An approach for the synthesis of State transition graphs from Use Cases*

Stéphane S. Somé
School of Information Technology and Engineering (SITE)
University of Ottawa
Ottawa, Ontario, K1N 6N5, Canada

Abstract *Use Cases describe possible interactions involving a system and its environment. They are widely used to represent user's requirements as a basis for software development. 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.*

Keywords: Requirements engineering, Use Cases, UML, State machines

1 Introduction

A use case partially describes a system behavior without revealing its internal structure. As such use cases are useful to capture and document the external requirements of systems. Several software development approaches including the Unified Software Development Process [3] recommend use cases for users' requirements description. The partial nature of use cases offers lot of benefits for requirements engineering. It allows several users with different views of a same system to provide different but possibly overlapping use cases describing its behavior. The partial nature also helps developing a system by incremental addition of services. A problem however, is that it is often difficult to visualize the global behavior resulting from the combination of the use cases. Moreover, separately defined use cases may be inconsistent one with the other and the set of use cases may be incomplete. A solution consists of deriving a specification integrating all the related use cases, such that the system global behavior can be examined, and verified for inconsistencies as well as incompleteness.

In this paper, we present an approach for the integration of related use cases into a global specification. We use finite state transition machines as a model for use cases composition and present an algorithm for the generation of finite state transition machines from use cases. Because finite state transition machines are executable, one of the possible applications of our work is early simulation of users requirements. 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.

Use cases are generally written informally using a natural language. This informal nature of use cases is a problem with our proposed approach. Automatic generation of specification from *free form* use cases would require analysis and understanding of unconstrained natural language; a task that is virtually impossible. To overcome this difficulty, the use cases in our approach have formal semantics with a front-end language based on a restricted form of English. The application domain model is used as a lexicon for use cases analysis.

Our approach is rooted in the UML. The domain model is a UML class diagram and we assume the use cases definition and semantics of

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

the Object Management Group (OMG) UML specification [7]. The UML seems appealing because of the great acceptance it has gained among software developers and tool vendors. An advantage of using UML is the possibility of integration of our approach to the various existing UML based methodologies and tools.

In the rest of this paper, we first present our notation and formalization of use cases. Section 3 then presents an approach for the generation of state transition models from use cases. Finally section 4 concludes the paper.

2 Use Cases

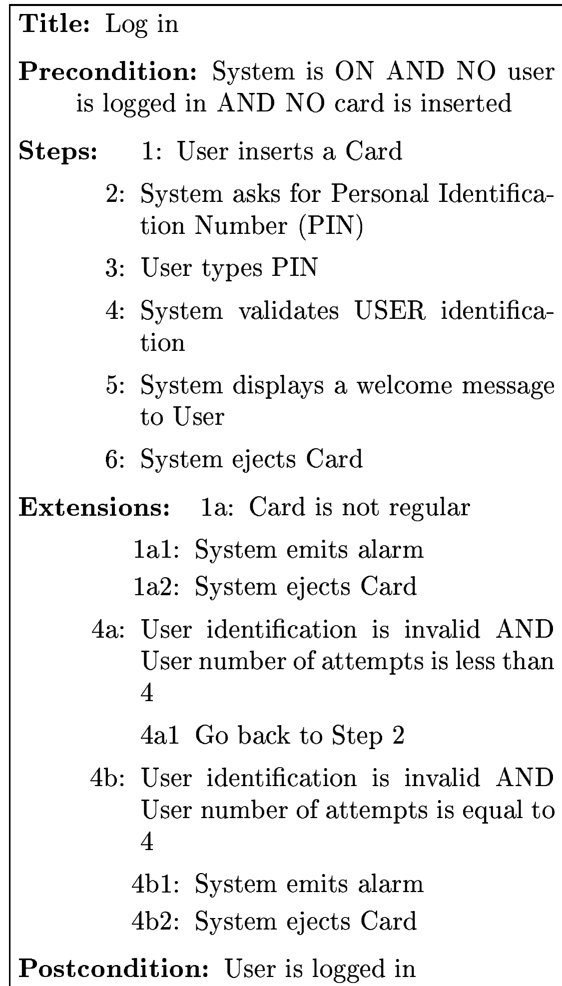
A use case consists of intertwined *scenarios* [4, 10, 1], each scenario being a possible *sequence of interactions*. Use cases are developed by adding secondary scenarios to a primary scenario. The primary scenario captures the "normal" or "most common" behavior. It is written as if everything goes right without any error. A secondary scenario describes an alternative outcome that may result from an error. Each secondary scenario is written by defining diverging behaviors from a specific point of a primary scenario.

Figure 1 shows an example of use case. The format used is inspired from [1]. This use case describes a login procedure that must be used by *doctors* and *nurses* for a *Patient Monitoring System* (PM System). 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 *Extensions*.

2.1 An abstract syntax for use cases

Figure 2 shows an UML notation of an abstract syntax for use cases. A use case can be seen as a tuple [*Title*, *Precondition*, *Steps*, *Postcondition*] with: *Title* a label that uniquely identifies the use case, *Precondition* a *condition*[1] that must be true before an instance of the

[1]The term *constraints* is used in the UML specification to refer to conditions.



Title: Log in

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

Extensions:
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 Go back to 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 1: Use case describing a login procedure in a Patient Monitoring System.

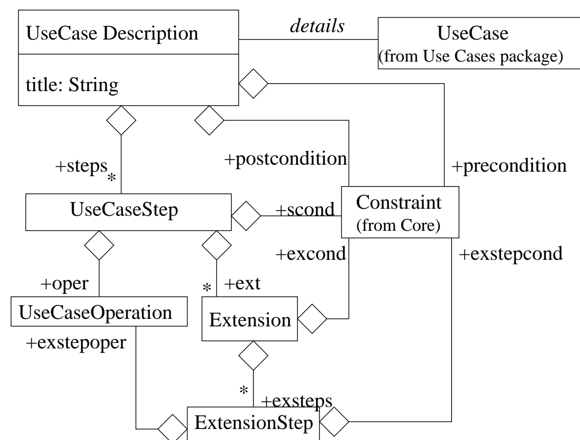


Figure 2: Abstract syntax for use cases description.

use case can be executed, *Steps* an ordered set of steps, and *Postcondition* a condition that must be true at the end of an instance of the use case execution. Each step in *Steps* is a tuple [*SCond*, *Oper*, *Ext*] with *SCond* a condition, *Oper* a *use case operation*, and *Ext* a set of extensions starting at this point. *SCond* is an additional condition that must hold for the step to be possible. A step can have one or more extensions specifying alternative behaviors that are possible following the step.

2.2 Natural language representation of 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 3 shows an excerpt of a Definite Clause Grammar (DCG) [8] 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. A domain model provides the context in our case. 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 the environment in which the system works [3]. 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. We use UML class diagrams [9] extended with *stereotypes* to describe domain models. Figure 4 shows a graphical representation of the PM system domain model using the UML notation.

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

Figure 3: A partial DCG for conditions.

The DCG in Figure 3 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 4 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 [11]. It is based on the following.

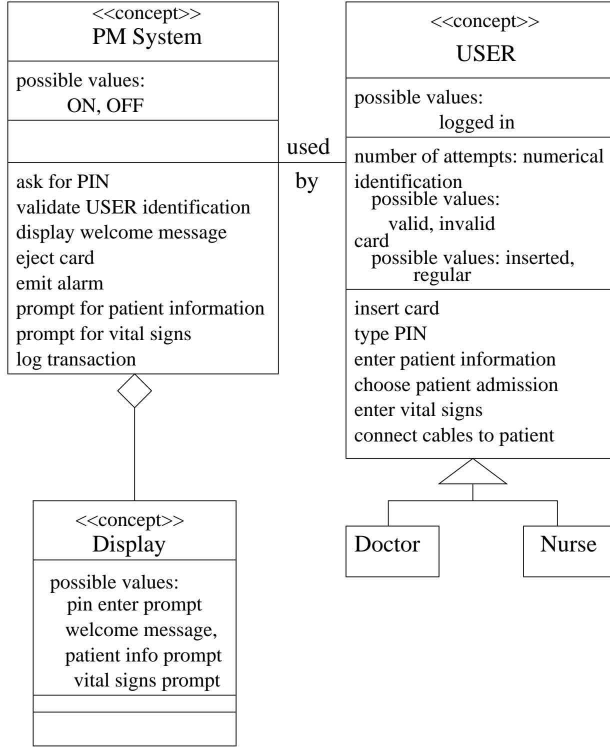


Figure 4: Partial representation of the PM system domain model.

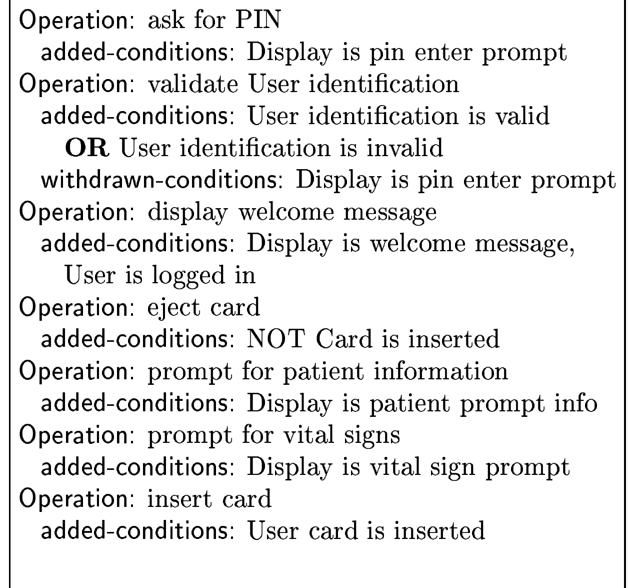

Operation: ask for PIN
added-conditions: Display is pin enter prompt
Operation: validate User identification
added-conditions: User identification is valid
OR User identification is invalid
withdrawn-conditions: Display is pin enter prompt
Operation: display welcome message
added-conditions: Display is welcome message, User is logged in
Operation: eject card
added-conditions: NOT Card is inserted
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 5: Description of the PM System domain model operations.

- Operations can have *withdrawn-conditions* and *added-conditions*. These conditions are expressed as predicates on the domain entities. As an example, Figure 5 shows an excerpt of the added-conditions and withdrawn-conditions of some of the operation in the PM System domain model.

- 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 behavior sequences corresponding to the use case is included as state transition sequences in the state transition graph. We use the operations withdrawn and added-conditions 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 6 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 4. Therefore, the resulting state obtained by considering this operation is

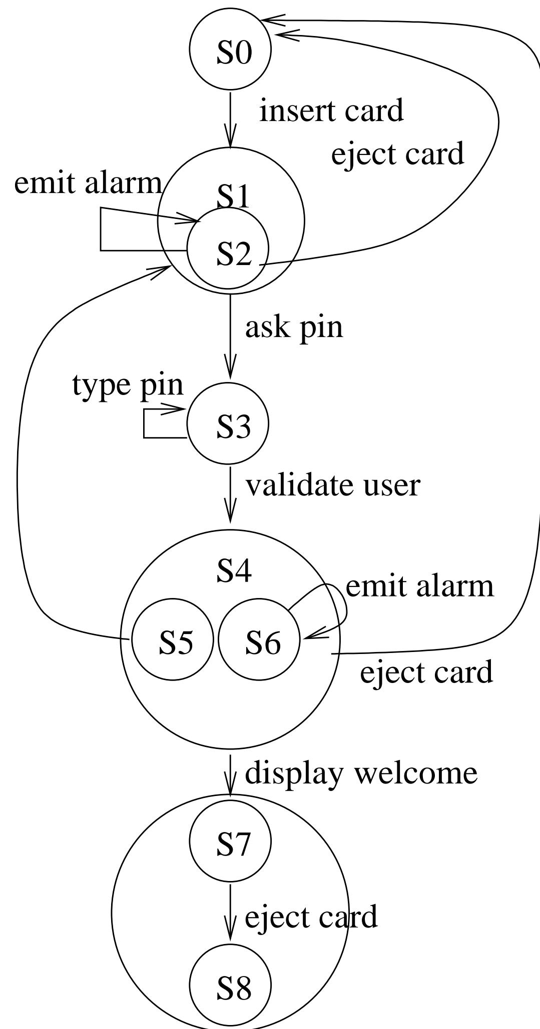
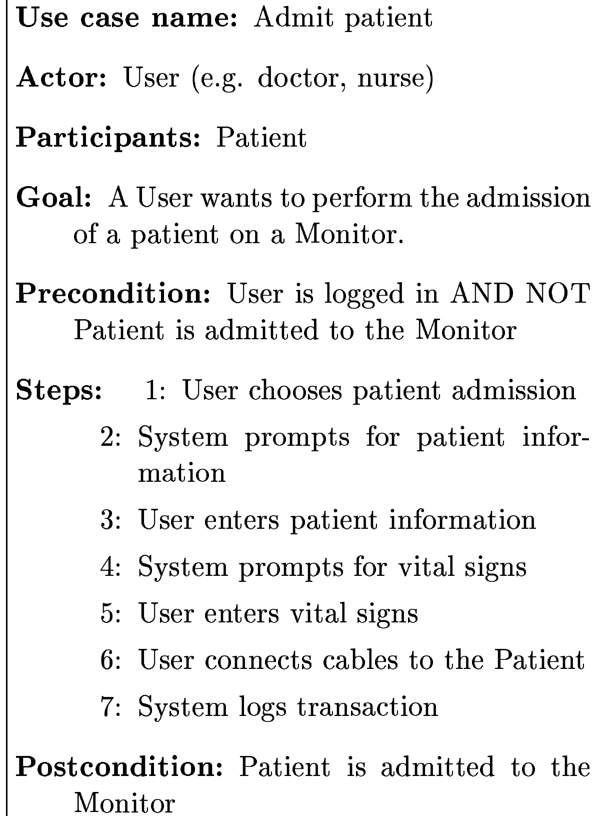


Figure 6: Finite state transition machine generated from use case User login.

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. We consider postconditions as *contractual* statements of *guarantees* at end of the successful execution of a use case (the primary scenario). Therefore, 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 7. *Admit patient* is supposed to follow the 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.



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 7: Admit patient use case.

4 Conclusion

This paper presented an approach for the generation of state transition models from use cases. A number of similar contributions discuss finite state machines synthesis from scenarios [5, 12, 2, 11, 6, 13]. The main difference between these approaches and ours is that they deal with scenarios. A scenario is a single linear sequence of interactions between external actors and a system, while use cases integrate set of scenarios. Another difference is that scenarios are often represented using formal graphical notations such as Sequence Diagrams or Message Sequence Charts (MSCs).

The approach presented here is being imple-

mented in a Use Case Edition (UCEd) tool. The objective of UCEd is to help use cases acquisition, use cases verification, prototype generation, and simulation. We are also extending the state model generation algorithm to support the *include* as well as the *extend* relationships between use cases.

References

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

[2] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In *Software Engineering - ESEC'95. Proceedings of the 5th European Software Engineering Conference*, pages 254–271. Springler LNCS 989, 1995.

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

[4] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering,* A Use Case Driven Approach. Addison-Wesley, ACM Press, 2 edition, 1993.

[5] K. Koskimies and E. Mäkinen. Automatic Synthesis of State Machines from Trace Diagrams. *Software-Practice and Experience*, 24(7):643–658, July 1994.

[6] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM Models from Message Sequence Chart Specifications. In *13th IEEE Conference on Automated Software Engineering*, October 1998.

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

[8] 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.

[9] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[10] G. Schneider and J. P. Winters. *Applying Use Cases a practical guide.* Addison-Wesley, 1998.

[11] 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.

[12] S. Somé, R. Dssouli, and J. Vaucher. From Scenarios to Timed Automata: Building Specifications from Users Requirements. In *Proceedings of the 2nd Asia Pacific Software Engineering Conference (APSEC'95).* IEEE, Dec. 1995.

[13] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, Jun. 2000.