DOI: 10.15625/vap.2016.0001

A METHOD TO SPECIFY SOFTWARE FUNCTIONAL REQUIREMENTS FOR SYSTEM TEST CASE GENERATION

Chu Thi Minh Hue[1, 2], Nguyen Ngoc Binh[2], Dang Duc Hanh[2]

[1] Hung Yen University of Technology and Education

[2] The VNU University of Engineering and Technology

huectm@gmail.com, hanhdd@vnu.edu.vn, nnbinh@vnu.edu.vn

ABSTRACT *— In this paper, we propose the next version of Use-case Specification Language (USL) that allows us to formally specify each use case at the user's requirement level with a single model, and without using the detailed use case design. We also propose a formal specification method for the functional requirements. The formal requirement specification can be used to automatically generate system level test cases. Version 2.0 of the USL is defined by extending the activity diagram with a Contract concept. This concept enables a more detailed specification of information when performing an action or a transition. The USL specifies the followings: sequences of actions in flows of events, input and output parameters when performing use case, precondition and post-condition of use case, satisfy conditions on input and output parameters when performing an action or a transition, <<include>> and <<extend>> relations with other use cases, maximum number of repeated times of an action (if any), and change effect in value of input parameters when performing an action or a transition. Our key contributions include a MetaModel (which describes an abstract syntax), a concrete syntax for USL, a concrete textual syntax via BNF, and a formal semantic definition of USL in general.*

Keyword *— Use case, functional requirement, USL, formal specification.*

I. INTRODUCTION

Software testing is an essential activity to ensure software quality in the software development process. Nowadays, the size and complexity of software is increasing, leading to the increase in the cost of time and resource needed for the testing activities. Thus, it is necessary to improve test automation techniques. The software testing process can be divided into three main phases: test case design, test execution and test evaluation. In practice, test case design is the most difficult to automatic. In functional testing, test case design is usually performed manually based on the software requirement specification document. Functional requirements are often specified by the use case diagram and the description of each use case is written in natural language in a given format. Therefore, the automatic generation of test cases from the use case specification is still a huge challenge for the research community.

In order to automatically generate the test cases from the software functional requirements, we need to transfer the use case description in the form of natural language into a formal specification. This specification can be used to automatically identify different test cases when performing a use case. Test case information includes input parameters, pre-conditions of test case performance, expected output, and test steps.

There have been many research focusing on the problem of how to formally specify use case towards automatically generating test scenarios and test cases. For example, the authors of [2] used a markup language structure to specify use case scenarios of a use case. This method focuses on the specification of the event flows of the use case. However, it does not specify other information of the use case.

The authors of [3] proposed a contract to specify pre and post conditions when performing the use case. The sequence diagram is used to describe use case scenarios. However, details of the method for determining the constraints were not mentioned in the [3].

The authors of [4] specified each use case by a structured natural language. The scope of specification includes pre- and post-conditions of the use cases and its set of interactive sequences that are described in the basic flow and the alternate flows. Each interaction is specified by a sentence of the structured Subject Verb Object (SVO). The use case scenarios are represented by activity or sequence diagram. The main drawback of this method is that a use case description requires 2 structures: the one is in natural language and the other is an activity or sequence diagram. This method is does not consider the relationship among the use cases.

The authors of [5] specified use case in a language named Restricted Use Case Modeling (RUCM). RUCM is defined based on a template containing a set of restriction rules in the natural language. This language defines a set of keywords for describing the flows of events. The drawbacks of the method include the use of natural language processing (NLP) to identify use case information and incomplete specification of all the relationships among event flows.

Other approaches of [6] - [9] used activity diagram to specify the use case behaviors. These researches focused on specifying the action sequences and guard conditions on the transitions. The authors of [10] used activity diagram to describe use case information, such as action sequences and transitions, guard conditions on transitions, and <<*include*>> relations. However, this method does not specify other information and it also does not separately specify

actions. The authors of [11] used activity diagram to describe use case. This research defines a method for specifying sequence interactions in the basic flows and alternate flows, and <<include>> and <<extend>> relations. However, this method does not specify information about pre- and post-conditions, constraints on the input and output parameters when executing an action or a transition.

However, none of the above researches clearly specifies the input and output parameters when executing a use case. Some of these researches do not consider these parameters, while others, such as [5], require using a complex extraction to extract the parameters from the OCL expressions. Further, there are no researches that propose to specify exactly the allowed maximum repeated times of an action. There are also no research, which separately specifies actor actions, the system actions, and actions that are performed outside of the system. Very few researches fully specify the <<include>> and <<extend>> relations among the use cases.

In this paper, we expand the Use case Specification Language (USL) of [12] and [13] as version 1.0 to version 2.0 to describe use cases. We also define a formal specification method for software functional requirements towards the automatical generation of system level test cases. In our method, software functional requirements are formally specified by use case diagram and each use case is specified by a USL diagram. A use case diagram in USL describes information more fully, which includes pre- and post-conditions, actor actions, system actions, actions that are performed outside of the system, input and output parameters when performing use case, satisfy constraints on the input and output parameters when performing an action or a transition, the numbers of maximum repeated times of an action, <<*include*>> and <<*extend*>> relations among the use cases. To define USL, we build a meta-model and define semantics for the language.

The idea of USL was first proposed in [12] as version 1.0. The concrete textual syntax of the USL is described via BNF in [13]. In these works, we proposed methods and algorithms that automatically generate system level test cases from USL diagram. In this paper, we present an expansion of the USL and the formalization of the language. More specifically, we specify more actions that are performed outside of the system, and <<*include*>> and <<*extend*>> relations among the use cases. In addition, we add some attributes into the *Contract* concept to specify input and output parameters of use case, the maximum repeated times of an action, and effect on input parameters when executing an action or a transition.

This paper is organized as follows: Section 2 introduces the motivation for developing USL. Section 3, we present syntax and semantics of our USL language. Section 4 focuses on the methods to use USL language for specifying functional requirements. We conclude the paper in Section 5.

II. MOTIVATION

In this paper, we use 3 use cases of the ATM system that are described in [15]. These use case are *Insert card, Withdraw, Help* to motivate and illustrate USL. Use case *Insert card* allows a user to enter a PIN code when inserting a bank card into the ATM machine. Use case *Withdraw* allows the user to withdraw money from the ATM machine. Use case *Help* allows the user to request the display of help information when performing a transition. Use case *Insert card* is included in the use case *Withdraw*. Use case *Help* is extended from the use case *Withdraw*. Figure 1 is a use case model of the 3 use cases above. Details of each use case are described in Table 1.

As show in Figure 1, a case diagram shows information about actors, use cases and relations. To reduce complication and increase their reuse, the use cases are reorganized using relations [16] as <<*include*>>, <<*extend*>>, <<*inheritance*>> relations…. In this research, we focus on the <<*include*>> and <<*extend*>> relations.

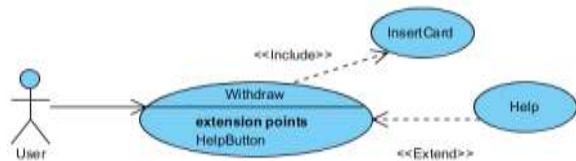

Figure 1. Use case diagram of Withdraw, Insert card, and Help

An <<include>> relation is considered to be a part of the basic flow. An extending use case can be triggered at an extension point or in any actions when a trigger condition is satisfied. The <<*extend*>> relation is considered to be a part of an alternate flow.

Table 1. *Use case description of Withdraw, Insert card, and Help*

Use Case UC1: Withdraw	**Alternate flows**
Primary Actor: User	3a Show Help Button
Pre-Condition : Insert card use case was successful	1. The actor press Help button, then the system calls the extending use case E1.
Post-Condition: If the use case was successful the system updates the balance, dispenses the cash, prints a receipt for the user. If not, the system displays an error message	5a. The actor enters an invalid amount 1. S: The system displays an error message “Amount is invalid, enter a valid amount” , return 3
Trigger: User clicks menu Withdraw	6a. The balance isn't sufficient
Basic flow	1. S: The system checks whether the user has

1. A: The user-actor Inserts Card into ATM (include) 2. A: The actor click menu Withdraw 3. S: System shows Withdraw screen 4. A: The actor enters the amount she/he wants to withdraw 5. S: The system checks whether the value entered is valid 6. S: The system will retrieve the balance of the user then check the balance is sufficient to withdraw 7. S: The system updates the balance, dispenses the cash and finally prints a receipt for the user 8. The user receive money and card	*overdraft* permission. If has go next, if don't has go to 3b 2. S. The system compares the amount required with the maximum allowed limit for overdraft, if the amount needed is within limit, go to 5, if not go next to 3 3. S. the system displays an error message "Amount is beyond Limit" 6b. the user has no overdraft permission 4. S: an error message is displayed " balance is not enough, No permission granted" **Extension Points** E1. Help about Withdraw: The extension point occurs at step 2 of the basic flow if user presses Help button
Use Case UC2: ATM *Insert Card* **Primary Actor:** User **Pre-condition :** ATM System is ready **Pos-condition:** payment is successful, credit card is updated. **Trigger** **:** User Insert Card into ATM	**Alternate flows** 2a. Card not valid 1. S: Display message and reject card 4a. PIN **n**ot valid 1. S: Display message and ask for retry (twice) 4b. PIN invalid 3 times 1. S: Eat card and exit **Use Case UC2:** Help **Primary Actor:** User
Basic Flow	**Pos-condition:** Return before screen
1. A: Insert card 2. S: Validates card and asks for PIN 3. A: Enters PIN 4. S: Validates PIN 5. S: Allows access to account	**Basic Flow** 1. S: System Show Help screen 2. A: If users enter *Close* then show Close Help Screen

Table 1 is the 3 use cases in natural language using the format defined in [1]. This description includes the use case's name, triggers, preconditions, post conditions, primary actors, secondary actors, basic flow, alternate flows and exceptions.

When building system level test cases, testers read information about a use case to generate different test scenarios. These test scenarios are generated based on the event flows of use case. A problem is that, we cannot always identify all the test scenarios. For example, a use case containing loops in the event flows may have an infinite number of execution paths. To ensure the coverage quality of the generated test cases, we need to indentify the test scenarios in two cases. The first case is for precondition checking loops. This case has three test scenarios. The first scenario is loop has no iterations. The second scenario is loop has one iteration. The third scenario is loop has n iterations (n>1) and n is the maximum repeated times of some actions. The second case is for post-condition checking loops. This case has three test scenarios. The first scenario is loop with one iteration. The second scenario is loop with two iterations. The scenario is loop with n iterations (n>1) and n is the maximum repeated times of some actions. For example, in *Insert card*'s description, the loop of action *3 (Enter Pin)* is described in alternate flow *4b,* has the repeated max times of 3. This is because when the user enters the wrong PIN code 3 times, the loop does not go back to action 3 but performs action *4b* inputting false PIN code 3 times, then the loop does not go back to action *3* but performing action *4b* and end the use case.

To identify different test cases, from a use case testers have to identify such information as input and output parameters of the use case. The input parameters of the test cases are provided by actor, and system status. Take use case description of *Insert card* as an example. The input parameters are: Card (ATM card), *PinNumber* (PIN code), *EPTimes* (number of times to enter PIN code). Card and *PinNumber* are provided by user. *EPTimes* is the system status. *EPTimes* status can change when performing an action or a transition. Before performing the action *EnterPin,* this status is assigned the value 0. When performing *EnterPin* action, this status is increased to 1. At each action or transition, the conditions concerning the input and output parameters need to be satisfied. From these conditions, we can identify different input and output value sets for each test scenario. Moreover, in each test case, testers need to identify the preconditions to perform the test case. Test steps are ordered actions that the actor takes when performing a use case.

There have been a lot of researches focusing on using use case specification methods to automatically generate test cases. However, there has been no research that produces a formal specification that contains all of the information mentioned above. To address these challenges, we proposed a USL language for use case formal specification. USL language is defined by extending the activity diagram with a *Contract* conception. The structure of activity diagram specifies the event flows of the use case. We redefine actions in order to separately specify the actor actions, the system

actions and the actions that are outside of the system. The <<*include*>> and <<*extend* >> relations are defined as abstract actions *includedAction* and *extendingAction*. The *Contract* concept associating with each action and transition can be specified with more use case information, such as input and output parameters when performing the use case, pre- and post-conditions of the use case, the constraints on input and output parameters at each step of performing actions or transition and effect on input parameters when performing an action or transition. The USL language allows us to specify a use case with a single diagram. The use case specification is as easy to understand as the user requirement.

III. USL FORMAL SPECIFICATION LANGUAGE

In this section, we present the syntax and semantics of the USL version 2.0 as extended from USL language considered as version 1.0 in the previous researches [12] [13]. The extension of version 2.0 compared to version 1.0 is addition of some concepts: *AgentAction, IncludedAction, ExtendingAction, ForkNode, JointNode*. We also add such new properties in the *Contract* as *Inputs, Outputs, Maxloop, Effect*. The USL provides the ability to specify information on the use case more fully.

Concepts

USL is proposed on the basis of extending UML activity diagram. We give concept of *Contract* to describe the constraints for steps of performing use case and providing more information for actions and moving flows. The concepts in USL include:

- *InitialNode* and *FinalNode* correspond to start and finish nodes of use case. A use case has only an InitialNode but may have a lot of *FinalNodes*. Concrete notation of *InitialNode* and *FinalNode* is shown as node 0 and node 9 in Figure 5.
- *ActorAction*, *SystemAction*, and *AgentAction* correspond to actor action, system action and actions outside the system. The actions outside the system is actions that supplement information for use case. The semantics of use case do not change where these actions can be specified or not (an example in use case *Sale* “Customers choose goods and bring them to counter, ask to pay”). Concrete notation of *ActorAction*, *SystemAction*, *AgentAction* is shown as nodes 2, 3, 8 in Figure 5.
- *IncludedAction* corresponds to another included use case which is included in a use case and is considered to be an action of the basic flow. Concrete notation of *IncludedAction* is shown as node 1 in Figure 5.
- *ExtendingAction* corresponds to another extending use case which is extended at extending point and is considered to be an action in an alternate flow of the use case. Concrete syntax of *ExtendingAction* is shown as node E1 in Figure 5.
- *DecisionNode* is a decision node which has one incoming transition and multiple out going transitions, of which only one will be taken. Condition declared in Contract associating on transition should be made to ensure that only one transition can be taken. Concrete notation of *DecisionNode* is shown as node d1 in Figure 5.
- *ForkNode* is a control node that splits an incoming flow into multiple concurrent outgoing flows. Tokens arriving at a fork are duplicated across each outgoing transition.
- *JoinNode* is a control node that synchronizes multiple transitions. If there is a token offered on all incoming transitions, then a token is offered on the single outgoing transition.
- *Transition* is moving flow with directions between actions.
- *Contract* is a concept adding information to use case, actions and transitions.
- *HasContract* is a association between actions or transitions and Contract attached on it.

The Contract is structured as follows:

Contract Name	- *Name* is the name of the *Contract;*
Inputs VariableName: Type **Outputs** VariableName: Type	- *VariableName* is the name of the input and output parameters when performing use case. They are declared correspondingly in *Inputs* and *Outputs*;
Pre Condition **Post** Condition	- *Condition* is a logical or OCL expression clause declared corresponding in *Pre* and *Post*. Conditions in *Pre* describe the constraints on the input parameters corresponding to execute actions or transition. Conditions in *Post* describe expected outputs that system returns when taking actions;
MaxLoop Intvalue **Effect** Expression	- *Intvalue* declared in *MaxLoop* is integer value to specify maximum number of repeated times of an action; Expression declared in Effect is the expression which assigns new values for the input parameters when performing an action or a transition.

We built abstract syntax of the USL by a *metamodel* as shown in Figure 2. The specifications as shown in Figure 4 and 5 conform to the *metamodel*. The USL concepts are represented by graphical notation signed as in Figure 5.

We use a BNF-like (Backus – Naur Form) notation to present the USL grammar with the following conventions. Terminal symbols are bold strings enclosed in single quote. The remaining strings are non-terminals. The non-terminal of the first production is the start symbol of the grammar. Elements of a sequence are separated by blanks, alternative elements are separated by a "|"symbol. Optional elements are closed in brackets "[" and "]", Elements that can occur zero or more times are enclosed in braces "{" and "}". Parentheses "(" and ")" are used to group elements. Non-terminal symbols of the form "xId" have the same definition as the non-terminal "id".

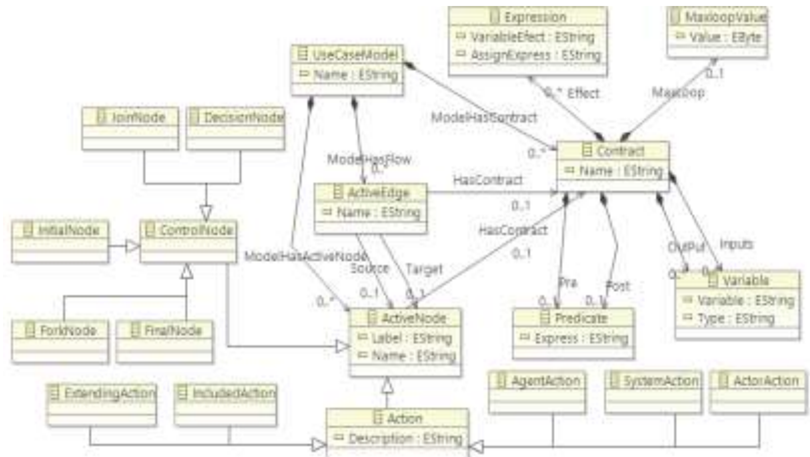

Figure 2. Metamodel of USL

USLLanguage ::= '**USLDiagram**' uId {USLDef}
USLDef ::= '**Actions**' listActions '**Transitions**' listTrans
'***Contracts***' list*Contracts*
listActions ::= (initialNode) {nodes} {finalNode}
initialNode ::= '**InitialNode**' iId {actionDef}
finalNode ::= '**FinalNode**' fId {actionDef}
nodes ::= aAction | sAction | agAction | iAction | eAction
dNode | fNode| jNode
aAction ::= '**ActorAction**' aId {actionDef }
sAction ::= '**SystemAction**' sId {actionDef}
agAction ::= '**AgentAction**' agId { actionDef }
iAction ::= '**IncludeAction**' iId { actionDef }
eAction ::= '**ExtendAction**'eId { actionDef }
dNode ::= '**DecisionNode**' dId {actionDef}
fNode ::= '**ForkNode**' fId {actionDef}
jNode ::= '**JoinNode**' jId {actionDef}
listTrans ::= {trans}
trans ::= '**Transition**' lId {tranDef}
listContract ::= {Contract}
Contract ::= '**Contract**' cId {ContractDef}
ContractDef ::= ['**Inputs**' defineVar]
['**Outputs**' defineVar] ['**Pre**' conditionEx]
['**Post**' conditionEx] ['**MaxLoop**' intvalue]
['**MaxLoop**' '**TRUE**' | '**FALSE**']
defineVar ::= (variable: type | '**DataBase**') { , variable: type | '**DataBase**' }

type ::= '**String**' | '**Int**' | object | '**Bool**' | '**Char**' | '**Double**' | ennum
sentence ::= atomicSentence | sentence connective sentence | quantifier variable,… sentence | '**NOT** ' sentence | '(' sentence ')';
connective ::= '**AND**' | '**OR** ' | '**IMPLIES**' | '**EQUIVALENT**';
quantifier ::= '∃' | '∀' ;
atomicSentence ::= term pt term | '**TRUE**' **|** '**FALSE** '
term ::= constant | variable;
variable ::= ('**a**' .. '**z**' | '**A**' .. '**Z**' | '_') {'**a**' .. '**z**' | '**A**' .. '**Z**' | '_' | '**0**' . .'**9**'}
pt::= '>' | '<' | '>=' | '<=' | '>' | ∈' | '∉' | '<>' | '='
constant::= {' ' | '*' | '+' | '-' | '\' | '**.**' | '**0**' . .'**9**' |'?' .. '~' }
function::= ('**a**' .. '**z**' | '**A**' .. '**Z**' | '_') {'**a**' .. '**z**' | '**A**' .. '**Z**' | '_' | '**0**' . .'**9**'}
intValue::= ('**1**' . .'**9**'){ '**0**' . .'**9**'}
tranDef ::= '**Out**' iId | sId | aId | dId
'**In**' fId | sId | aId | did
['**Has*Contract***' cId]
actionDef ::= ['**Description**' '"' actionDes '"']
['**Has*Contract***' cId]
id ::= ('**a**' .. '**z**' | '**A**' .. '**Z**' | '_') {'**a**' .. '**z**' | '**A**' .. '**Z**' | '_' | '**0**' . .'**9**'}
actionDes ::= {' ' | '**!**' .. '~' | '0' .. '9' }

Formal semantics of USL

This section will provide formal semantics of the USL. The functional requirements are specified by some use case diagrams. Each use case is specified by a graph in the USL, in which each use case scenario is an executable path in the USL. The USL allows us to specify a use case description as a diagram. The USL diagram includes collection of nodes which are connected by edges. It also includes a set of *Contracts* associated with nodes or edges. It is a kind of directed graph. Tokens which indicate control flow along edges from source node to the target nodes driven by the actions and contracts. The USL diagram has four kinds of modeling elements: nodes, edges, Contracts, Contract relations.

Definition 1: A USL diagram is a seven-tuple D = $(A, T, F_{out}, F_{in}, C, a_0, a_L)$, where

- $A = A_a \cup A_s \cup A_I \cup A_E \cup A_g \cup \{a_0, a_L\}$ with:
    - $A_a = \{a_1, a_2,..., a_m\}$ is a finite set of actor actions (*ActorAction*);
    - $A_s = \{s_1, s_2,..., s_n\}$ is a finite set of system actions (*SystemAction*); $A_I = \{b_1, b_2, ..., b_z\}$ is a finite set of included use cases which is included into a use case (*IncludedAction*); $A_E = \{d_1, ..., d_y\}$ is a finite set of extending use cases which is extended from a use case (*ExtendingAction*);

- $A_g = \{o_1, ..., o_h\}$ is a finite set of actions outside the system. The semantics of use case do not change whether these actions can be specified or not.

- $T = \{t_1, t_2, ..., t_l\}$ is a finite set of completion transitions.

- $C = \{c_1, c_2, ..., c_k\}$ is a finite set of Contract associated with actions or transitions in the model, which c_i is in correspondence with $a_j \in A$ or $t_j \in T$. *ConA* is a mapping from a_j to c_i so that $ConA(a_j)=c_i$. *ConF* is a mapping from t_j to c_i so that $ConF(t_j) = c_i$

- $F_{out} \subseteq A \times T$, $F_{in} \subseteq T \times A$ is the flow relation between the nodes and transitions. F_{out} is the transition coming out from a node. F_{in} is the transition coming into a node.

- a_0 is the initial node so that $\exists t \in T$ & $(a_0, t) \in F_{out}$, and if $\exists t' \in T$ & $(t', a_0) \in F_{in}$ then $\forall a \in \{w \mid (w, t') \in F_{out}\} \mid a \in A_I$ & $\nexists t'' \in T$ & $(t'', a) \in F_{in}$. That means that the start node must has flow come out, if any, the flow come in initial node then must come from Included action and no transition come in *IncludedAction*.

- a_L is the final node so that $\exists t \in T$ & $(t, a_L) \in F_{in}$, and $\nexists t' \in T$ & $(a_L, t') \in F_{out}$.

Definition 2: A use case scenario, $u_s \in U_S$, in a USL diagram D, can be defined as an execution path from the *InitialNode* to the *FinalNode* consisting of activities and transitions. i.e. $\forall u_s$, where $u_s \in U_S$, $u_s = e_0 \rightarrow t_0 \rightarrow e_1 \rightarrow t_1 \rightarrow ... \rightarrow t_{x-1} \rightarrow e_x$ with $e_0 = \{a_0\}$ or $e_0 \in A_I$, $e_x = \{a_L\}$, and $\forall i (0<i<=x)$ then, and $(e_{i-1}, t_{i-1}) \in F_{out}$ *and* $(t_{i-1}, e_i) \in F_{in}$ and *pre* of $ConF(t_{i-1})$ *and* $ConA(e_i)$ is satisfied.U_S is the set of use case scenarios. Each u_s corresponds to a test scenario.

Definition 3: V is a set of all collective contracts on all use case scenarios U_S, in a USL diagram D, can be defined, each v_s corresponds to a u_s is a set of Contracts in order where $v_s = \bigcup_{i=0}^{x-1}(ConA(e_i) \cup ConF(t_i)) \cup ConA(e_x)$.

IV. FUNCTIONAL REQUIREMENTS SPECIFICATION METHOD USING USL

Figure 2 shows an overview of the formal method for specifying the software functional requirements using USL. In this method, the software functional requirements are specified by a set of use case diagrams and for each use case description is specified by a USL diagram. The use case diagram determines information about the use cases and the performing sequence of use cases. The detailed USL specifications of each use case determine the use case scenarios and the set of contracts of each use case scenario.

From the use case diagram, we can automatically determine the order in which the use cases are performed by using the Graph traversal algorithms and by analyzing the relationship among the use cases. Developers will transform the specification of each use case from the textual description into a USL diagram. We can use the Depth-first search algorithm to automatically generate use case scenarios and the set of contracts of each scenario. From these, we can define the test cases that have such information as input and output parameters, values of the input and expected output parameters, preconditions to perform testing, and test steps.

The transformation from the use case textual description into USL model

The steps to transform the use case textual description into USL diagram are stated as below.

1. Read the information in the use case diagrams to get the information about the relationship among the use cases.

2. Transform each use case textual description format as in [1] to the USL diagram. The included case uses and extending use cases are transformed firstly. The use cases containing the included use cases and extending use cases are transformed later.

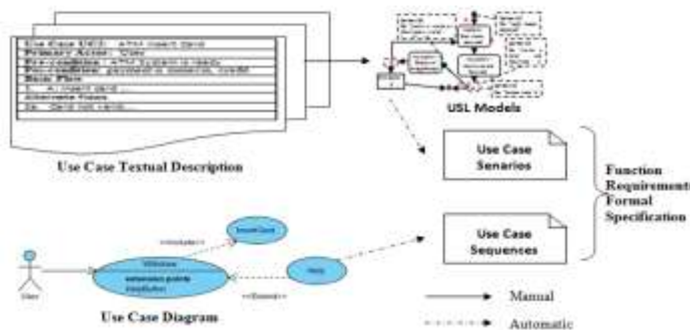

Figure 3. Overview of the proposed approach

3. Each use case is specified by a USL diagram based on the following principles:

- Transfer each step in the basic flow and Alternative flows into the corresponding actions such as *ActorAction, SystemAction, AgentAction.* The semantic of use case does not change the meaning whether the actions outside the system transfer or not. Use *InitialNode* and *FinalNode* to specify the starting and the end point of the use case. There is only one *InitialNode* but there can be lots of *FinalNodes*.
- If in each system action or actor action, there is the selection of the next actions add *DecisionNode* to specify.
- Use the *Transition* directional arrows to specify transition of the action.
- Add *ForkNode* and *JoinNode* at the beginning and the ending of concurrence actions.
- Use *IncludedAction* and *ExtendingAction* to specify the <<include>> and <<extend>> relation.

- For each relation, the use template has a redefined structure. For instance, the <<include>> relation is considered as an abstract action of a basic flow, while <<extend>> relation is considered as an abstract action of an alternate flow.
- The use case is extended at any action when satisfying the trigger conditions are regulated to specify as an *ExtendingAction* with the transition coming out from *initialNode*.
- Using the *Contract* concept to specify information for each action and transition as below
  - The contract associated to *InitialNode* allows us to specify the input and output parameters of use case. It is corresponding declared in *Inputs* and *Outputs*. Preconditions of use case are declared in the *pre*.
  - The contract associated to nodes or transitions, except for *AgentAction* nodes allows us to specify the constraints on input and output parameters when performing actions or transitions. The constraints on input parameters are described in *pre.* The constraints on output parameters described in *post*. The effects on input parameters when performing an action or a transition are declared in *effect*.
  - If the action has limited times of repetition, the limited times shall be declared in *MaxLoop*, for example in *Contract* C3 associated to action 3 in use case *Insert card* as in Figure 4.

Case Study

The USL specifications of the three use cases *Insertcard, Help, Withdraw* are shown in Figure 4 and Figure 5 below. There, actions in the basic flow and alternate flows are specified by corresponding actions and added with controlling nodes. The *Contracts* attached to actions and transitions allow us to specify detailed information when performing the use cases. In Figure 5, node 0 (*InitialNode*) is the starting point of the use case. Node 9 (*FinalNode*) is the ending point of the use case. The corresponding actions 2, 3, 8 are actions of the actor (*ActorAction*), the system action (*SystemAction*), and the agent action (*AgentAction*). Action 1 is the *IncludedAction* describing the <<*include*>> relation between *Insert card* and *Withdraw*. Action E1 is the *ExtendingAction* describing the *<<extend>>* relation of *Help* from *Withdraw*. Contract C1 describes input and output parameters when performing the use case declared in *Inputs* and *Outputs*, and the precondition of the use case declared in *Pre*. Contract C6 describes the constraints on input parameters to allow the action 5a.1 to occur. These constraints are declared in *Pre*. The expected outputs when performing action 2a are declared in *Post*. In Figure 4, Contract C2 describes the constraints for the transition from d to 3. These constraints are declared in *Pre*. Before executing action 3, the input parameter *EPTimes* = 0 is declared in *Effect*. Contract C3 describes action 3 that is repeatable up to 3 times. The maximum of repeated times is declared in *MaxLoop*. Whenever action 3 is performed, *EPTimes* will be increased to 1 and this is declared in *Effect*.

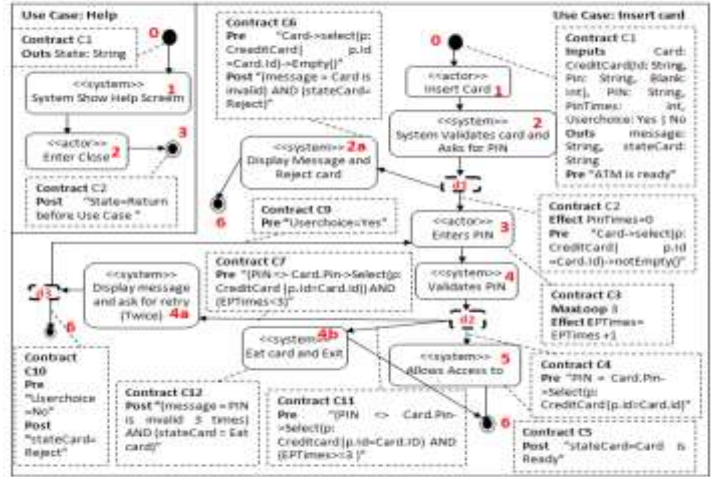

Figure 5. USL use case specifications of Withdraw

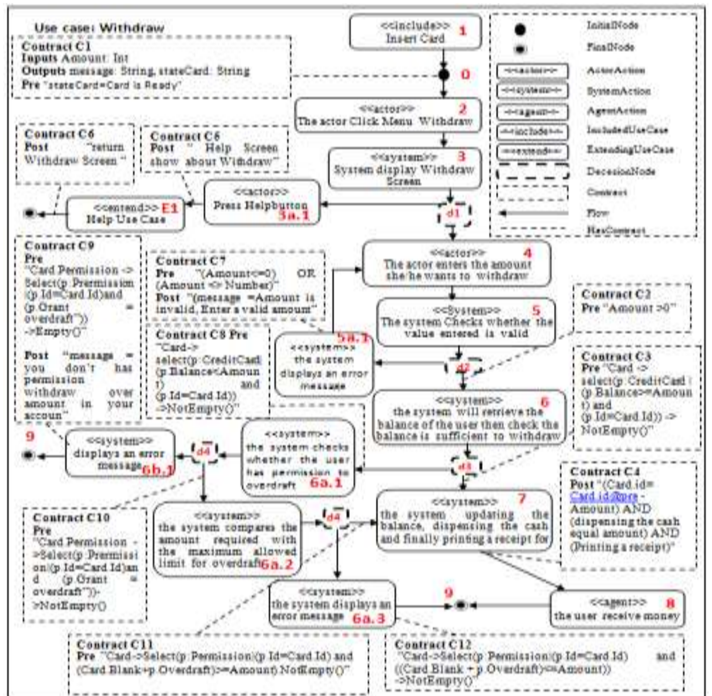

Figure 4. USL use case specifications of Help and Insert Card

Through these use case specification examples in the ATM system, we illustrated the different cases when specifying use cases. These specifications will be the input for automatic test case generation form software functional specifications. The method of automatical test case generation will be presented in our future work.

V. CONCLUSION

In this paper, we extended the USL formal specification language from version 1.0 to version 2.0 and proposed a more complete method for specifying the software functional requirements. We solved the problems of other researches which not being able to simultaneously specify information the following: pre- and post-conditions of use

case, the relations with the use cases, input and output parameters when performing use case, the maximum number of repetitions of an action, pre and post conditions when performing an action or a transition. Our method is simple and close to the natural language descriptions. In addition, USL specifications can be used to automatically generate test cases. In the future, we will continue to research and improve the method of automatic generation of test cases to achieve better quality. We also aim set up the algorithms for generating the textual form test case.

REFERENCES

[1] A. Cockburn, *Writing Effective Use Cases*.: Addison-Wesley, 2001.

[2] Nhuan D, Lai V. Y. N, Tho T. Q, and Thuan D. L, "A framework for automatic construction of test scenarios from use-cases," in *Ho Chi Minh City Software Testing Conference January 2015*, Ho Chi Minh, 2015.

[3] C. Nebut, F. Fleurey, Y. Le Traon, and J. M. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, pp. 140-155, 2006.

[4] M. Smialek and T.Straszak, "Automating acceptance testing with tool support," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, 2014, pp. 1569-1574.

[5] C. Wang, F.Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015*, New York, 2015, pp. 385-396.

[6] M. Chen, P. Mishra, and D. Kalita, "Coverage-driven automatic test generation for uml activity diagrams," in *" Proceedings of the 18th ACM Great Lakes Symposium on VLSI, GLSVLSI '08*, USA, 2008, pp. 139-142.

[7] R. K. Swain, V. Panthi, and P. K. Behera, "Generation of test cases using activity diagram," *International Journal of Computer Science and Informatics*, vol. 3, 2013.

[8] P. N. Boghdady, N. L. Badr, M. Hashem, and M. F. Tolba, "A proposed test case generation technique based on activity diagrams," *International Journal of Engineering & Technology IJET-IJENS*, vol. 11 , 2011.

[9] N. N. Patil and P. E. Patel, "Testcases formation using uml activity diagram," in *Communication Systems and Network Technologies (CSNT),* 2013.

[10] Jes´ us M. Almendros-Jim´enez and Luis Iribarne, "Describing Use Cases with Activity Charts," in *Lecture Notes in Computer Science*.: Springer Berlin Heidelberg, 2005, pp. 141-159.

[11] S. Tiwari and A.l Gupta, "An approach of generating test requirements for agile software development," in *Proceedings of the 8th India Software Engineering Conference, ISEC '15*, New York, 2015, pp. 186-195.

[12] C. T. M. Hue, D. D. Hanh, N. N. Binh, "A Method to Generate Test Cases from Use Cases," in *Fair'8*, Ha Noi, 2015, pp. 590-599.

[13] C. T. M. Hue, N. N. Binh, and D. D. Hanh, "Automatic Test-case Generation from Detailed Use-case Specification for System Testing," *submit to Research, Development and Application on Information and Communication Technology*, June 2016.

[14] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Redwood, USA: Addison Wesley Longman Publishing Co, 2004.

[15] IBM, *Object-Oriented Analysis and Design with UML2*.: IBM Corporation, 2006.

[16] Ivar Jacobson and Pan-Wei Ng, *Aspect-Oriented Software Development with Use Cases*. Redwood, USA: Addison-Wesley, 2004.

[17] Richard C. Gronback, *Eclipse Modeling Project A Domain-Specific Language*.: Addison-Wesley, 2009.

[18] V. Panthi and D.P. Mohapatra, "Automatic test case generation using sequence diagram," in *International Journal of Applied Information Systems*, New York, 2012, pp. 277-284.

PHƯƠNG PHÁP ĐẶC TẢ CHỨC NĂNG PHẦN MỀM ĐỂ SINH CÁC CA KIỂM THỬ HỆ THỐNG

Chu Thị Minh Huệ, Nguyễn Ngọc Bình, Đặng Đức Hạnh

TÓM TẮT *— Trong bài báo này, chúng tôi mở rộng ngôn ngữ đặc tả ca sử dụng (USL) từ phiên bản 1.0 sang phiên bản 2.0 cho phép đặc tả hình thức mô tả của từng ca sử dụng ở mức yêu cầu người dùng chỉ với một mô hình duy nhất, mà không sử dụng các thiết kế chi tiết của ca sử dụng. Chúng tôi cũng nêu hướng dẫn phương pháp đặc tả hình thức các yêu cầu chức năng. Các đặc tả yêu cầu có thể được sử dụng để sinh tự động các ca kiểm thử mức hệ thống. Ngôn ngữ USL được mở rộng từ biểu đồ hoạt động trong UML và thêm vào khái niệm Contract nhằm đặc tả đầy đủ hơn các thông tin khi thực hiện ca sử dụng như: chuỗi các hành động trong các luồng sự kiện, các tham số đầu vào và đầu ra khi thực hiện ca sử dụng, tiền điều kiện và hậu điệu kiện thực hiện ca sử dụng, điều kiện ràng buộc trên dữ liệu đầu vào và đầu ra tại mỗi hành động hoặc luồng chuyển, mối quan hệ <<include>> và <<extend>>, số lần lặp tối đa của một hành động, hiệu ứng thay đổi giá trị tham số đầu vào khi thực hiện một hành động hoặc luồng chuyển. Chúng tôi xây dựng một MetaModel mô tả cú pháp trừu tượng, xây dựng cú pháp cụ thể dạng văn bản bằng BNF và định nghĩa ngữ nghĩa hình thức cho ngôn ngữ.*