USL: A Domain-Specific Language for Precise Specification of Use Cases and Its Transformations

Chu Thi Minh Hue[1], Dang Duc Hanh[2], Nguyen Ngoc Binh[3] and Le Minh Duc[4]
Department of Software Engineering, VNU University of Engineering and Technology
[1]Hung Yen University of Technology and Education, Vietnam
[2]Corresponding author
[3]Visiting professor, Hosei University, Japan
[4]Hanoi University, Vietnam
E-mail: `{huectm.di12 | hanhdd | nnbinh}@vnu.edu.vn, duclm@hanu.edu.vn`

Keywords: use cases, pre- and postcondition, labelled transition systems, model transformations, domain-specific languages

Received: March 29, 2018

A use case model is often represented by a UML use case diagram and loosely structured textual descriptions. The use case model expressed in such a form contains ambiguous and imprecise parts. This prevents integrating it into model-driven approaches, where use case models are often taken as the source of transformations. In this paper, we introduce a domain-specific language named the Use case Specification Language (USL) to precisely specify use cases. We define the abstract syntax of USL using a metamodel together with OCL wellformedness rules and then provide a graphical concrete syntax for the usability goal. We also define a precise semantics for USL by mapping USL models to Labelled Transition Systems (LTSs). It opens a possibility to transform USL models to software artifacts such as test cases and design models. We focus on a transformation from a USL model to a template-based use case description in order to illustrate our method. A language evaluation of USL is also performed in this paper.

Povzetek: Zasnovan je domensko specifični jezik USL za natančno specifikacijo primerov in transformacij.

1 Introduction

Use case is a software artifact that is commonly used for capturing and structuring the functional requirements. A use case is defined as "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" [1]. As a requirements artifact, the use case model is commonly specified by a UML use case diagram and loosely structured textual descriptions [2]. A key benefit of this use case specification is that it is easy for non-technical stakeholders to learn and use. However, the use case models expressed in this form often contain ambiguous and imprecise parts. This prevents the models from being used directly in model-driven approaches, as a transformation source to produce other analysis and design models. An important challenge here is how to achieve a balance between two seemingly conflicting goals: to specify use case sufficiently precise for model transformation purposes, while achieving the ease-of-use required by non-technical stakeholders.

To this end, a considerable number of works, including [3, 4, 5, 6, 7] and those discussed in [8], have attempted to introduce rigor into use case description. More specifically, T. Yue et al. [3] proposed adding keywords and restriction rules into use case descriptions and then using natural language processing techniques in order to analyze them. Unlike [3, 4, 9, 5, 6, 7], which used natural language description, the works in [10, 11] proposed a formal semantics for use case. On the other hand, UML activity and sequence diagrams are proposed in [12, 13, 14, 15] to model the control flows in use case. A number of other works [4, 16, 17] proposed using a domain specific language (DSL) to specify use case. DSL [18] is a language that is designed specifically for a certain domain to ease the task of describing concepts in the domain.

However, the main limitation of the existing work is that they do not focus on precisely capturing the relevant use case information. These include control flows, steps, system actions, actor actions, and constraints on the use case and its flows. In this paper, we propose a DSL named Use Case Specification Language (USL) to overcome this limitation. The goal of USL is to precisely specify use cases and its model transformation abilities. The USL's domain consists in the task of specifying use cases that capture the system behavior.

Our approach is to define the abstract syntax of USL by extending the metamodels of the UML use case and activity diagrams [2]. Our extension consists in a set of meta-concepts needed for the following purposes: (1) to describe the elements of a typical use case description template; (2) to represent the *basic* and *alternate flows* of a use case in the form of sequential, branched, repeating steps, or concurrent steps; (3) to categorize *steps* and *actions* based on the interaction subjects, which include the *system*, *actor*s and *included/extending use cases*; and (4) to represent *constraints* on the use case, *actions*, and *flows*.

Our precise specification of USL makes it possible to automatically transform USL models into other software artifacts using model transformation techniques. In brief, the main contributions of our work are as follows:

– A DSL named USL to precisely specify use cases. We define the abstract syntax of USL using a metamodel constrained by OCL wellformedness rules [19]. For usability, we define a graphical concrete syntax for USL.

– A formal semantics specification for USL using Labelled Transition System [20]. This semantics enables the automatic transformation of USL models into other software artifacts, such as test cases and class models.

– A support tool that includes a visual editor for constructing USL models. We use this tool and two commonly-used case studies to illustrate our method. We also evaluate USL by comparing it to other related languages.

This work makes four significant improvements from our earlier conference paper [21]. First, regarding to USL specification, we make the abstract syntax precise with the OCL wellformedness rules [19] and define a graphical concrete syntax. Second, we develop an additional case study in order to illustrate how to apply USL in practice. Third, we define a numqber of typical model transformation scenarios for USL model and explain, in more detail, the transformation into template-based use case description. Fourth, we provide an evaluation for USL.

The rest of this paper is organized as follows. Section 2 presents the background and an example for our work. Section 3 overviews our approach. Section 4 presents the USL abstract syntax and explains its formal semantics. Section 5 explains how USL models are transformed into other software artifacts. Section 6 introduces our support tool and illustrates how to apply USL to the *ATM* system case study. This section also presents an evaluation of USL. Section 7 comments on the related works. The paper is closed with the conclusions and future work.

2 Background and motivation

Figure 1 shows a simplified requirement model of a `Library system` including a UML use case model depicted in the part (a) and a UML class diagram capturing corresponding domain concepts of the system which is presented in the part (b). Our paper uses the use case *Lend Book* in the part (a) as a motivating example. This use case is invoked when the librarian executes the book lending

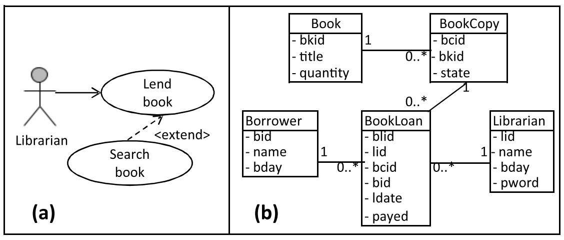


Figure 1: The simplified use case and the conceptual domain model of the `Library system`.

Table 1: A typical use case description template

Use case name: *Lend Book* **Brief description:** The Librarian processes a book loan. **Actors:** Librarian. **Precondition:** The librarian has logged into the system successful. **Postcondition:** If the use case successfully ends, the book loan is saved and a complete message is shown. In the other case, the system displays an error message. **Trigger:** The Librarian requests a book-loan process. **Special requirement:** There is no special requirement.
Basic flow
1. The Librarian selects the *Lend Book* function. 2. The system shows the Lend-book window, gets the current date and assigns it to the book-loan date. 3. The Librarian enters a book copy id. 4. The system checks the book copy id. If it is invalid, it goes to step 4a.1 5. The Librarian enters a borrower id. 6. The system validates the borrower id. If it is invalid, it goes to step 6a.1 7. The Librarian clicks the save-book-loan button. 8. The system validates the conditions to lend book. If it is invalid, the system goes to step 8a.1 9. The system saves the book loan record, then executing two steps 10 and 11 concurrently. 10. The system shows a complete message. 11. The system prints the borrowing bill.
Alternate flows
E1. request searched book 1. The Librarian selects the search function after step 4a.1. 2. The system executes the extending use case Search book. 4a. The book copy id is invalid 1. The system shows an error message, then going to step 3. 6a. The Borrower id is invalid 1. The system shows an error message, then going to step 5. 8a. The lending condition is invalid 1. The system shows an error message. 2. The system ends the use case.

transaction. The use case is represented in a typical template as shown in Table 1.

A typical use case description template [22] often includes two parts, the overview information elements and the detailed description of flows. The first part consists of the following elements: the **use case name**, the use case's **brief description**, the **actors** participating in the use case, the use case's **precondition** and **postcondition**, the **trigger** that initiates the use case and the **special requirement** that describes the non-functional requirements of the use case. The second part contains two types of flows, the **basic flow** and **alternative flows**. The **basic flow** covers what normally happens when the use case is performed. Each use case description has only one **basic flow**. The **alternative flows** cover optional or exceptional behaviour as well as the variations of the normal behaviour. Both the **basic** and **alternative flows** are often further structured into steps or subflows [23, 1]. Moreover, one can smooth use case flows to contain only a **basic flow** and some **alternate flows**.

Each step in flows consists of actions performed either by the system or actors. We refer to actors, the system, and other relation use cases as `interactive subjects`. For example, Step 1 in the **basic flow** is carried out by the Librarian actor, while Step 2 is performed by the system. A step may also contain the information to decide the next moving is another step or another flow or the starting or finishing of concurrent actions. As illustrated in Table 1, Step 2 includes three system actions, "The system shows the Lend-book window", "The system gets the current date" and "The system assigns the current time to the book-loan date". Step 5 contains a branching decision, "If it is invalid, the system goes to step 6a". Step 9 contains the starting point of two concurrent actions: "The system executes two steps 10 and 11 concurrently".

In our work, we consider sentences describing execution of an extending or an included use case as the system's actions. Our previous work [21] divides use case's actions into nine types as follows:

Actor-Input is an actor action to enter data into the system, e.g., the action "The Librarian enters a book copy id" at Step 3 in Table 1 is an `Actor-Input`.

Actor-Request is an actor action to send requests into the system, e.g., the action "The Librarian clicks the save-book-loan button" at Step 7 in Table 1 is an `Actor-Request`.

System-Display is a system action that the system performs operations with the user interface, e.g., the action "The system shows the lend-book window" at Step 2 in Table 1 is a `System-Display`.

System-Operation is a system action to validate a request and data, or process and calculate data, e.g., the action "The system gets the current date" at Step 2 in Table 1 is a `System-Operation`.

System-State is a system action to query or update its internal states, e.g., the action "The system saves the book loan record" at Step 9 in Table 1 is a `System-State`.

System-Output is a system action to send outputs to the actors, e.g., the action "The system shows an error message" at Step 1 of the alternate flow 4a in Table 1 is a `System-Output`.

System-Request is a system action to send requests to a secondary actor, e.g., the action "The system prints the borrowing bill" at Step 11 shown in Table 1 is a `System-Request`.

System-Include is a system action to include another use case.

System-Extend is a system action to extend another use case, e.g., the action "The system executes the extending use case Search book" within Step 2 of the alternate flow E1 in Table 1 is a `System-Extend`.

A use case is successfully executed only if the pre- and postcondition of the use case as well as of the actions of the current flow are satisfied.

Within the context of model-driven development, a use case model, as illustrated in Fig. 1 tends to be taken as a source model of transformations in order to obtain other software artifacts such as analysis models, design models, and test cases. However, the ambiguous and imprecise parts within use case descriptions prevents us from achieving such transformations. In order to integrate use cases into model-driven approaches, we aim to tackle the following challenges:

Capturing the overview structure. The use case model needs to preserve the overview structure of use case descriptions so that a template-based representation of use cases might be generated for non-technical stakeholders.

Specifying precisely control follows. A use case includes a set of scenarios, each of which corresponds to a control flow of the use case. Therefore, the use case model needs to preserve the information of control flows of use cases. This allows us to automatically generate artifacts like test scenarios and behaviour models.

Specifying precisely actions. The use case model needs to precisely represent actions within use case scenarios. A precise specification of actions allows us to capture use case relationships and to generate other artifacts from use cases such as class diagrams, test scenarios, and test objects.

Specifying use case constraints. For the aim to automatically generate test data, the use case model needs to preserve the constraints within use case descriptions, including the pre- and postcondition of use cases, the pre- and postcondition of use case actions, and their guard conditions.

3 Overview of the approach

Figure 2 illustrates our approach. First, we take as input a use case diagram, the textual descriptions of use cases, and a class diagram capturing the conceptual model of the system. Then, we aim to represent each use case specification as a model element of a so-called *use-case domain*. In order to define the use-case domain, we define meta-concepts w.r.t. the structural elements of the typical use-case-description template and the use case concepts as explained in Sect. 2. The meta-concepts allow us (1) to represent the basic and alternate flows of a use case in form of sequential, branched, or repeating steps, (2) to categorize use case steps and actions based on the interactive subjects including the system and actors, and (3) to represent constraints on the use case and its flows.

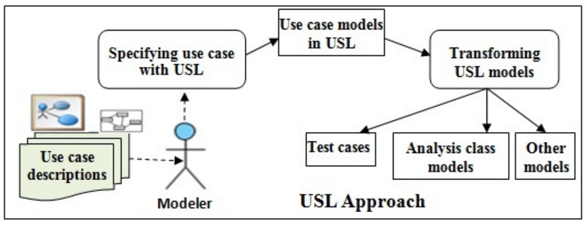


Figure 2: Overview of the USL Approach.

In order to represent textual descriptions of actions or constraints within a use case specification, we consider them as operations on an object-oriented model w.r.t. the input conceptual model of the system. In that way, we could employ pairs of pre- and postcondition as contracts on actions in order to obtain a more precise specification of the use case. The constraints are often expressed using constraint languages such as the OCL [24], JML [25], and natural language as mentioned in [17]. In this research, we employ the OCL to represent the constraints.

Specifically, our approach is realized as follows. We propose a domain specific language named USL in order to represent use cases within the use-case domain. Further, we define a formal semantics of USL so that we could transform USL models in to other artifacts such as test cases and analysis class models. To illustrate this point, a transformation from a USL model to a template-based use case description will be explained in details in SubSect. 5.3.

4 The USL language

This section first explains the abstract syntax and the graphical concrete syntax of USL. Here, we utilize the metamodeling approach as mentioned in [26] to define USL. Then, we focus on defining a precise semantics for USL by mapping a USL model to a Labelled Transition System (LTS) [20].

4.1 The USL abstract syntax

We define the USL metamodel w.r.t. the use-case domain based on (1) UML use case specification (Chapt. 18 of [2]), (2) the Use Case Descriptions (UCDs) [1], [23], [22], and (3) the UML activity specification (Chapt.s 15, 16 of [2]). We will refer to these as the `domain sources` (1), (2), and (3), respectively.

Figure 3 shows the metamodel of USL. For brevity, we divide the metamodel into four blocks: (a), (b), (c), and (d). Figure 3-a (*i.e.*, block (a)) presents the top-level concepts. Figure 3-b presents the `FlowStep` hierarchy. Figure 3-c presents the `ControlNode` hierarchy. Figure 3-d presents the `Action` hierarchy and how it is related to the `FlowStep` hierarchy. Figure 3-e presents the concept `Constraint` and how it is used to specify `Action`, `InitialNode`, `FinalNode`, and `FlowEdge`.

To conserve space, we will not repeat here the definitions of all of the USL concepts that are described in the three domain sources. We will instead focus on a key sub-set of the concepts – those that will be used later to define the transformation of USL models. Figure 4 presents the USL model of the *Lend Book* use case as shown in Table 1. We will use this example USL model in order to illustrate our definitions.

Action (domain sources (1, 3)) represents a action that is performed either by an actor or by the system. An `Action` is characterised by the following attributes: `actionName` and `parameters`. The parameters are represented by concept `Parameter` inherited the concept `Parameter` of UML (as presented in Sect. 19.9.13 of [2]). `Action` is specialized into two main types (as illustrated in Fig. 3-d): `ActorAction` and `SystemAction`. `ActorAction` is further specialized into `ActorRequest` and `ActorInput`. `SystemAction` is specialized into `SystemOperation`, `SystemOutput`, `SystemDisplay`, `SystemState`, `SystemInclude` and `SystemExtend` that were explained in Sect. 2.

FlowStep (domain source (2)) is a sequence of `Actions` that represents a step in a basic flow or an alternate flow of the use case. It is characterised by the following attributes: `number` (the order number of step), `description` (the content of the step) and `maxloop` (the maximum iteration of the step if existing). `FlowStep` is specialized into two types (as shown in Fig. 3-b): `ActorStep` and `SystemStep`, as mentioned in Sect. 2. We define three utility functions as shown in Table 2.

Example 4.1.1. The USL model shown in Fig. 4 consists of the `FlowSteps` s1, ..., s16. Among these, s1 is an `ActorStep` and s2 is a `SystemStep`. Step s3 contains the `ActorInput` a5. Step s1 contains the `ActorRequest` a1. Step s2 contains `SystemOperation` a3. Step s10 contains the `SystemOutput` a12. Step s4 contains the `SystemState` a6. Step s11 contains the `SystemRequest` a13. Step s14 contains the `SystemExtend` a16. The `Action` a5 has `Parameter` "bcid".

Control Node (domain source (3)) represents a control action that regulates the flows across other `USLNodes`. A `ControlNode`, as illustrated in Fig. 3-c, is specialized into `InitialNode`, `FinalNode`, `DecisionNode`, `ForkNode` and `JoinNode`. These respectively represent the starting and ending points of use case, the branching points of steps, and the starting and ending points of concurrent actions in steps. To ease notation, we define two overloading functions w.r.t. `ControlNode` and a function w.r.t. `DecisionNode` as shown in Table 2.

Example 4.1.2. The USL model as shown in Fig. 4 contains nine `ControlNodes` c0, ..., c8. In particular, c0 is an `InitialNode`, c7 and c8 are different `FinalNodes`, c1, ..., c3 and c6 are `DecisionNodes`, c4 is a `ForkNode`, and c5 is a `JoinNode`.

USLNode represents all the nodes `FlowStep` or `ControlNode` that make up a USL model.

FlowEdge (domain source (3)) is a binary directed edge between two `USLNodes`. If both steps are a part of a basic flow, we call the transition a `BasicFlowEdge`. On the other hand, if both steps are a part of an alternate flow, we call the transition an `AlternateFlowEdge`. As shown in Table 2, we define two utility functions `source` and `target`, two overloading functions `guardE` and a function `isCompleted` w.r.t. the concept `FlowEdge`.

Example 4.1.3. The USL model as shown in Fig. 4 contains b1, ..., b18 as `BasicFlowEdges` and al_1, ..., al_10 as `AlternateFlowEdges`.

Variable (domain source (3)) represents variables that

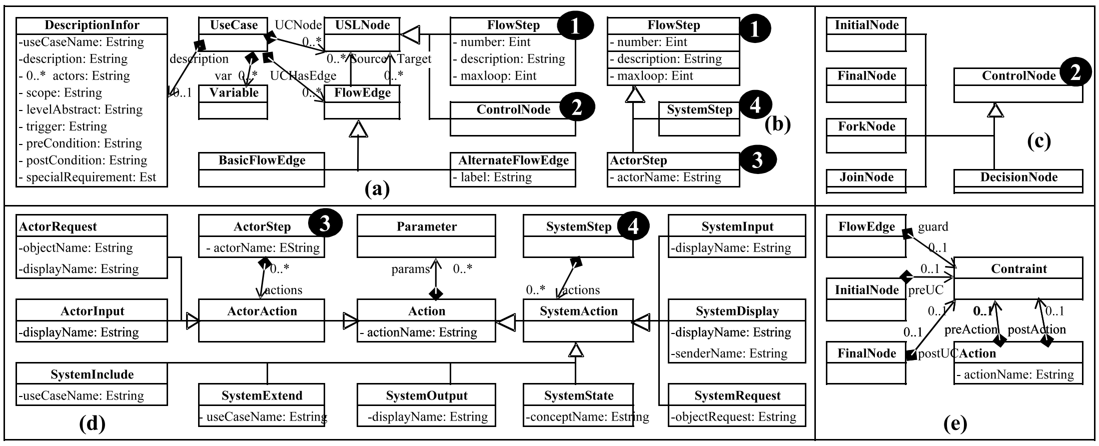


Figure 3: The USL metamodel.

Table 2: List of utility functions w.r.t. USL concepts

Utility function	Description
`firstAct`: `FlowStep` → `Action`	Returning the first `Actions` of a `FlowStep`.
`lastAct`: `FlowStep` → `Action`	Returning the last `Actions` of a `FlowStep`.
`actions`: `FlowStep` → `Actions`	Returning a set of Actions of a `FlowStep`.
`firstAct`: `ControlNode` → `ControlNode`	Returning the `ControlNode` itself.
`lastAct`: `ControlNode` → `ControlNode`	Returning the `ControlNode` itself.
`source`: `FlowEdge` → `USLNode`	Returning the source `USLNodes` of a `FlowEdge`.
`target`: `FlowEdge` → `USLNode`	Returning the target `USLNodes` of a `FlowEdge`.
`guardE`: `FlowEdge` → `Constraint`	Returning the guard condition.
`guardE`: `USLNode` → `USLNode` → `Constraint`	Taking the source and target `USLNodes` as input and returning the guard condition.
`isCompleted`: `FlowEdge` → `Boolean`	Determining whether or not **lastAct**(**source**(e)) has completed its execution.
`preA`: `Action` → `Constraint`	Returning the precondition of an `Action`.
`preA`: `ControlNode` → `Constraint`	If the `ControlNode` is not a `InitialNode`, returning true, else returning the `Constraint` of the `InitialNode`
`postA`: `Action` → `Constraint`	Returning the postcondition of an `Action`.
`postA`: `ControlNode` → `Constraint`	If the `ControlNode` is not a `FinalNode`, returning true, else returning the `Contraint` of the FinalNode.
`preC`: `USLModel` → `Constraint`	Returning the precondition of a `USLModel`.
`postC`: `USLModel` → `Constraint`	Returning the postcondition of a `USLModel`.
`postC`: `USLModel` → `FinalNode` → `Constraint`	Returning the postcondition of a particular `FinalNode` of a `USLModel`.

hold data values during the execution of a use case scenario. It is inherited the concept `Variable` of UML presented Sect. 15.7.25 of [2].

DescriptionInfor (domain source (2)) maintains the other textual description of use case.

Constraint (domain source (1,3)) represents constraints that are formed by use case variables: (1) the precondition of use case associated with `InitialNode`; (2) the postcondition of use case associated with `FinalNodes`; (3) guard conditions of a transition; and (4) the pre- and postcondition of an `Action`. This concept is inherited the concept `Constraint` in UML, shown in Sect. 7.6 of [2]. As depicted in Table 2, we define utility functions w.r.t. `Constraints` to get the pre- and postcondition of actions and use case.

Example 4.1.4. The USL model as shown in Fig. 4 contains g1, . . . , g6 as guard conditions and p1, . . . , p6 as postconditions of `Actions`.

We define a set of OCL wellformedness rules as restrictions on the USL metamodel. These rules are defined in the context of the `UseCase` concept and listed as follows.

Rule 1. A USL model has one `InitialNode`:

```
self.uslnode->selectByType(InitialNode)
->size()=1
```

Rule 2. A USL model has at least one `FinalNode`:

```
self.uslnode->selectByType(FinalNode)
->size() >= 1
```

Rule 3. A USL model has at least one `FlowStep`:

```
self.uslnode->selectByKind(FlowStep)
```

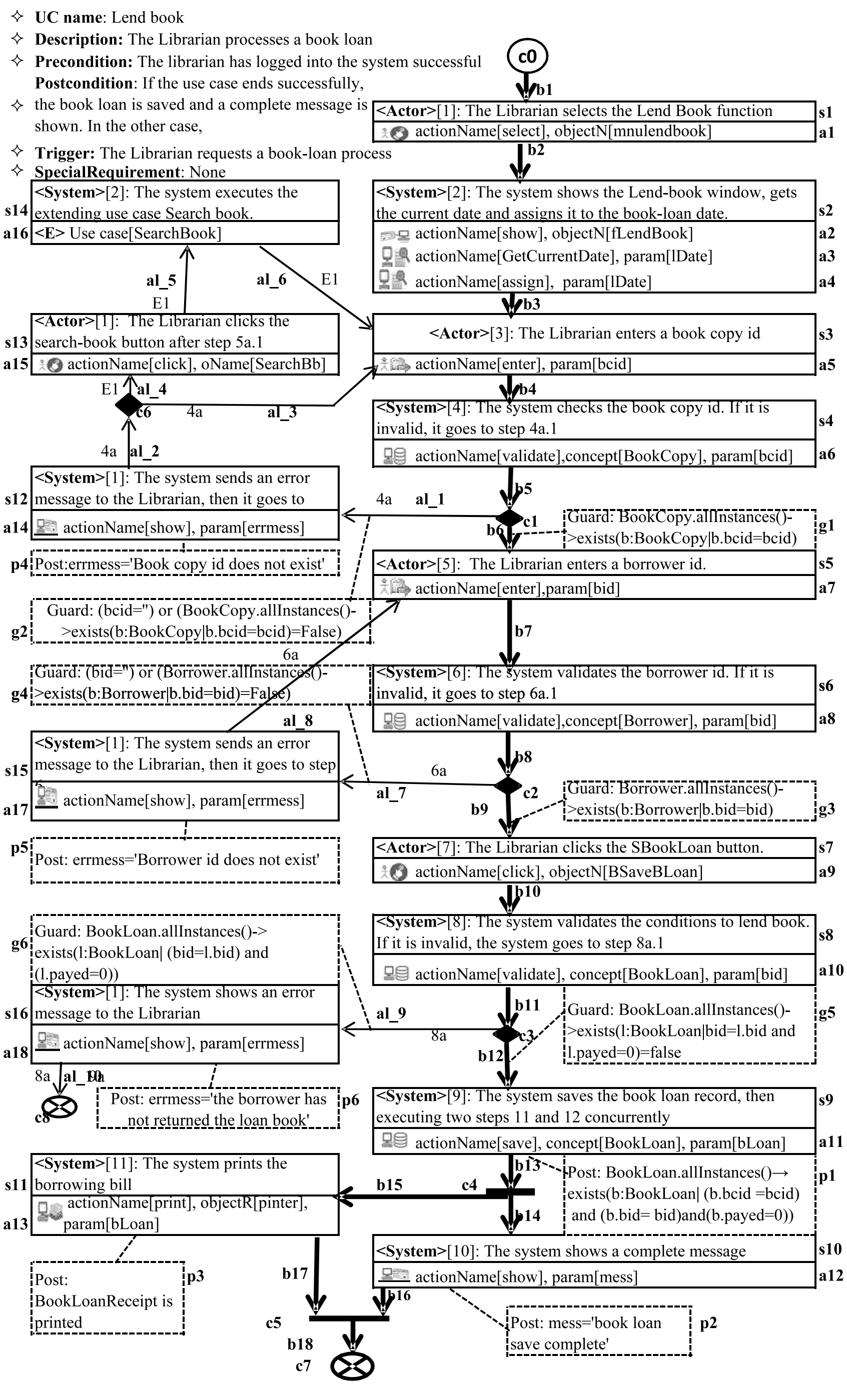

UC name: Lend book
Description: The Librarian processes a book loan
Precondition: The librarian has logged into the system successful
Postcondition: If the use case ends successfully, the book loan is saved and a complete message is shown. In the other case,
Trigger: The Librarian requests a book-loan process
SpecialRequirement: None
c0
b1
<Actor>[1]: The Librarian selects the Lend Book function s1
actionName[select], objectN[mnulendbook] a1
b2
<System>[2]: The system shows the Lend-book window, gets the current date and assigns it to the book-loan date. s2
actionName[show], objectN[fLendBook] a2
actionName[GetCurrentDate], param[lDate] a3
actionName[assign], param[lDate] a4
b3
<Actor>[3]: The Librarian enters a book copy id s3
actionName[enter], param[bcid] a5
b4
<System>[4]: The system checks the book copy id. If it is invalid, it goes to step 4a.1 s4
actionName[validate],concept[BookCopy], param[bcid] a6
b5
c1
b6
Guard: BookCopy.allInstances()->exists(b:BookCopy|b.bcid=bcid) g1
<Actor>[5]: The Librarian enters a borrower id. s5
actionName[enter],param[bid] a7
b7
<System>[6]: The system validates the borrower id. If it is invalid, it goes to step 6a.1 s6
actionName[validate],concept[Borrower], param[bid] a8
b8
c2
b9
Guard: Borrower.allInstances()->exists(b:Borrower|b.bid=bid) g3
<Actor>[7]: The Librarian clicks the SBookLoan button. s7
actionName[click], objectN[BSaveBLoan] a9
b10
<System>[8]: The system validates the conditions to lend book. If it is invalid, the system goes to step 8a.1 s8
actionName[validate], concept[BookLoan], param[bid] a10
b11
c3
b12
Guard: BookLoan.allInstances()->exists(l:BookLoan|bid=l.bid and l.payed=0)=false g5
<System>[9]: The system saves the book loan record, then executing two steps 11 and 12 concurrently s9
actionName[save], concept[BookLoan], param[bLoan] a11
b13
Post: BookLoan.allInstances()→exists(b:BookLoan| (b.bcid =bcid) and (b.bid= bid)and(b.payed=0)) p1
c4
b14
b15
<System>[10]: The system shows a complete message s10
actionName[show], param[mess] a12
b16
Post: mess='book loan save complete' p2
<System>[11]: The system prints the borrowing bill s11
actionName[print], objectR[pinter], param[bLoan] a13
Post: BookLoanReceipt is printed p3
b17
c5
b18
c7
s14 <System>[2]: The system executes the extending use case Search book.
a16 <E> Use case[SearchBook]
al_5
E1
al_6
E1
s13 <Actor>[1]: The Librarian clicks the search-book button after step 5a.1
a15 actionName[click], oName[SearchBb]
E1
al_4
c6
4a
al_3
4a
al_2
s12 <System>[1]: The system sends an error message to the Librarian, then it goes to
a14 actionName[show], param[errmess]
4a
al_1
p4 Post:errmess='Book copy id does not exist'
g2 Guard: (bcid='') or (BookCopy.allInstances()->exists(b:BookCopy|b.bcid=bcid)=False)
6a
g4 Guard: (bid='') or (Borrower.allInstances()->exists(b:Borrower|b.bid=bid)=False)
al_8
s15 <System>[1]: The system sends an error message to the Librarian, then it goes to step
a17 actionName[show], param[errmess]
6a
al_7
p5 Post: errmess='Borrower id does not exist'
g6 Guard: BookLoan.allInstances()->exists(l:BookLoan| (bid=l.bid) and (l.payed=0))
s16 <System>[1]: The system shows an error message to the Librarian
a18 actionName[show], param[errmess]
al_9
8a
8a
al_10
c8
Post: errmess='the borrower has not returned the loan book' p6

Figure 4: Representing the *Lend Book* use case as a USL model.

```
->size()>=1
```

Rule 4. An `InitialNode` has one out-going `BasicFlowEdge` and does not have any in-coming `FlowEdges`:

```
(self.flowedge->select(t:FlowEdge|t.source.
    oclIsTypeOf (InitialNode))->size()=1)and(
    self.flowedge ->select (b:FlowEdge|(b.
    source.oclIsTypeOf(InitialNode)) and (b.
    oclIsTypeOf(BasicFlowEdge)))->size()=1)
    and  (self.flowedge->select(t:FlowEdge| t
    .target.oclIsTypeOf(InitialNode))->size()
    =0)
```

Rule 5. A `FinalNode` has one in-coming `FlowEdge` and does not have any out-going `FlowEdge`:

```
self.uslnode->selectByType(FinalNode)->
    forAll (f:FinalNode|(self.flowedge->
    select(e:FlowEdge|e.target=f) ->size()
    =1) and (self.flowedge->select (e:
    FlowEdge|e.source=f)->size()=0))
```

Rule 6. A `DecisionNode` has one in-coming `FlowEdge` and at least two out-going `FlowEdges`:

```
(d:DecisionNode|(self.flowedge->select(e:
    FlowEdge|e.target=d)->size()=1)and (
    self.flowedge->select(e:FlowEdge|e.
    source=d)->size()>=2) )
```

Rule 7. A `ForkNode` has at least one in-coming `FlowEdge` and at least two out-going `FlowEdges`:

```
self.uslnode->selectByType(ForkNode)->forAll(
    f:ForkNode|(self.flowedge->select(e:
    FlowEdge|e.target=f)->size()>=1) and (
    self.flowedge->select(e:FlowEdge|e.source
    =f)->size()>=2) )
```

Rule 8. A `JoinNode` has at least two in-coming `FlowEdges` and one out-going `FlowEdge`.

```
self.uslnode->selectByType(JoinNode)->forAll
    (j:JoinNode|(self.flowedge->select(e:
    FlowEdge|e.target=j)->size()>=2)and(self.
    flowedge->select (e:FlowEdge|e.source=j)
    ->size()=1)).
```

Rule 9. A `SystemStep` or `ActorStep` has at least one in-coming `FlowEdge` and one out-going `FlowEdge`:

```
self.uslnode->selectByKind(FlowStep)->forAll(
    f:FlowStep|(self.flowedge->select(e:
    FlowEdge|e.target=f)->size()>=1)and(self.
    flowedge->select(e:FlowEdge|e.source=f)->
    size()=1))
```

Rule 10. A USL model is valid if the `FlowEdges` that connect the `USLNodes` of the model are valid, i.e., the type and label are correctly defined:

```
self.uslnode->forAll(n:USLNode|
if (n.oclIsTypeOf(InitialNode))then
 self.flowedge->select(b:FlowEdge|
 (b.source.oclIsTypeOf(USL::InitialNode)) and
     (b.oclIsTypeOf(USL::BasicFlowEdge)))->
     size()=1
else
 if (self.flowedge->selectByType(
     BasicFlowEdge)->select(b:BasicFlowEdge|b
     .target=n))->size()>=1 then
  if (n.oclIsTypeOf(DecisionNode))then
   self.flowedge->selectByType(BasicFlowEdge)
       ->select (b:BasicFlowEdge|b.source=n)
       ->size()=1
  else
   if (n.oclIsTypeOf(ForkNode)) then
    self.flowedge->select(f:FlowEdge|f.source
        =n)->forAll(b:FlowEdge|b.oclIsTypeOf(
        BasicFlowEdge))
   else
    if (n.oclIsTypeOf(JoinNode)) then
     (self.flowedge->select(f:FlowEdge|f.
         source=n)->forAll(b:FlowEdge|b.
         oclIsTypeOf(BasicFlowEdge)))and(self
         .flowedge->select(f:FlowEdge|f.
         target=n)->forAll (b:FlowEdge|b.
         oclIsTypeOf(BasicFlowEdge)))
    else
     if(n.oclIsKindOf(FlowStep)) then
      self.flowedge->select(f:FlowEdge|(f.
          source=n) and (f.oclIsTypeOf(
          BasicFlowEdge)))-> size() = 1
     else true
    endif
   endif
  endif
 endif
else ((self.flowedge->selectByType(
    BasicFlowEdge)->select(b:BasicFlowEdge|b.
    source=n))->size()=0) and if(n.
    oclIsTypeOf(FinalNode))then
  true else self.flowedge ->selectByType (
      AlternateFlowEdge) ->exists (f:
      AlternateFlowEdge|f.label=self.
      flowedge->selectByType(
      AlternateFlowEdge)->select(a:
      AlternateFlowEdge|a.target=n)->first()
      .label)endif)
endif
endif)
```

Rule 11. The `number` property of each `FlowStep` in a **Basic flow** is unique:

```
self.uslnode->selectByKind(FlowStep)->select(
    n:FlowStep|self.flowedge->selectByType(
    BasicFlowEdge)->exists(t:BasicFlowEdge|(t
    .source=n)or(t.target=n)))->forAll(n1:
    FlowStep, n2:FlowStep|n1.number=n2.number
     implies n1=n2).
```

Rule 12. The `number` property of each `FlowStep` in an **Alternate flow** is unique:

```
self.uslnode->selectByKind(FlowStep)->select
    (n:FlowStep|self.flowedge->selectByType(
    AlternateFlowEdge)->exists(t:
    AlternateFlowEdge|(t.source=n)or(t.target
    =n)))->forAll(n1:FlowStep,n2:FlowStep|(n1
    .number=n2.number)and(self.flowedge->
    selectByType(AlternateFlowEdge)->select(
    t1:AlternateFlowEdge|t1.target=n1)->first
    ().label=self.flowedge->selectByType(
    AlternateFlowEdge)->select(t2:
    AlternateFlowEdge|t2.target=n2)->first().
    label) implies n1=n2).
```

Example 4.1.5. Let us focus on the USL model as shown in Fig. 4:

- If we remove c0 from or add a new `InitialNode` to this model then it will violate Rule 1.
- If we remove both c7, c8 from the model then it will violate Rule 2.
- If the model only has `ControlNodes` then it will violate Rule 3.
- If `FlowEdge` b1 is not a `BasicFlowEdge` but an `AlternateFlowEdge` then the model will violate Rule 4.
- If we connect b17 to c8 and remove c7 then the model will violate Rule 5.
- If we add an `AlternateFlowEdge` to connect s4 to c6 then the model will violate Rules 6 and 9.
- If we remove b14, s11, b16, p3 then the model will violate Rules 7 and 8.

- If either `FlowEdge` b6 is not a `BasicFlowEdge` but an `AlternateFlowEdge` or the value of the `label` property of `AlternateFlowEdge` al_1 is not "4a" then the model will violate Rule 10.
- If the value of the `number` property of s2 is not 2 but 1 then the model will violate Rule 11.
- If the value of the `number` property of s14 is not 2 but 1 then the model will violate Rule 12.

4.2 The USL concrete syntax

In order to help the user to easily create USL models, we propose a concrete syntax for USL with the graphical notations as shown in Table 3. We have implemented this syntax in a visual editor for USL modelling. A detailed explanation of this tool will be presented in Sect. 6.

4.3 Formal semantics of USL

We formally define a `USL model` as follows. Here, we consider a USL model as a graph consisting of nodes and edges. A node represents either a step or a control action performed by the system. Further, we will take into account the fact that the underlying use case references the domain concepts captured in a UML class diagram.

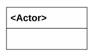
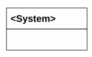
Definition 1. A `USL Model` of a use case is the tuple $D = \langle D_C, A, E, C\rangle$ such that:

- D_C is a class diagram to represent the underlying domain;
- A is the set of `USLNodes`;
- E is the set of `FlowEdges`;
- $C = G \cup C_{preUC} \cup C_{postUC} \cup C_{preA} \cup C_{postA}$ is the set of `Constraints`,

where:

- $A = A_{cNode} \cup A_f$;
- $A_{cNode} = N_I \cup N_F \cup N_d \cup N_j \cup N_f$, where
  $N_I = \{a \mid a \in A, \text{InitialNode}(a)\}$
  $N_F = \{a \mid a \in A, \text{FinalNode}(a)\}$,
  $N_d = \{a \mid a \in A, \text{DecisionNode}(a)\}$,
  $N_j = \{a \mid a \in A, \text{JoinNode}(a)\}$, $N_f = \{a \mid a \in A, \text{ForkNode}(a)\}$;
- $|N_I| = 1$; $|N_F| \geq 1$;
- $A_f = A_a \cup A_s$, where
  $A_f = \{a \mid a \in A, \text{FlowStep}(a)\}$, $A_a = \{a \mid a \in A, \text{ActorStep}(a)\}$,
  $A_s = \{a \mid a \in A, \text{SystemStep}(a)\}$;
- $|A_s| \geq 1$; $\forall s \in A_f.|\texttt{actions}(s)| \geq 1$;
- $E = E_b \cup E_a$ and $E_b \cap E_a = \emptyset$, where
  $E_b = \{e \mid e \in E, \text{BasicFlowEdge}(e)\}$,
  $E_a = \{e \mid e \in E, \text{AlternateFlowEdge}(e)\}$.

Table 3: The graphical notations of USL

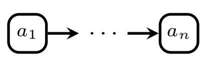
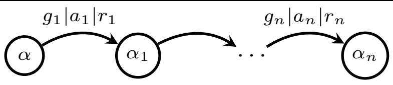
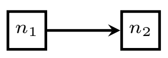
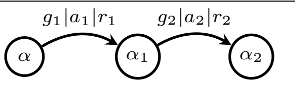
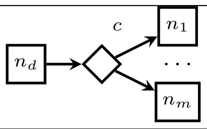
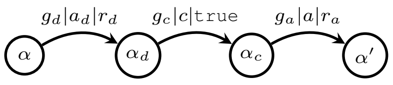
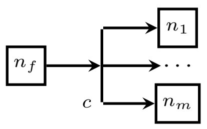
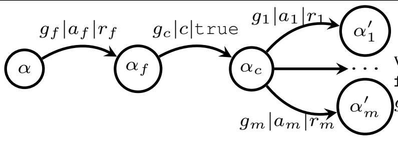
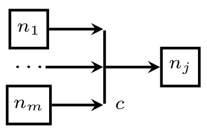
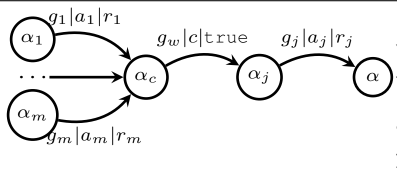
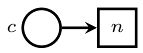
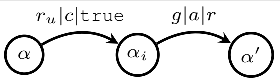
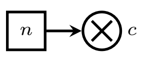
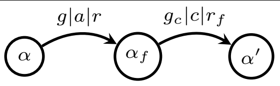
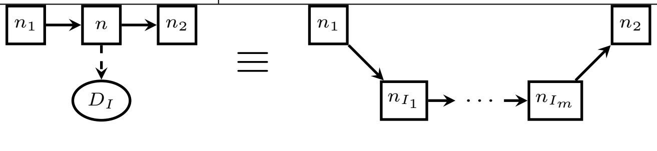
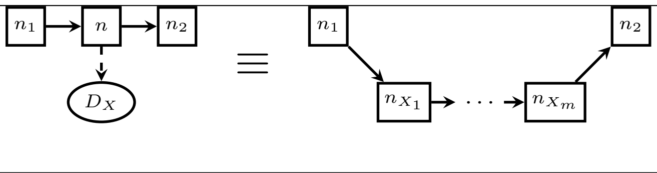
Concepts	Presentation	Notation
DescriptionInfor	A borderless text box that properties are listed in the text box	⋮
InitialNode	An unfilled circle	○
FinalNode	A circle with a crosshairs symbol	⊗
DecisionNode	A filled diamond with one in-coming arrowed line and at least two out-going arrowed lines	
ForkNode	A solid line segment with one in-coming arrowed line and at least two out-going arrowed lines	
JoinNode	A solid line segment with at least two in-coming arrowed lines and one out-going arrowed line	
BasicEdge	A thick arrowed line	➔
AlternateEdge	A labelled, thin arrowed line (the label is the name of the flow)	$\xrightarrow{\text{label}}$
ActorStep	A labeled, 2-part rectangle. The first part contains the label **<Actor>** and two properties `numberStep` and `description` of the `ActorStep`. The second part contains the `ActorActions` of the `ActorStep`	
SystemStep	A labelled, 2-part rectangle. The first part contains the label **<System>** and two properties `numberStep` and `description` of the `SystemStep`. The second part contains the `SystemActions` of the `SystemStep`	
Action	Information of a `Action` are presented by textual form in the second part of `FlowSteps`	┄

Example 4.1.5. The USL model as shown in Fig. 4 contains the following elements: $N_I = \{c0\}$; $N_F = \{c7, c8\}$; $A_{cNode} = \{c0, \ldots, c8\}$; $A_a = \{s1, s3, s5, s7, s13\}$; $A_s = \{s2, s4, s6, s8, \ldots, s12, s14, s15, s16\}$; $E_b = \{b1, \ldots, b17\}$; $E_a = \{al\_1, \ldots, al\_10\}$; $G = \{g1, \ldots, g6\}$; $C_{preUC} = \emptyset$; $C_{postUC} = \emptyset$; $C_{preA} = \emptyset$; and $C_{postA} = \{p1, \ldots, p6\}$. D_C corresponds to the conceptual model shown in the part (b) of Fig. 1. There are sixteen constraints for guard conditions and pre- and postconditions, e.g., the postcondition $p1$ of `Action` $a11$ is expressed by the following OCL contraint: `BookLoan.allInstances()->exists(b:BookLoan| (b.bcid = bcid) and (b.bid = bid) and (b.payed=0))`.

We use LTS [20] to formally define the operational semantics of USL. Conceptually, the execution of a USL model is modelled by an LTS, whose transitions are caused by the execution of use case actions, and whose states are defined by variable assignments during the execution. We define the LTS of a USL model recursively from the basic USL concepts. The semantics of these concepts are defined as summarized in Table 4. Definition 2 formalizes the notion of the LTS of the USL model.

Definition 2. Given a USL model $D = \langle D_C, A, E, C\rangle$, an `LTS` that results from the execution of D is the tuple $\langle \Sigma(\mathcal{V}), \mathbb{P}(\mathcal{G} \times \mathcal{A} \times \mathcal{P}), \mathcal{T}, \alpha_{init}, \mathcal{F}\rangle$ such that:

Table 4: LTS-based semantics of the basic USL concepts

USL concepts	Notation	LTS-based semantics
Step		 $g_1\|a_1\|r_1$ $g_n\|a_n\|r_n$ α α_1 α_n where $g_i = \texttt{preA}(a_i)$, $r_i = \texttt{postA}(a_i)$ ($\forall i = 2, \ldots, n$).
Flow edge		 $g_1\|a_1\|r_1$ $g_2\|a_2\|r_2$ α α_1 α_2 where $a_2 = \texttt{firstAct}(n_2)$, $g_2 = \texttt{guardE}(n1, n2) \wedge \texttt{preA}(a_2)$ $r_2 = \texttt{postA}(a_2)$.
Decision node		 $g_d\|a_d\|r_d$ $g_c\|c\|\texttt{true}$ $g_a\|a\|r_a$ α α_d α_c α' where $a_d = \texttt{lastAct}(n_d)$; $g_c = \texttt{guardE}(n_d, c)$, $a = \texttt{firstAct}(n)$, $g_a = \texttt{guardE}(c, n) \wedge \texttt{preA}(a)$ s.t $n \in \{n_1, \ldots, n_m\}$.
Fork node		 $g_f\|a_f\|r_f$ $g_c\|c\|\texttt{true}$ $g_1\|a_1\|r_1$ $g_m\|a_m\|r_m$ α α_f α_c α'_1 α'_m where $a_f = \texttt{lastAct}(n_f)$; $g_c = \texttt{guardE}(n_f, c)$, $a_i = \texttt{firstAct}(n_i)$, $g_i = \texttt{preA}(a_i)$, $r_i = \texttt{postA}(a_i)$ ($\forall i = 1, \ldots, m$).
Join node		 $g_1\|a_1\|r_1$ $g_w\|c\|\texttt{true}$ $g_j\|a_j\|r_j$ $g_m\|a_m\|r_m$ α_1 α_m α_c α_j α where $a_i = \texttt{lastAct}(n_i)$, ($\forall i = 1, \ldots, m$); $g_w = \left(\bigwedge_{(e \in D.E, \texttt{target}(e)=c)} \texttt{isCompleted}(e) \wedge \texttt{guardE}(e)\right)$; $a_j = \texttt{firstAct}(n_j)$, $g_j = \texttt{guardE}(c, n_j) \wedge \texttt{preA}(a_j)$, $r_j = \texttt{postA}(a_j)$.
Initial node		 $r_u\|c\|\texttt{true}$ $g\|a\|r$ α α_i α' where $\alpha = \alpha_{init}$, $r_u = \texttt{preC}(D)$; $a = \texttt{firstAct}(n)$, $g = \texttt{guardE}(c, n) \wedge \texttt{preA}(a)$, $r = \texttt{postA}(a)$.
Flow final node		 where $\alpha' \in \mathcal{F}$, $r_f = \texttt{postC}(D, c)$, $g_c = \texttt{guardE}(\texttt{n}, \texttt{c})$; $a = \texttt{lastAct}(n)$,
USL model with include action		where $n \in A_s$, $\|\texttt{actions}(n)\| = 1$, $\texttt{SystemInclude}(a)$ ($a \in \texttt{actions}(n)$), $t = (\alpha, (g_a\|a\|r_a), \alpha')$; $n_{I_1}, \ldots, n_{I_m} \in D_I.A$, $g_a = \texttt{guardE}(n1, n) \wedge \texttt{preA}(a) \wedge \texttt{preC}(D_I)$, $r_a = \texttt{postC}(D_I)$
USL model with extend action		where $n \in A_s$, $\|\texttt{actions}(n)\| = 1$, $\texttt{SystemExtend}(a)$ ($a \in \texttt{actions}(n)$), $t = (\alpha, (g_a\|a\|r_a), \alpha')$; $n_{X_1}, \ldots, n_{X_m} \in D_X.A$, $g_a = \texttt{preA}(a) \wedge \texttt{preC}(D_X)$, $g_a = \texttt{guardE}(n1, n) \wedge \texttt{preA}(a) \wedge \texttt{preC}(D_X)$, $r_a = \texttt{postC}(D_X)$
Legend	a a action in a step; s a step; D_u use case; α a state	

– $\mathcal{V}$ is a finite set of variables whose types include the basic types and the classes of the D_C;

– $\Sigma(\mathcal{V})$ is the set of states (α), each of which is a set of value assignments to a subset of variables in $\mathcal{V}$;

– $\mathcal{P} \subseteq C_{postA} \cup C_{postUC}$ is the set of constraints as the postconditions of D;

– $\mathcal{A} = A_{cNode} \cup A_{act}$ is the set of actions;

– $\mathcal{G} \subseteq G \cup C_{preUC} \cup C_{preA}$ is the set of guard conditions of the transitions;

– $\mathcal{T} \subseteq \Sigma(\mathcal{V}) \times \mathbb{P}(\mathcal{G} \times \mathcal{A} \times \mathcal{P}) \times \Sigma(\mathcal{V})$ is the transition relation defined as follows: A transition $t = (\alpha, (g, a, d), \alpha') \in \mathcal{T}$, written as $\alpha \xrightarrow{g\|a\|r} \alpha'$, where $a \in \mathcal{A}$ is the action that causes t, $g = \texttt{defGuard}(a) \in \mathcal{G}$ is the guard condition to execute a, $r \in \mathcal{P}$ is the postcondition of a, and $\alpha, \alpha' \in \Sigma(\mathcal{V})$ are the pre- and post-states of t (resp.) such that α' satisfies r;

– $\alpha_{init} \in \Sigma(\mathcal{V})$ is the initial state;

– $\mathcal{F} \subset \Sigma(\mathcal{V})$ is the set of final states,

where:

– $A_{act} = \bigcup_{s \in A_f} \texttt{actions}(s)$;

– `defGuard` is defined as follows (summarized from Table 4).

$$
= \begin{cases}
\texttt{preC}(D), \texttt{ifInitialNode}(a) \\
\texttt{guardE}(e)(e \in D.E, \texttt{target}(e) = a), \texttt{ifDecisionNode}(a) \\
\vee \ \texttt{ForkNode}(a) \vee \ \texttt{FinalNode}(a) \\
\bigwedge_{(e \in D.E, \texttt{target}(e)=a)} \texttt{isCompleted}(e) \wedge \texttt{guardE}(e), \\
\text{if } \texttt{JoinNode}(a) \\
\texttt{preC}(D_I) \wedge \texttt{preA}(a) \wedge \texttt{guardE}(e)(e \in D.E, \texttt{target}(e) = a), \\
\text{if } \texttt{SystemInclude}(a) \\
\texttt{preC}(D_X) \wedge \texttt{preA}(a) \wedge \texttt{guardE}(e)(e \in D.E, \texttt{target}(e) = a), \\
\text{if } \texttt{SystemExtend}(a) \\
\texttt{preA}(a) \wedge \texttt{guardE}(e)(s \in A_f, \texttt{target}(e) = s), \\
\texttt{if}((a \in A_{act}) \wedge \ (a = \texttt{firstAct}(s)) \\
\texttt{preA}(a)(s \in A_f, a \in actions(s)), \texttt{if}otherwise
\end{cases}
$$

Brrower

bid	name	bday
123	Joney	6/2/90
124	Mary	2/3/91

BookCopy

bcid	bkid	state
001	N01	'normal'
002	N01	'normal'

BookLoan

lbid	bcid	bid	lid	ldate	payed
1	002	123	110	2/3/17	0

Librarian

lid	name	bday	pword
110	Davi	5/8/80	12334
111	Bob	9/3/91	12344

Figure 5: A snapshot of the *Lend Book* use case.

Example 4.3.1. We assume that the snapshot shown in Fig. 5 is captured when the USL model as shown in Fig. 4 is executed at Step `a8`. We have the following value assignments: $(bcid, \text{“001”}) \equiv bcid = \text{“001”}$, $(lid, \text{“110”}) \equiv lId = \text{“100”}$, $(ldate, \text{“25/8/17”}) \equiv ldate = \text{“25/8/17”}$, $(bid, \text{“1234”}) \equiv bcid = \text{“1234”}$. The objects of the snapshot are as follow: `BookCopy`:“001”, `BookCopy`:“002”, `Borrower`:“123”, `Borrower`:“124”, `Librarian`:“100”, `Librarian`:“111”, `BookLoan`:“1”. Then, we have $\alpha_{a8} = \{(bcid, \text{“001”}), (ldate, \text{“001”}), (lid, \text{“110”}), (bid, \text{“124”}), (bLoan, (\text{“2”}, \text{“001”}, \text{“124”}, \text{“110”}, \text{“25/8/17”}, 0))$, `BookCopy`:“001”, `BookCopy`:“002”, `Borrower`:“123”, `Borrower`:“124”, `Librarian`:“100”, `Librarian`:“111”, `BookLoan`:“1”$\}$.

Certain use case actions are concurrent actions, whose executions cause `concurrent transitions` between states. The next two definitions define precisely what this means.

Definition 3. Given a current state α of an LTS L of a USL model D and a transition $t = \alpha \xrightarrow{g|a|r} \alpha' \in L.\mathcal{T}$, we define the following terms:

- $\texttt{preT}(t) = \alpha$, $\texttt{postT}(t) = \alpha'$, $\texttt{guard}(t) = g$, $\texttt{postC}(t) = r$, and $\texttt{act}(t) = a$.
- $\texttt{eval}(g)$ is the evaluation of Constraint g.
- $\texttt{reachable}(\alpha) = \{t \mid \texttt{preT}(t) = \alpha\}$ is the set of transitions that start from α.
- $\texttt{firable}(\alpha) = \{t \in \texttt{reachable}(\alpha), \texttt{eval}(\texttt{guard}(t)) = \texttt{true}\}$ is the set of transitions that can be fired from α.

Example 4.3.2. When the USL model as shown in Fig. 4 executes at action $a11$, we have $\alpha_{a11} = \{(bcid, \text{“001”}), (lDate, \text{“001”}), (lid, \text{“110”}), (bid, \text{“124”}), (bLoan, (\text{“2”}, \text{“001”}, \text{“124”}, \text{“110”}, \text{“25/8/17”}, 0))$, `BookCopy`:“001”, `BookCopy`:“002”,`Borrower`:“123”, `Borrower`:“124”, `Librarian`:“100”, `Librarian`:“111”, `BookLoan`:“1”, `BookLoan`:“2”$\}$.

Transition $t_{a11,c4} = \alpha_{a11} \xrightarrow{true|c4|true} \alpha_{c4}$. $\texttt{reachable}(\alpha_{a11}) = \{t_{a11,c4}\}$ and $\texttt{firable}(\alpha_{a11}) = \{t_{a11,c4}\}$.

Definition 4. Given a current state α of an LTS L of a USL model D, a `concurrent transition` $\tau \in L.\mathcal{T}$ is a set of transitions $t_1, t_2, \ldots, t_n \in \texttt{firable}(\alpha)$.

Example 4.3.3. When the USL model as shown in Fig. 4 executes at Step $c4$, we have two transitions $t_{c4,a12} = \alpha_{c4} \xrightarrow{true|a12|p2} \alpha_{a12}$ and $t_{c4,a13} = \alpha_{c4} \xrightarrow{true|a13|p3} \alpha_{a13}$, $\texttt{reachable}(\alpha_{c4}) = \{t_{c4,a12}, t_{c4,a13}\}$ and $\texttt{firable}(\alpha_{c4}) = \{t_{c4,a12}, t_{c4,a13}\}$. Hence, $\{t_{c4,12}, t_{c4,13}\}$ is a concurrent transition and α_{a12}, α_{a13} satisfy p2, p3, respectively.

Within our approach the LTS of a USL model may contain both concurrent and non-concurrent transitions. We next define the semantics of a use case scenario.

Definition 5. Given a `use case scenario` of a USL model D that consists of the following sequence of actions $(a_0, \ldots, a_{n-1})$. The execution of this scenario is realized as a path in the LTS L of D: $p = \alpha_0 \xrightarrow{t_0} \alpha_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} \alpha_n$, where $t_i = \alpha_i \xrightarrow{g_i|a_i|r_i} \alpha_{i+1}$ ($\forall i = 0, \ldots, n-1$), $\alpha_0 = L.\alpha_{init}$, $\alpha_n \in L.\mathcal{F}$, and $t_i \in L.\mathcal{T}$.

Example 4.3.4. When the USL model as shown in Fig.4 executes at Step α_{a11} as mentioned above and `eval`(g1), `eval`(g3), and `eval`(g5) are true, then the use case scenario is as follows:

$$p = \alpha_{init} \xrightarrow{true|a1|true} \alpha_{a1} \xrightarrow{true|a2|true} \alpha_{a2} \xrightarrow{true|a3|true} \alpha_{a3} \xrightarrow{true|a4|true} \alpha_{a4} \xrightarrow{true|a5|true} \alpha_{a5} \xrightarrow{true|a6|true} \alpha_{a6} \xrightarrow{true|c1|true} \alpha_{c1} \xrightarrow{g1|a7|true} \alpha_{a7} \xrightarrow{true|a8|true} \alpha_{a8} \xrightarrow{true|c2|true} \alpha_{c2} \xrightarrow{g3|a9|true} \alpha_{a9} \xrightarrow{true|a10|true} \alpha_{a10} \xrightarrow{g5|a11|true} \alpha_{a11} \xrightarrow{true|c4|true} \alpha_{c4} \xrightarrow{\{true|a12|p2, true|a13|p3\}} \alpha_{a12-a13} \xrightarrow{true|c5|true} \alpha_{c5} \xrightarrow{true|c7|true} \alpha_{c7} \ (\alpha_{c7} \in \mathcal{F}).$$

5 Transforming USL models to other software artifacts

This section explains how USL models can be transformed to software artifacts including test cases, structural and behavioral models, and textual template-based use case descriptions (TUCDs). We particularly focus on the last transformation (to obtain TUCDs) and show how the transformation could be realized.

5.1 Generating test cases

A test scenario is used to create a set of test cases [27]. A test case results from combining a test scenario with some test data. According to the use case-driven testing approach [27], a use case scenario identifies one test scenario (a use case description consists of one or more use case scenarios). The constraints of a use case scenario help identifying the test data of the corresponding test scenario.

The model-based testing (MBT) method [28] presents a specific technique for automatically generating test cases from a use case model. Specifically, the control flows of a use case model are used to generate the use case scenarios. For example, Linzhang [29] first presents a technique to represent the control flows using UML activity diagram. He then proposes an algorithm to traverse all the possible basic paths of the activity diagram to generate the test scenarios.

Two other works [30, 31] focus on the problem of automatically generating test data from the test scenario constraints, written in OCL. They develop OCL constraint solvers for this task.

Since our USL captures the necessary information elements of the use case description, we argue that USL models can also be used as an input to generate test cases. More specifically, USL has meta-concepts for representing the different control nodes of the UML activity diagram. Further, the `Constraint` meta-concept of USL captures the different types of constraints that are needed to generate test data.

5.2 Generating structural and behavioural models

In the requirement analysis activity, the behaviours described in a use case description are analysed in order to create other structural and behavioural models. The target models are often represented using UML diagrams, including activity diagram, class diagram, collaboration diagram, and sequence diagram.

D. Savić *et al.* [16] and M. Smialek *et al.* [17] propose a specific method for the above. In particular, they first use different types of actions to precisely model the use case behaviours. They then present a model transformation technique that automatically transforms the behaviours and other relevant model elements into a class diagram. Examples of these elements that are discussed in [28] include sender and receiver objects, messages, and parameters.

Our USL specification was inspired by this work. Specifically, we use `Action` meta-concept to represent use case behaviours and the relevant model elements discussed above. Regarding to behavioural modelling, a USL model can be used as input to generate activity and sequence diagrams. The reason is because USL represents all the control nodes of UML activity diagram. For example, a specific technique for generating sequence diagram is presented in [12].

5.3 Generating TUCDs

According to [32, 33, 34], textual template-based use case descriptions (TUCDs) [1, 22, 8] enable the customer to positively participate in requirement analysis, to identify and resolve conflicts in the requirement drafts, and to ensure that it is consistent with their intention. Table 1 shown earlier is an example of such a template.

In order to automatically generate a TUCD from a USL model, we develop a transformation USL2TUCD using the model-to-text transformation language Acceleo [18]. The transformation USL2TUCD is shown in Listing 1. We illustrate this transformation using the USL model of the use case named `Withdrawal` (shown in Fig. 9). The output TUCD is a text file named `Withdrawal.txt` that is shown in Fig. 10.

Briefly, the USL2TUCD transformation uses five queries to extract information from the input USL model (`uc`). The first query is `getBasicFlow(uc)` at line 19. It is used is to find all the `BasicFlowSteps` in `uc`. The second query is `getDecisionNode (uc)` at line 25. It is used to get all the `DecisionNodes` in `uc`. The third query is `getPreAlternateFlowLabel(uc,d)` at line 27. It is used to get the label of the in-coming `AlternateFlowEdge` of some `DecisionNode d` in `uc`. It returns empty if no such `AlternateFlowEdges` exist. The fourth query is `getAFEdges(uc,d)` at lines 29 and 37. This query is used to get the out-going `AlternateFlowEdges` from a `DecisionNode d` in `uc`. The fifth query is `getAlternateFlow(uc, l)` at lines 30 and 39. This query is used to find the `FlowSteps` in the `AlternateFlow` in `uc` that is labeled `l`.

The definitions of all five queries are written in another transformation named `libraryUCD`. The transformation is shown in Listing 2.

Listing 1: The USL2TUCD transformation

```
1  [module GenUCDescription('http://eclipse
       .USLModel/USL')]
2  [import org::eclipse::acceleo::module::
       sample::service::libraryUCD]
3  [template public generateElement(uc:
       UseCase)]
4  [comment @main/]
5  [file (uc.descriptioninfor->r at(0).
       useCaseName.concat('.txt'), false, '
       UTF-8')]
6  [let d:DescriptionInfor =uc.
       descriptioninfor-> at(0)]
7  ----------------------------------
8  UC name: [d.useCaseName/]
9  Description: [d.description /]
10 Actor: [for(a:String|d.actor)] a, [/for]

11 Level abstract: [d.levelAbstract /]
12 Precondition: [d.preCondition /]
13 Postcondition: [d.postCondition /]
14 SpecialRequirement: [d.
       specialRequirement/]
15 [/let]
16 ----------------------------------
17 BasicFlow
18 ----------------------------------
19 [let Bsteps:OrderedSet(FlowStep)=
       getBasicFlow(uc)]
20 [for(s:FlowStep|Bsteps)]
21   [s.number/].  [s.description/]
22 [/for] [/let]
```

```
----------------------------------
AlternateFlow
[let dList:OrderedSet(DecisionNode)=
    getDecisionNode(uc)]
[for(d:DecisionNode|dList)]
 [let preALabel:String=
     getPreAlternatFlowLabel(uc, d)]
 [if(preALabel='')]
  [for(af:AlternateFlowEdge|getAFEdges(
      uc, d))]
   [let Asteps:OrderedSet(FlowStep)=
       getAlternateFlow(uc, af.label)]
   [af.label/]. [af.description]
   [for(s:FlowStep|Asteps)]
     [s.number/].  [s.description/]
   [/for] [/let]
  [/for]
 [else]
  [for(af:AlternateFlowEdge|getAFEdges(
      uc, d))]
   [if(af.label <>preALabel)]
    [let Asteps:OrderedSet(FlowStep)=
        getAlternateFlow(uc, af.label)]
    [af.label/]. [af.description/]
    [for(s:FlowStep|Asteps)]
      [s.number/].  [s.description/]
    [/for] [/let]
   [/if]
  [/for]
 [/if] [/let]
[/for] [/let]
----------------------------------
[/file]
[/template]
```

Listing 2: The `libraryUCD` transformation

```
[comment encoding = UTF-8 /]
[module libraryUCD('http://eclipse.
    USLModel/USL')]
[query public getBasicFlow(uc:UseCase):
    OrderedSet(FlowStep)=uc.uslnode->
    select (n:USLNode|uc.flowedge->
    selectByType (BasicFlowEdge)->exists(
    b:BasicFlowEdge|(n=b.source) or (n=b.
    target)))->selectByKind(FlowStep)/]

[query public getAlternateFlow(uc:
    UseCase,l:String): OrderedSet(
    FlowStep)=uc.uslnode->select(n:
    USLNode|uc.flowedge->selectByType(
    AlternateFlowEdge)->select(a:
    AlternateFlowEdge|a.label=l)->exists(
    f:AlternateFlowEdge|(f.target=n)or(f.
    source=n)))->selectByKind(FlowStep)/]

[query public getAFEdges(uc:UseCase, d:
    DecisionNode): OrderedSet(
    AlternateFlowEdge) =uc.flowedge->
    select(f:FlowEdge|(f.source=d)and(f.
    oclIsTypeOf(AlternateFlowEdge)))->
    selectByType (AlternateFlowEdge) /]

[query public getDecisionNode(uc:UseCase
    ): OrderedSet(DecisionNode)=uc.
    uslnode->selectByType(DecisionNode)
    /]

[query public getPreAlternatFlowLabel (
    uc:UseCase,d:DecisionNode):String=
 if uc.flowedge->selectByType(
     AlternateFlowEdge)->select(f:
     AlternateFlowEdge| f.target=d)->size
     ()>0 then
  uc.flowedge->selectByType(
      AlternateFlowEdge)->select(f:
      AlternateFlowEdge| f.target=d)->at
      (0).label
 else
   ''
 endif /]
```

6 Tool support and evaluation

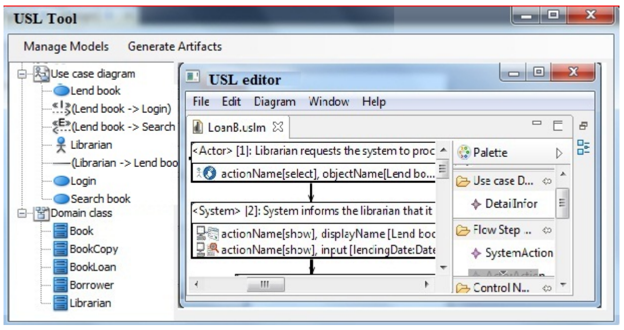
In this section, we first describe a USL tool that we have developed for visually creating USL models. After that, we explain two case studies for USL. We conclude this section with an evaluation of USL.

6.1 Tool support

We developed a support tool for our approach as illustrated in Fig. 6. This USL tool provides three main functions. The first function (displayed on the left of the figure) is called the "Loading function". It is responsible for loading the use cases and domain concepts of a system from a UML use case diagram and a class diagram. The second function (shown on the right of the figure) is called "USL Editor". It is used to create the USL models for the loaded use cases. This editor has a user-friendly GUI. The third function is called "Generating Artifacts". It automatically generates other software artifacts.

In our tool, the "Loading function" was developed using a Java project. The "USL Editor" was implemented using an EMF project and an GMF project within the Eclipse tool [26]. Specifically, the EMF project is to build the abstract syntax of USL and the GMF project is to build the concrete syntax and to implement the OCL constraint rules on the metamodel. The "Generating artifacts" function was written using model transformation languages, such as M2T and M2M [18]. To illustrate, Fig. 8 shows a USL model for the use case *Session*, that is created by the "USL Editor". Figure 10 shows a TUCD text file that is automatically generated by a transformation that was specified earlier in Listings 1 and 2. This transformation was written in the Acceleo M2T language.

Note that when working with a generation relationship between use cases, the modeler needs to create USL models only for the specific use cases rather than for the abstract ones.



<Actor> [1]: Librarian requests the system to proc
actionName[select], objectName[Lend bo...
<System> [2]: System informs the librarian that it
actionName[show], displayName [Lend boc
actionName[show], input [lendingDate:Date

Figure 6: The USL tool.

6.2 Case study

In order to demonstrate the applicability of our method, we chose another system case study named *ATM*, which is described in Bjork [35]. The system includes three actors, seven specific use cases, one abstract use case, and two use case relationships. Figure 7 shows the use cases of the *ATM* system. Figure 8 and Fig. 9 show two USL models corresponding to these two use cases: *Session* and *Withdrawal*. Figure 10 and Fig. 11 show two TUCD text files that are generated from these two USL models, by applying the function "Generating Artifact". These files are the use case descriptions of the two corresponding use cases.

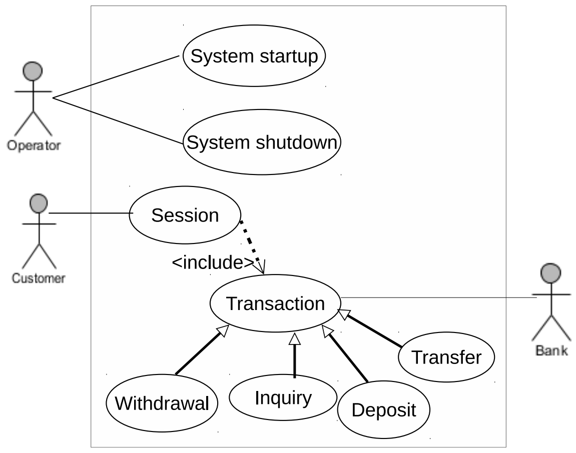


Figure 7: The use case diagram of the `ATM` system.

6.3 Language evaluation

This section presents our evaluation of USL's expressiveness, compared to five languages: RUCM [3], UC-B [10], MBD-L[1] [4], SiLabReq [16] and RSL [17]. We use the following four sub-criteria of expressiveness:

C1. Template-based representation of use case descriptions
C2. Control flow-based representation of use case behaviour
C3. Action specification
C4. Use case constraint representation

[1] 'MBD' stands for the author's names, 'L' for language.

Table 5 lists the evaluation results for the above criteria. In the table, we use three letters 'F', 'I', 'N' to denote the specification method that is used for each language: 'F' denotes *formal specification method*, 'I' denotes *informal specification method* and 'N' denotes that the specification method is not discussed.

Table 5: Expressiveness comparison between use case specification languages

Use case information	RUCM [3]	UC-B [10]	MBD-L [4]	SelabReq [16]	RSL [17]	USL
(c1) Overview elements	I	N	F	N	N	F
(c1) Flows of use case	I	I	F	N	N	F
(c1) Use case scenarios	N	N	N	F	F	N
(c2) Control flows	I	N	F	N	F	F
(c2) Concurrent actions	N	N	N	N	N	F
(c3) Action types	I	N	F	F	F	F
(c4) Use case scenario's pre- and postcondition	I	F	F	N	I	F
(c4) Guard conditions	I	F	F	N	I	F
(c4) Action's pre- and postcondition	N	F	N	N	N	F

We will discuss in detail the results shown in the table in the first five subsections that follow. In the last subsection, we discuss the possibility of applying USL in practice.

6.3.1 Template-based representation of use case descriptions

As discussed in Sect. 4, USL enables us to capture all the information elements of the use case description template shown in Table 1. In particular, the elements of overview information are described by the properties of the `DescriptionInfo` object in the model. The steps in a basic flow are represented by `FlowSteps` (including `ActorStep` and `SystemStep`) and are connected by `BasicFlowEdges` and `ControlNodes`. Similarly, steps in an alternate flow are represented by `FlowSteps` (including `ActorStep` and `SystemStep`) and are connected by `ControlNodes` and `AlternateFlowEdges`. USL represents this template precisely using the corresponding USL meta-concepts. Briefly, we draw the following conclusions from Table 5:

- USL is more expressive and more precise than three other languages, namely UC-B, SilabReq, and RSL.
- USL is more precise than RUCM.
- USL is as expressive and precise as MBD-L.

Specifically, our USL is more expressive than UC-B, SilabReq, and RSL because of the following reasons. First,

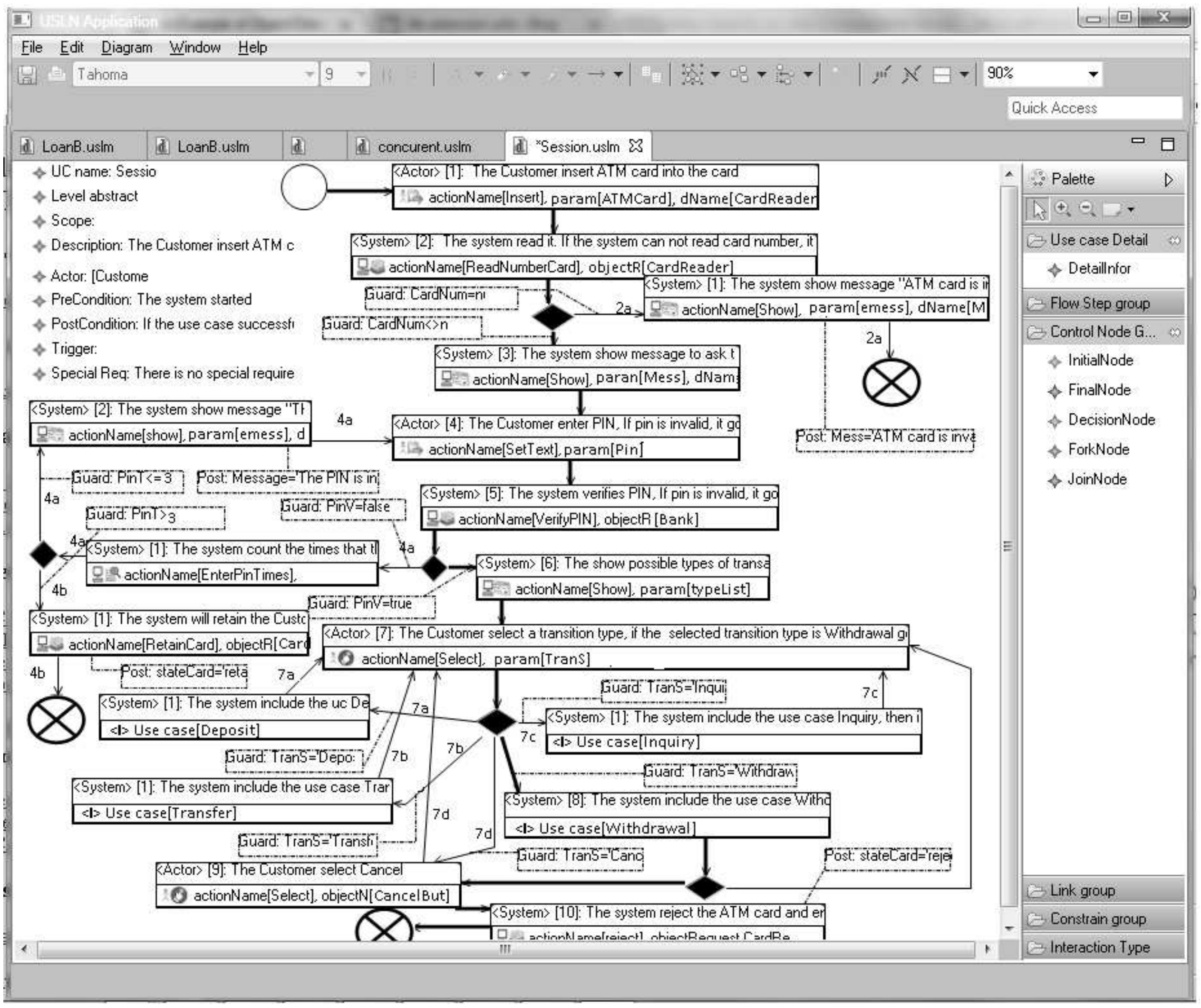

File Edit Diagram Window Help
LoanB.uslm
concurent.uslm
*Session.uslm
UC name: Sessio
Level abstract
Scope:
Description: The Customer insert ATM c
Actor: [Custome
PreCondition: The system started
PostCondition: If the use case successf
Trigger:
Special Req: There is no special require
<Actor> [1]: The Customer insert ATM card into the card
actionName[Insert], param[ATMCard], dName[CardReader
<System> [2]: The system read it. If the system can not read card number, it
actionName[ReadNumberCard], objectR[CardReader]
<System> [1]: The system show message "ATM card is i
actionName[Show], param[emess], dName[M
Guard: CardNum=n
Guard: CardNum<>n
<System> [3]: The system show message to ask t
actionName[Show], paran[Mess], dNam
Post: Mess="ATM card is inv
<System> [2]: The system show message "T
actionName[show], param[emess], d
<Actor> [4]: The Customer enter PIN. If pin is invalid, it g
actionName[SetText], param[Pin]
Guard: PinT<=3
Post: Message="The PIN is in
<System> [5]: The system verifies PIN. If pin is invalid, it go
actionName[VerifyPIN], objectR [Bank]
Guard: PinT>3
Guard: PinV=false
<System> [1]: The system count the times that t
actionName[EnterPinTimes],
<System> [6]: The show possible types of transa
actionName[Show], param[typeList]
Guard: PinV=true
<System> [1]: The system will retain the Custo
actionName[RetainCard], objectR[Car
<Actor> [7]: The Customer select a transition type, if the selected transition type is Withdrawal g
actionName[Select], param[TranS]
Post: stateCard=reta
Guard: TranS="Inqui
<System> [1]: The system include the uc De
<I> Use case[Deposit]
<System> [1]: The system include the use case Inquiry, then i
<I> Use case[Inquiry]
Guard: TranS="Depo
Guard: TranS="Withdraw
<System> [1]: The system include the use case Trar
<I> Use case[Transfer]
<System> [8]: The system include the use case Withd
<I> Use case[Withdrawal]
Guard: TranS="Transf
Guard: TranS="Canc
Post: stateCard="reje
<Actor> [9]: The Customer select Cancel
actionName[Select], objectN[CancelBut]
<System> [10]: The system reject the ATM card and er
2a
4a
4b
7a
7b
7c
7d
Palette
Use case Detail
DetailInfor
Flow Step group
Control Node G...
InitialNode
FinalNode
DecisionNode
ForkNode
JoinNode
Link group
Constrain group
Interaction Type
Quick Access
Tahoma
90%

Figure 8: Modelling use case Session in the USL Editor tool.

the use case information elements that are captured in USL are more formal than what are represented in UC-B. UC-B provides a GUI for informally describing use case scenarios. Second, UC-B only represents the steps of a use case scenario and the trigger of a use case. SilabReq and RSL only capture flows corresponding to use case scenarios. With USL, we can express more use case information, such as the pre- and postcondition of an action.

On the other hand, USL captures information elements as expressively as RUCM. The RUCM method proposes a Restricted Use Case Modeling (RUCM) language, using a set of keywords and restricted description rules. Specifications in USL are more formal than those in RUCM, because RUCM's specifications are expressed in natural language.

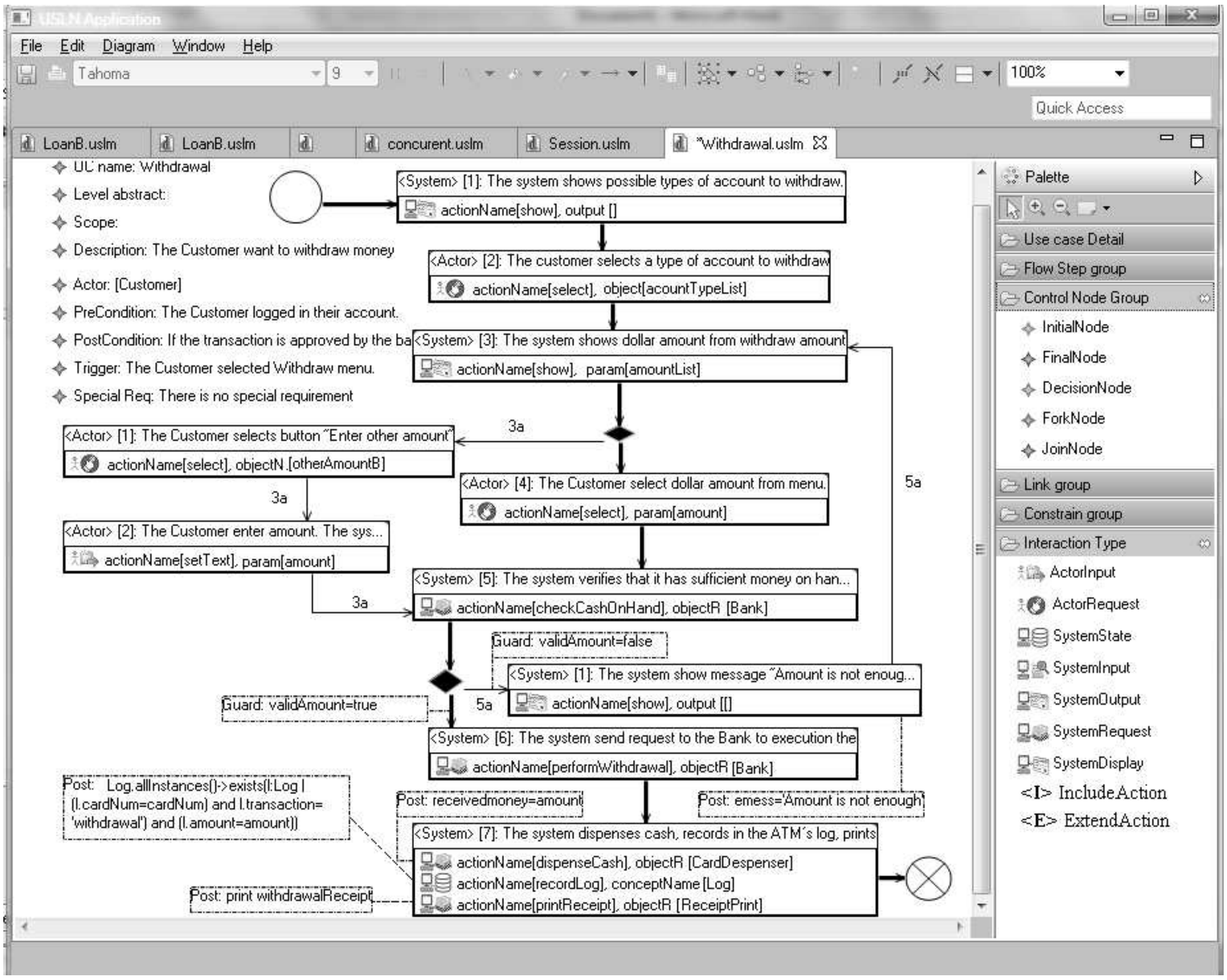
In comparison with MBD-L, USL lacks concepts for specifying sub-flows. However, as discussed in Sect. 2, use cases containing sub-flows can be smoothed so that they are suitable for modelling in USL. On the other hand, MBD-L is only specified with the abstract syntax. Unlike USL, it does not contain a concrete syntax and a formal semantic.

6.3.2 Control flow representation for use case behaviour

Our USL language is built on UML activity diagram. A USL model includes `USLNodes` (corresponding to `Nodes` in UML activity diagram) and `FlowEdges` (corresponding to `Edges` in UML activity diagram) to specify control flows which pass through steps in the use case's flows. USL captures the different control flow types of UML activity diagram (such as sequence, branch, loop, and concurrence flow). In addition, USL also specifies steps with a limited number of iterations. For example, in the use case `Session` in SubSect. 6.2, Step 4 executes a maximum of three times. Briefly, we draw the following conclusions from Table 5:

– USL can represent concurrent steps, while the other languages do not.

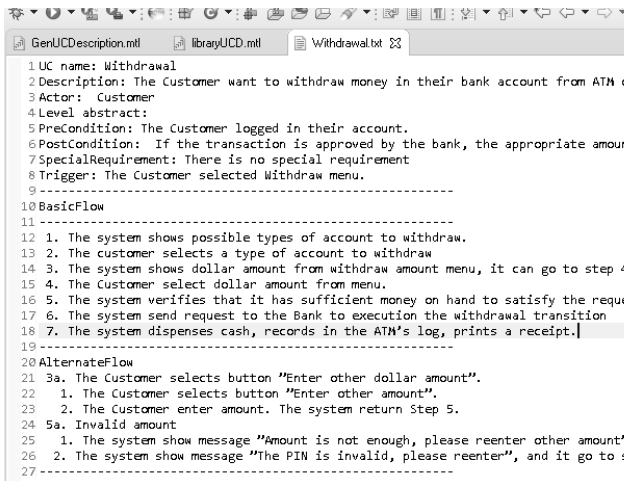
– USL *directly* represents control flows using `USLNodes` and `FlowEdges`, while the other languages do not model the flows directly.



UC name: Withdrawal
Level abstract:
Scope:
Description: The Customer want to withdraw money
Actor: [Customer]
PreCondition: The Customer logged in their account.
PostCondition: If the transaction is approved by the ba
Trigger: The Customer selected Withdraw menu.
Special Req: There is no special requirement

<System> [1]: The system shows possible types of account to withdraw.
actionName[show], output []
<Actor> [2]: The customer selects a type of account to withdraw
actionName[select], object[acountTypeList]
<System> [3]: The system shows dollar amount from withdraw amount
actionName[show], param[amountList]
<Actor> [1]: The Customer selects button "Enter other amount"
actionName[select], objectN [otherAmountB]
<Actor> [4]: The Customer select dollar amount from menu.
actionName[select], param[amount]
<Actor> [2]: The Customer enter amount. The sys...
actionName[setText], param[amount]
<System> [5]: The system verifies that it has sufficient money on han...
actionName[checkCashOnHand], objectR [Bank]
Guard: validAmount=false
<System> [1]: The system show message "Amount is not enoug...
actionName[show], output [[]
Guard: validAmount=true
<System> [6]: The system send request to the Bank to execution the
actionName[performWithdrawal], objectR [Bank]
Post: Log.allInstances()->exists(l:Log | (l.cardNum=cardNum) and l.transaction='withdrawal') and (l.amount=amount))
Post: receivedmoney=amount
Post: emess='Amount is not enough'
<System> [7]: The system dispenses cash, records in the ATM's log, prints
actionName[dispenseCash], objectR [CardDespenser]
actionName[recordLog], conceptName [Log]
actionName[printReceipt], objectR [ReceiptPrint]
Post: print withdrawalReceipt

Figure 9: Modelling use case withdrawal in the USL Editor tool.



```
UC name: Withdrawal
Description: The Customer want to withdraw money in their bank account from ATM
Actor:  Customer
Level abstract:
PreCondition: The Customer logged in their account.
PostCondition:  If the transaction is approved by the bank, the appropriate amour
SpecialRequirement: There is no special requirement
Trigger: The Customer selected Withdraw menu.
---------------------------------------------------------
BasicFlow
---------------------------------------------------------
 1. The system shows possible types of account to withdraw.
 2. The customer selects a type of account to withdraw
 3. The system shows dollar amount from withdraw amount menu, it can go to step
 4. The Customer select dollar amount from menu.
 5. The system verifies that it has sufficient money on hand to satisfy the reque
 6. The system send request to the Bank to execution the withdrawal transition
 7. The system dispenses cash, records in the ATM's log, prints a receipt.
---------------------------------------------------------
AlternateFlow
 3a. The Customer selects button "Enter other dollar amount".
   1. The Customer selects button "Enter other amount".
   2. The Customer enter amount. The system return Step 5.
 5a. Invalid amount
   1. The system show message "Amount is not enough, please reenter other amount'
  2. The system show message "The PIN is invalid, please reenter", and it go to
---------------------------------------------------------
```

Figure 10: The TUCD generated from the use case *Withdrawal*.

In comparison with all other works [10, 3, 4, 16, 17], our method can additionally specify concurrent steps. Moreover, these works do not directly specify control flows. They only capture rejoin points or refer to other steps.

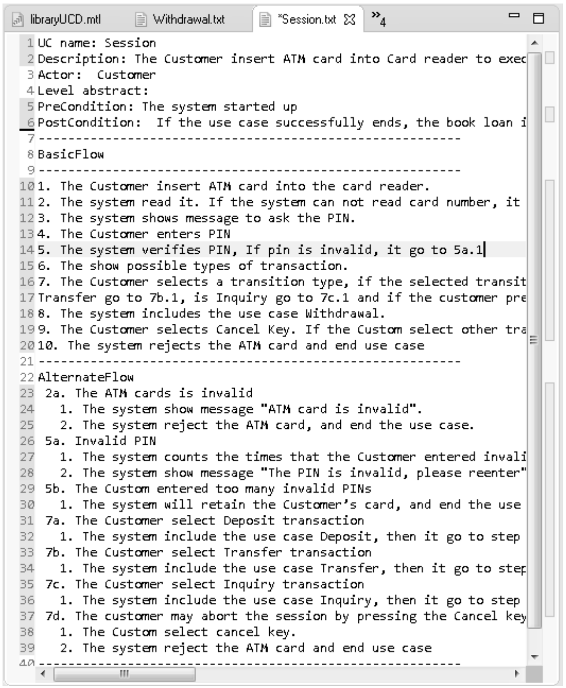
6.3.3 Action specification

As discussed in Sect. 4, USL precisely specifies use case behaviors using nine action types. These action types are represented by meta-concepts in the USL metamodel. The action type of each behavior enables us to identify sender objects, receiver objects, messages, parameters of actions and object types. Briefly, we draw the following conclusions from Table 5:

– *Action type coverage*:

  – USL represents all the action types that are supported in other languages.

  – USL complements several action types, compared to four related languages, MBD-L, SiLabReq, RSL, and RUCM. USL employs two new action types `IncludeAction` and `ExtendAction` to represent use case relationships.



```
UC name: Session
Description: The Customer insert ATM card into Card reader to exec
Actor:  Customer
Level abstract:
PreCondition: The system started up
PostCondition:  If the use case successfully ends, the book loan i
-----------------------------------------------------------
BasicFlow
-----------------------------------------------------------
1. The Customer insert ATM card into the card reader.
2. The system read it. If the system can not read card number, it
3. The system shows message to ask the PIN.
4. The Customer enters PIN
5. The system verifies PIN, If pin is invalid, it go to 5a.1
6. The show possible types of transaction.
7. The Customer selects a transition type, if the selected transit
Transfer go to 7b.1, is Inquiry go to 7c.1 and if the customer pre
8. The system includes the use case Withdrawal.
9. The Customer selects Cancel Key. If the Custom select other tra
10. The system rejects the ATM card and end use case
-----------------------------------------------------------
AlternateFlow
 2a. The ATM cards is invalid
   1. The system show message "ATM card is invalid".
   2. The system reject the ATM card, and end the use case.
 5a. Invalid PIN
   1. The system counts the times that the Customer entered invali
   2. The system show message "The PIN is invalid, please reenter"
 5b. The Custom entered too many invalid PINs
   1. The system will retain the Customer's card, and end the use
 7a. The Customer select Deposit transaction
   1. The system include the use case Deposit, then it go to step
 7b. The Customer select Transfer transaction
   1. The system include the use case Transfer, then it go to step
 7c. The Customer select Inquiry transaction
   1. The system include the use case Inquiry, then it go to step
 7d. The customer may abort the session by pressing the Cancel key
   1. The Custom select cancel key.
   2. The system reject the ATM card and end use case
-----------------------------------------------------------
```

Figure 11: The TUCD generated from the use case *Session*.

- *Precise specification*:
  - USL uses the USL meta-concepts to represent actions.
  - The actions in USL are precisely specified using pre- and postconditions. Some languages, e.g., MBD-L, SiLabReq, and RSL also support this feature. Others, namely UC-B and RUCM, do not support it.

We use more action types to classify behaviors and we capture the behavior's information more precisely. More specifically, by using different concepts in USL to specify action types our approach captures behaviors more formally than UC-B. In UC-B, behaviors are not precisely specified and are divided into different action types. Similarly, behaviors are better captured in USL than in RUCM, because the latter only uses keywords and restricted rules in natural language to divide behaviors into action types. Moreover, RUCM does not support the action type named `SystemDisplay`, that is captured in USL.

In comparison with MBD-L, SiLabReq, and RSL, actions in USL are better classified with nine action types. MBD-L uses only four categories of action types: Request, DataValidate, Expletive and Response. Similarly, SeLabReq divides actions into four groups: *Actor prepares Data (APDExecuteSO)*, *Actor calls System (ACSExecuteSO)*, *System executes SystemOperation (SExecuteSO)*, and *System replies and returns Result (SRExecutionSO)*. The classification method of RSL is less specific than USL's, because it does not support the type of system action that sends a request to a primary actor. This system action type is specified in USL by `SystemRequest`.

6.3.4 Constraint representation

USL employs OCL to define constraints in use case. Specifically, a use case's precondition is specified by a `Constraint` associated with the `InitialNode`. A use case scenario's postcondition is specified by a `Constraint` associated with a `FinalNode` of scenario. Similarly, guard conditions on flows and actions' pre- and postconditions are captured by `Constraint` associated with `FlowEdges` and actions, respectively. Briefly, we draw the following conclusions from Table 5:

- USL supports a more complete set of constraints than four other languages, namely RUCM, MBD-L, SiLabReq, and RSL.
- Constraint representation in USL (using OCL) is more precise than two other languages, RUCM and RSL (these languages use natural language to write constraints).

USL specifies more constraint types than four other language: RUCM, MBD-L, SiLabReq, and RSL. Unlike USL, these languages do not support actions' pre- and postcondition. Moreover, USL is better than RUCM and RSL in terms of precision, because several languages, such as MBD-L, SiLabReq, and RSL, also support this feature. The other languages, UC-B and RUCM, do not support it.

It is worth mentioning that constraints specified in USL are quite similar to constraints in UC-B. In the latter, constraints are specified using Event-B's mathematical language. However, this language is rather inconvenient and difficult for non-technical stakeholders to understand.

6.3.5 Applying USL in practice

It is possible to apply USL in practice for two main reasons. First, as discussed in Sect. 5, use cases are precisely specified and represented in USL as models, which conform to a metamodel. This enables them to be automatically transformed into other software artifacts, such as textual use case descriptions, structural and behavioral models and test cases. These generated models are necessary artifacts in software development.

Second, the USL tool realizes our USL approach as an Eclipse modeling project (DSL toolkit) [26]. This tool enables the modeler to visually create USL models and to integrate these models into the existing UML use case models and class model (the latter captures the domain concepts of a system). Moreover, our DSL toolkit provides the meta-metamodel language MOF to build USL. It also enables the definition of model transformation languages in order to realize the transformations discussed in Sect. 5.

However, USL is not without limitations. The graphical concrete syntax of the language might be inconvenient for modelers who prefer writing use cases in the textual form. In order to accommodate for this, the USL tool would be extended with a textual editor, similar to one used in the RSL approach [17]. This textual editor would enable a modeler to specify use cases by entering descriptive sentences about actions in steps, constraints, and relations between steps. The tool would then process these to create the corresponding USL model.

7 Related work

We position our work in the intersection between use case-driven development [1] and model-driven development [18]. Within this context, a use case model is usually represented as a combination of a UML use case diagram and a textual description written in natural language. Such a use case specification tends to be ambiguous, unclear, and inconsistent. In order to precisely specify use cases several approaches as in [10], [4], [16], [17], [3] have been proposed.

T. Yue *et al.* [3] proposed a use case modeling language called Restricted Use Case Modeling (RUCM), which is composed of a use case description template, a set of keyword, and a set of well-defined restrictions for a restricted natural language to specify use cases. However, the RUCM is semi-formal textual language and it does not mention some important information such as concurrent actions, the pre- and postcondition of actions. Hence, in other work that use RUCM to express use case specifications to automatically generate other artifacts, they have to use NLP(Natural Language Processing) technique to extract information. For example, C. Wang *et al.* [30] uses use case specifications expressed in RUCM in order to generate test cases. After use NLP technique to extract test scenarios and constraints described in natural language, they use OCL to precisely specify constraints and use these precise specifications to automatically generate test data.

R. Murali *et al.* [10] proposed using a mathematical language w.r.t. Event-B in order to formalize the pre- and postcondition of triggers and actions within use case flows. However, other descriptions of a use case are still informal. Their proposition only focus automatically generates a corresponding Event-B model that is then amenable to the Rodin verification tools that enable system-level properties to be verified.

M. Misbhauddin *et al.* [4] extended the metamodel of UML use case models in order to capture both the structural and behavioural aspects of use cases. To specify a use case, they developed a prototype tool called `UCDest`. However, concurrent actions, pre- and postcondition of actions have not been mentioned. Moreover, action types are defined inadequately.

D. Savić *et al.* [16] and M. Smialek *et al.* [17] proposed the DSLs named `SilabReq` and `RSL` in order to capture use cases as the functional requirements models. The DSLs only focus on flows describing use case scenarios while other description information of use case is omitted. In addition, the RSL does not define distinguish actions inserting an extending use case and an included use case, both are defined <invoke> action. Furthermore, the DSLs do not mention concurrent actions, pre and postcondition of actions.

In comparison with all the work above, We provide for USL a formal semantic which use LTS to express , while other works lack a formal semantics.

Our previous work in [36, 9] proposed a metamodel to specify use cases. In that work we also tried to define a precise semantics for use cases based on graph transformation. Our work here continues it by enhancing the use case metamodel as well as proposing a new LTS-based technique in order to characterize the operational semantics of use case.

Furthermore, all above mentioned approaches still lack a method specifying use cases satisfying all relevant information of use cases including flows, steps, system actions, actor actions, control flows, relationships, and constraints on the use case and its flows.

The USL language, introduced in this work, aims to cover all relevant information of a use case including both structural and behavioural aspect. Comparing to the current works in literature, USL could obtain the following advantages: (1) to specify concurrent actions in flows; (2) to capture and represent nine action types in which there are the system action including another use case and the system action extending another use case that have not been mentioned in other research; (3) to present not only constraints on the use case and its flows but pre- and postcondition of each action in flows; (4) to present control flows of steps within the use case. In addition, in this paper we also defined operational semantics of USL to specify dynamic information when use case scenarios execute. In that way, from USL models we could obtain software artifacts by transformations.

8 Conclusion

This paper proposed a DSL named USL to specify use cases. A USL model can cover the relevant information of a use case description including flows, steps, system actions, actor actions, relationships, control flows, and constraints. We built the abstract using a metamodel together with wellformedness rules and the graphical concrete syntax of USL. Moreover, we defined precise semantic for the USL by mapping USL models to LTSs. We also developed a USL Editor to create the USL models visually. In addition, we explained how USL models can be transformed to some software artifacts and developed a model transformation program to automatically generate textual template-based use case descriptions. Moreover, we evaluated USL's expressiveness.

In the future work, we will focus on realizing transformations from USL models in order to generate test cases as well as other software artifacts automatically. In addition, we will enrich the abstract syntax and enhance the concrete syntax of USL in order to support better for modelers.

Acknowledgement

We wish to thank the anonymous reviewers for their useful comments.

References

[1] I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley Longman Publishing Co., Inc., 2004.

[2] OMG, "UML 2.5," May 2005.

[3] T. Yue, L.C. Briand, and Y. Labiche, "Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments," ACM Trans. Softw. Eng. Methodol., vol.22, no.1, pp.5:1–5:38, March 2013.

[4] M. Misbhauddin and M. Alshayeb, "Extending the UML Use Case Metamodel with Behavioral Information to Facilitate Model Analysis and Interchange," Software & Systems Modeling, vol.14, no.2, May.

[5] P. Kruchten, The Rational Unified Process: An Introduction, 3 ed., Addison-Wesley Professional, 2004.

[6] D. Liu, K. Subramaniam, B.H. Far, and A. Eberlein, "Automating Transition from Use-Cases to Class Model," Proc. Canadian Conf. Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436)(CCECE), 2003.

[7] P. Haumer, "Use case-based software development," in Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle, ed. I. Alexander and N. Maiden, ch. 12, pp.237–264, Wiley, 2004.

[8] S. Tiwari and A. Gupta, "A Systematic Literature Review of Use Case Specifications Research," Inf. Softw. Technol., vol.67, no.C.

[9] D.H. Dang, "Triple Graph Grammars and OCL for Validating System Behavior," Proc. 4th Int. Conf. Graph Transformations (ICGT), LNCS 5214, pp.481–483, Springer, 2008.

[10] R. Murali, A. Ireland, and G. Grov, "UC-B: Use Case Modelling with Event-B," Proc. 5th Int. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z(ABZ), ed. M. Butler, K.D. Schewe, A. Mashkoor, and M. Biro, LNCS 9675, Springer, 2016.

[11] W. Grieskamp and M. Lepper, "Using Use Cases in Executable Z," Proc. 3th Int. Conf. Formal Engineering Methods (ICFEM), pp.111–119, IEEE, 2000.

[12] J.S. Thakur and A. Gupta, "Automatic Generation of Sequence Diagram from Use Case Specification," Proc. 7th India Conf. Software Engineering (ISEC), pp.20:1–20:6, ACM, 2014.

[13] L. Li, "Translating Use Cases to Sequence Diagrams," Proc. 15th Int. Conf. Automated Software Engineering (ASE), pp.293–298, IEEE Computer Society, 2000.

[14] J.M. Almendros-Jiménez and L. Iribarne, "Describing Use Cases with Activity Charts," Proc. Int. Conf. Metainformatics (MIS 2Generation of System Test Cases from Use Case Specifications,"004), LNCS 3511, pp.141–159, Springer-Verlag, 2005.

[15] S. Tiwari and A. Gupta, "An Approach of Generating Test Requirements for Agile Software Development," Proc. 8th on India Conf. Software Engineering (ISEC), ACM, 2015.

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

[17] M. Smialek and W. Nowakowski, From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice, Springer.

[18] M. Brambilla, J. Cabot, and M. Wimmer, Model-Driven Software Engineering in Practice, 1st ed., Morgan & Claypool Publishers, 2012.

[19] OMG, "OCL 2.0," May 2006.

[20] R.M. Keller, "Formal verification of parallel programs," Commun. ACM, vol.19, no.7, pp.371–384, July 1976.

[21] M.H. Chu, D.H. Dang, N.B. Nguyen, M.D. Le, and T.H. Nguyen, "USL: Towards Precise Specification of Use Cases for Model-Driven Development," Proc. 8th Int. Conf. Information and Communication Technology (SoICT), pp.401–408, 2017.

[22] A. Cockburn, Writing Effective Use Cases, 1 edition ed., Addison-Wesley Professional, Boston, Oct. 2000.

[23] I. Jacobson, I. Spence, and K. Bittner, USE-CASE 2.0 The Guide to Succeeding with Use Cases, Ivar Jacobson International SA., 2011.

[24] M. Giese and R. Heldal, "From Informal to Formal Specifications in UML," Proc. Int. Conf. The Unified Modeling Language: Modelling Languages and Applications (UML), ed. T. Baar, A. Strohmeier, A.M.D. Moreira, and S.J. Mellor, LNCS 3273, 2004.

[25] P. Schmitt, I. Tonin, C. Wonnemann, E. Jenn, S. Leriche, and J.J. Hunt, "A Case Study of Specification and Verification Using JML in an Avionics Application,"

[26] R.C. Gronback, Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit, 1 edition ed., Addison-Wesley Professional, Boston, March 2009.

[27] J. Heumann, "Generating Test Cases From Use Cases," tech. rep., Rational Software, 2001.

[28] s. results and B. Legeard, Practical Model-Based Testing: A Tools Approach, 1 edition ed., Morgan Kaufmann, Amsterdam ; Boston, Dec. 2006.

[29] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating Test Cases from UML Activity Diagram Based on Gray-Box Method," Proc. 11th Asia-Pacific Conf. Software Engineering (APSEC), IEEE Computer Society, 2004.

[30] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic Generation of System Test Cases from Use Case Specifications," Proc. Int. Symposium Conf. Software Testing and Analysis (ISSTA), pp.385–396, ACM, 2015.

[31] "Generating Test Data from OCL Constraints with Search Techniques," vol.39.

[32] B. Regnell, M. Andersson, and J. Bergstrand, "A Hierarchical Use Case Model with Graphical Representation," Proc. IEEE Symposium and Workshop on Engineering of Computer-Based Systems (ECBS), pp.270–277, IEEE Computer Society, 1996.

[33] G. Kotonya and I. Sommerville, Requirements Engineering: Processes and Techniques, 1st ed., Wiley Publishing, 1998.

[34] M. Genero Bocco, A. Durán Toro, and B. Bernárdez Jiménez, "Empirical Evaluation and Review of a Metrics-Based Approach for Use Case Verification," Journal of Research and Practice in Information Technology, vol.36, no.4, pp.247–258, 2004.

[35] Russell C. Bjork, "An Example of Object-Oriented Design: An ATM Simulation." `http://www.math-cs.gordon.edu/courses/cs211/ATMExample/`. Accessed: 2018-01-01.

[36] D.H. Dang, A.H. Truong, and M. Gogolla, "Checking the Conformance between Models Based on Scenario Synchronization," Journal of Universal Computer Science, vol.16, no.17, pp.2293–2312, 2010.