# USL: Towards Precise Specification of Use Cases for Model-Driven Development

[redacted]

## ABSTRACT

Use cases have been widely employed as an efficient means to capture and structure software requirements. A use case model is often represented by a loose combination between a UML use case diagram and a textual description in natural language. The use case model expressed in such a form often 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. This paper introduces a domain specific language named the Use case Specification Language (USL) to precisely specify use cases with two main features: (1) The USL has a concrete syntax in graphical form that allows us to achieve the usability goal; (2) The precise semantics of USL that is defined by mapping the USL to a Labelled Transition System (LTS) opens a possibility for transformations from USL models to other artifacts such as test cases and analysis class models.

## CCS CONCEPTS

• **Software and its engineering → Domain specific languages**;

## KEYWORDS

use cases, model transformation, labelled transition systems, domain specific languages, pre and postcondition.

**ACM Reference Format:**
Minh-Hue Chu, Duc-Hanh Dang, Ngoc-Binh Nguyen, and Minh-Duc Le, and Thi-Hanh Nguyen. 2017. USL: Towards Precise Specification of Use Cases for Model-Driven Development. In *SoICT '17: Eighth International Symposium on Information and Communication Technology, December 7–8, 2017, Nha Trang City, Viet Nam.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3155133.3155194

*Also with Faculty of Information Technology,
Hung Yen University of Technology and Education
My Hao, Hung Yen,Vietnam.
†The author is a visiting professor, Hosei University, Japan

## 1 INTRODUCTION

Use cases have achieved wide use for capturing and structuring software requirements. A use case is typically represented as a combination between an informal UML use case diagram and loosely structured textual descriptions [15]. Such a use case specification is quite convenient for the users as they could express requirements in their own language. However, use cases expressed in the current form often contain ambiguous and imprecise parts. This prevents integrating them into model-driven approaches, where they are often taken as the source of model transformations.

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" [9]. Many research [23] have been attempted to introduce rigor into use case descriptions. In [24], the authors proposed adding keywords and restriction rules into use case descriptions and then using natural language processing techniques in order to specify and analyze use cases. The works in [7, 14] proposed a formal semantics for use cases. The works in [1, 12, 21, 22] proposed using UML activity and sequence diagrams in order to represent the control structures of use cases. Several other works [13, 18, 20] proposed defining Domain Specific Languages (DSLs) in order to specify use cases. However, as mentioned in [23], it still lacks a method to precisely capture the relevant information of use cases including control flows, steps, system actions, actor actions, and constraints on the use case and its flows. In addition, the use case specification must be precise enough for transformations as well as understandable for non-technique stakeholders.

In this paper, we introduce a DSL named the Use case Specification Language (USL) to precisely specify use cases. Here, as explained in [8] we refer to a DSL as a language designed to be useful for a specific set of task domains. The set of tasks w.r.t the USL is to build use case models to represent the system behaviour. To define the abstract syntax of USL we extend the meta models of UML use case and activity diagrams. The new meta-concepts are defined for the following purposes: (1) to describe the elements for a typical use-case-description template; (2) to represent the `basic` and `alternate flows` of a use case in form of sequential, branched, or repeating steps; (3) to categorize `use case steps` and `actions` based on the interactive subjects including the `system, actors`, and included/extending use cases; and (4) to represent constraints on the use case, actions and flows. The main contributions of our work are as follows:

- To propose a DSL to precisely specify use cases. The USL language has a concrete syntax in graphical form that allows us to achieve the usability goal.
- To map the USL to a `Labelled Transition System (LTS)` for a precise semantics. Use cases in USL are now precise enough for transformations to obtain other artifacts such as test cases and analysis class models.

The rest of this paper is organized as follows. Section 2 presents an example for our work. Section 3 overviews our approach and then explains the abstract syntax and formal semantics of the USL. Section 4 introduces our support tool and discusses the potential usability of the USL. Section 5 comments on related works. The paper is closed with conclusions and future work.

## 2 A RUNNING EXAMPLE

According to Pohl [17], software-intensive systems are divided into information systems and embedded systems. In our research, we only focus on the use case description of information systems.

Figure 1 shows a simplified requirements model of a Library system including a use case model depicted in the part (A) and a corresponding conceptual domain presented in the part (B). Here, the main use case `Lend book` describes a book-loan function. The use case is represented in a typical template as illustrated in Table 1.

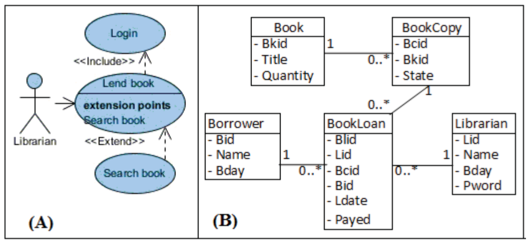


**Figure 1: The simplified use case and conceptual domain model of the `Library system`.**

A typical use case description template [3] often includes two parts, the overview information elements of use case and the detailed description of use case flows. The first part consists of the following elements: the `use case name`, the `brief description` of use case, the `actors` participating in the use case, the `precondition` and `postcondition` of the use case, and the `trigger` that initiates the use case. The second part contains two types of event flows, the `basic flow` and `alternative flows`. The basic flow covers what normally happens when the use case is performed. 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 [9, 10]. Each step consists of actions performed either by the system or actors. We refer to actors, the system, and other relation use cases as `interactive subjects` of a use case. For example, Step 1 of 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 of concurrent actions. As illustrated in Table 1, Step 3 includes two system actions, "The system shows the Lend book window" and "The system assigns the current time to the date lending book". Step 5 contains a branching decision, "If it is invalid, the system goes to step 5a". Step 10 contains the starting point of two concurrent actions: "The system executes two step 11 and 12 concurrently".

**Table 1: A typical use-case-description template**

| |
|---|
| **Use case name:** Lend book<br>**Brief description:** The Librarian processes a book loan.<br>**Actors:** Librarian.<br>**Precondition:** There is no constraints to start the use case.<br>**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.<br>**Trigger:** The Librarian requests a book-loan process.<br>**Special requirement:** There is no special requirement. |
| **Basic flow** |
| 1. The Librarian selects the Lend Book function.<br>2. The system gets the Librarian id. If it is null, it goes to step 2a.<br>3. The system shows the lend-book window and sets the book-loan date.<br>4. The Librarian enters a book copy id.<br>5. The system checks the book copy id. If it is invalid, it goes to step 5a.<br>6. The Librarian enters a borrower id.<br>7. The system validates the borrower id. If it is invalid, it goes to step 7a.<br>8. The Librarian clicks the save-book-loan button.<br>9. The system validates the conditions to lend book. If it is invalid, the system goes to step 9a.<br>10. The system saves the book loan record, then executing two steps 11 and 12 concurrently.<br>11. The system shows a complete message.<br>12. The system prints the borrowing bill. |
| **Alternate flows** |
| E1. request searched book<br>1. The Librarian clicks the search-book button after step 5a.1.<br>2. The system executes the extending use case Search book.<br>2a. The Librarian does not Login<br>1. The system executes the included use case Login, if it is successful, it goes to step 3.<br>2. The system shows an error message.<br>3. The use case ends.<br>5a. The book copy id is invalid<br>1. The system shows an error message, then it goes to step 4.<br>7a. The Borrower id is invalid<br>1. The system shows an error message, then it goes to step 6.<br>9a. The lending condition is invalid<br>1. The system shows an error message.<br>2. The system ends the use case. |

Apart from the seven types of use case actions proposed in [20, 24], we introduce two other types of actions: one is for extending and the other is for including another use case into a given use case.

**Actor-Input** is an actor action to enter data, e.g., the action "The Librarian enters a book copy id" at Step 4 in Table 1 is an `Actor-Input`.

**Actor-Request** is an actor action to send requests, e.g., the action "The Librarian clicks the save-book-loan button" at Step 8 in Table 1 is an `Actor-Request`.

**System-Display** is a system action to send outputs to the actors, e.g., the action "The system shows the lend-book window" at Step 3 in Table 1 is a `System-Display`.

**System-Input** is a system action to validate or update input data, e.g., the action "The system sets the book-loan date" at Step 3 in Table 1 is a `System-Input`.

**System-State** is a system action to query or update current system state, e.g., the action "The system saves the book loan record" at Step 10 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 5a in Table 1 is a `System-Output`.

**System-Request** is a system action to send a request to the actors, e.g., the action "The system prints the borrowing bill" at Step 12 shown in Table 1 is a System-Request.

**System-Include** is a system action to include another use case e.g., the action "The system executes the included use case Login" at Step 1 of the alternate flow 2a in Table 1 is a System-Include.

**System-Extend** is a system action to extend with 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 its are satisfied. Furthermore, the pre and postcondition of the actions of current flow need to be fulfilled also.

## 3 USE CASE SPECIFICATION IN USL

This section first overviews our approach. The basic idea is to propose a domain specific language named USL to specify use cases. Next, the section presents the abstract syntax of the USL and then provides a formal semantics for it.

### 3.1 Overview of the USL Approach

Figure 2 illustrates our approach. First, we take as input a set of use case diagrams, the textual descriptions of some use cases in these diagrams and a class diagram presenting the domain concepts. 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 (with regard to) 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.

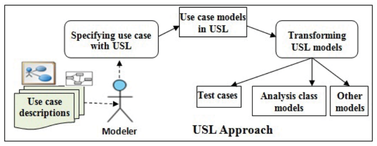


**Figure 2: Overview of the USL Approach.**

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 use case. The constraints are often expressed using constraint languages such as the OCL [6], JML [19], and event natural language as mentioned in [20]. In this research, we employ the OCL in order to present 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 the USL so that we could transform USL models in to other artifacts such as test cases and analysis class models.

**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. |

### 3.2 The Abstract Syntax of USL

We define the USL metamodel w.r.t the use-case domain based on (1) UML use case specification (Chapt. 18 of [15]), (2) the Use Case Descriptions (UCDs) [3, 9, 10] and (3) the UML activity specification (Chapt.s 15, 16 of [15]). 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 model. 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 use case 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 [15]). Action is specialized into two main types (as illlustrated in Fig. 3-D): ActorAction and SystemAction. ActorAction is further specialized into ActorRequest and ActorInput. SystemAction is specialized into SystemInput, 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 (order number of step), description (the content of the step) and maxloop (the

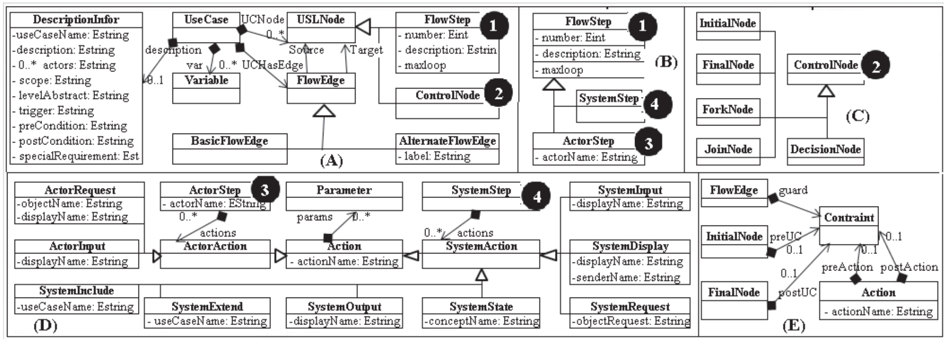


**Figure 3: The USL metamodel.**

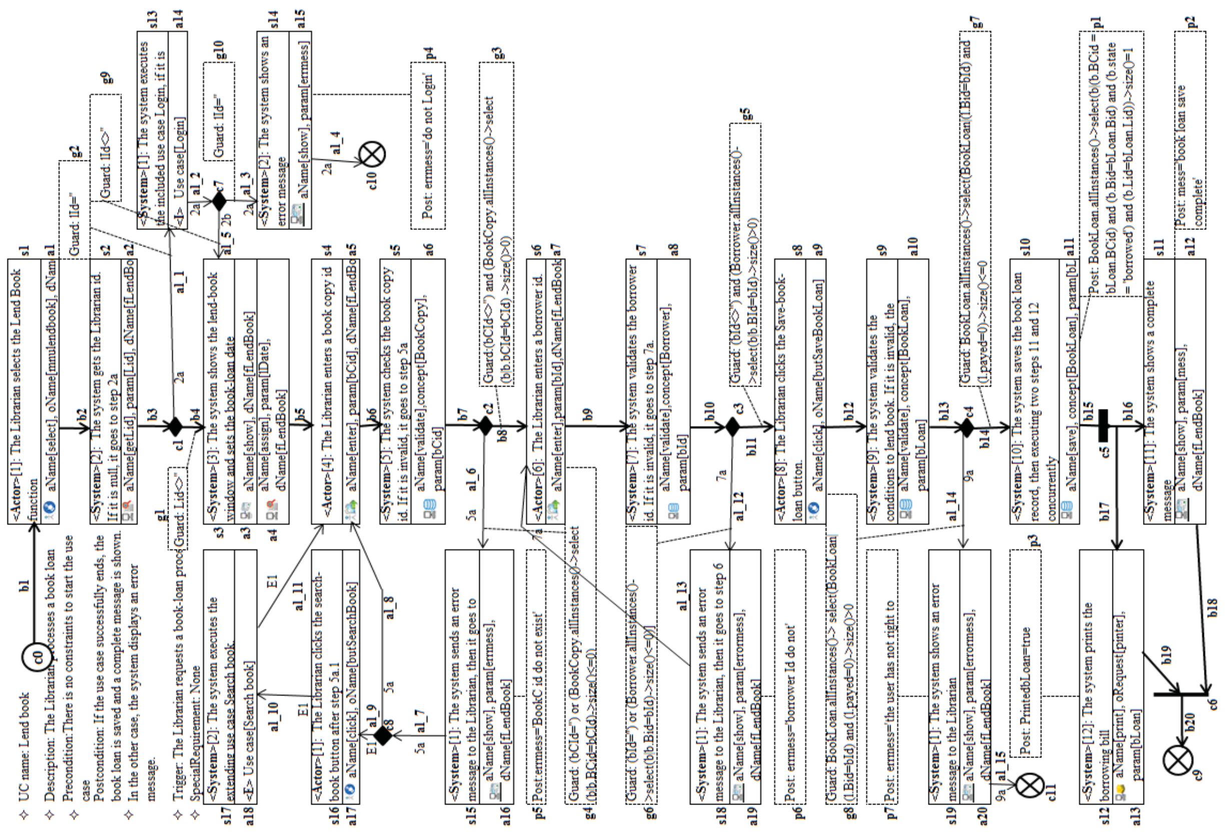


**Figure 4: The USL model of Lend book use case.**

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 3.2.1.** The USL model shown in Fig. 4 consists of the FlowSteps s1, . . . , s19. Among these, the s1 is an ActorStep, and s2 is a SystemStep. Step s4 contains the ActorInput a5. Step s1 contains the ActorRequest a1. Step s2 contains the SystemInput a2. Step s11 contains the SystemOutput a12. Step s5 contains the SystemState a6. Step s12 contains the SystemRequest a13. Step s13 contains the SystemInclude a14. Step s16 contains the SystemExtend a17. Action a5 has a the 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 3.2.2.** The USL model as shown in Fig. 4 contains twelve `ControlNode`s, the c0, ..., c11. In particular the c0 is an `InitialNode`, the c9, c10 and c11 are different `FinalNode`s, the c1, ..., c4, c7, and c8 are `DecisionNode`s, the c5 is a `ForkNode`, and the c6 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 `USLNode`s. If both steps are part of a basic flow, we call the transition a `BasicFlowEdge`. On the other hand, if both steps are 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 3.2.3.** The USL model as shown in Fig. 4 contains the b1, ..., b20 as `BasicFlowEdge`s and the al_1, ..., al_15 as `AlternateFlowEdge`s.

**Variable** (domain source (3)) represents variables that 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 [15].

**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 `FinalNode`s; (3) guard conditions of a transition; and (4) pre and postcondition of an `Action`. This concept is inherited the concept `Constraint` in UML, shown in Sect. 7.6 of [15]. 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 3.2.4.** The USL model as shown in Fig. 4 contains the guard conditions g1, ..., g10 and the postcondition of `Actions` p1, ..., p7.

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, which are 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 present the underlying domain;
- $A$ is the set of `USLNode`s;
- $E$ is the set of `FlowEdge`s;
- $C = G \cup C_{preUC} \cup C_{postUC} \cup C_{preA} \cup C_{postA}$ 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
  $E_b = \{e \mid e \in E, \text{BasicFlowEdge}(e)\}$,
  $E_a = \{e \mid e \in E, \text{AlternateFlowEdge}(e)\}$.

**Example 3.2.5.** The USL model as shown in Fig. 4 contains the following elements: $N_I = \{c0\}$; $N_F = \{c9, c10, c11\}$; $A_{cNode} = \{c0, \ \ldots, \ c11\}$; $A_a = \{s1, s4, s6, s8, s16\}$; $A_s = \{s2, s3, s5, s7, s9, \ \ldots, \ s15, s17, s18, s19\}$; $E_b = \{b1, \ \ldots, \ b20\}$; $E_a = \{al\_1, \ \ldots, \ al\_15\}$; $G = \{g1, \ \ldots, \ g10\}$; $C_{preUC} = \emptyset$; $C_{postUC} = \emptyset$; $C_{preA} = \emptyset$; and $C_{postA} = \{p1, \ \ldots, \ p7\}$. $D_C$ corresponds to the conceptual model shown in the part (B) of Fig. 1. There are 16 constraints for guard conditions and pre and postcondition, e.g., the postcondition $p1$ of $a11$ is expressed by the following OCL contraint [16]:
```
BookLoan.allInstances()->select(b|(b.BCid= bLoan.BCid)  and
(b.Bid=bLoan.Bid)        and        (b.payed='0')       and
(b.Lid=bLoan.Lid))->size()=1.
```

## 3.3 Formal Semantics of USL

We use labelled transition system (LTS) [11] 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 Table 3. Definition 2 formalizes the notion of the LTS of the USL model.

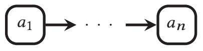
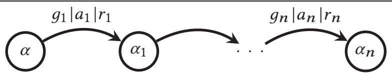
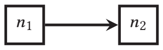
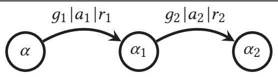
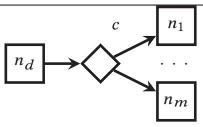
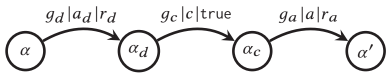
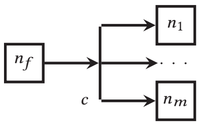
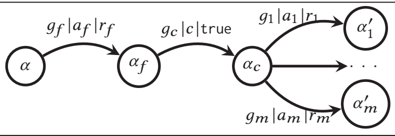
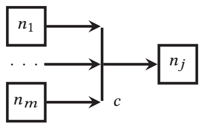
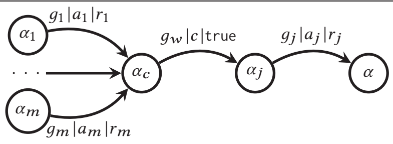
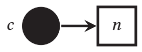
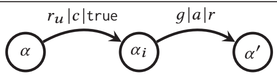
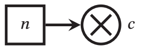
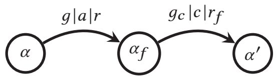
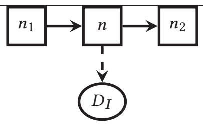
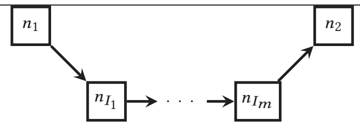
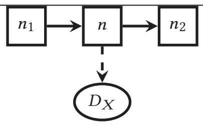
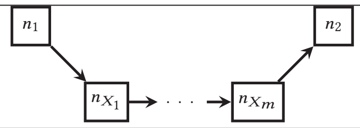
**Definition 2.** Given a USL model $D = \langle D_C, A, E, C\rangle$, an `LTS` that results from the execution of $D$ is the tuple $\langle \Sigma(\mathcal{V}), \mathbb{P}(\mathcal{G} \times \mathcal{A} \times \mathcal{P}), \mathcal{T}, \alpha_{init}, \mathcal{F}\rangle$ such that:

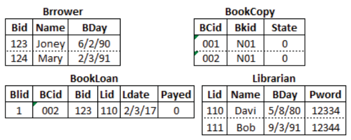
- $\mathcal{V}$ is a finite set of variables whose types include the basic types and the classes of the $D_C$;
- $\Sigma(\mathcal{V})$ is the set of states ($\alpha$), each of which is a set of value assignments to a subset of variables in $\mathcal{V}$;
- $\mathcal{P} \subseteq C_{postA} \cup C_{postUC}$ 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 $\alpha'$ 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 3).

**Table 3: LTS-based semantics of the basic USL concepts**

| USL concepts | Notation | LTS-based semantics | |
|---|---|---|---|
| Step |  |  | where $g_i$ = preA($a_i$), $r_i$ = postA($a_i$) ($\forall i = 2, \ldots, n$). |
| Flow edge |  |  | where $a_2$ = firstAct($n_2$), $g_2$ = guardE($n1$, $n2$) $\wedge$ preA($a_2$) $r_2$ = postA($a_2$). |
| Decision node |  |  | where $a_d$ = lastAct($n_d$); $g_c$ = guardE($n_d$, $c$), $a$ = firstAct($n$), $g_a$ = guardE($c$, $n$) $\wedge$ preA($a$) s.t $n \in \{n_1, \ldots, n_m\}$. |
| Fork node |  |  | where $a_f$ = lastAct($n_f$); $g_c$ = guardE($n_f$, $c$), $a_i$ = firstAct($n_i$), $g_i$ = preA($a_i$), $r_i$ = postA($a_i$) ($\forall i = 1, \ldots, m$). |
| Join node |  |  | where $a_i$ = lastAct($n_i$), ($\forall i = 1, \ldots, m$); $g_w = \left(\bigwedge_{(e \in D.E, \text{target}(e)=c)}\right.$ isCompleted($e$) $\wedge$ guardE($e$)$\left.\right)$; $a_j$ = firstAct($n_j$), $g_j$ = guardE($c$, $n_j$) $\wedge$ preA($a_j$), $r_j$ = postA($a_j$). |
| Initial node |  |  | where $\alpha = \alpha_{init}$, $r_u$ = preC($D$); $a$ = firstAct($n$), $g$ = guardE($c$, $n$) $\wedge$ preA($a$), $r$ = postA($a$). |
| Flow final node |  |  | where $\alpha' \in \mathcal{F}$, $r_f$ = postC($D$, $c$), $g_c$ = guardE(n, c); $a$ = lastAct($n$), |
| USL model with include action |  | $\equiv$  | where $n \in A_s$, \|actions($n$)\| = 1, SystemInclude($a$) ($a \in$ actions($n$)), $t = (\alpha, (g_a\|a\|r_a), \alpha')$; $n_{I_1}, \ldots, n_{I_m} \in D_I.A$, $g_a$ = guardE($n1$, $n$) $\wedge$ preA($a$) $\wedge$ preC($D_I$), $r_a$ = postC($D_I$) |
| USL model with extend action |  | $\equiv$  | where $n \in A_s$, \|actions($n$)\| = 1, SystemExtend($a$) ($a \in$ actions($n$)), $t = (\alpha, (g_a\|a\|r_a), \alpha')$; $n_{X_1}, \ldots, n_{X_m} \in D_X.A$, $g_a$ = preA($a$) $\wedge$ preC($D_X$), $g_a$ = guardE($n1$, $n$) $\wedge$ preA($a$) $\wedge$ preC($D_X$), $r_a$ = postC($D_X$) |
| **Legend** | | $a$ a action in a step; $s$ a step; $D_u$ use case; $\alpha$ a state | |



Brrower

| Bid | Name | BDay |
|---|---|---|
| 123 | Joney | 6/2/90 |
| 124 | Mary | 2/3/91 |

BookCopy

| BCid | Bkid | State |
|---|---|---|
| 001 | N01 | 0 |
| 002 | N01 | 0 |

BookLoan

| Blid | 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 w.r.t the Lend book use case.**

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

**Example 3.3.1.** We assume that the snapshot shown in Fig. 5 is captured when the USL model shown in Fig. 4 is executed at Step a9. 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 $\alpha_{a9} = \{(bCId, \text{“001”}), (lDate, \text{“001”}), (lId, \text{“110”}), (bId, \text{“124”}), (bLoan, (\text{“2”}, \text{“001”}, \text{“124”}, \text{“110”}, \text{“25/8/17”}, 0))$, 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 $\alpha$ 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:

- preT($t$) = $\alpha$, postT($t$) = $\alpha'$, guard($t$) = $g$, postC($t$) = $r$, and act($t$) = $a$.
- eval($g$) is the evaluation of Constraint $g$.

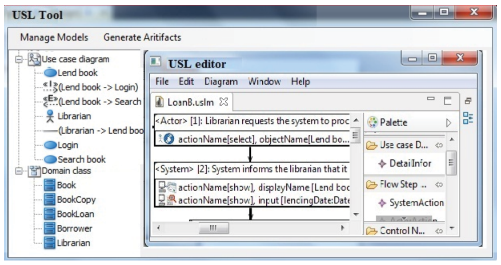


**Figure 6: USL tool.**

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

**Example 3.3.2.** When the USL model shown in Fig. 4 executes at Step $a11$, 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” $\}$.

The transition $t_{a11,c5} = \alpha_{a11} \xrightarrow{true|c5|true} \alpha_{c5}$. $\mathtt{reachable}(\alpha_{a11}) = \{t_{a11,c5}\}$ and $\mathtt{firable}(\alpha_{a11}) = \{t_{a11,c5}\}$.

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

**Example 3.3.3.** When the USL model shown in Fig. 4 executes at Step $c5$, we have two transitions $t_{c5,a12} = \alpha_{c5} \xrightarrow{true|a12|p2} \alpha_{a12}$ and $t_{c5,a13} = \alpha_{c5} \xrightarrow{true|a13|p3} \alpha_{a13}$, $\mathtt{reachable}(\alpha_{c5}) = \{t_{c5,a12}, t_{c5,a13}\}$ and $\mathtt{firable}(\alpha_{c5}) = \{t_{c5,a12}, t_{c5,a13}\}$. Hence, $\{t_{c5,12}, t_{c5,13}\}$ is a concurrent transition, and $\alpha_{a12}$, $\alpha_{a13}$ 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 $\mathtt{use\ \ case\ \ scenario}$ 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 3.3.4.** When the USL model shown in Fig.4 executes at Step $\alpha_{a11}$ as mentioned above, and the $\mathtt{eval}$(g1), $\mathtt{eval}$(g3), $\mathtt{eval}$(g5), and $\mathtt{eval}$(g7) are true, then the use case scenario is as follows:

$$p = \alpha_{init} \xrightarrow{true|a1|true} \alpha_{a1} \xrightarrow{true|a2|true} \alpha_{a2} \xrightarrow{true|c1|true} \alpha_{c1} \xrightarrow{g1|a3|true} \alpha_{a3} \xrightarrow{true|a4|true} \alpha_{a4} \xrightarrow{true|a5|true} \alpha_{a5} \xrightarrow{true|a6|true} \alpha_{a6} \xrightarrow{true|c2|true} \alpha_{c2} \xrightarrow{g3|a7|true} \alpha_{a7} \xrightarrow{true|a8|true} \alpha_{a8} \xrightarrow{true|c3|true} \alpha_{c3} \xrightarrow{g5|a9|true} \alpha_{a9} \xrightarrow{true|a10|true} \alpha_{a10} \xrightarrow{g7|a11|true} \alpha_{a11} \xrightarrow{true|c5|true} \alpha_{c5} \xrightarrow{\{true|a12|p2, true|a13|p3\}} \alpha_{a12-a13} \xrightarrow{true|c6|true} \alpha_{c6} \xrightarrow{true|c9|true} \alpha_{c9}$$

$(\alpha_{c9} \in \mathcal{F})$.

## 4 TOOL SUPPORT AND DISCUSSIONS

We realize our USL approach with a support tool illustrated in Fig. 6. The main functionality of the tool is to take as input UML models and to offer means for the modeler to produce a corresponding USL model.

The left part of Fig. 6 illustrates the function loading as input a UML use case diagram and a class diagram for the conceptual domain. These diagrams are then displayed on a tree structure. The right part of Fig. 6 corresponds to the function of USL Editor to create USL models. The function is realized using the EMF project and the GMF project within the Eclipse tool. The former helps us build the abstract syntax of USL, while the latter supports the concrete syntax together with constraint rules expressed in OCL expressions. The left part of the USL Editor is used to draw USL models whereas the right part contains a palette containing notations. The USL editor allows creating a USL model by dragging and dropping the notations on the palette. The loaded elements in the first part will help the created USL models in the second part achieving consistences with the built elements into the UML models.

**Discussion** The USL approach allows us to obtain the following benefits:

- *Consistency:* With USL we can specify use cases for different stakeholders in a whole model. Hence, the USL model makes a consistency view for different stakeholders about functional requirements of the system.
- *Usability:* The USL concepts correspond to the concepts of the use case description domain. Therefore, non-technical stakeholders an apply their own language when creating USL models.
- *Specification ability:* USL models are used not only to document functional requirements of the system but also to manipulate them. The precise semantics of USL, defined by mapping the USL to a Labelled Transition System (LTS), opens a possibility for transformations from USL models to other artifacts such as test cases and analysis class models. The artifact generating transformations from USL models will be part of our future works. Besides, use case descriptions may contain constraints on the real time elements. The ability presenting such constraints depend on choosing a suitable constraint language. Within our approach, we use the OCL to present constraints.
- *Tool support:* The graphical concrete syntax of the USL allows creating models visually. The graphical syntax of USL language might be not as flexible as a textual syntax. Thus, it might be necessary to extend the USL editor in order to specify USL models in a textual form.

## 5 RELATED WORKS

We position our work in the intersection between use case-driven development [9] and model-driven development [2]. 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 [13, 14, 18, 20, 24] have been proposed.

Tao Yue *et al* . [24] proposed a use case modeling language called Restricted Use Case Modeling (RUCM), which is composed of a use case template 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.

Murali *et al.* [14] 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 description of a use case are still informal.

Misbhauddin *et al.* [13] extended the meta-model of UML use case models in order to capture both the structural and behavioral aspects of use cases. In order to specify a use case, they developed a prototype tool called UCDest. However, concurrent actions, pre and postcondition of actions have not been mention. In addition, action types are defined inadequately.

Savic *et al.* [18] and Smialek *et al.* [20] 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. They also lack a formal semantics.

Our previous work in [4, 5] 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 behavioral aspect. Comparing to the current works in literature, the 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 the USL to specify dynamic information when use case scenarios execute. The result of that, USL models are precise source models for transformations generating other artifacts in MDD.

## 6 CONCLUSIONS AND FUTURE WORK

This paper proposed a DSL named USL for use case specification. The USL allows specifying more complete and precise use cases. A USL model can cover the relevant information of a use case including flows, steps, system actions, actor actions, relationships, control flows, and constraints on the use case. We built the abstract syntax of the USL and a modeling tool to create the USL models. We also defined a formal semantics for USL by mapping to an LTS. Therefore, the USL model can be transformed into other artifacts such as test cases and analysis class models.

In the future work, we focus on realizing transformations from USL models in order to generate test cases as well as other model artifacts automatically.

## ACKNOWLEDGMENTS

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

## REFERENCES

[1] Jesús M. Almendros-Jiménez and Luis Iribarne. 2005. Describing Use Cases with Activity Charts. In *Proc. 2004th Int. Conf. on Metainformatics (MIS'04)*. Springer-Verlag, Berlin, Heidelberg, 141–159.
[2] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice* (1st ed.). Morgan & Claypool Publishers.
[3] Alistair Cockburn. 2000. *Writing Effective Use Cases* (1 edition ed.). Addison-Wesley Professional, Boston.
[4] Duc-Hanh Dang. 2008. Triple Graph Grammars and OCL for Validating System Behavior. In *Proc. 4th Int. Conf. Graph Transformations (ICGT)*, Vol. LNCS 5214. Springer, 481–483.
[5] Duc-Hanh Dang, Anh-Hoang Truong, and Martin Gogolla. 2010. Checking the Conformance between Models Based on Scenario Synchronization. *Journal of Universal Computer Science* 16, 17 (2010), 2293–2312.
[6] Martin Giese and Rogardt Heldal. 2004. From informal to formal specifications in UML. In *Proc. of UML2004, Lisbon, volume 3273 of LNCS.* Springer, 197–211.
[7] Wolfgang Grieskamp and Markus Lepper. 2000. Using use cases in Executable Z. In *ICFEM 2000. Third IEEE International Conf. on Formal Engineering Methods.* IEEE, York, England, 111–119.
[8] Richard C. Gronback. 2009. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit* (1 edition ed.). Addison-Wesley Professional, Boston.
[9] Ivar Jacobson. 2004. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley Longman Publishing Co., Inc.
[10] Ivar Jacobson, Ian Spence, and Kurt Bittner. 2011. *USE-CASE 2.0 The Guide to Succeeding with Use Cases.* Ivar Jacobson International SA.
[11] Robert M. Keller. 1976. Formal Verification of Parallel Programs. *Commun. ACM* 19, 7 (July 1976), 371–384. https://doi.org/10.1145/360248.360251
[12] Liwu Li. 2000. Translating Use Cases to Sequence Diagrams. In *Proc. 15th IEEE Int. Conf. on Automated Software Engineering (ASE '00).* IEEE Computer Society, Washington, DC, USA, 293–298.
[13] Mohammed Misbhauddin and Mohammad Alshayeb. 2015. Extending the UML use case metamodel with behavioral information to facilitate model analysis and interchange. *Software & Systems Modeling* 14, 2 (May 2015), 813–838.
[14] Rajiv Murali, Andrew Ireland, and Gudmund Grov. 2016. UC-B: Use Case Modelling with Event-B. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z (LNCS),* Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro (Eds.). Springer International Publishing, Switzerland, 297–302.
[15] OMG. 2005. UML 2.5. (May 2005). http://www.omg.org/spec/UML/2.5/
[16] OMG. 2006. OCL 2.0. (May 2006). http://www.omg.org/spec/OCL/2.0/
[17] Klaus Pohl. 2010. *Requirements Engineering - Fundamentals, Principles, and | Klaus Pohl | Springer.* Springer-Verlag Berlin Heidelberg.
[18] Dušan Savić, Siniša Vlajić, Saša Lazarević, Ilija Antović, Vojislav Stanojević, Miloš Milić, and Alberto Rodrigues da Silva. 2016. Use Case Specification Using the SILABREQ Domain Specific Language. *Computing and Informatics* 34, 4 (Feb. 2016), 877–910.
[19] Peter Schmitt, Isabel Tonin, Claus Wonnemann, Eric Jenn, Stéphane Leriche, and James J. Hunt. 2006. A Case Study of Specification and Verification Using JML in an Avionics Application. In *Proc. 4th Int. Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '06).* ACM, New York, NY, USA, 107–116.
[20] Michal Smialek and Wiktor Nowakowski. 2015. *From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice.* Springer, Switzerland.
[21] Jitendra Singh Thakur and Atul Gupta. 2014. Automatic Generation of Sequence Diagram from Use Case Specification. In *Proc. 7th India Software Engineering Conf. (ISEC '14).* ACM, New York, NY, USA, 20:1–20:6.
[22] Saurabh Tiwari and Atul Gupta. 2015. An Approach of Generating Test Requirements for Agile Software Development. In *Proc. 8th Conf. on India Software Engineering (ISEC '15).* ACM, New York, NY, USA, 186–195.
[23] Saurabh Tiwari and Atul Gupta. 2015. A Systematic Literature Review of Use Case Specifications Research. *Inf. Softw. Technol.* 67, C (Nov. 2015), 128–158.
[24] Tao Yue, Lionel C. Briand, and Yvan Labiche. 2013. Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments. *ACM Trans. Softw. Eng. Methodol.* 22, 1 (March 2013), 5:1–5:38.