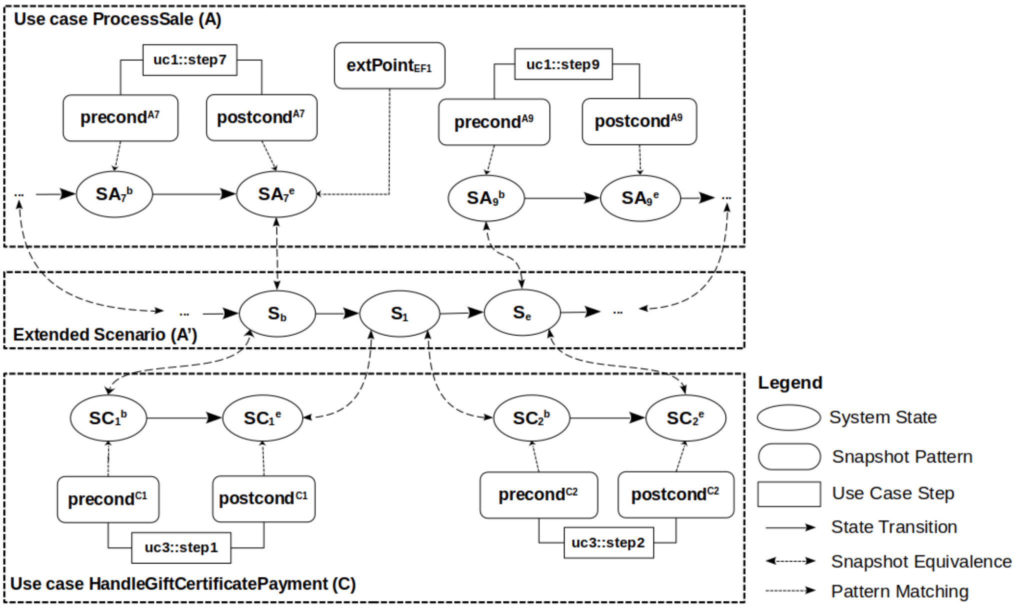


Figure 6. An illustration for use case extension.

- The scenario sc is referred to as *a basic flow* when $\forall i \in \{0, ..., n\} : \delta(a_i)$.
- We write $s_i \overset{sc}{\Longrightarrow} s_j (\forall 0 \leq i < j \leq n)$ to denote a transition sequence of sc. Therefore, we could write $s_0 \overset{sc}{\Longrightarrow} s_n$ to denote the scenario sc.
- We write $s_i \xrightarrow{sc} s_{i+1} (\forall 0 \leq i < n)$ to denote a transition step of sc.
- $|sc|$ denotes all the states of the scenario sc, i.e., $|sc| = \{s_0, s_1, ..., s_n\}$.
- $|uc|$ denotes all the scenarios of the uc. □

Definition 9 (Use Case Inclusion)**.** A use case A includes a use case B iff $\exists sc \in |A|, s_b \overset{sc}{\Longrightarrow} s_e$ such that the following conditions are fulfilled.

i) $\forall s^A \in |A|, s_b = s_{A_1} \overset{s^A}{\Longrightarrow} s_{A_n} = s_e : \exists (s_{B_1} \overset{s^B}{\Longrightarrow} s_{B_n}) \in |B|$ (a scenario of B) such that $\forall 1 \leq i \leq n, s_{A_i} \xrightarrow{c^A_{p1_i}, a^A_i, c^A_{p2_i}} s_{A_{i+1}}, s_{B_i} \xrightarrow{c^B_{p1_i}, a^B_i, c^B_{p2_i}} s_{B_{i+1}} : s_{A_i} \equiv s_{B_i} \wedge s_{A_{i+1}} \equiv s_{B_{i+1}} \wedge c^A_{p1_i} \equiv c^B_{p1_i} \wedge c^A_{p2_i} \equiv c^B_{p2_i} \wedge a^A_i \equiv a^B_i$;

ii) $\forall (s_{B_1} \overset{s^B}{\Longrightarrow} s_{B_n}) \in |B|, s_{B_1} \equiv s_b, s_{B_n} \equiv s_e :$ $\exists s^A \in |A|, s_b = s_{A_1} \overset{s^A}{\Longrightarrow} s_{A_n} = s_e, \forall 1 \leq i \leq n,$ $s_{A_i} \xrightarrow{c^A_{p1_i}, a^A_i, c^A_{p2_i}} s_{A_{i+1}}, s_{B_i} \xrightarrow{c^B_{p1_i}, a^B_i, c^B_{p2_i}} s_{B_{i+1}} :$ $s_{A_i} \equiv s_{B_i} \wedge s_{A_{i+1}} \equiv s_{B_{i+1}} \wedge c^A_{p1_i} \equiv c^B_{p1_i} \wedge c^A_{p2_i} \equiv c^B_{p2_i} \wedge a^A_i \equiv a^B_i$. □

Example 6. Figure 5 illustrates how the `ProcessSales` use case includes the `HandleCreditPayment` use case. The description of these use cases is shown in Fig. 1. The post-state of `Step7` of the `ProccessSales`, denoted by SA^e_7, coincides with the pre-state of `Step1` of the `HandleCreditPayment`, denoted by SB^b_1. Since then, each state transition of the `ProccessSales` is defined by a corresponding step of the `HandleCreditPayment`: (SA^b_i, SA^e_i) coincides with (SB^b_i, SB^e_i). The post-state of the last step of the `HandleCreditPayment`, denoted by SB^e_6, coincides with the pre-state of `Step9` of the `ProcessSales`, denoted by SA^b_9.

Definition 10 (Use Case Extension)**.** Let be given A, B as two use cases of a unified domain model

UD and p as a snapshot pattern of *UD*. The use case A is extended by the use case B from a so-called *extension point* p, resulting a new use case A' of *UD* such that $\forall s^A \in |A|, s_b \xrightarrow{s^A} s_e, s_b \xrightarrow{c^A_{p1}, a^A, c^A_{p2}} s_e, \forall (s_{B_1} \overset{s^B}{\Longrightarrow} s_{B_n}) \in |B|$ there exists a corresponding scenario $s_{A'_1} \overset{s^{A'}}{\Longrightarrow} s_{A'_n}$ that fulfils the following conditions.

i) $s_b \xrightarrow{s_{A'}} s_{B_1} \wedge s_{B_n} \xrightarrow{s_{A'}} s_e \wedge s_b \xrightarrow{s_{A'}} s_e$;

ii) $s_b \xrightarrow{p, \epsilon', p} s_{B_1} \wedge s_{B_n} \xrightarrow{\top, \varepsilon'', \top} s_e \wedge s_b \xrightarrow{\neg p \wedge c^A_{p1}, a^A, c^A_{p2}} s_e$;

iii) $\forall 1 \leq i \leq n, s_{A'_i} \overset{c^{A'}_{p1_i}, a^{A'}_i, c^{A'}_{p2_i}}{\Longrightarrow} s_{A'_{i+1}}, s_{B_i} \xrightarrow{c^B_{p1_i}, a^B_i, c^B_{p2_i}} s_{B_{i+1}} : s_{A'_i} \equiv s_{B_i} \wedge s_{A'_{i+1}} \equiv s_{B_{i+1}} \wedge c^{A'}_{p1_i} \equiv c^B_{p1_i} \wedge c^{A'}_{p2_i} \equiv c^B_{p2_i} \wedge a^{A'}_i = a^B_i$;

iv) $\forall s \in |s^A|, ((s \overset{s^A}{\Longrightarrow} s_b) \iff (s \overset{s^{A'}}{\Longrightarrow} s_b)) \wedge ((s_e \overset{s^A}{\Longrightarrow} s) \iff (s_e \overset{s^{A'}}{\Longrightarrow} s))$. □

Example 7. Figure 6 illustrates a use case extension: The `ProcessSales` is extended by the `HandleGiftCertificatePayment`. The extension point $extPoint_{EF1}$, that refers to `Step8` of the use case `ProcessSales` as shown in Fig. 1, would match the post-state SA^e_7 of `Step7` (represented by *uc1* :: *step7*). This state coincides with the state S_b and the pre-state SC^b_1 of `Step1` of the `HandleGiftCertificatePayment` (represented by *uc3* :: *step1*). After finishing this extending use case at the post-state SC^e_2, the execution rejoins the extended use case `ProcessSales` at `Step9`.

6. Tool Support

We have implemented a tool support for FRSL. The VNU-FRSL tool[2] is an open source project developed based on the framework Xtext and the OCL Eclipse plugin named OCLInEcore[3] [37]. Figure 7 shows part of the FRSL specification of the use case `ProcessSales`. The panel on the bottom-left of the figure shows operations of the use case class corresponding the steps of the underlying use case.

Figure 8 depicts the overall architecture of VNU-FRSL. The tool is developed based on the Eclipse platform with a plugin architecture that allows for extensibility. The plugin architecture consists of two main parts: the core module and the additional modules. This design allows for modular functionality to be added and plugged to the core module, providing scalability, flexibility, and separation of application features. The core module allows specifying the abstract syntax, which will be taken as input to other functional modules for generating other artifacts.

Figure 9 shows the VNU-FRSL source code structure based on the Eclipse architecture. The left pane of the figure shows the main plugins, which realized the functions: i) the FRSL specification; ii) the constraint specification in UML/OCL; iii) the model-to-model transformations in ATL [38]; and iv) the model-to-text transformations in Acceleo [39]. The FRSL's abstract syntax is specified based on the Eclipse-based metamodel system (i.e., as an ecore model). Meanwhile, the FRSL's textual syntax is built using the Eclipse/Xtext technology.

7. Evaluation and Discussion

This section presents an evaluation of the expressiveness of FRSL in comparison with current use case specification languages: RUCM [40], UC-B [8], UC2AD [41], UCM [21], SelabReq [12], RSL [11], and USL [13]. Based on recent surveys [2, 3] of use case specification, we propose four criteria of expressiveness as follows:

[2]https://github.com/vnu-dse/frsl

[3]https://projects.eclipse.org/projects/modeling.mdt.ocl

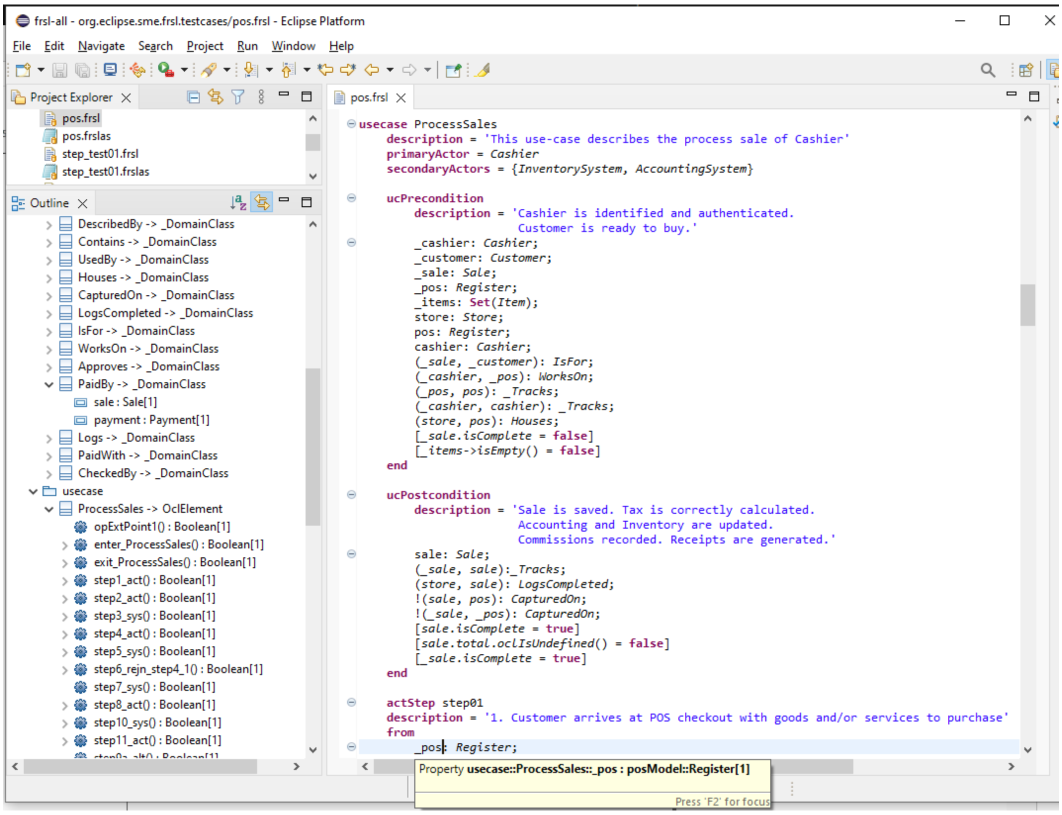


Figure 7. The FRSL specification of the use case `Process Sales`.

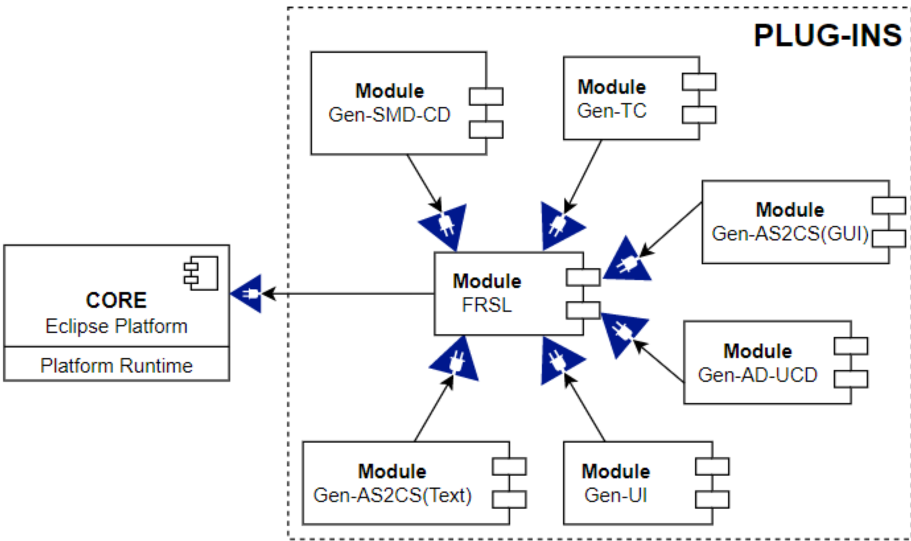


Figure 8. The overall architecture of VNU-FRSL.

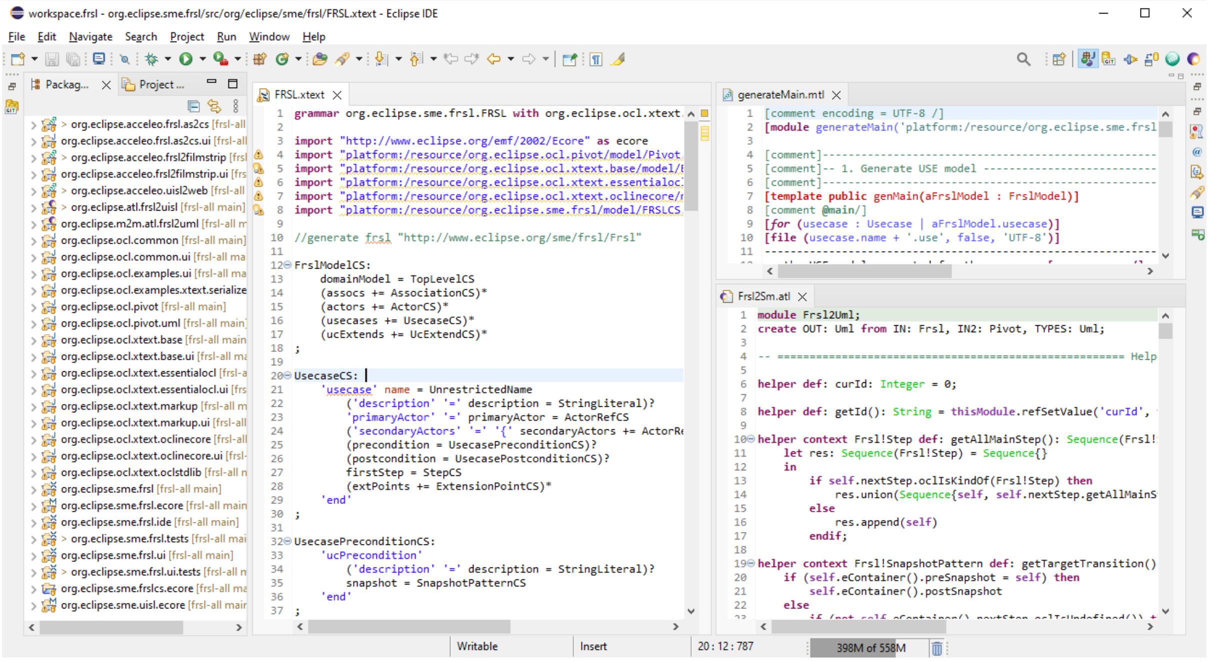


Figure 9. The VNU-FRSL source code structure based on the Eclipse architecture.

C1. This criterion is intended to compare the ability to express use case specification based on templates. According to the survey in [2], among many proposal templates, a use case specification should consist of two parts: a general description of the use case and a specification of the use case behavior. The behavior specification describes the flows of events of the use case with two kinds: the basic flow and alternative flows. A use case scenario then is obtained as a combination of the base flow and alternative ones.

C2. This criterion aims to evaluate the ability to specify i) the control flow over use case behaviors such as branching and looping; and ii) the control flow for concurrently executed behaviors.

C3. This criterion concentrates on the ability to specify actions in the use case behavior description. A precise specification of actions with different kinds of actions [11–13, 21] would open possibility for generating software artifacts from use case specifications as well as for formally verifying system behaviors.

C4. This criterion aims to evaluate the ability to express use case constraints for the following situations: i) pre- and postconditions of scenarios; ii) guard conditions of use case flows; and iii) pre- and postconditions of actions. These constraints are also the basis for generating artifacts and formally verifying use cases.

Table 1 lists the evaluation results for the above criteria. In the table, we use three characters ‘F’, ‘I’, ‘N’ to denote method specification used for each language: ‘F’ stands for formal specification method, ‘I’ stands for informal specification method, and ‘N’ means the information is not captured in the specification. First, considering the criterion C1, the FRSL language allows us to express both the general description and the use case behavior of a use case. This feature is basically based on the meta-concepts of the FRSL metamodel as explained in Section 4.

Table 1. Compare expressiveness between FRSL and the other use case specification languages

	Use case information (UC)	RUCM [40]	UC-B [8]	UC2AD [41]	UCM [21]	SelabReq [12]	RSL [11]	USL [13]	FRSL
(c1)	General information	I	N	N	F	N	N	F	F
	UC flows of events	I	N	I	F	N	N	F	F
	UC scenarios	N	N	N	N	F	F	N	F
(c2)	UC control flows	I	N	F	F	N	F	F	F
	Concurrent actions	N	N	F	N	N	N	F	N
(c3)	Action types	I	N	N	F	F	F	F	F
(c4)	Pre- and postconditions of scenarios	I	F	N	F	N	I	F	F
	Guard conditions	I	F	I	F	N	I	F	F
	Pre- and postconditions of actions	N	F	N	N	N	N	F	F
	Problem domain concepts	N	N	N	N	N	N	N	F

Second, considering the criterion C2, like the four languages UC2AD, UCM, RSL and USL, the FRSL allows us to specify the control flows of a use case. This feature is based on the meta-concepts `RejoinStep` and `AltFlow`, as depicted in Fig. 4: Each of them is associated with the meta-concept `SnapshotPattern` for a guard condition. However, unlike USL, FRSL currently does not support specifying concurrent actions.

Third, considering the criterion C3, like the languages RUCM, UCM, SelabReq, RSL, and USL, the FRSL allows specifying actions with different types. Specifically, the FRSL supports two basic types `ActorAction` and `SystemAction`. The other types are defined based on the attributes `type` and `description` of the meta-concept `Action`.

Finally, considering the criterion C4, the FRSL allows expressing constraints for (p1) guard conditions, (p2) pre- and postconditions of actions, and (p3) pre- and postcondition of scenarios. As depicted in Fig. 4, this feature is based on the association between the meta-concept `SnapshotPattern` and each of these meta-concepts: `RejoinStep`, `AltFlow`, and `ExtensionPoint` (for p1); `ActStep` (for p2); `UsecasePrecondition` and `UsecasePostcondition` (for p3). Additionally, a FRSL specification might refer to the domain concepts in order to specify actions using pre- and postconditions, as explained in Section 4.1.4.

Discussion. This section mainly focuses on evaluating the expressiveness of FRSL qualitatively. The aim is to highlight the key features of FRSL for a precise specification of use cases. A quantitative assessment of this feature as well as other features such as usability is beyond the scope of this paper. We would take such a task as part of our future work.

8. Conclusion

In this paper, we propose a domain-specific language named FRSL (Functional Requirements Specification Language) to precisely specify use cases. The FRSL allows us to obtain a precise specification of functional requirements, thereby enabling increased automation in software development. Specifically, we define the abstract syntax and textual concrete syntax for FRSL, and provide a formal semantics for it. This formal semantics enables a precise explanation of the meaning of use cases and their relationships. Additionally, this serves as an initial effort to define transformations from use cases for the automatic generation of software artifacts. We have implemented an Eclipse plugin to support the FRSL and conducted an evaluation to highlight the language's key features and compare it with current use case specification languages.

In our future plans, we have several goals. First, we aim to define additional features for

FRSL, such as specifying concurrent actions and use case generalization. These features will be incorporated into FRSL's formal semantics framework and implemented using the VNU-FRSL tool. Second, we plan to conduct further studies to quantitatively evaluate FRSL. Finally, another aspect of our future work involves developing transformations for automatic generation of software artifacts from the FRSL specification.

Acknowledgements

This work has been supported by Vietnam National University, Hanoi under Project No. QG.20.54. We wish to thank the anonymous reviewers for numerous insightful feedback on the first version of this paper.

References

[1] OMG, Unified Modeling Language 2.5.1, Object Management Group, 2017.

[2] S. Tiwari, A. Gupta, A Systematic Literature Review of Use Case Specifications Research, Inf. Softw. Technol., Vol. 67, No. C, 2015, pp. 128–158, https://doi.org/10.1016/j.infsof.2015.06.004.

[3] Hurlbut, Russell R., A Survey of Approaches for Describing and Formalizing Use Cases, Tech. rep., Expertech, Ltd., Wheaton, Illinois (1997).

[4] A. C. Cockburn, Writing Effective Use Cases, 1st Edition, Addison-Wesley Professional, 2000.

[5] I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley, 1992.

[6] M. El-Attar, J. Miller, Constructing High Quality Use Case Models: A Systematic Review of Current Practices, Requirements Engineering, Vol. 17, No. 3, 2012, pp. 187–201, https://doi.org/10.1007/s00766-011-0135-y.

[7] A. R. da Silva, D. Savić, S. Vlajić, I. Antović, S. Lazarević, V. Stanojević, M. Milić, A Pattern Language for Use Cases Specification, in: Proc. 20th Int. European Conf. Pattern Languages of Programs (EuroPLoP), ACM, 2015, pp. 8:1–8:18, https://doi.org/10.1145/2855321.2855330.

[8] R. Murali, A. Ireland, G. Grov, UC-B: Use Case Modelling with Event-B, in: Abstract State Machines, Alloy, B, TLA, VDM, and Z, Vol. 9675 of LNCS, Springer International Publishing, 2016, pp. 297–302, https://doi.org/10.1007/978-3-319-33600-8_24.

[9] L. Ribeiro, L. Duarte, R. Machado, A. Costa, E. Cota, J. Santos Bezerra, Use Case Evolution Analysis Based on Graph Transformation with Negative Application Conditions, Science of Computer Programming, Vol. 198, 2020, pp. 102495, https://doi.org/10.1016/j.scico.2020.102495.

[10] T. Yue, L. C. Briand, Y. Labiche, aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models, ACM Transactions on Software Engineering and Methodology, Vol. 24, No. 3, 2015, pp. 13:1–13:52, https://doi.org/10.1145/2699697.

[11] M. Smialek, W. Nowakowski, From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice, Springer, 2015, https://doi.org/10.1007/978-3-319-12838-2.

[12] D. Savić, S. Vlajić, S. Lazarević, I. Antović, V. Stanojević, M. Milić, A. R. da Silva, Use Case Specification Using the SILABREQ Domain Specific Language, Computing and Informatics, Vol. 34, No. 4, 2016, pp. 877–910.

[13] M.-H. Chu, D.-H. Dang, N.-B. Nguyen, M.-D. Le, T.-H. Nguyen, USL: Towards Precise Specification of Use Cases for Model-Driven Development, in: Proc. 8th Int. Symp. Information and Communication Technology (SoICT), Association for Computing Machinery, 2017, pp. 401–408, https://doi.org/10.1145/3155133.3155194.

[14] D. Mairiza, D. Zowghi, N. Nurmuliani, An Investigation into the Notion of Non-functional Requirements, in: Proc Int. Symp. Applied Computing (SAC), ACM, 2010, pp. 311–317, https://doi.org/10.1145/1774088.1774153.

[15] M. D. S. Soares, J. Vrancken, Model-Driven User Requirements Specification using SysML, Journal of Software, Vol. 3, No. 6, 2008, pp. 57–68.

[16] A. van Lamsweerde, Requirements Engineering - From System Goals to UML Models to Software Specifications, Wiley, 2009.

[17] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Bruckner, P. D. Mosses, D. Sannella, A. Tarlecki, CASL: the Common Algebraic Speciÿcation Language, Theoretical Computer Science, Vol. 286, 2002, pp. 153 – 196.

[18] A. Raschke, D. Méry (Eds.), Rigorous State-Based - 8th International Conference, ABZ , Proceedings, Vol. 12709 of LNCS, Springer, 2021, https://doi.org/10.1007/978-3-030-77543-8.

[19] Shweta, R. Sanyal, B. Ghoshal, Automated Class Diagram Elicitation Using Intermediate Use Case Template, IET Software, Vol. 15, No. 1, 2021, pp. 25–42, https://doi.org/10.1049/sfw2.12010.

[20] M. A. Miranda, M. G. Ribeiro, H. T. Marques-Neto, M. A. J. Song, Domain-Specific Language for Automatic Generation of UML Models, IET Software, Vol. 12, No. 2, 2018, pp. 129–135, https://doi.org/10.1049/iet-sen.2016.0279.

[21] M. Misbhauddin, M. Alshayeb, Extending the UML Use Case Metamodel with Behavioral Information to Facilitate Model Analysis and Interchange, Software & Systems Modeling, Vol. 14, No. 2, 2015, pp. 813–838, doi:ttps://doi.org/10.1007/s10270-013-0333-9.

[22] J. M. Almendros-Jiménez, L. Iribarne, Describing Use Cases with Activity Charts, in: U. K. Wiil (Ed.), Proc. Int. Conf. Metainformatics (MIS), Vol. 3511 of LNCS, Springer, 2004, pp. 141–159, https://doi.org/10.1007/11518358_12.

[23] L. Li, Translating Use Cases to Sequence Diagrams, in: Proc. 15th Int. Conf. Automated Software Engineering (ASE), IEEE Computer Society, 2000, pp. 293–298, https://doi.org/10.1109/ASE.2000.873681.

[24] A. Lorenz, H.-W. Six, Tailoring UML Activities to Use Case Modeling for Web Application Development, in: Proc. Int. Conf. the Center for Advanced Studies on Collaborative Research (CASCON), IBM Corp., 2006, pp. 1–26, https://doi.org/10.1145/1188966.1189001.

[25] D.-H. Dang, H. Truong, M. Gogolla, Checking the Conformance between Models Based on Scenario Synchronization, J. UCS, Vol. 16, 2010, pp. 2293–2312, https://doi.org/10.3217/jucs-016-17-2293.

[26] S. Tiwari, D. Ameta, A. Banerjee, An Approach to Identify Use Case Scenarios from Textual Requirements Specification, in: Proc. 12nd India Software Engineering Conference (ISEC), ACM, 2019, pp. 1–11, https://doi.org/10.1145/3299771.3299774.

[27] J. S. Thakur, Automatic Generation of Analysis Class Diagrams from Use Case Specifications, CoRR, Vol. abs/1708.01796, 2017, pp. 1–41, https://doi.org/10.48550/arXiv.1708.01796.

[28] T. Skersys, P. Danenas, R. Butleris, Extracting SBVR Business Vocabularies and Business Rules from UML Use Case Diagrams, Journal of Systems and Software, Vol. 141, 2018, pp. 111–130, https://doi.org/10.1016/j.jss.2018.03.061.

[29] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, M. Blackburn, NAT2TEST$_{SCR}$: Test Case Generation from Natural Language Requirements Based on SCR Specifications, Science of Computer Programming, Vol. 95, 2014, pp. 275–297, https://doi.org/10.1016/j.scico.2014.06.007.

[30] C. Wang, F. Pastore, A. Goknil, L. Briand, Z. Iqbal, Automatic Generation of System Test Cases from Use Case Specifications, in: Proc. 24th Int. Conf. Software Testing and Analysis (ISSTA), ACM, 2015, pp. 385–396, https://doi.org/10.1145/2771783.2771812.

[31] E. Sarmiento, J. Leite, E. Almentero, G. Alzamora, Test Scenario Generation from Natural Language Requirements Descriptions based on Petri-Nets, Electronic Notes in Theoretical Computer Science, Vol. 329, 2016, pp. 123–148, https://doi.org/10.1016/j.entcs.2016.12.008.

[32] P. Metz, J. O'Brien, W. Weber, Specifying Use Case Interaction: Clarifying Extension Points and Rejoin Points, The Journal of Object Technology, Vol. 3, No. 5, 2004, pp. 87–102, https://doi.org/10.5381/jot.2004.3.5.a1.

[33] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Edition, Addison Wesley Professional, 2004.

[34] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, 2nd Edition, Addison-Wesley Professional, 2004.

[35] J. Warmer, A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition, Addison-Wesley Professional, 2003.

[36] M. Richters, A Precise Approach to Validating UML Models and OCL Constraints, Ph.D. thesis, Universität Bremen, Fachbereich Mathematik und Informatik (2002).

[37] E. D. Willink, Aligning OCL with UML, Electron. Commun. Eur. Assoc. Softw. Sci. Technol., Vol. 44, 2011, pp. 1–20, https://doi.org/10.14279/tuj.eceasst.44.664.

[38] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Science of Computer Programming, Vol. 72, No. 1-2, 2008, pp. 31–39, https://doi.org/10.1016/j.scico.2007.08.002.

[39] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, 2nd Edition, Springer Cham, 2017, https://doi.org/10.1007/978-3-031-02549-5.

[40] T. Yue, L. C. Briand, Y. Labiche, Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments, ACM Trans. Softw. Eng. Methodol., Vol. 22, No. 1, 2013, pp. 5:1–5:38, https://doi.org/10.1145/2430536.2430539.

[41] S. Tiwari, A. Gupta, An Approach of Generating Test Requirements for Agile Software Development, in: Proc. 8th Int. Conf. India Software Engineering Conference (ISEC), Association for Computing Machinery, 2015, pp. 186–195, https://doi.org/10.1145/2723742.2723761.