Beyond Scenarios: Generating State Models from Use Cases*

Stéphane S. Somé
School of Information Technology and Engineering (SITE)
University of Ottawa
150 Louis Pasteur, P.O. Box 450, Stn. A
Ottawa, Ontario, K1N 6N5, Canada
ssome@site.uottawa.ca

ABSTRACT

A Use Case is a textual representation of requirements consisting of related scenarios. This paper discusses state models generation from Use Cases. We propose a formalization of use cases, a natural language based syntax for use cases description, and an algorithm that incrementally composes a set of use cases as a finite state transition machine.

1. INTRODUCTION

The terms "use case" and "scenario" are often considered synonymous. However, there is a fundamental difference between the two artifacts. A use case is a collection of possible scenarios between the system under discussion and external actors [3], while a scenario is a linear sequence of interactions between external actors and a system. In the Unified Modeling Language (UML) [5], a use case is "the specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system". Typically each 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. In software development approaches such as the Unified Software Development Process [4], users requirements are first captured as use cases that are refined afterward into scenarios. The scenarios are usually represented graphically as Sequence Diagrams or Message Sequence Charts (MSCs).

In the recent years lot of progresses have been made on automatic generation of state models from scenarios (reviews on the subject are presented in [9, 1, 2]). These progresses bring us one step closer to bridging the gap between requirements and the design process. However, most of the scenario-based approaches start from sequence diagram or MSCs. Supposing a manual derivation of scenarios from use cases.

In this paper, we discuss the generation of state transition graphs from use cases. Because state transition models are executable, one of the possible applications of our work is early simulation of users requirements. Simulation helps uncovering hidden behaviors and making requirement changes happen earlier in the development process. Other applications include verification of use cases and conformance validation of the early requirement model with models derived at later stages from refinements of uses cases.

The informal nature of use cases is a problem with our proposed approach. Automatic generation of specification from *free form* use cases resumes to the analysis and understanding of unconstrained natural language; a task that is virtually impossible. Fortunately much of the use cases based approaches advocate the use of *restricted forms* of natural language [7, 3]. As an example, according to [3], action steps in use cases should: (1) be sentences in the present tense, (2) with an active verb in the active voice, and (3) describing an actor successfully achieving a goal that moves the process forward. We propose a restricted form of natural language, which in combination with a dedicated Use Case Editor (see section 4) helps writing textual use cases in natural language that can still be automatically analyzed with little effort. The use cases in our approach have formal semantics with a front-end language based on a restricted form of English.

The rest of the paper is organized as follow. We present our notation and formalization of use cases in the next section. Section 3 presents an approach for the generation of state transition models from use cases. Our work is being implemented in a tool called Use Case Editor (UCEd). We briefly outline this tool in section 4. Finally section 5 concludes the paper.

2. USE CASES

Figure 1 shows an example of use case. The format used is from [3]. A use case includes sections such as title, scope, level, etc. The description of these different parts is beyond the scope of this paper.

The use case in Figure 1 describes a login procedure that must be used by *doctors* and *nurses* for a *Patient Monitoring System* (PM System). Although all the parts in a use case help requirements documentation, for the purpose of state model generation, we are interested in only the functional

*This work is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Title: User login

Scope: Patient Monitoring System

Level: Sub function

Primary Actor: User (e.g. Doctor, Nurse)

Participants: User

Goal: A User wants to use a PM system to perform an activity such as admitting a patient or changing system parameters.

Precondition: System is ON AND NO user is logged in AND NO card is inserted

Steps:
1: User inserts a Card
2: System asks for Personal Identification Number (PIN)
3: User types PIN
4: System validates USER identification
5: System displays a welcome message to USER
6: System ejects Card

Exceptions:
1a: Card is not regular
1a1: System emits alarm
1a2: System ejects Card
4a: User identification is invalid
4a1: IF number of attempts < 4
4a11 Go back to Step 2
4a2: IF number of attempts ≥ 4
4a21: System emits alarm
4a22: System ejects Card

Postcondition: User is logged in

Figure 1: Use case describing a login procedure in a Patient Monitoring System.

description aspect of use cases defined by the preconditions, the steps, the exceptions and the postcondition. We also consider use case titles for traceability. In the following, we restrict our view of use cases on these parts.

2.1 Elements of use cases

A use case can be seen as a tuple [*Title*, *Pre*, *Steps*, *Post*] with *Title* a label that uniquely identifies a use case, *Pre* a set of preconditions, *Steps* an ordered set of steps, and *Post* a set of postconditions.

Each step in *Steps* is a tuple [*SCond*, *Oper*, *Ext*] with *SCond* a set of conditions, *Oper* an operation, and *Ext* a set of extensions starting at this point.

The conditions in *SCond* are additional conditions that must hold for the step to be possible. An operation *Oper* is an actor action or a system response. An operation may also be a branching statement to a step in the use case as in step *4a1* of the use case *User login*.

An extension defines alternative behaviors that are possible following a step. An extension is a tuple [*ExCond*, *ExSteps*] with *ExCond* set of the extension conditions, and *ExSteps* an ordered set of extension steps. Each extension step is a tuple [*ExStepCond*, *ExStepOper*] with *ExStepCond* a set of conditions and *ExStepOper* an operation (actor action or system reaction).

Formally, a use case is a graph of behavior sequences where each node represents a state and links correspond to operations. Figure shows the graph of behavior sequences corresponding to the use case *User login*.

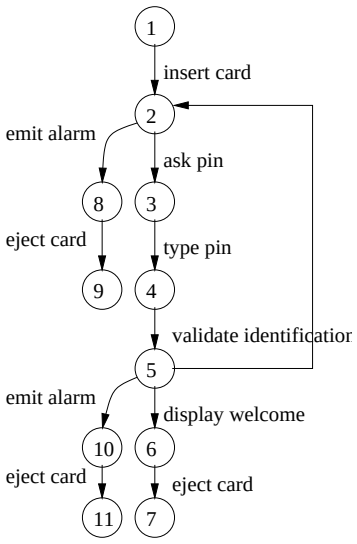


Figure 2: Graph of behavior sequence corresponding to use case User login. The graph corresponds to a network of connected scenarios.

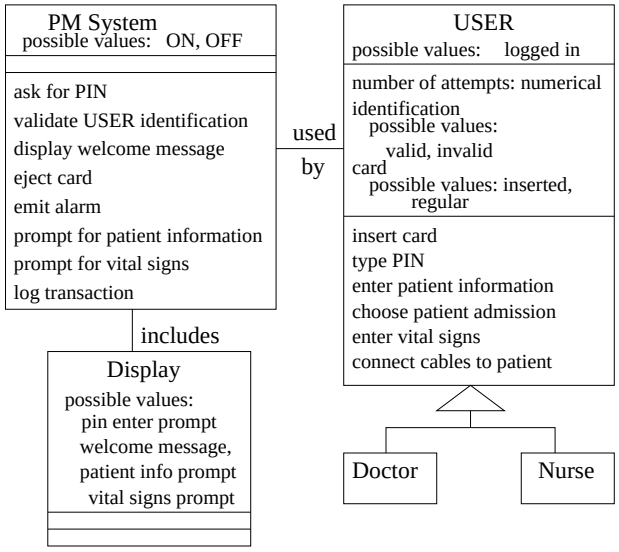
2.2 Domain model

A *domain model* is an integral part of requirements in approaches such as the Unified Software Development Process. We use a domain model as a complementary part of use cases description. Preconditions, postconditions and other conditions in use cases are predicates on elements from the domain model. The application domain model also formalizes the operations referred in use cases (actor actions and system responses).

A domain model is a *high-level class model* that captures the most important types of objects in the context of the system. The domain concepts include the system as a black box with the "things" that exist or events that transpire in environment in which the system works [4]. In the PM system, domain concepts include the PM system and User (a generalization of Doctor and Nurse). These concepts in turn may have sub-concepts, attributes and operations.

Figure 3 shows a graphical representation of the PM system domain model using the UML notation. In the UML, the description of each class includes attributes, associations and operations. We extend this description as follow.

- Domain entities (concepts and sub-concepts) and their attributes have possible values. We distinguish between discrete and non-discrete entities (such as numerical ones). The domain model enumerates all the possible values of discrete entities.



Operation: ask for PIN
  added-conditions: Display is pin enter prompt
Operation: validate USER identification
  added-conditions: USER identification is valid
    OR USER identification is invalid
  withdrawn-conditions: Display is pin enter prompt
Operation: display welcome message
  added-conditions: Display is welcome message,
    USER is logged in
Operation: eject card
  added-conditions: NOT Card is inserted
  withdrawn-conditions: all conditions on Card
Operation: prompt for patient information
  added-conditions: Display is patient prompt info
Operation: prompt for vital signs
  added-conditions: Display is vital sign prompt
Operation: insert card
  added-conditions: USER card is inserted

Figure 3: Partial representation of the PM system domain model.

- We specify operations by giving a set of *effects* consisting of *withdrawn-conditions* and *added-conditions*. Withdrawn-conditions are conditions that are removed after the operation execution, while added-conditions are conditions that are known to hold after the operation execution. These conditions are expressed as predicates on the domain entities. We use the same natural language form of description for conditions in the use cases and the domain model. Withdrawn-conditions and added-conditions could be expressed in OCL [5]. However, the natural language form provides a single simple notation in both use cases and the domain model.

The domain model serves two purposes: it is a *knowledge base* for natural language analysis of use cases, and a basis for state model generation from use cases.

2.3 Natural language in use cases

We use a form of natural language for conditions and operations. Conditions describe *situations* prevailing within a system and environment. Operations are active sentences in which a component performs an action given as a verb. Another component may be included in the sentence as the one affected by the operation. We briefly present a part of the syntax used for conditions in the rest of this section.

A condition is written as a *predicative phrase*, seeking a certain quality on an entity of the domain model. As an example the precondition of the above use case is a clause where the system has the quality of being *ON*. Figure 4 shows an excerpt of a Definite Clause Grammar (DCG) [6] for conditions. DCGs are contextual grammars used for natural language description. The context in a DCG is defined as predicates that are checked with the grammar productions. The domain model provides the context in our case.

condition	$\longmapsto$	pred_phrase
condition	$\longmapsto$	pred_phrase, conj, condition
condition	$\longmapsto$	negation, condition
pred_phrase	$\longmapsto$	noun_phrase(N), verb, value(N)
noun_phrase(C)	$\longmapsto$	determinant, [C] {*concept(C)*}
noun_phrase([C,A])	$\longmapsto$	determinant, [C], [A], {*concept_attribute(C,A)*}
value(N)	$\longmapsto$	determinant [A], {*discrete(N)*, *possible_value(N,A)*}
value(N)	$\longmapsto$	comparison {*not(discrete(N))*}
conj	$\longmapsto$	[AND]|[OR]
verb	$\longmapsto$	{*derived_from(be)*}
verb	$\longmapsto$	{*derived_from(become)*}

Figure 4: A partial DCG for conditions.

The DCG in Figure 4 references the domain model through the predicates *concept*, *concept_attribute*, *discrete*, and *possible_value*. The domain model definition is mapped into these predicates. As an example, some of predicates corresponding to the model in Figure 3 are *concept*("User"), *concept*("PM System"), *concept_attribute*("User", "number of attempts"), *concept_attribute*("User", "Card"), *discrete*("Card"), *possible_value*(["User", "Card"], "inserted"), *possible_value*(["User", "Card"], "regular").

3. FROM USE CASES TO STATE MODELS

Based on our formalization, we have developed an algorithm to generate a hierarchical type of finite state transition machines from use cases. The algorithm is an adaptation of [8]. It is based on the following.

- Each state is defined by *characteristic predicates* which hold in it. These predicates are formulated on the domain model entities.

  Two states are *identical* if they have the same characteristic predicates.

- A state s_b is a *sub-state* of a state s_a (its *sup-state*), if its characteristic predicates include those of s_a in the logical sense.

  Any transition going from a state s also applies to all *sub-state* of s.

For each use case, we augment a state transition graph with states and transitions such that the graph of behavior sequence corresponding to the use case is included as state transition sequences in the state transition graph. We use

the operations effects to determine states. Suppose "-" is an operator such that C_1 and C_2 being 2 sets of predicates, $C_1 - C_2$ is a set obtained by removing all the predicates in C_2 from C_1, and $C_1 + C_2$ is a set obtained by adding all the predicates in C_2 to C_1. Given a state s such that *pred(s)* are the characteristic predicates of s, the execution of operation *op* with added-condition *add_conds(op)* and withdrawn conditions *withdr_conds(op)* produces a state s' such that *pred(s')* = (*pred(s)* - *withdr_conds(op)*) + *add_conds(op)*.

A finite state transition machine is a tuple [Σ, S, F, $S0$] where: Σ is a finite alphabet, S is a finite set of states, F is a transition function, and $S0 \subseteq S$ is a set of initial states. F is defined as $S \times \Sigma \times S$. Given a use case [*Title*, *Pre*, *Steps*, *Post*], the algorithm enriches a state transition machine M as follow. Before the first use case composition, M is initially such that $\Sigma = S = F = \emptyset$.

1 Let s be a state such that *pred(s)* = *Pre*

2 For each step = [*SCond*, *Oper*, *Ext*] in *Steps*

  2.1 Let t be a state such that *pred(t)* = *pred(s)* + *SCond* (t is either *identical* to s or is a *sub-state* of s).

  2.2 If *Oper* is an actor action or a system response, let u be a state obtained by executing *Oper* from state t, add a transition $t \times Oper \times u$ to F, and add *Oper* to Σ

  If *Oper* is a branching to step i, let u be the state from which step i was considered, add a transition $t \times \{\} \times u$ to F

  2.3 For each extension *ext* = [*ExCond*, *ExSteps*] in *Ext*, let u' be a state such that *pred(u')* = *pred(u)* + *ExCond*

  For each extension step *exst* = [*ExStepCond*, *ExStepOper*] in *ExCond*

    2.4.1 Let v be the state such that *pred(v)* = *pred(u')* + *ExStepCond*

    2.4.2 add a transition corresponding to *ExStepOper* the same way as in steps 2.2, suppose v' the resulting state, $u' = v'$

  2.5 $s = u$

3 Add any new state to S

Figure 5 shows the finite state transition machine corresponding to the use case *User login*. State *S0* characteristic predicates are the use case preconditions {*"System is ON"*, *"No user is logged in"*, *"No card is inserted"*}. State *S1* is obtained by considering the use case step 1. It is characterized by the set of predicates {*"System is ON"*, *"No user is logged in"*, *"Card is inserted"*} since the operation *"insert Card"* adds the predicate *"Card is inserted"*, replacing the previous predicate *"No card is inserted"*. The algorithm generates state *S2* when adding the extension of step 1 *1a*. *S2* is a *sub-state* of *S1* because of *1a extension condition "Card is not regular"*. The state *S2* characteristic predicates are {*"System is ON"*, *"No user logged in"*, *"Card is inserted"*, *"Card is not regular"*}. The extension step *1a1* creates a transition that loops back to state *S2*. This is due to the fact that the operation *"emit alarm"* has no effects according to the domain model shown in Figure 3. Therefore, the resulting state obtained by considering this

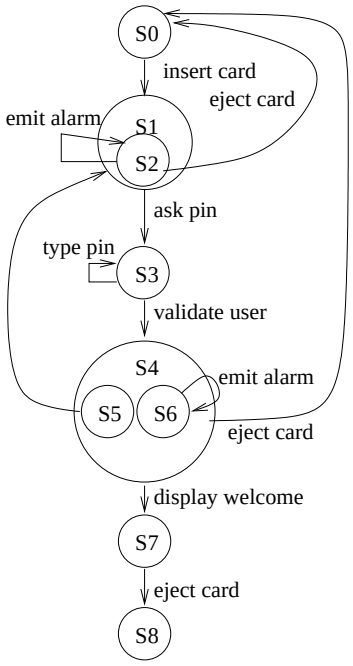


Figure 5: Finite state transition machine generated from use case User login.

operation is identical to the originating state. The extension step *1a2* operation *"eject card"* withdraws all the predicates on *Card* and add the predicate *"No Card is inserted"* to *S2* characteristic predicates, resulting in the characteristic predicates {*"System is ON"*, *"No user is logged in"*, *"No Card is inserted"*} of state *S0*. The remaining states are as follow. State *S3* characteristic predicates are {*"System is ON"*, *"No user is logged in"*, *"Card is inserted"*, *"Display is pin enter prompt"*}. State *S4* is characterized by {*"System is ON"*, *"No user is logged in"*, *"Card is inserted"*, *"USER identification is valid OR USER identification is invalid"*}. States *S5* and *S6* are sub-states of *S4* because their characteristic predicates logically include those of *S4*. *S5* characteristic predicates are {*"System is ON"*, *"No user is logged in"*, *"Card is inserted"*, *"USER identification is invalid"*, *"number of attempts* < 4*"*}, while *S6* characteristic predicates are {*"System is ON"*, *"No user is logged in"*, *"Card is inserted"*, *"USER identification is invalid"*, *"number of attempts* ≥ 4*"*}. States *S7* characteristic predicates are {*"System is ON"*, *"USER is logged in"*, *"Card is inserted"*, *"USER identification is valid"*, *"Display is welcome message"*}. Finally states *S8* is characterized by {*"System is ON"*, *"USER is logged in"*, *"No Card is inserted"*, *"USER identification is valid"*, *"Display is welcome message"*}.

We do not directly use use cases postconditions for finite state machine generation. Postconditions are rather used for verification. The postconditions of a use case should be included in the characteristic predicates of the last state corresponding to the use case main course of events. In the above example, state *S8* characteristic predicates effectively includes the postcondition of the use case which is *"USER is logged in"*.

Our algorithm supports overlapping and connected use cases. Suppose the use case *Admit patient* of the PM System

shown in Figure 6. *Admit patient* is supposed to follow the

Use case name: Admit patient

Actor: User (e.g. doctor, nurse)

Participants: Patient

Goal: A User wants to perform the admission of a patient on a Monitor.

Precondition: User is logged in AND NOT Patient is admitted to the Monitor

Steps:
1: User chooses patient admission
2: System prompts for patient information
3: User enters patient information
4: System prompts for vital signs
5: User enters vital signs
6: User connects cables to the Patient
7: System logs transaction

Postcondition: Patient is admitted to the Monitor

Figure 6: Use case describing a patient admission procedure to the Patient Monitoring System.

use case *User login*. The algorithm adds *Admit patient* from sub-states of states *S7* and *S8* since the predicate *"User is logged in"* holds in both these states.

4. USE CASE EDITOR

We are implementing our results in a Use Case Editor (UCEd). The objective of UCEd is to help use cases acquisition, use cases verification, prototype generation, and simulation. UCEd will include a Use Case Writing Tool, a Domain Model Editor, a Use Case Composition Module and a Use Case Simulator.

The Use Case Writing Tool shown in Figure 7 is a graphical interface for the edition of use cases. The Use Case Writing Tool allows use cases to be entered in a field-oriented form. Use cases elements are written in a restricted form of natural language according to the syntax outlined in section 2.3. The benefit of a field-oriented form is that use case writers do not have to worry about delimiting the different parts of their use cases. The Domain Model Editor allows users to describe application domain models as in section 2.2. Since we use the UML notation, one of our plans is to provide a UXF (UML eXchange Format) based interface for interoperability with UML diagramming tools. UCEd checks use cases and the domain model against each other and reports inconsistencies and omissions.

The Use Case Composition Module implements the finite state transition machine generation algorithm presented in Section 3. The state models generated can be used as prototypes and animated with the Use Case Simulator.

5. CONCLUSION

We propose generating state transition models from use cases by looking at a use case as a network of connected scenarios. Our work aims at helping use cases based requirements engineering with a framework as well as a tool for

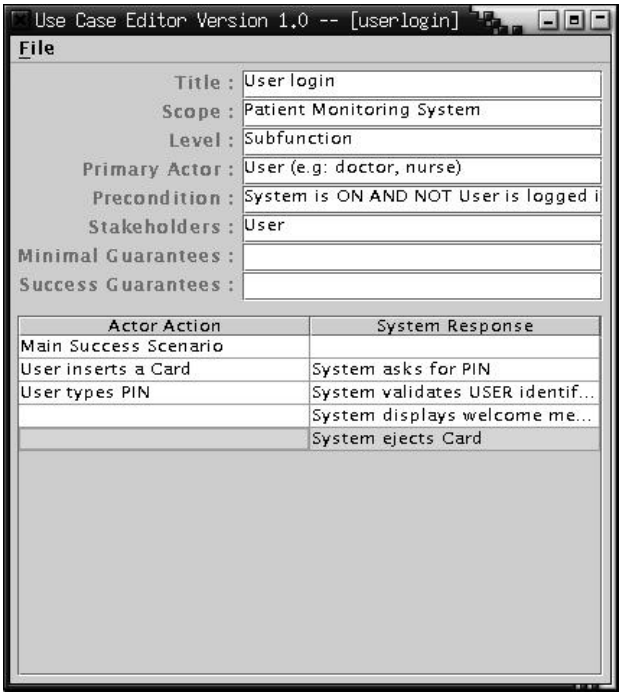


Figure 7: UCEd Use Case Writing Tool.

use case edition, clarification, and early prototyping. This paper presented an ongoing work. We plan to extend the scenario notation and support other aspects of uses cases such as sub use cases and extension of use cases.

6. REFERENCES

[1] C. B. Achour and C. Souveyet. Bridging the gap between users and requirements engineering the scenario-based approach. Technical Report 99-07, Cooperative Requirements Engineering With Scenarios (CREWS), 1999.

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

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

[4] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1998.

[5] OMG. Unified modeling language specification, 1999.

[6] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis–a survey of the formalism and comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[7] C. Rolland and C. B. Achour. Guiding the construction of textual use case specifications. *Data & Knowledge Engineering Journal*, 25(1-2):125–160, march 1998.

[8] S. Somé and R. Dssouli. An Enhancement of Timed Automata generation from Timed Scenarios using Grouped States. *Electronic Journal on Network and Distributed Processing (EJNDP)*, (6), 1998.

[9] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenario Usage in System Development: A Report on Current Practice. *IEEE Software*, March 1998.