Generation of Test Scenarios from Use Cases

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 present an approach for the generation of test scenarios from use cases. Test scenarios can be used to validate use cases. They also constitute a step toward complete test cases. We focus on use cases described in a restricted form of natural language. Test scenario generation proceeds in three steps. We start by extracting a control flow graph from use cases, then we extract path sequences from the control flow graph and finally, we derive test scenarios based on these path sequences. The approach is an extension of the UCEd approach for use cases based requirements engineering.***

Keywords: Use Case, Scenario, Requirements Validation, Testing

1 Introduction

Use cases describe interactions involving systems and their environments. A use case is the specification of a sequence of actions, including variants, that a system can perform, interacting with actors of the system [4]. Use cases have become one of the favorite approaches for requirements capture. In our previous work [7, 8], we developed an approach for use cases based requirements elaboration. This approach is supported by a tool called Use Case Editor (UCEd) that allows use cases capture and use cases validation based on simulation. In [9], we introduced *scenarios* to allow repeatability of simulation sessions. A scenario describes an execution sequence of a system with input/output events as well as assertions. This paper presents a further extension of our approach with automated generation of scenarios. One of our objective is to allow automated validation of use cases based requirements. Another objective is to use generated scenarios as a basis for test cases development.

This paper is organized as follow. The next section introduces the UCEd approach as well as use cases and scenarios. Section 3 presents our approach for the automated generation of scenario. In Section 4 we discuss some related work and finally, section 5 concludes the paper and presents our ongoing and future work.

2 Background

Figure 1 describes our requirements engineering process. It is supported by the UCEd tool. Requirements are captured as use cases from

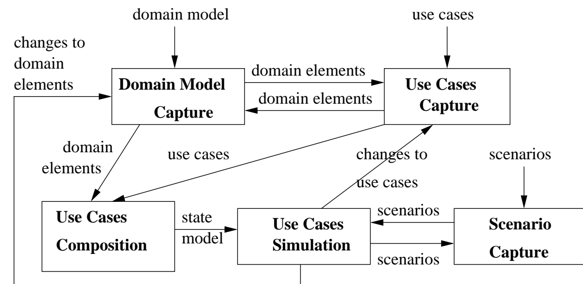

Figure 1: Use cases based requirements engineering process.

which we automatically generate executable state machines using a domain model [7]. The generated state machines are used as proto-

types for requirements validation by simulation [8]. Scenarios are used to automate the simulation process [9].

2.1 Use Cases

A use case model consists of use cases, actors and relationships. A use case diagram is a graphical depiction of a use case model. It shows use case names, actors, relationships between actors and use cases, and relationships between use cases. A relationship between an actor and a use case captures the fact that the actor participates in the use case. Relationships between use cases include the *include* and *extend* relationships. The *include* relationship denotes the inclusion of a use case as a sub-process of another use case (the *base use case*). The *extend* relationship, denotes an extension of a use case as addition of "chunks" of behaviors defined in an *extension use case*. These chunks of behaviors are included at specific places in a base use case called *extension points*. Figure 2 shows an example of use case diagram. The system under consideration is a *Patient Monitoring System* (PM System).

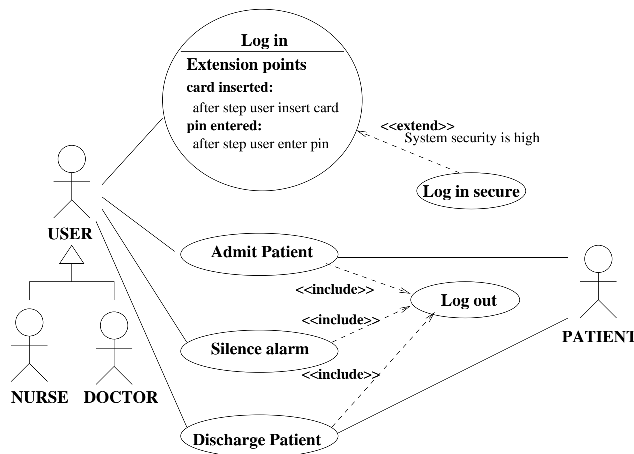

Figure 2: Example of Use Case diagram for a PM System.

Each use case in a use case diagram describes sequences of interactions between the system and its actors. In a previous work [7], we defined an abstract syntax and a concrete syntax for use case description. The abstract syntax is based on Cockburn's template [2] and a restricted form of natural language is used as concrete syntax. Figures 3 and 4 show examples of use case description using our syntax.

Title: Log in

Precondition: System is ON

Steps:
1: User inserts a Card in the card slot
Extension Point: card inserted
2: System asks for Personal Identification Number (PIN)
3: User types PIN
Extension Point: pin entered
4: System validates User identification
5: System displays a welcome message to User
6: System ejects Card

Alternatives: 1a: Card is not regular
1a1: System emits alarm
1a2: System ejects Card
4a: User identification is invalid AND User number of attempts is less than 4
4a1 GOTO Step 2
4b: User identification is invalid AND User number of attempts is equal to 4
4b1: System emits alarm
4b2: System ejects Card

Postcondition: User is logged in

Figure 3: Use case describing a login procedure in a Patient Monitoring System.

Title: Log in secure

Parts: p1 At extension point card inserted
1p1: System logs transaction
p2 At extension point pin entered
1p2: System logs transaction

Figure 4: Extension use case.

The use case is Figure 3 is a *normal use case* while the use case in Figure 4 is an *extension use case*.

Each normal use case includes a *primary*

scenario (or main course of events) and 0 or more *secondary scenarios* that are alternative courses of events to the primary scenario [6]. The primary scenario is described in the section titled *Steps* while the secondary scenarios consist of interactions in the primary scenario followed by behaviors defined in the section titled *Alternatives*.

Formally, we define a normal use case as 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 *Steps* is a tuple [*SCond*, *Oper*, *Alt*, *ExPoint*] with *SCond* a *condition*, *Oper* a *use case operation*, *Alt* a set of *alternatives* starting at this point and *ExPoint* a possibly null extension point. The condition *SCond* if present, is an additional condition that must hold for the step to be possible. A step can have one or more alternatives specifying exceptional behaviors that are possible following the step. The set of use case operations includes *triggers* (actors actions), *system reactions*, *branching statements* (GOTO) and *use case inclusion statements*.

Formally an alternative is a tuple [*AltCond*, *AltSteps*] with *AltCond* a condition that must be true for the alternative to be possible, and *AltSteps* a sequence of alternative steps.

An extension use case includes one or more *parts* that are to be inserted at specific *extension points* in a *base use case*. An extension use case is a tuple [*Title*, *Parts*] with: *Title* as defined previously and *Parts* a set of parts. Each part is a tuple [*ExtPoint*, *Steps*] with *ExtPoint* a reference to an extension point (defined in the UML specification) and *Steps* a sequence of steps. Extension points are defined in the use case diagram and each extension point refers to a step in a base use case. As an example, use case *Log in secure* shown in Figure 4 is an extension of use case *Log in* (a base use case) such that any information provided by a user logging in is recorded.

The use case diagram specifies the *extend* relations between extension use cases and base use cases. Formally an extend relation between a base use case *UCbase* and an extension use case *UCext* is a tuple [*UCbase*, *UCext*, *ExtCond*, *ExtPoints*] where *ExtCond* is an *extend condition* under which the extension can take place, and *ExtPoints* are a set of extension points referred to in the extension use case. For instance, the relation between use case *Log in* and use case *Log in secure* formally corresponds to tuple [*Log in*, *Log in secure*, "System security is high", {card inserted, pin entered}].

2.2 Scenarios

A *scenario* is a sequence of *triggers*, *system reactions*, *guard realizations* and *assertions*. We define a *guard realization* as a condition set to hold at a certain point in a scenario, and an *assertion* as a condition that needs to be true at a certain point in a scenario. Figure 5 shows a scenario related to the *PMSystem*.

```
Scenario: successful login high security
1. Assertion:System is ON
2. Trigger: User inserts a Card in the
            card slot
3. Guard: Security is high
4. Reaction:log transaction
5. Guard: Card is regular
6. Reaction:asks for Personal
            Identification Number
7. Trigger: User types PIN
8. Guard: Security is high
9. Reaction:log transaction
10. Reaction:validate User identification
11. Guard: User identification is valid
12. Reaction:displays a welcome message
             to User
13. Reaction:eject Card
14. Assertion:User is logged in
```

Figure 5: Example of scenario in the PMSystem.

Scenarios are used as input for simulation. An assertion must be valid according to the system's state when evaluated, triggers are provided as input events to the system, reactions specify operations that need to be produced by the system and guards are conditions

that must be enabled for a scenario to proceed. A scenario execution fails when an assertion is not verified or the system doesn't produce a specified reaction.

3 Automated generation of scenarios

In this paper, we extend UCEd simulation capabilities with automated generation of *scenarios*. One objective is automatic validation of use case based requirements. Given a use case, scenario generation is performed in the following three steps: We start by generating a use case control flow graph from the use case, then we extract *path sequences* from the control flow graph, and finally we generate *scenarios* from path sequences.

3.1 Use Case Control Flow Graph

Figure 6 shows a Use Case Control Flow Graph (UCCFG) corresponding to use case *Log in* augmented with extension use case *Log in secure*. A UCCFG is a abstract representation of a use case. It is a directed graph with a *root* node corresponding to the beginning of the use case. The graph might include one or more *leaf* nodes corresponding to the use case ending. For instance node *0* is the root node in Figure 6. The UCCFG includes three leaves corresponding to the end of the use case primary scenario (node *12*) and secondary scenarios (nodes *15* and *19*).

Control flow graphs are obtained from parsing use cases [7, 8]. The parsing process considers <<*include*>> and <<*extends*>> relations. For instance, the UCCFG in Figure 6 considers the extend relation between extension use case *Log in secure* and use case *Log in*. *c1* is the extend condition "System security is high". The control flow graph incorporates behaviors defined by use case *Log in secure* under condition *c1*. Similarly the control flow graph of an included use case would be integrated to the including use case control flow graph.

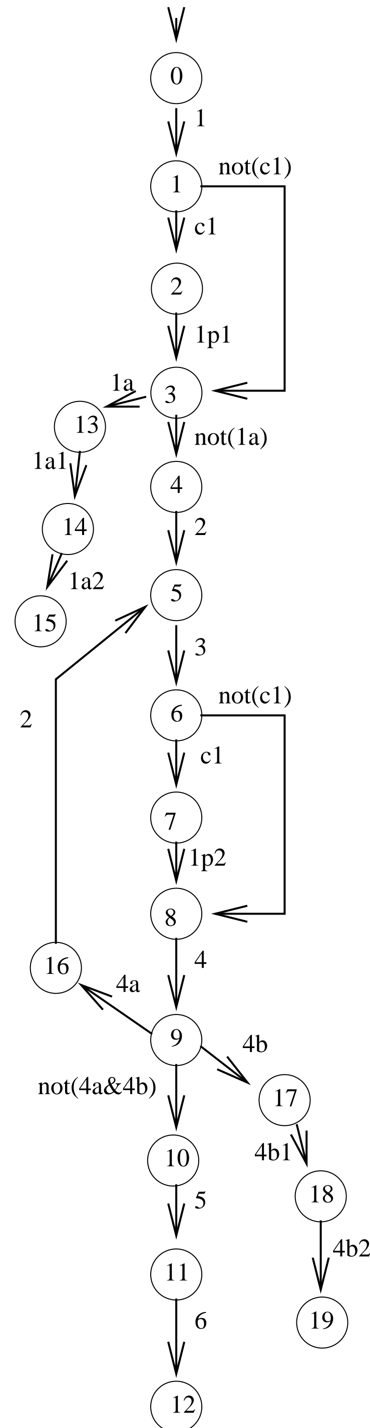

Figure 6: Use Case Control Flow Graph corresponding to use case "Log in" extended by use case "Log in secure". Edges are labeled according to their corresponding *trigger*, *reaction* or *condition* in the use case.

Table 1: Path sequences extracted from the UCCFG in Figure 6

Path		Type
1	$0 \xrightarrow{1} 1 \xrightarrow{not(c1)} 3 \xrightarrow{1a} 13 \xrightarrow{1a1} 14 \xrightarrow{1a2} 15$	alternative path
2	$0 \xrightarrow{1} 1 \xrightarrow{c1} 2 \xrightarrow{1p1} 3 \xrightarrow{1a} 13 \xrightarrow{1a1} 14 \xrightarrow{1a2} 15$	alternative path
3	$0 \xrightarrow{1} 1 \xrightarrow{not(c1)} 3 \xrightarrow{not(1a)} 4 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{not(c1)} 8 \xrightarrow{4} 9 \xrightarrow{not(4a\&4b)} 10 \xrightarrow{5} 11 \xrightarrow{6} 12$	main path
4	$0 \xrightarrow{1} 1 \xrightarrow{c1} 2 \xrightarrow{1p1} 3 \xrightarrow{not(1a)} 4 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{c1} 7 \xrightarrow{1p2} 8 \xrightarrow{4} 9 \xrightarrow{not(4a\&4b)} 10 \xrightarrow{5} 11 \xrightarrow{6} 12$	main path
5	$0 \xrightarrow{1} 1 \xrightarrow{c1} 2 \xrightarrow{1p1} 3 \xrightarrow{not(1a)} 4 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{c1} 7 \xrightarrow{1p2} 8 \xrightarrow{4} 9 \xrightarrow{4b} 17 \xrightarrow{4b1} 18 \xrightarrow{4b2} 19$	alternative path
6	$0 \xrightarrow{1} 1 \xrightarrow{not(c1)} 3 \xrightarrow{not(1a)} 4 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{not(c1)} 8 \xrightarrow{4} 9 \xrightarrow{4b} 17 \xrightarrow{4b1} 18 \xrightarrow{4b2} 19$	alternative path
7	$0 \xrightarrow{1} 1 \xrightarrow{not(c1)} 3 \xrightarrow{not(1a)} 4 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{not(c1)} 8 \xrightarrow{4} 9 \xrightarrow{4a} 16 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{not(c1)} 8 \xrightarrow{4} 9 \xrightarrow{not(4a\&4b)} 10 \xrightarrow{5} 11 \xrightarrow{6} 12$	main path
8	$0 \xrightarrow{1} 1 \xrightarrow{c1} 2 \xrightarrow{1p1} 3 \xrightarrow{not(1a)} 4 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{c1} 7 \xrightarrow{1p2} 8 \xrightarrow{4} 9 \xrightarrow{4a} 16 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{c1} 7 \xrightarrow{1p2} 8 \xrightarrow{4} 9 \xrightarrow{not(4a\&4b)} 10 \xrightarrow{5} 11 \xrightarrow{6} 12$	main path
9	$0 \xrightarrow{1} 1 \xrightarrow{not(c1)} 3 \xrightarrow{not(1a)} 4 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{not(c1)} 8 \xrightarrow{4} 9 \xrightarrow{4a} 16 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{not(c1)} 8 \xrightarrow{4} 9 \xrightarrow{4b} 17 \xrightarrow{4b1} 18 \xrightarrow{4b2} 19$	alternative path
10	$0 \xrightarrow{1} 1 \xrightarrow{c1} 2 \xrightarrow{1p1} 3 \xrightarrow{not(1a)} 4 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{c1} 7 \xrightarrow{1p2} 8 \xrightarrow{4} 9 \xrightarrow{4a} 16 \xrightarrow{2} 5 \xrightarrow{3} 6 \xrightarrow{c1} 7 \xrightarrow{1p2} 8 \xrightarrow{4} 9 \xrightarrow{4b} 17 \xrightarrow{4b1} 18 \xrightarrow{4b2} 19$	alternative path

3.2 Path sequences generation

A path sequence is a sequence of nodes and edges $n_0 \xrightarrow{ev1} n_1 \cdots \xrightarrow{evi} n_i$ starting from the root of a use case control flow graph. We define a *complete* path sequence as one that ends in a leaf node. Path sequences are generated by a *depth first* traversal of a use case control flow graph. Figure 7 shows the basic traversal algorithm. Path sequences are generated according to a repetition limit l, such that a node does not appear in a path more than l times. Table 1 shows the path sequences generated from the UCCFG in Figure 6, with repetition limit *2*. Notice that some paths are prevented because of conditions. For instance a path with condition *c1* true can not be such that *c1* is subsequently false because *extend conditions* are set for a whole use case execution according to use cases semantics.

3.3 Scenario inference from path sequences

Scenarios are generated from a selection of path sequences according to a given *coverage objective*. We distinguish the following three coverage objectives.

- All-nodes coverage achieved with a set of complete path sequences such that each node in the UCCFG appears at least once. For instance, the set of paths {2, 4, 5, 8} satisfies all-nodes coverage.
- All-edges coverage achieved with a set of complete path sequences such that each edge in the UCCFG appears at least once. For instance, the set of paths {1, 4, 6, 7} satisfies all-edges coverage.
- All-complete-paths with repetition limit n coverage. For instance all paths in Table 1 are needed to satisfy all-complete-paths with repetition limit 2 coverage.

Selected paths are directly mapped to scenarios according to the following.

Generate_path_sequences(g: UCCFG,l: integer) *returns* a set of path sequences

A1. $Allpaths = \emptyset$, $path = \emptyset$, let R be graph g root node

A2. Traverse(R,l, $path$, $Allpaths$)

A3. return $Allpaths$

Traverse(n: node, l: integer, p: path, All: set of paths)

B1. IF n is a *leaf* node, $All = All \cup \{p\}$

B2. FOR EACH edge $n \xrightarrow{e} n_i$ starting from n

B2.1. IF n_i doesn't appear in p at least l times

B2.1.1. $p = p \xrightarrow{e} n_i$

B2.1.2. Traverse(n_i, l, p, All)

Figure 7: Path sequence generation algorithm.

1. Each scenario starts with an assertion corresponding to the use case pre-condition.

2. Each trigger, system reaction or condition in a path sequence corresponds respectively to a trigger, system reaction or guard realization in the resulting scenario.

3. A scenario for an alternative path is completed with an assertion corresponding to the negation of the use case post-condition.

4. A scenario for a main course path is completed with an assertion corresponding to the use case post-condition.

The scenario in Figure 5 is obtained from the main path sequence *8*. Generated scenarios are used in our use cases based requirements engineering approach to automate specification simulation and validation.

3.4 From scenario to test cases

Scenarios can be used as basis for test cases construction. For instance the partial test case in Figure 8 is obtained from the scenario in Figure 5. A partial test case is created by providing a setup such that the initial assertion

Test setup: *System is ON*, *Security is high*, *Card is regular*, *User identification is valid*

Test sequence

Input: *insert Card*

Expected Output: *log transaction, asks for Personal Identification Number*

Input: *type PIN*

Expected Output: *log transaction, validate User identification, display welcome message, eject Card*

Expected end outcome: *User is logged in*

Figure 8: Partial test case corresponding to the scenario in Figure 5.

as well as all guards are verified. Inputs correspond to trigger events. Expected outputs correspond to system reactions and the test case final expected end outcome corresponds to the last assertion of the scenario. Prior to execution, partial test cases need to be completed by providing concrete values for the setup. As an example, a complete test case corresponding to the partial test case in Figure 8 would specify a concrete *Card* and corresponding *PIN*.

4 Related work

In [5], Ryser and Glinz propose a method for Scenario-Based Validation and Test of Software (SCENT). In this approach, use cases are first formalized into state machines. These state machines are then annotated with information such as pre-conditions, data and non-functional requirements. Finally test cases are generated by path traversal of the state machines. The SCENT approach introduces dependency charts to capture dependencies among use cases and generate system wide test cases. Another approach where use cases are transformed to state machines and then test cases are generated from these state machines

is presented in [3]. In this approach, test case generation is seen as a planning problem and a planning tool is used for test generation from state machines. In [1], Briand and Labiche present a methodology for system testing based on UML models such as use case models, interaction diagrams, class diagrams, and OCL. Use cases are refined as UML interaction diagrams and UML activity diagrams are used to capture use case sequential dependencies.

A key difference between our approach and the above approaches is that we propose automated generation of test scenarios from the original use cases in text form. In [5, 3] state machines are first derived from use cases while in [1], scenarios represented as UML interaction diagrams need to be derived from the use cases. The derivation of state machines or interaction diagrams from use cases involves some design decision. Our objective is to derive system level tests as much as possible from requirements models. In fact, a primary goal is to test state machines derived from use cases.

5 Conclusions

In this paper, we presented an approach for the generation of test scenarios from use cases. Our primary objective is to automatically produce test scenarios for the validation of state machines derived from use cases [9]. Test scenarios produced can also be used as a basis for the production of complete test cases by providing details such as concrete data values. We use three basic coverage criteria for test scenario generation. We are exploring other coverage criteria and looking for ways to automate scenario selection based on these criteria.

A limitation of the current approach is that we only generate *positive* scenarios. While positive scenarios give a good indication on how well an implementation conforms to requirements, *negative* scenarios may be needed to check constraints violation [9]. We are looking for a way to enhance our approach with negative scenarios generation. Another of our current work involves test cases finalization.

References

[1] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. *Software and System Modeling*, 1(1):10–42, 2002.

[2] A. Cockburn. *Writing Effective Use Cases.* Addison Wesley, 2001.

[3] Peter Fröhlich and Johannes Link. Automated test case generation from dynamic models. In Elisa Bertino, editor, *ECOOP*, volume 1850 of *Lecture Notes in Computer Science*, pages 472–492. Springer, 2000.

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

[5] J. Ryser and M. Glinz. A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts. In *Proc. 12th International Conference on Software and Systems Engineering and their Applications*, Dec 1999.

[6] Geri Schneider and Jason P. Winters. *Applying Use Cases a practical guide.* Addison-Wesley, 1998.

[7] 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.

[8] S. Somé. Supporting use cases based requirements simulation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'04)*, volume I, pages 381–386, june 2004.

[9] S. Somé. Enhancement of a Use Cases based Requirements Engineering approach with Scenarios. In *Proceedings of the 12th Asia Pacific Software Engineering Conference (APSEC 2005)*, Dec 2005.