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

Chu Thi Minh Hue¹, Dang Duc Hanh², Nguyen Ngoc Binh³ and Le Minh Duc⁴
Department of Software Engineering, VNU University of Engineering and Technology
¹Hung Yen University of Technology and Education, Vietnam
²Corresponding author
³Visiting professor, Hosei University, Japan
⁴Hanoi University, Vietnam
E-mail: {huectm.dil12 | 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, actors 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 number 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 transaction. The use case is represented in a typical template as shown in Table 1.

![Figure 1: The simplified use case and the conceptual domain model of the Library system.](image)

### Table 1: A typical use case description template

<table>
<thead>
<tr>
<th>Use case name: Lend Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brief description: The Librarian processes a book loan.</td>
</tr>
<tr>
<td>Actors: Librarian.</td>
</tr>
<tr>
<td>Precondition: The librarian has logged into the system successfully.</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>Trigger: The Librarian requests a book-loan process.</td>
</tr>
<tr>
<td>Special requirement: There is no special requirement.</td>
</tr>
</tbody>
</table>

#### 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.
3. The Librarian enters the extended search id.
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.
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 post-condition 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.](Image)

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 meta-modeling 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 “bcd”.

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 consists 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 a1_1, ..., a1_10 as AlternateFlowEdges.

Variable (domain source (3)) represents variables that
Table 2: List of utility functions w.r.t. USL concepts

<table>
<thead>
<tr>
<th>Utility function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstAct: ControlNode → ControlNode</td>
<td>Returning the ControlNode itself.</td>
</tr>
<tr>
<td>lastAct: ControlNode → ControlNode</td>
<td>Returning the ControlNode itself.</td>
</tr>
<tr>
<td>source: FlowEdge → USLNode</td>
<td>Returning the source USLNode of a FlowEdge.</td>
</tr>
<tr>
<td>target: FlowEdge → USLNode</td>
<td>Returning the target USLNode of a FlowEdge.</td>
</tr>
<tr>
<td>guardE: FlowEdge → Constraint</td>
<td>Returning the guard condition.</td>
</tr>
<tr>
<td>guardE: USLNode → USLNode → Constraint</td>
<td>Determining whether or not lastAct(source(e)) has completed its execution.</td>
</tr>
<tr>
<td>isCompleted: FlowEdge → Boolean</td>
<td>Returning the precondition of an Action.</td>
</tr>
<tr>
<td>preA: Action → Constraint</td>
<td>If the ControlNode is not a InitialNode, returning true, else returning the Constraint of the InitialNode</td>
</tr>
<tr>
<td>postA: Action → Constraint</td>
<td>If the ControlNode is not a FinalNode, returning true, else returning the Constraint of the FinalNode</td>
</tr>
<tr>
<td>preC: USLModel → Constraint</td>
<td>Returning the precondition of a USLModel.</td>
</tr>
<tr>
<td>postC: USLModel → Constraint</td>
<td>Returning the postcondition of a USLModel.</td>
</tr>
<tr>
<td>postC: USLModel → FinalNode → Constraint</td>
<td>Returning the postcondition of a particular FinalNode of a USLModel.</td>
</tr>
</tbody>
</table>

Example 4.1.4. The USL model as shown in Fig. 4 contains g1, . . . , g6 as guard conditions and p1, . . . , p6 as post-conditions 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:

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

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

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

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

```ocl
def self.uslnode->selectByKind(FlowStep)
```
Rule 4. An InitialNode has one out-going BasicFlowEdge and does not have any in-coming FlowEdges:

\[
\text{size}(\text{BasicFlowEdge}) = 1 \\
\text{ocllsTypeOf(InitialNode)} \rightarrow \text{size}(0) \\
\text{self.flowedge} \rightarrow \text{select}(t: \text{FlowEdge} | t.\text{source}.\text{ocllsTypeOf(InitialNode)} \rightarrow \text{size}(1) \text{and} (t.\text{target.oclIsTypeOf(InitialNode)}) \text{and} (t.\text{source.oclIsTypeOf(BasicFlowEdge)})) \rightarrow \text{size}(1) \\
\text{self.flowedge} \rightarrow \text{select}(t: \text{FlowEdge}) \text{t.size} = 1 \text{and} (t.\text{source.oclIsTypeOf(InitialNode)}) \text{and} (t.\text{target.oclIsTypeOf(BasicFlowEdge)})) \rightarrow \text{size}(1) \\
\]

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

\[
\text{size}(\text{FinalNode}) = 1 \\
\text{self.uslnode} \rightarrow \text{selectByType(FinalNode)} \rightarrow \text{for all f:FinalNode} \text{self.flowedge} \rightarrow \text{select}(e: \text{FlowEdge} | e.\text{target} = f \rightarrow \text{size}(1) \text{and} (e.\text{source.oclIsTypeOf(FlowEdge)} \rightarrow \text{size}(0)) \\
\]
Rule 6. A DecisionNode has one in-coming FlowEdge and at least two out-going FlowEdges:

\[
(d:\text{DecisionNode}) \rightarrow \text{self.flowedge->select(e: FlowEdge|e.target=d)->size()>=1 and } \text{self.flowedge->select(e:FlowEdge|e.source=d)->size()>=2)}
\]

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

\[
\text{self.uslnode->selectByType(ForkNode)->forAll(f:ForkNode)\rightarrow \text{self.flowedge->select(e:FlowEdge|e.target=f)->size()>=1 and } \text{self.flowedge->select(e:FlowEdge|e.source=f)->size()>=2)}
\]

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

\[
\text{self.uslnode->selectByType(JoinNode)->forAll(n:JoinNode)(self.flowedge->select(f:FlowStep|n.number=f.number)->size()>=1 and } \text{self.flowedge->select(e:FlowEdge|e.source=f)->size()<=1)}
\]

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

\[
\text{self.uslnode->selectByKind(FlowStep)->forAll(f:FlowStep)(self.flowedge->select(e:FlowEdge|e.target=f)->size()>=1 and } \text{self.flowedge->select(e:FlowEdge|e.source=f)->size()<=1)}
\]

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:

\[
\text{self.uslnode->forAll(n:USLNode)}; \quad \text{if (n.oclIsTypeOf(InitialNode)) then } \text{self.flowedge->select(b:FlowEdge)}; \quad \text{if (b.oclIsTypeOf(USL::InitialNode)) and (b.oclIsTypeOf(AlternateFlowEdge)) \rightarrow size()=1}
\]

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.
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;
- $C = G \cup C_{\text{preUC}} \cup C_{\text{postUC}} \cup C_{\text{preA}} \cup C_{\text{postA}}$ is the set of Constraints,

where:

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

**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;
- $C = G \cup C_{\text{preUC}} \cup C_{\text{postUC}} \cup C_{\text{preA}} \cup C_{\text{postA}}$ is the set of Constraints,

where:

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

**Example 4.1.5.** The USL model as shown in Fig. 4 contains the following elements: $N_f = \{ f0 \}$; $N_F = \{ f7, f8 \}$; $A_{\text{Node}} = \{ c0, \ldots, c8 \}$; $A_a = \{ s1, s3, s5, s7, s13 \}$; $A_s = \{ s2, s4, s6, s8, \ldots, s12, s14, s15, s16 \}$; $E_b = \{ b1, \ldots, b17 \}$; $E_a = \{ a1, \ldots, a10 \}$; $G = \{ g1, \ldots, g6 \}$; $C_{\text{preUC}} = \emptyset$; $C_{\text{postUC}} = \emptyset$; $C_{\text{preA}} = \emptyset$; and $C_{\text{postA}} = \{ p1, \ldots, p6 \}$. $D_C$ 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 all is expressed by the following OCL constraint: $\text{BookLoan.allInstances()}\rightarrow\exists(b:BookLoan| (b.bcuid = bcid) and (b.pid = bid) and (b.payed=0))$.

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 $(\Sigma(V), P(G \times A \times P), T, \alpha_{\text{init}}, F)$ such that:

**Table 3: The graphical notations of USL**

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Presentation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DescriptionInfor</td>
<td>A borderless text box that properties are listed in the text box</td>
<td>☐</td>
</tr>
<tr>
<td>InitialNode</td>
<td>An unfilled circle</td>
<td>☐</td>
</tr>
<tr>
<td>FinalNode</td>
<td>A circle with a crosshairs symbol</td>
<td>☐</td>
</tr>
<tr>
<td>DecisionNode</td>
<td>A filled diamond with one in-coming arrowed line and at least two out-going arrowed lines</td>
<td>☐</td>
</tr>
<tr>
<td>ForkNode</td>
<td>A solid line segment with one in-coming arrowed line and at least two out-going arrowed lines</td>
<td>☐</td>
</tr>
<tr>
<td>JoinNode</td>
<td>A solid line segment with at least two in-coming arrowed lines and one out-going arrowed line</td>
<td>☐</td>
</tr>
<tr>
<td>AlternateEdge</td>
<td>A labelled, thin arrowed line (the label is the name of the flow)</td>
<td>☐</td>
</tr>
<tr>
<td>BasicEdge</td>
<td>A thick arrowed line</td>
<td>☐</td>
</tr>
<tr>
<td>ActorStep</td>
<td>A labeled, 2-part rectangle. The first part contains the label &lt;Actor&gt; and two properties numberStep and description of the ActorStep. The second part contains the ActorActions of the ActorStep</td>
<td>☐</td>
</tr>
<tr>
<td>SystemStep</td>
<td>A labeled, 2-part rectangle. The first part contains the label &lt;System&gt; and two properties numberStep and description of the SystemStep. The second part contains the SystemActions of the SystemStep</td>
<td>☐</td>
</tr>
<tr>
<td>Action</td>
<td>Information of a Action are presented by textual form in the second part of Flowsteps</td>
<td>☐</td>
</tr>
</tbody>
</table>
Table 4: LTS-based semantics of the basic USL concepts

<table>
<thead>
<tr>
<th>USL concepts</th>
<th>Notation</th>
<th>LTS-based semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>$n_1 \rightarrow \cdots \rightarrow n_n$</td>
<td>$g_1[n_1]r_1 \cdots g_n[n_n]r_n$ where $g_i = \text{preA}(a_i), r_i = \text{postA}(a_i)$ ($\forall i = 2, \ldots, n$).</td>
</tr>
<tr>
<td>Flow edge</td>
<td>$n_1 \rightarrow n_2$</td>
<td>$g_1[n_1]r_1 g_2[n_2]r_2$ where $a_2 = \text{firstAct}(n_2), g_2 = \text{guardE}(n_1, n_2) \land \text{preA}(a_2), r_2 = \text{postA}(a_2)$.</td>
</tr>
<tr>
<td>Decision node</td>
<td>$n_i$</td>
<td>$g_1[n_i]r_d g_2[n_2]r_d$ where $a_d = \text{lastAct}(n_d), g_2 = \text{guardE}(n_d, c), a = \text{firstAct}(n), g_2 = \text{guardE}(c, n) \land \text{preA}(a) \land n \in {n_1, \ldots, n_m}$.</td>
</tr>
<tr>
<td>Fork node</td>
<td>$n_f \rightarrow n_1 \rightarrow \cdots \rightarrow n_m$</td>
<td>$g_1[n_f]r_f g_2[n_2]r_f$ where $a_f = \text{lastAct}(n_f), g_2 = \text{guardE}(n_f, c), a = \text{firstAct}(n), g_2 = \text{guardE}(c, n) \land \text{preA}(a), r_f = \text{postA}(a)$.</td>
</tr>
<tr>
<td>Join node</td>
<td>$n_1 \rightarrow \cdots \rightarrow n_m$</td>
<td>$r_1[n_1]r_1 \cdots r_m[n_m]r_m$ where $\alpha_1 = \text{lastAct}(n_1), (\forall i = 1, \ldots, m)$; $g_1[n_1]r_1 \cdots g_m[n_m]r_m \cdots g_1[n_1]r_1 \cdots g_m[n_m]r_m$ where $\alpha_1 = \text{lastAct}(n_1), (\forall i = 1, \ldots, m)$; $\exists \text{guardE}(c, n) \land \text{postA}(a)$.</td>
</tr>
<tr>
<td>Initial node</td>
<td>$c \rightarrow n_1$</td>
<td>$r_1[n_1]r_1 \cdots r_m[n_m]r_m$ where $\alpha_1 = \text{lastAct}(n_1), (\forall i = 1, \ldots, m)$; $\exists \text{guardE}(c, n) \land \text{postA}(a)$.</td>
</tr>
<tr>
<td>Flow final node</td>
<td>$n_1 \rightarrow c$</td>
<td>$g_1[n_1]r_1 g_2[n_2]r_2$ where $\alpha_1 = \text{lastAct}(n_1), (\forall i = 1, \ldots, m)$; $\exists \text{guardE}(c, n) \land \text{postA}(a)$.</td>
</tr>
<tr>
<td>USL model with include action</td>
<td>$n_1 \rightarrow \cdots \rightarrow n_2$</td>
<td>$D_f \rightarrow n_1 \rightarrow \cdots \rightarrow n_2$ where $n \in A_n,</td>
</tr>
<tr>
<td>USL model with extend action</td>
<td>$n_1 \rightarrow \cdots \rightarrow n_2$</td>
<td>$D_x \rightarrow n_1 \rightarrow \cdots \rightarrow n_2$ where $n \in A_n,</td>
</tr>
</tbody>
</table>

Legend

- $\alpha$ a action in a step; $s$ a step; $D_x$ one case; $\alpha$ a state

- $V$ is a finite set of variables whose types include the basic types and the classes of the $D_C$;
- $\Sigma(V)$ is the set of states ($\alpha$), each of which is a set of value assignments to a subset of variables in $V$;
- $\mathcal{P} \subseteq C_{\text{postA}} \cup C_{\text{postUC}}$ is the set of constraints as the postconditions of $D$;
- $A = A_{\text{Node}} \cup A_{\text{act}}$ is the set of actions;
- $\mathcal{G} \subseteq G_{\text{preUC}} \cup C_{\text{preA}}$ is the set of guard conditions of the transitions;
- $\mathcal{T} \subseteq \Sigma(V) \times \mathcal{P}(\mathcal{G} \times A \times \mathcal{P}) \times \Sigma(V)$ is the transition relation defined as follows: A transition $t = (\alpha, (g, a, d), \alpha') \in \mathcal{T}$, written as $\alpha \xrightarrow{g[a]} \alpha'$, where $a \in A$ is the action that causes $t, g = \text{defGuard}(a) \in \mathcal{G}$ is the guard condition to execute $a, r \in \mathcal{P}$ is the postcondition of $a$, and $\alpha, \alpha' \in \Sigma(V)$ are the pre- and post-states of $t$ (resp.) such that $\alpha'$ satisfies $r$;
- $\alpha_{\text{init}} \in \Sigma(V)$ is the initial state;
- $\mathcal{F} \subseteq \Sigma(V)$ is the set of final states,
where:
- $A_{\text{act}} = \bigcup_{a \in A_f} \text{actions}(a)$;
- defGuard is defined as follows (summarized from Table 4).
Example 4.3.1. We assume that the snapshot shown in Fig. 5 is captured when the USL model as shown in Fig. 4 executes at Step a8. We have the following value assignments: \((\text{bcid}, "001"), (\text{lid}, "110"), (\text{iDate}, "25/8/17"), (\text{ldate}, "25/8/17"), (\text{bid}, "2018"), (\text{ifDecisionNode}), (\text{guardE})).\) The objects of the snapshot are as follows: BookCopy: 001, Borrower: 123, Librarian: 111. Then, we have \(\alpha_{a8} = \{(\text{bcid}, "001"), (\text{iDate}, "001"), (\text{lid}, "110"), (\text{bid}, "2018"), (\text{ifDecisionNode}), (\text{guardE})\} = \{\text{BookCopy: 001}, \text{Borrower: 123}, \text{Librarian: 111}\}.\) 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 \(\alpha\) of an LTS \(L\) of a USL model \(D\) and a transition \(t = \alpha \xrightarrow{\text{pre}(t)} \alpha' \in L.T\), we define the following terms:

- \(\text{preT}(t) = \alpha, \text{postT}(t) = \alpha', \text{guard}(t) = g, \text{postC}(t) = r, \text{act}(t) = a.\)

- \(\text{eval}(g)\) is the evaluation of Constraint \(g.\)

- \(\text{reachable}(\alpha) = \{t \mid \text{preT}(t) = \alpha\}\) is the set of transitions that start from \(\alpha.\)

- \(\text{firable}(\alpha) = \{t \in \text{reachable}(\alpha), \text{eval}(\text{guard}(t)) = \text{true}\}\) is the set of transitions that can be fired from \(\alpha.\)

Example 4.3.2. When the USL model as shown in Fig. 4 executes at action a11, we have \(\alpha_{a11} = \{(\text{bid}, "2018"), (\text{lid}, "110"), (\text{ifDecisionNode}), (\text{guardE})\}.\) 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 Transforming USL models to other software artifacts

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 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

```plaintext
[module GenUCDescription('http://eclipse.USLModel/USL')]
[import org::eclipse::acceleo::module::sample::service::libraryUCD]
[template public generateElement(uc: UseCase)]
[comment @main/]
[file (uc.descriptioninfor->r at(0).useCaseName.concat(’.txt’), false,’UTF-8’)]
[let d:DescriptionInfor =uc. descriptioninfor->r at(0)]
[ UC name: [d.useCaseName/]
[ Description: [d.description/]
[ Actor: [for(a:String|d.actor)] a, [/for]
[ Level abstract: [d.levelAbstract/]
[ Precondition: [d.preCondition/]
[ Postcondition: [d.postCondition/]
[ SpecialRequirement: [d.specialRequirement/]
[/let]
[BasicFlow]
[let Bsteps:OrderedSet(FlowStep)=
 getBasicFlow(uc)]
[for(s:FlowStep|Bsteps]
[s.number/]. [s.description/]
[/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]

[/for]

[/if]

[/else]

[/for] [/let]

[/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:1[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) /]

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.
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.

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

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.

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,
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 concurrency 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.
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.
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, and RUCM, do not support it.

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, SiLabReq 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 languages: 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 keywords, 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. Mishbhaudin 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 well-formedness 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


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


