A Formal Approach to Requirement Verification

Divya K. Nair, Stéphane S. Somé
School of Information Technology and Engineering (SITE)
University of Ottawa
800 King Edward, P.O. Box 450, Stn. A Ottawa, Ontario, K1N 6N5, Canada

Abstract

In this paper, we propose an approach for the verification of a state model against use cases written in natural language. An objective of this approach is to provide a quick way to ensure that requirements modeling proceeds in the right direction in the context of iterative development. The approach consists of extracting logical statements from use cases and verifying these statements with corresponding statements obtained from state machines. We are able to detect some requirement violations symptomatic of wrong assumptions.

1 Introduction

Use cases/Scenarios are considered as one of the most effective ways to capture requirements and guide software development. Use cases are easily traceable to the later development stages namely implementation and testing. A use case describes a set of sequential interactions between a system and its actors. A use case scenario represents a sequential flow of events involving actors entitled to perform a specific function. According to [1], scenarios are intuitive and very close to users' requirements. At the early requirement analysis stage, use cases are built in informal textual format as plainly stated by the user. Then these use cases are usually formalized using UML Sequence Diagram, Message Sequence Charts (MSC) or State Machines. But once the requirements are formalized, the software developer takes little concern in checking whether the formalized design still follows all the basic functionalities correctly with reference to the informal textual requirements. The above factors were instrumental to the development of the approach described in this paper. The objective of this approach is to provide a mechanism for checking a design model represented as state machines, against use cases. We focus on "early" use cases represented in a form of natural language. The conversion of use cases to state machines is essential to refine system behavior and to keep consistency between requirements and design [10]. Our goal is to provide a *lightweight* and semi-automated approach that could be applied in the context of iterative elaboration of state machines from use cases. The approach is lightweight in the sense that it relies on simple conversion rules and a proof mechanism based on first order logic. It does not proscribe full-blown validation based on simulation or theorem proving. But, our position is that such approaches are more *heavyweight* and are better used later after enough requirements are gathered.

The essence of our verification approach is the derivation of logical statements corresponding to use case scenarios and state machines, followed by a formal matching of these statements. We use a form of the Object Constraint Language (OCL) [7] for logical statements. The approach is based on the assumption that use cases and state machines are syntactically consistent; that is a same terminology is used in both models. Another assumption is that states are formally specified in state machines using predicates. Because of these assumptions, the approach naturally fits into the UCEd process [9].

The remainder of this paper is organized as follows. Section 2 summarizes our use case based requirements engineering approach and its supporting tool UCEd. In Section 3, we present our verification approach. This section is subdivided further to explain the extraction of OCL statements from use cases and state machines, and the algorithm used for validating these OCL operations. Finally section 4 presents some related works and concludes the paper.

2 Use case based requirements engineering

Figure 1 describes our requirements engineering process as supported by the UCEd tool. Requirements represented as use cases are captured and executable state machines are automatically generated using domain information [8]. The generated state machines are used as prototypes for requirements validation by simulation.

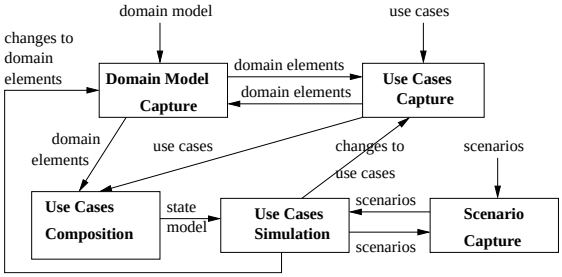

Figure 1: Use cases based requirements engineering process.

2.1 Use cases

Figure 2 shows a use case in the context of an ATM banking application. A use case captures requirements as sequences of interactions between a system and its actors. Formally, it is a tuple [*Title*, *Precondition*, *Steps*, *Postcondition*] with: *Title* a label that uniquely identifies the use case, *Precondition* a *condition* that must be true before an instance of the use case can be executed, *Steps* a sequence of steps, and *Postcondition* a *condition* that must be true at the end of the normal execution of an instance of the use case. Each step in a use case may include a set of *extensions* which are alternative behaviors that may follow the step. We developed a restricted form of natural language for writing use cases [9].

2.2 Domain model

We use a *domain model* where entities and operations are declared, in order to allow for use cases parsing and state machine generation. Figure 3 shows an excerpt of a domain model for the ATM banking application (because of space limitation, this domain model is incomplete). Operations are specified using *withdrawn* and *added* conditions. Withdrawn conditions are those conditions that are eliminated after the operation execution and added conditions are condition that hold after operation execution.

2.3 State machines

A state machine may integrate one or more use cases in terms of states and transitions. Figure 4 shows a state machine (also termed as state chart or state diagram) corresponding to the use case in Figure 2. This state machine is produced automatically using UCEd according to an algorithm described in [8]. We distinguish *choice-points* from other states. A choice-point allows only guarded transitions. For instance, state *2* in Figure 4 is a choice-point. A state machine also

```
Title: Cash withdrawal
Precondition: ATM status is ON and
              ATM display is Welcome message
Post condition: ATM status is ON and
               ATM display is operation menu
STEPS
1. User inserts card in card reader slot
2 .ATM asks PIN
3. Before 60 sec, user enters PIN
4. ATM asks user validation to bank
5. Bank checks user id
6. If user identification is valid,
              ATM display is operation menu
7. User selects cash withdrawal
8. ATM asks withdrawal amount
9. Customer enters withdrawal amount
10.ATM checks customer update
11.If user withdrawal OK,
 ATM dispenses cash in cash dispenser slot
 and ATM ejects card
12.User takes cash and
   ATM display is operation menu
ALTERNATIVES
1.a. User card is not valid
 1.a.1.  ATM ejects card
 1.a.2.  ATM displays error message
 1.a.3. After 30 sec,
        ATM displays welcome message
2.a. After 60 sec,
 2.a.1.  ATM ejects card
 2.a.2.  ATM displays error message
 2.a.3. After 30 sec,
        ATM displays welcome message
5.a. User identification is invalid
 5.a.1.  ATM ejects card
 5.a.2.  ATM displays error message
 5.a.3. After 30 sec,
        ATM displays welcome message
10.a. User withdrawal is not OK
 10.a.1. ATM displays error
 10.a.2. After 10 sec, Go to step 8
```

Figure 2: Cash withdrawal Use case.

includes an initial state (state *1* in Figure 4). Each state s is defined by a set of *characteristic conditions* $Cond(s)$, such that all conditions in $Cond(s)$ are verified when the system is in state s. Figure 5 shows the characteristic conditions of the states in Figure 4 .

State machine synthesis is based of the specification of domain model operations as sets of added conditions and withdrawn conditions (see Figure 3). When an operation, *op* is applied to a state *s1* with characteristic conditions $Cond(s1)$, the resulting state is a state *s2* such that $Cond(s2) = Cond(s1)$ - *withdrawn-cond*(op) + *added-cond*(op).

```
Concept:ATM, Attributes:Display,Transaction status
 Operation:display welcome message
  AddedCondition:ATM Display is welcome message
  WithdrawCondition:ANY ON USER Card
 Operation:ask pin
  AddedConditions:User PIN is requested, ATM
   Display is pin enter prompt
 Operation:ask user validation
  AddedCondition:User Validation status is bank
                 inquired
 Operation:display operation menu
  AddedCondition:ATM Display is operation menu
 Operation:eject card
  AddedCondition:User Validation status is
                 card ejected
 Operation:display error message
  AddedCondition:ATM display is error message
Concept:User, Attributes:PIN, Validation status
 Operation:insert Card
  AddedCondition:USER Validation status is
                 card inserted
  WithdrawCondition:ANY ON ATM Display
 Operation:enter pin
  AddedCondition:USER Validation status is
                 pin entered
 Operation:select cash withdrawal
  AddedCondition:USER Transaction is cash withdrawal,
   ATM Transaction status is withdrawal initiated
  WithdrawCondition:ANY ON ATM Display
```

Figure 3: ATM application domain model.

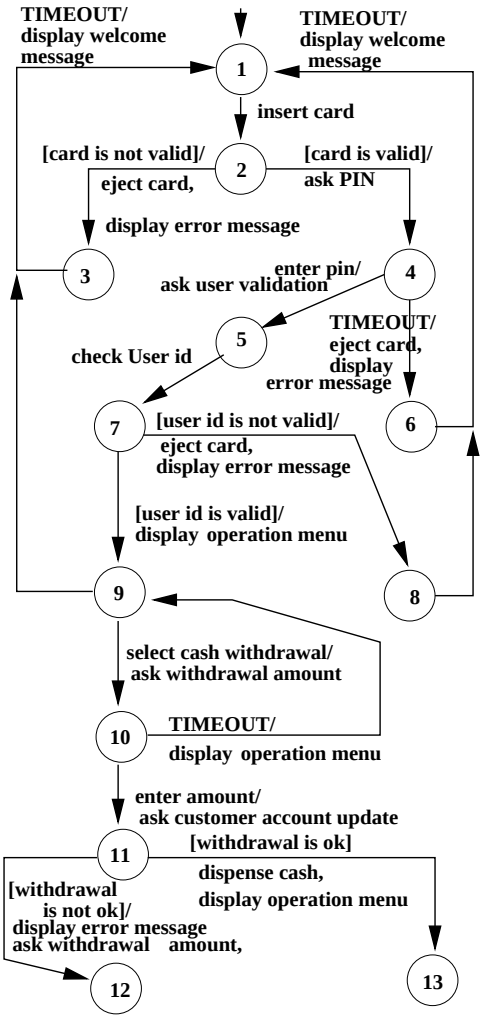

Figure 4: State Machine corresponding to Use Case Cash withdrawal.

3 Verification approach

A use case corresponds to a set of scenarios that are sequences of events. One of these scenarios is usually referred as the "main scenario" while other scenarios are "secondary scenarios". A *precondition* and a *postcondition* can be associated to each of these scenarios. The precondition denotes necessary conditions for the scenario execution, while the postcondition is a set of conditions that must hold at the end of the scenario. The underlying idea of state machines verification is to check that when a given scenario precondition hold, the state machine ensures that its postcondition is verified at the end of the scenario. This is possible if there is a specific mechanism to represent use case scenarios and state chart diagram into similar form of logical operational statements. Once this conversion to logical statements is done, the preconditions to be verified are generated automatically from these use case based logical statements. Then the post conditions corresponding to these use case based precondition logical statements are compared against the post conditions of state chart based logical statements. Hence, the approach can be decomposed into the following major sequences: (1) conversion of use case scenarios to a form of OCL based operational statements, (2) conversion of state chart diagrams to OCL based operational statements, and (3) matching of the scenario based OCL operational statements (step1) and state chart based OCL operational statements (step2). It is worth noticing that verification is necessary even in the context automated state machine generation. The domain model as well as use cases upon which state machine generation is based, may be incomplete and may include inconsistencies. It is also possible that generated state machines are manually altered and developers need to ensure that no inconsistency is introduced.

3.1 Conversion of use case scenarios

We generate pre and post conditions corresponding to scenarios according to the three following cases.

```
S1 [ATM is ON,ATM Display is welcome message]
S2 Choice point
S3 [Timer11:30.0 second,ATM is ON,
    User Validation status is card ejected,
    User Card is NOT valid,
    ATM Display is error message]
S4 [ATM Display is pin enter prompt,
    ATM is ON, User PIN is requested,
    User Validation status is card inserted]
S5 [ATM Display is pin enter prompt,
    ATM is ON, User PIN is requested,
    User Validation status is bank inquired]
S6 [ATM is ON, User Validation status
    is card ejected, User PIN is requested,
    ATM Display is error message]
S7 Choice point
S8 [ATM Display is operation menu, ATM is ON]
S9 [ATM Display is operation menu, ATM is ON,
    User Identification is valid,
    User PIN is requested,
    User Validation status is bank responded]
S10 [ATM is ON,
    ATM Display is withdrawal amount,
    User Transaction is cash withdrawal]
S11 ChoicePoint
S12 [ATM Transaction status is amount checking,
     User withdrawal is NOT ok,ATM is ON,
     ATM Display is error message,
     User Transaction is cash withdrawal]
S13 [ATM Display is operation menu,
     ATM Transaction status is cash dispensed,
     ATM is ON,
     User Validation status is card ejected,
     User Transaction is cash withdrawal]
```

Figure 5: Cash withdrawal Use case.

1. The use case precondition and respective postcondition (for the main scenario).

2. Preconditions sequences and their respective postconditions sequences for secondary scenarios.

3. Condition sequences to account for conditional behavior embedded within scenarios.

From the pre and postconditions in Figure 2, we generate the following OCL statement (A).

```
Context ATMControl:: cashwithdrawal ()
Pre: (ATM.status=ON) and (ATM.Display=Welcome)
Post: (ATM.status=ON) and
      (ATM.Display=Operation menu)
```

This statement corresponds to the use case main scenario. Notice that the conversion is a straightforward translation from our natural language based syntax.

For secondary scenarios, the generated precondition corresponds to the step that has the extension "anded" with the extension condition. For instance alternative *10.a* in Figure 2 corresponds to the following statement (B).

```
Context ATMControl:: cashwithdrawal ()
Pre: (User.withdrawal = NOT OK) and
   (ATM.Transaction status = amount checking)
Post: (ATM.Display = error message) and
   (ATM.Display = askwithdrawalmount)
```

The precondition is obtained from step *10* operation added-condition combined with the extension condition "*User withdrawal is not OK*". The postcondition corresponds to the added/withdrawn conditions of the operations in the extension.

For the third category, the pre condition is the *if* part or any other conditional parts present in any of the use case steps and the corresponding post condition is revealed by the succeeding action sequence. For instance step *11* in Figure 2 corresponds to the following statement (C).

```
Context ATMControl :: cashwithdrawal ()
Pre: (User.withdrawal = OK)
Post: (User.validation status= card ejected) and
 (ATM.Transaction status = cash dispensed)
```

The *If* statement introduces the precondition and the operation yields the postcondition. In the remainder of this paper, we will refer to preconditions and postconditions derived from use case scenarios as `UCS-pre` and `UCS-post` respectively.

3.2 Conversion of state machines

State machine conversion is performed according to the algorithm in Figure 6. We generate plain OCL based state chart statements (denoted by `SCS` in future explanations) corresponding to each state and its respective transitions. SCS statements are generated by pairs. Therefore during the verification, for a given SCS corresponding to a UCS-pre, the respective post condition will be the `next` OCL state chart statement (next SCS) immediately following this SCS under reference. Procedure `generate_OCL` generates the SCS corresponding to a state in a similar way as in section 3.1. A count of the number of transitions starting from a state is stored in variable `SCcount`. If there are more than one transitions starting from a state, procedure `generate_OCL_trans` ensures that the first transition is taken into account and the corresponding SCS will contain the predicates in that state along with the transition guards if any. The next SCS will correspond to the resultant state of the transition in the state diagram if that state has no departing transitions. Otherwise this process is repeated until a state with no SC transitions is reached. This particular function is implemented by the procedure, `generate_OCL_resultant`.

```
Procedure Generate_SCL_spec_statechart
         (S: state machine)
Begin
 Mainpre = generate_SCL(s1:state1,S:state machine);
 n= count (S: state chart);
 Mainpost = generate_ SCL (sn: last state,
                           S: state machine);
 Print (mainpre, mainpost)
 For each state (s S) and (s! = sn) do
  Begin
   If (SCtransitions! = null)
   Begin
    SCcount = number_of_trans (s)
    I = 0
    While (SCcount! = I )
    Begin
     If (s! = 'choicepoint')
      Pre = generate_SCL (s,S) U
            generate_SCL_trans (s,S,SC)
      Post = generate_SCL_resultant (s, S, SC)
     Else if (s = 'choicepoint')
      Pre = generate_SCL_prestate (s, S) U
            generate_SCL_trans (s, S, SC)
      Post = generate_SCL_resultant (s, S, SC)
     End if
     Print (pre, post)
     I = I +1
   End while
   Else if (SCtransitions = null)
    Pre = generate_SCL (s, S)
    Post = generate_SCL_next(s, S)
    Print (pre, post)
   End if
  End for
End
```

Figure 6: Algorithm for state chart conversion.

The procedure keeps a list of visited states to avoid infinite loops. Procedures, `generate_OCL_prestate` and `generate_OCL_resultant` are used when considering a *choice point*. In this case the corresponding SCS is obtained from the state that caused the transition to the choice-point. Figure 7 shows the SCS statements generated from the state machine in Figure 4.

3.3 Statement Matching

Figure 8 summarizes our matching technique for logical statements derived from use cases and state machines. Given a UCS-pre, we first attempt to find a matching SCS. If successful, we then verify that the immediate next SCS includes that of the corresponding UCS-post. In case, the UCS-post does not match, the verification is interrupted by allowing the developer to see the generated inconsistency details (the erroneous UCS-pre, UCS-post and SCS).

1. (ATM.status=ON) and (ATM.Display=Welcome)

2. (ATM.Display=Operation menu) and (ATM.status=ON) and (User.validation status=card ejected) and (ATM.Transaction status=cash dispensed)

3. (ATM.status=ON) and (ATM.Display=Welcome) and if (User.validation status = card inserted)

4. (ATM.status=ON) and (ATM.Display=Welcome) and if (User.validation status=card inserted) and if (User.card= NOT valid)

5. ...

x. (ATM.status=ON) and (ATM.Transaction status=amount checking) and (ATM.Display=askwithdrawalamount) and if (User.withdrawal=NOT OK)

x+1. (ATM.Display=error message) and (ATM.status=ON) and and (ATM.Transaction.status=amount checking)

...

Figure 7: SCL statements generated from state machine in Figure 4.

```
Procedure generate_matching (S: state machine;
                             M: scenario model)
Begin
 IN1= Generate_OCL_spec_scenario (M)
 IN2= Generate_OCL_spec_statechart (S)
 For each pre condition, p in IN1
   Pre=Findcorrespre (IN2,IN1,p);
   Scenpost = scencheckpost (p, IN1);
   Statepost = statecheckpost ( Pre, IN2);
   If (statepost = includes (scenpost)) then
     Print ("no ambiguity");
   Else
     Print ("ambiguity", pre, scenpost, statepost);
 End for
End
```

Figure 8: Algorithm for matching statements.

As an example, consider instance `A` in section 3.1. Recall that the OCL statement in this instance corresponds to use case *Cash withdrawal* main scenario. The UCS-pre (ATM.status=ON) and (ATM.Display=Welcome) matches the SCS statement *1* in Figure 7. So, the next step is to check the corresponding UCS-post, (ATM.status = ON) and (ATM.Display = Operation menu) with the SCS statement *2* (ATM.Display = Operation menu) and (ATM.status = ON) and (User.validation status = card ejected) and (ATM.Transaction status = cash dispensed). We can notice that the SCS statement *2* includes the UCS-post. Therefore, the verification is successful.

Consider now instance B. The UCS-pre can be matched to the SCS statement *x*. However, the corresponding UCS-post (ATM.Display = error message) and (ATM.Display = askwithdrawalmount) cannot be matched to the SCS statement *x+1* (ATM.Display = error message) and (ATM.status = ON) and and (ATM.Transaction.status=amount checking). The SCS statement is missing condition (ATM.Display = askwithdrawalmount), which means that when the user withdrawal amount is invalid, an error message is generated but the user is not asked to re-enter the withdrawal amount. An analysis of the inconsistency shows a contradiction in use case *Cash withdrawal*. According to steps *10.a.1* and *10.a.2* (*8*), if the withdrawal amount is not OK, ATM Display should be "error message" and ATM Display should be "ask withdrawal amount" which suggests two different values for ATM Display at the same time. The inconsistency can be eliminated by performing a correction to the use case. For instance, step *10.a.2* could be rewritten as "10.a.2. After 30 seconds, Goto step 8".

4 Conclusions

In this paper, we have proposed an approach to ensure the consistency of use cases against state machines. There are very few works related ours. Martin Giese and Rogardt Heldal [3] suggest an approach for relating formal and informal requirements using OCL. However, the focus is on the extraction of OCL statements from use cases formalized as state machines. In [4], Martin Glinz presents an approach aiming at improving the quality of requirements with scenarios. The approach concentrates on systematic representation of use case relationships. Grieskamp and Lepper [5] selected Z language to model use cases giving way to executable specifications, but this was only good enough to verify the use cases and gives little concern to the informal specifications. Levy, Marcano and Souquières [6] also used a similar briefed approach as [2] using the B formal language.

An objective of our approach is to help bridge the gap between informal and formal requirements by logically checking the presence of contradictions between scenarios as stated in use cases and as realized in state machines. Because our approach relies on a "*contract-like*" specification of domain operations, we also provide an early validation of high-level design assumptions. The approach does not preclude for extensive validation for which, we are planning on using theorem-proving techniques.

References

[1] D. Amyot and A. Eberlein. An Evaluation of Scenario Notations for Telecommunication Systems Development. In *Proceedings 9 th ICTS*, March 2001.

[2] Achim D. Brucker and Burkhart Wolff. A proposal for a formal ocl semantics in isabelle/hol. In *Proceedings 15th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2002*, pages 99–114, August 2002.

[3] Martin Giese and Rogardt Heldal. From Informal to Formal Specifications in UML. In *Proceedings of the 7th UML International Conference - The Unified Modelling Language: Modelling Languages and Applications*, pages 197–211, October 2004.

[4] Martin Glinz. Improving the quality of requirements with scenarios. In *Proceedings of the Second World Congress on Software Quality*, pages 55–60, September 2000.

[5] Wolfgang Grieskamp and Markus Lepper. Using use cases in executable z. In *ICFEM*, pages 111–120, 2000.

[6] N. Levy, R. Marcano, and J. Souquires. From requirements to formal specification using UML and B. In *2nd International Conference in Computer Systems and Technologies CompSysTech2002*, June 2002.

[7] OMG. OMG Unified Modeling Language Specification version 1.4, 2001.

[8] S. Somé. An approach for the synthesis of state transition graphs from use cases. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'03)*, volume I, pages 456–462, june 2003.

[9] S. Somé. Supporting use cases based requirements engineering. *Information and Software Technology*, 48:43–58, 2006.

[10] H. Wang, K. Zhang, T. Feng, H. Che, and Y. Zheng. Synthesizing Statecharts through Sequence Diagrams Analysis. In *Software Engineering and Applications (SEA 2004)*, 2004.