Supporting Use Case based Requirements Engineering

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
ssome@site.uottawa.ca

Abstract

Use Cases that describe possible interactions involving a system and its environment are increasingly being accepted as effective means for functional requirements elicitation and analysis. In the current practice, informal definitions of use cases are used and the analysis process is manual. In this paper, we present an approach supported by a tool for use cases based requirements engineering. Our approach includes use cases formalization, a restricted form of natural language for use cases description, and the derivation of an executable specification as well as a simulation environment from use cases.

Key words:
Use cases, domain modeling, UML, requirements engineering, prototyping

1 Introduction

Requirements engineering includes the elicitation, understanding and representation of customers needs for a system. It is a critical task in software engineering; the source of a great number of software failures.

The main reason for requirements induced failures is a gap existing between customers and the system development process. This gap is due to the manual nature of the requirement engineering process. Requirements are informally sought by analysts from customers who then pursue other development activities according to what they understand about customers needs. The *understanding* of requirements is generally represented as an abstract specification often not comprehensible by customers. That added to the difficulty to automatically ensure consistency between specifications and informal requirements makes difficult to ascertain, before later phases of a development process, if a

specification is right according to its requirements and if there are no missing requirements.

According to the UML specification (1), a use case is "the specification of a sequence of actions, including variants, that a system (or a subsystem) can perform, interacting with actors of the system". A use case describes a piece of behavior of a system without revealing that system internal structure. As such use cases are useful to capture and document external requirements. They are ideal for requirements validation through prototyping. Several software development approaches including the Unified Software Development Process (2) recommend use cases for users' requirements description.

A use case is a partial behavior description. Stakeholders with different views of a same system may thus provide different but possibly overlapping use cases as part of their requirements. It is also possible to develop 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 separate use cases. Moreover, use cases may be inconsistent one with the other and a set of use cases may define the requirements of a system incompletely. A common solution consists of deriving a global specification model integrating all the related use cases, such that the system can be simulated, its global behavior can be examined, and verified for inconsistencies as well as incompleteness. In the current practice specification derivation from use cases is informal and manual. In this paper, we present an approach to support of use cases based requirements elicitation, clarification, composition and simulation. We introduce a *restricted form* of natural language for use cases such that automated derivation of specification is possible while readability and understandability of use cases by all stakeholders is retained. Our approach is supported by a tool called *UCEd* (Use Case Editor (3)) that takes a set of related use cases written in a restricted form of natural language and generates an executable specification integrating the partial behaviors of the use cases. We use a *domain model* for syntactical analysis of use cases and specification generation. The approach is rooted in the Unified Modeling Language (UML) (1). Domain models are UML class diagrams (4). We also assume the UML specification and semantics of use cases. The UML is 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 UCEd to the various existing UML based methodologies and tools. UCEd allows importation of XMI (1) represented use cases and domain models.

This paper is organized as follows. In the next section, we situate our work in the context of some related works. Section 3 presents and abstract syntax and a concrete natural language syntax for use cases. The analysis of use cases is based on a domain model. We describe an extension of UML class diagrams for domain models in section 4. In section 5 we present an algorithm for

state models generation from use cases. Requirements verification is discussed in section 6 and validation by simulation in section 7. Section 8 presents a use cases based requirements elaboration approach that involves state model generation from use cases and simulation. Finally, section 9 concludes the paper and outlines some future works.

2 Related work

The two main research areas related to our work are natural language analysis of use cases and scenario composition.

Works on natural language analysis of use cases include (12), (13), (14) and (15). They mainly focus on providing guidelines and restricted languages for writing use cases in natural language while avoiding ambiguities and errors. In (12) Rolland and Ben Achour present an approach for guiding use cases development. They propose linguistic patterns and structures for use case specification as well as an iterative process for writing use case specification as an unambiguous natural language text. A similar approach is discussed in (14) where a restricted language based on a set of guidelines is defined for use cases. Fantechi et al (15) use linguistic techniques to analyze use cases expressed in natural language. They collect quality metrics and detect defects related to use cases inherent ambiguity. Martin Glinz (13) proposes a natural language based notation for use cases similar to our notation and a manual approach for statecharts synthesis. Our work differs from that of Glinz in that we propose an automated approach for state model synthesis.

Our work is closely related to scenario composition approaches using finite state machines (16; 17; 18; 19; 20; 21; 22; 23). UCEd is a successor of REST (Requirements Engineering with Scenario Tool) (24). REST generates timed-automata specification by incremental composition of scenarios. A key difference between the present work and scenario-based approaches lies on the differences between the notion of use cases and that of scenarios. Although the terms "use case" and "scenario" are often considered synonymous, there are fundamental differences between the two artifacts. A scenario is a single linear sequence of interactions between external actors and a system. A use case integrates a set of scenarios (a main scenario and zero or more secondary scenarios). While natural language text is used for use cases, scenarios are often represented using more formal graphical notations such as Sequence Diagrams, Message Sequence Charts (MSCs) or Live Sequence Charts (LSCs). Finally, use cases are considered earlier in the development process and at a higher level of abstraction than scenarios. In software development approaches such as the Unified Software Development Process (2), users requirements are first captured as use cases that are refined afterward into scenarios. A *black box*

view of the system is considered for use cases while a *gray box* view involving some internal components of the system is considered for scenarios.

Our work also shares similarities with the play-in/play-out approach of Harel and Marelly (25). The play-in/play-out approach is a specification methodology where a system required behavior is captured (played-in) as scenarios using a Graphical User Interface. A play-engine automatically generates a formal version of the played scenarios in the language of Live Sequence Charts (LSCs). This formal specification can then be simulated (played-out) using the same Graphical User Interface as for scenarios capture. UCEd automatic generation of a Graphical User Interface and simulation through that interface is similar to the way scenarios are played-out in the play-in/play-out approach. Differences between the two approaches include the use of textual use cases and a domain model as a basis for requirement capture in UCEd. One of our objectives is to support requirements engineering with textual use cases.

3 Use cases

A use case is a description of interactions between a system and actors in its environment. A *use case model* includes use cases, actors and relationships. A use case diagram is a graphical depiction of a use case model in the UML. Use case diagrams show 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 1 shows a UML graphical representation of use case diagram. The system under consideration is a *Patient Monitoring System* (PM System). A system used to monitor patients vital signs in a hospital.

Use case diagrams are abstract high-level view of functionality. They do not describe the interactions in use cases. According to the UML specification, the realization of a use case may be specified by a set of *collaborations* that define how instances in the system interact to perform the sequences of the use case. The collaborations may be captured using a variety of notations including natural language, sequence diagrams, activity diagrams and state diagrams.

In practice in order to allow for an easy communication with stakeholders, use cases collaborations are usually written as *structured natural language* interac-

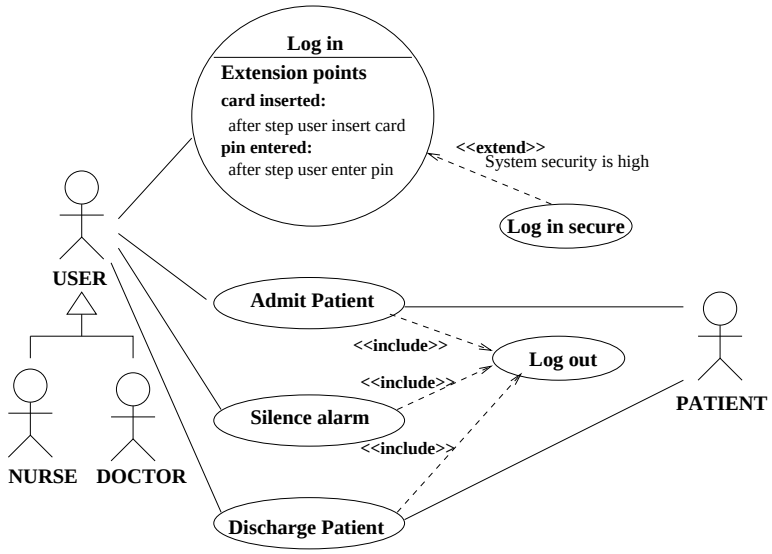

Fig. 1. Example of Use Case diagram for a PM System.

tions between actors and a system. Different *templates* and *guidelines* for use cases edition have been proposed in the literature. An example is Cockburn's template (5) where use cases are described using structured text. Our representation of use cases is inspired from that template. In the remainder of this section, we first present an abstract syntax for use cases collaborations, and then we present a concrete syntax using a restricted form of natural language.

3.1 Use cases abstract syntax

Figure 2 shows our abstract syntax for use cases in the UML. We distinguish two kinds of use case descriptions corresponding to the two types of use cases: *normal use cases* and *extension use cases*.

3.1.1 Normal use cases

Figure 3 shows the details of use case *Log in* (the concrete syntax used is presented in section 3.2). The use case *Log in* describes a login procedure that must be used by the users of the PM System.

A normal use case can be seen as a tuple [*Title*, *Precondition*, *Steps*, *Use Case alternatives*, *Postcondition*,] with: *Title* a label that uniquely identifies the use case, *Precondition* a *Constraint* that must be true before an instance of the use case can be executed (the term *constraint* is used in the UML specification to refer to conditions), *Steps* a sequence of steps, *Use Case alternatives* a set of *alternatives* that apply to all the steps in the use case, and *Postcondition* a *Constraint* that must be true at the end of an instance of the use case

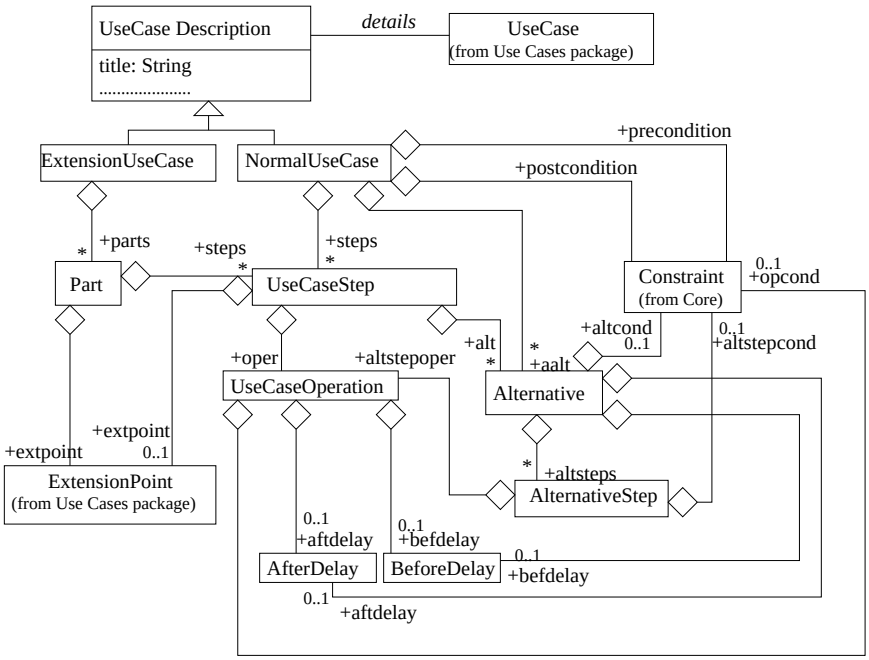

Fig. 2. UML representation of an abstract syntax for use cases description.

execution. As an example the use case in Figure 3 title is "Log in". The use case precondition is "PM System is ON". Use case *Log in* includes six steps listed in section titled Steps and the postcondition is "User is logged in". Use cases generally include more sections than here. Although all these sections help requirements documentation, for the purpose of state model generation, we are interested in only the functional description aspect of use cases defined by the preconditions, steps, alternatives and postconditions. We also consider use case titles for traceability.

Each step in *Steps* is a tuple [*Oper*, *ExtPoint*,*Alt*] with *Oper* a *use case operation*, *ExtPoint* a possibly null definition of an *extension point*, and *Alt* a set of *alternatives* that are possible after the step. We distinguish the following types of use case operations: *instances of concept operations*, *branching statements* and *use cases inclusion directives*. Examples of instances of concept operations in use case *Log in* are "insert a Card" by concept "User" and "ask for PIN" by concept "PMSystem". Extension step *4a1* includes a branching to Step *2*. A use case operation may include a *guard* (*OpCond*), an *AfterDelay* and a *BeforeDelay*. *OpCond* is an additional condition that must hold for the operation to be possible. Step *5* operation is constrained by the condition "USER identification is valid". Step *6* operation includes an *AfterDelay* "After 45 sec". Delays are counted from the completion moment of the previous step or from when the precondition became *true* when applied to the first step of a use case. An *AfterDelay* specifies a minimum time amount that must pass for the operation to be possible while a *BeforeDelay* specifies a maximum time amount after which an operation is no more possible. As an example step *6* delay means that 45 seconds must pass before the execution

Title: Log in

Primary Actor: User

Participants:

Goal: A User wants to identify herself in order to be able to use the PM system to perform a task such as admitting a patient or changing silencing an alarm.

Precondition: PMSystem is ON

Postcondition: User is logged in

Steps:1: User inserts a Card in the card slot
`Extension Point ==>` card inserted
2: PMSystem asks for PIN
3: User types her PIN
`Extension Point ==>` pin entered
4: PMSystem validates the User identification
5: IF the User identification is valid THEN PMSystem displays a welcome message to User
6: AFTER 45 sec PMSystem ejects the User Card

Alternatives:

1a: User Card status is irregular
1a1: PMSystem starts System status alarm
1a2: AFTER 20 sec PMSystem ejects Card
2a: AFTER 60 seconds
2a1: PMSystem starts System status alarm
2a2: AFTER 20 sec PMSystem ejects Card
4a: User identification is invalid AND User number of attempts is less than 4
4a1 GO TO Step 2
4b: User identification is invalid AND User number of attempts is equal to 4
4b1: PMSystem starts System status alarm
4b2: AFTER 20 sec PMSystem ejects Card

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

of operation "eject card" by the "PMSystem". An extension point is a label that references a particular point in a use case where interactions defined in extension use cases may be inserted. Use case *Log in* includes extension points *card inserted* at step *1* and *pin entered* at step *3*.

An alternative specifies a possible continuation of a use case after a step. Alternatives describe exceptions, error situations or less common courses of events. Formally an alternative is a tuple [*AltCond*, *AftDelay*, *BefDelay*, *AltSteps*] with *AltCond* a constraint that must be true for the alternative to be possible, *AftDelay* a possibly null *AfterDelay*, *BefDelay* a possibly null *BeforeDelay* and *AltSteps* a sequence of alternative steps. Each of an alternative step

in turn is a tuple [*AltStepCond*, *AltStepOper*] with *AltStepCond* a constraint and *AltStepOper* a use case operation. Use Case *Log in* includes alternatives to steps *1* (*1a*), *2* (*2a*) and *4* (*4a* and *4b*). Alternative *1a* condition is "User Card is not regular". Alternative *2a* is constrained by an *AfterDelay*. That delay is related to the completion of step *2* operation. In order to keep use cases complexity low, we make the restriction that there cannot be alternatives to alternatives. Situations requiring several levels of alternatives are conveniently specified with included use cases.

3.1.2 Extension use cases

An extension use case includes one or more *parts*. These parts are inserted at specific *extension points* in a *base use case* in presence of an *extend* relationship. An extension use case is formally a tuple [*Title*, *Parts*] with: *Title* as previously defined , and *Parts* a set of parts. Each part is a tuple [*ExtPoint*, *Steps*]. *ExtPoint* is a reference to an extension point (defined in the UML specification) and *Steps* a sequence of steps. As an example, use case *Log in secure* shown in Figure 4 is an extension use case with two parts. One to be

Title: Log in secure

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

Fig. 4. Extension use case.

included at an extension point card inserted, and the other at an extension point pin entered. In the use case diagram shown in Figure 1, there is and extend relationship between *Log in secure* and the base use case *Log in*. According this relationship, use case *Log in secure* extends use case *Log in* when condition *PMSystem security is high* holds. Data provided by users logging in are then recorder. Formally, an extend relationship between a base use case *UCbase* and an extension use case *UCext* is a tuple [*UCbase*, *UCext*, *ExtCond*, *ExtPoints*], where *ExtCond* is a constraint under which the extension can take place and *ExtPoints* is a set of extension points referred to in the extension use case.

3.1.3 Scenarios

A use case consists of a set of *scenarios* each being a sequence of events from the beginning of the use case to one of its possible ends. 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 (6). In

our notation, primary scenarios are described in the section titled Steps while secondary scenarios consist of interactions in the primary scenario followed by interactions defined in the section titled Alternative.

Figure 5 shows a partial list of use case *Log in* scenarios. The number of

Scenario	Sequence of Events	Type
1	1 - 2 - 3 - 4 - 5 - 6	Primary scenario
2	1 - 1a1 - 1a2	Secondary scenario
3	1 - 2 - 2a1 - 2a2	Secondary scenario
4	1 - 2 - 3 - 4 - 4a1 - 2 - 3 - 4 - 5 - 6	Secondary scenario
5	1 - 2 - 3 - 4 - 4a1 - 2 - 3 - 4 - 4a1 - 2 - 3 - 4- 5 - 6	Secondary scenario
6	1 - 2 - 3 - 4 - 4b1 - 4b2	Secondary scenario
...	...	...

Fig. 5. Use Case *Log in* scenarios.

scenarios in a use case may be infinite in presence of branching statements. In use case *Log in*, the number of scenarios is limited by conditions at lines *4a* and *4b*.

3.1.4 Conditions

Several elements in our abstract syntax are conditions. These elements include *preconditions*, *postconditions* operation *guards* and alternative conditions. We formally define conditions as *predicates* or combinations of predicates. A predicate is a pair $<E,V>$ where E is an *entity* and V a *value*. Entities refer to *concepts* (actors or the system under consideration) or *attributes* of *concepts*. We distinguish *atomic* values and *set* values. Atomic values are denoted as units such as 'ON' or 'logged in', while set values are denoted by way of comparisons. We also consider a special value 'unknown' that may be used with any entity.

As an example, condition "User Card is not regular" is formally a predicate $<$*User Card*,*not regular*$>$ that evaluates to true when the concept *User Card* has the value *not regular*. An example of set value denoted by "less than 4" is used in condition "User number of attempts is less than 4".

3.2 A concrete natural language syntax

The basic components of use cases are *conditions* and *operations*. We propose a form of natural language for the concrete description of these elements. This language is *context dependent*. The context information needed for use case analysis is provided in a *domain model*, which is described in section 4.

3.2.1 Concrete syntax for conditions

Figure 6 outlines our grammar for conditions. Conditions are *predicative sentences* describing *situations* prevailing within the system and its environment. We distinguish *simple conditions* from *composite conditions*. Composite con-

```
<condition>   -> <acondition> "and" <condition>
  | <acondition> "or" <condition>
  | "("<condition>")"
  | <negation> <condition>
<acondition>  -> [<determinant>] <entity> [<verb>] <value>
<determinant> -> "a" | "an" | "the"
<negation>    -> "not" | "no"
<entity>      ->  <concept> | <attribute>
<concept>     -> (<word>)+ {member of the model concepts}
<attribute>   -> (<word>)+ {attribute of concept}
<verb>        -> {derived from to be in present tense}
<value>       -> (<word>)+ {value of the entity}
 | <comparison> {entity is non-discrete ?}
<comparison>  -> <comparator> <word>
<comparator>  -> ">" | "<" | "=" | "<=" | ">=" | "<>"
 | "greater than" | "less than" | "equal to" | "different to"
 | "greater or equal to" | "less or equal to"
```

Fig. 6. Grammar for conditions. Elements between '<>' are non-terminals and those between "'' are terminals. Elements between "[]" are optional and | is used for alternative rules.

ditions are negations, conjunctions or disjunctions of conditions. A simple condition starts with an optional *determinant* followed by an *entity*, a *verb*, and a *value*. An *entity* refers to a *concept* or an *attribute* of a *concept*. Concepts are elements of the domain. They include the system under consideration as well as actors of the system environment. The *verb* is derived from the verb *to be* and the present tense must be used. A value is an entity *qualifier*. Atomic values are declared as *possible values* of corresponding entities in the domain model. As an example the precondition "*PMSystem is ON*" is a clause where the system has the quality of being *ON*. The entity is *PMSystem*; the system under consideration. The verb form "*is*" is used and the value used as qualifier is the *atomic element* "ON".

3.2.2 Concrete syntax for use case operations

Figure 7 shows our syntax for use case operations. A use case operation may be

```
<operation_spec> -> <concept_operation> | <branching_statement>
  | <useCase_inclusion>
<concept_operation> -> [<before_delay>] [<after_delay>]
                       [<condition_spec>] <operation_reference>
<condition_spec> ->  "IF" <condition> "THEN"
<operation_reference> -> (<word>)+ {derived from an operation of
                                    the current entity}
<after_delay>  -> "AFTER" <duration_spec>
<before_delay> -> "BEFORE" <duration_spec>
<duration_value> -> <duration_value> <duration_unit>
<branching_statement> ->
    "GO" "TO" "Step" <word> {corresponding to a step label}
<useCase_inclusion> -> [<condition_spec>]
                       "INCLUDE" <use-case-name>
```

Fig. 7. Grammar for use case operations. The grammar refers to some definitions in Figure 6.

an instance of *concept operation*, a *branching statement*, or a use case *inclusion directive*.

An instance of concept operation denotes the execution of an operation by a concept. It is *active sentence* in which a component performs an action given as a verb. We assume concept operations declaration in the format *action_verb* [*action_object*];
where the *action_verb* is a verb in infinitive and the *action_object* refers to a concept or an attribute of a concept affected by the action. As an example, "validate user identification" is an operation name where the action verb is "validate" and the action object is "user identification" (an attribute of concept User). Given this naming convention, an operation reference has the following form:

[*determinant*] *concept_name action_specification* [*preposition action_participant*]

The *action_specification* has the form

conjugated_action_verb [*action_object*]

The *conjugated_action_verb* is the *action_verb* used in the concept operation declaration in the present tense.

As an example, given the operation "validate user identification", "The System validates user identification with the Branch" is an operation reference where:

- the active concept name is "System",

- the action specification is “validates user identification”, and
- “Branch” is an action participant introduced by the preposition “with”.

4 Domain model

A domain model is a high-level *class model* that captures *domain concepts* and their relationships. Domain concepts are the most important types of objects in the context of a system. They include the system as a black box with the “things” that exist or events that transpire in the environment in which the system works (2). A domain model is essential for use cases analysis. It serves as a lexicon for natural language analysis of use cases, and provides a formalization of operations necessary to state model generation.

Figure 8 shows a graphical representation of the *PMSystem* domain model. We use the UML class diagrams (4) to describe domain models. The basic

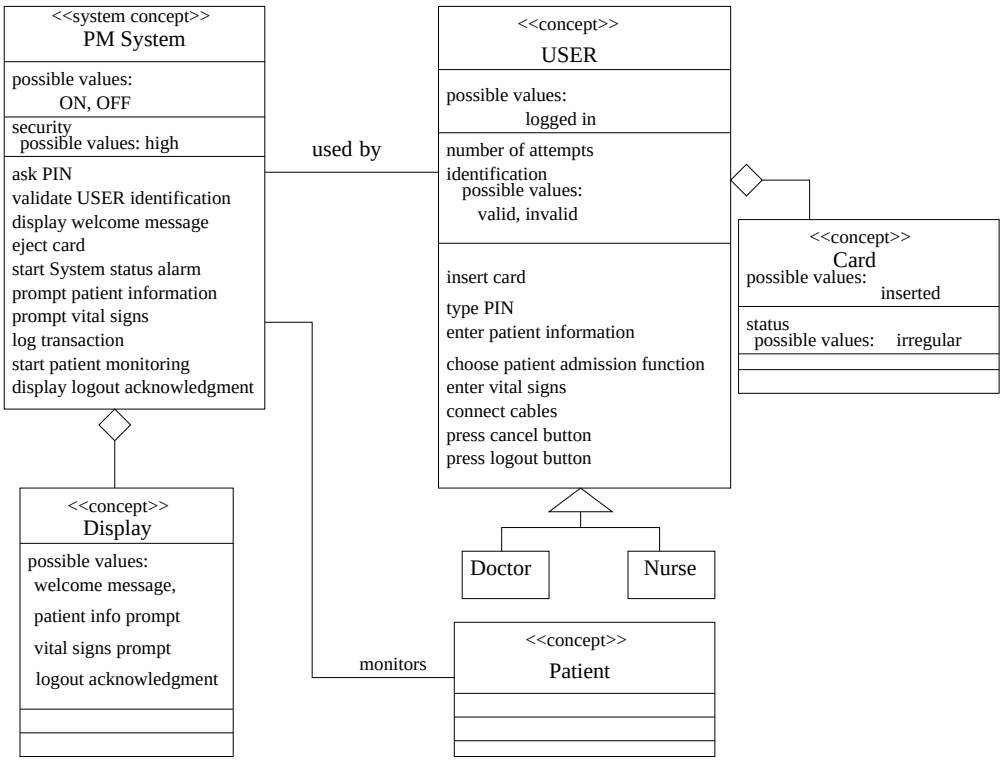

Fig. 8. Partial representation of the PM system domain model.

UML class diagram notation needed to be extended such that it is possible to specify possible values of entities. Indeed, in the concrete syntax of conditions, when a value qualifies an entity, that value must be known as a possible value of the entity. The traditional way to extend the UML is through *stereotypes*, *tagged values* and *constraints*. We defined a stereotype of the UML metaclass

Class called <<*concept*>> to represent concepts. This stereotype includes a *tag possibleValues* for the enumeration of the possible values of the concept. We also added a constraint to <<concept>> such that the attributes of a Concept are instances of a stereotype <<conceptAttribute>>, and each operation of a Concept is an instance of a stereotype <<conceptOperation>>. The stereotype <<conceptAttribute>> extends the metaclass *Attribute* with a *tag possibleValues*.

We define the following sets for each operation.

- An operation *added-conditions* set is a set of predicates that become true after the operation execution.
- An operation *withdrawn-conditions* set is a set of predicates that are removed after the operation execution.
  More formally, the semantic of a withdrawn-condition $<E, V>$ is as follow. "If the entity E value is V before the invocation of the operation, E value becomes *unknown* after the operation".

In the UML, it is possible to attach *preconditons* and *postconditions* to operations. A *preconditon* denotes a constraint that must hold for the invocation of the operation while a *postcondition* denotes a constraint that must hold after the invocation of the operation. Added-conditions and withdrawn-conditions are postconditions. We defined stereotype <<conceptOperation>> for the specialization of operations postconditions as added-conditions and withdrawn-conditions. Figure 9 shows added-conditions and withdrawn-conditions in the PMSystem example.

Withdrawn-conditions can be expressed as *any-conditions*. An *any condition* refers to a set of predicates on a same entity. Recall that withdrawn postconditions are conditions that are removed or become irrelevant after an operation. As shown in the *PM System* domain model, a withdrawn condition may be specified as a 'single' condition the same way as an added-condition. However, it is sometime useful to refer to all the conditions on an entity. As an example let us assume that after his/her card has been ejected, no information about a User is anymore relevant (in order words, the System forgets all about the User i.e. identification, numbers of attempts made, etc). It is not always feasible as in that case to list all the possible individual conditions that must be withdrawn. An any condition can be conveniently used in that case. More formally given an entity E, an any-conditions written as "ANY ON E" correspond to a set of predicates $P = \{<E, V>\}$. A wildcard '*' may be used to refer to the sub-entities of an entity in addition to the entity itself. As an example 'ANY ON User' refers to all the predicates with 'User' as entity (e.g. User is logged in), but does not include conditions on 'User Card' or 'User identification'. The withdrawn-condition 'ANY ON User*' on the other hand refers to all the predicates on 'User', as well as predicates on 'User identification', 'User

```
CONCEPT: PMSystem
  Operation:display logout acknowledgment message
    WithdrawConditions:ANY ON USER*
  Operation:ask Pin
  Operation:eject Card
    AddedCondition:Not USER card is inserted
    WithdrawCondition:ANY ON PMSystem Alarm
  Operation:validate USER identification
    AddedConditions:User identification is valid OR
                    User identification is invalid
    WithdrawConditions:ANY ON PMSystem display
  Operation:display welcome message
    AddedConditions:PMSystem display is welcome message, User is logged in
    WithdrawConditions:ANY ON USER identification
  Operation:start System status alarm
  Operation:prompt patient information
    AddedConditions:PMSystem display is patient info prompt
  Operation:prompt vital signs
    AddedConditions:PMSystem display is vital signs prompt
  Operation:log transaction
    AddedConditions:PMSystem log status is logged
  Operation:start patient monitoring
    AddedConditions:Patient status is monitoring
CONCEPT: User
  Operation:press logout button
    AddedConditions:USER is loging out
  Operation:insert card
    AddedConditions:USER Card is inserted
  Operation:type PIN
    AddedConditions:USER identification is entered
    WithdrawConditions:ANY ON PMSystem Display
  Operation:enter patient information
    AddedConditions:Patient status is identification entered
  Operation:choose patient admission
    AddedConditions:Patient status is admission initiated
    WithdrawConditions:ANY ON PMSystem Display
  Operation:enter vital signs
    AddedConditions:Patient status is vital signs entered
    WithdrawConditions:PMSystem Display is vital signs prompt
  Operation:connect cables
    AddedConditions:Patient status is connected
```

Fig. 9. Added and withdrawn conditions for domain operations.

number of attempts', and 'User Card'.

5 State Models generation from Use Cases

In (7; 8), we presented an algorithm for the generation of a hierarchical type of finite state transition machines from use cases. We briefly provide an outline of the algorithm here.

State model generation is based on: the specification of operations effects (withdrawn and added-conditions), and a relation between *states* and conditions. Each state is defined by *characteristic conditions* holding in it. These conditions are formulated as predicates $<E, V>$ with E a domain entity, and V a value. We consider a special value "*unknown*" that may qualify any entity. By default any entity whose value is not explicit in a state has an *unknown*

value.

We define a state machine $\mathcal{M}$ as a tuple $[Trig_c, Reac_c, G_c, S_c, S0_c, F_c, T_c]$.

- $Trig_c$ is a set of *triggers*. $Trig_c$ includes operations from the environment and *timeout events*.
- $Reac_c$ is a set of *reactions* that are operations executed by the system.
- G_c is a set of *guard* conditions.
- S_c is a set of states.
- $S0_c \in S_c$ is the initial state of the state machine.
- F_c is a transition function in domain $S_c \times Trig_c \times 2^{G_c} \times 2^{Reac_c} \times S_c$. Each transition $s \times trig \times g \times reac \times s'$ includes a start state s, an optional trigger *trig*, a set of guards g, a set of reactions *reac* and an ending state s'.
- T_c is a set of timers.

Given a use case *uc* and a state model $\mathcal{M}$, we create states and transitions into $\mathcal{M}$ such that the following are true.

(1) $\mathcal{M}$ includes a state s_0 with s_0 characteristic conditions $cond(s_0)$ equal to use case *uc* preconditions.
(2) For each actor operation *op* in use case *uc*, F_c includes a transition $s \times op \times g \times reac \times s'$ with:
   - s the state from which *op* is considered,
   - *reac* a set of all the system operation that may follow *op* according to use case *uc*,
   - g additional conditions constraining the operations, and
   - s' a state with characteristic conditions $cond(s')$. $cond(s')$ is a set of conditions obtained by considering operation *op* and operations in *reac* effects given the characteristic conditions of state s.
(3) F_c includes timeout triggered transitions reflecting delays in the use case.

$\mathcal{M}$ is an initially empty state model, which is incrementally expanded with states and transitions.

Our algorithm supports overlapping and connected use cases. We also support UML relationships << *extends* >> and << *include* >>. As an example, consider use case *Admit patient* of the PMSystem shown in Figure 10. *Admit patient* includes use case *Log out* and is supposed to follow use case "Log in". Figure 11 shows a state model obtained from use cases *Log in*, *Admit patient* and *Log in secure*. The state model description includes a definition of states in term of characteristic conditions followed by state transitions. Characteristic conditions define a hierarchy of state inclusion. A state s is a *substate* of a state s' if s' characteristic conditions include those of s. For instance, state *3* is a substate of state *1*. Transitions are in the format `trigger/reactions` or `conditions/reactions`. Any transition going from a state s also applies to all substates of s.

Title: Admit patient
Primary Actor: User
Participants: Patient
Goal: A User wants to perform the admission of a patient on a Monitor.
Precondition: User is logged in AND PMSystem Display is welcome message AND PMSystem is ON
Postcondition: Patient is admitted
Steps:1: User chooses patient admission function
2: PMSystem prompts for Patient information
3: User enters Patient information
4: PSystem prompts for vital signs
5: User enters vital signs
6: User connects cables to the Patient
7: PMSystem starts patient monitoring
8: Include Log out

Title: Log out
Primary Actor: User
Participants:
Goal: An authorized user want to logout from the PMS system
Precondition: User is logged in
Postcondition:
Steps:1: User presses logout button
2: PMSystem displays logout acknowledgment message

Fig. 10. Use case describing a patient admission procedure to the Patient Monitoring System with included use case "Log out".

A state machine generated from a use case includes all the use case scenarios. It is possible to find a sequence of transitions in the state machine corresponding to the sequence of operations of each scenario. A generated state machine may however include extra sequences of events that do not correspond to any scenario. These extra sequences are a consequence of the relation between the specification of operation effects in the domain model and the generated state machine. An insufficiency of information in the domain model might result in generalizations in generated state machines. In the extreme case that no operation effect is defined, the resulting system would be modeless. The generated state model would include a single state with looping transitions and every possible input would be accepted at any moment. Although, such a system would allow the sequence of events defined in the use cases, many other additional sequences would also be permitted.

Notice that such extra behaviors are not necessarily bad. We consider use cases as possible behavior descriptions from which more behavior can be inferred. Some of the extra sequences allowed by generalization may therefore be valid.

```
**** STATES ****
1:  [PMSystem is ON]
3:  [USER Card is inserted, PMSystem is ON]
4:  [USER Card is NOT inserted, PMSystem is ON]
6:  [USER Card is inserted, PMSystem is ON, USER Card status is
irregular]
7:  [USER Card status is irregular, USER Card is NOT inserted, PMSystem is ON]
9:  [USER Card is inserted, PMSystem Display is welcome message,
     USER is logged in, PMSystem is ON]
11: [USER Card is inserted, PMSystem is ON, USER identification is invalid,
     USER number of attempts == 4]
12: [USER Card is NOT inserted, PMSystem is ON, USER number of attempts == 4,
     USER identification is invalid]
13: [PMSystem Display is welcome message, USER is logged in, USER Card
     is NOT inserted, PMSystem is ON]
14: [USER Card is inserted, PMSystem security is high,
     PMSystem log status is logged, PMSystem is ON]
15: [PMSystem Display is welcome message, USER is logged in, PMSystem is ON]
16: [Patient status is admission initiated, PMSystem Display is patient info prompt,
     USER is logged in, PMSystem is ON]
17: [PMSystem Display is vital signs prompt, USER is logged in,
     Patient status is identification entered, PMSystem is ON]
18: [Patient status is vital signs entered, USER is logged in, PMSystem is ON]
19: [USER is logged in, Patient status is monitoring, PMSystem is ON]
20: [Patient status is monitoring, PMSystem is ON]
21: [USER Card is inserted, PMSystem is ON, USER identification is invalid]
**** TRANSITIONS ***
   1---insert card/-->2
   2---[PMSystem security is high]/log transaction, ask Pin-->3
   2---[USER Card status is NOT irregular, PMSystem security is NOT high]/ask Pin-->3
   2---[USER Card status is irregular]/start System status alarm-->6
   3---TIMEOUT(Timer2:20.0 second)/eject Card-->4
   3---TIMEOUT(Timer2:60.0 second)/start System status alarm-->2
   3---type PIN/-->5
   5---[PMSystem security is NOT high]/validate USER identification-->8
   5---[PMSystem security is high]/log transaction, validate USER identification-->8
   6---TIMEOUT(Timer3:20.0 second)/eject Card-->7
   8---[USER identification is invalid]/-->10
   8---[USER identification is valid]/display welcome message-->9
   9---TIMEOUT(Timer17:45.0 second)/eject Card-->13
   10---[USER number of attempts < 4]/-->2
   10---[USER number of attempts == 4]/start System status alarm-->11
   10---[USER number of attempts > 4]/-->21
   11---TIMEOUT(Timer13:20.0 second)/eject Card-->12
   14---TIMEOUT(Timer2:20.0 second)/eject Card-->4
   14---TIMEOUT(Timer2:60.0 second)/start System status alarm-->2
   14---type PIN/-->5
   15---choose patient admission/prompt patient information-->16
   16---enter patient information/prompt vital signs-->17
   17---enter vital signs/-->18
   18---connect cables/start patient monitoring-->19
   19---press logout button/display logout acknowledgment message-->20
```

Fig. 11. State machine obtained from use cases "Log in", "Admit patient" and "Log in secure". Transitions are specified in the UML format: $[guard]trigger/reaction_1, \cdots reaction_n$.

However, in case an extra sequence is unwanted, use cases and/or the domain model need to be modified in order to remove it. We present a process by which a generated state model is validated and then corrected in section 8.

6 Requirements verification

A single unifying domain model as advocated by our approach helps avoid some inherent ambiguities such as the use of different names for a same entity.

Conflicts can arise at four levels in our requirements model: between elements of a domain model, between a domain model and use cases, within a use case, and between separate use cases. We detect inconsistencies during syntactic analysis of domain and use case models, during use cases composition, and during state machines generation.

The syntactical analysis of a domain model ensures that there is no duplicate declaration (use of same name to refer to different things). Syntactical analysis of the domain model also ensures that references made in conditions are valid. As an example, when analyzing a condition, we check if the entity used exists in the model, and if the condition refers to an atomic value, we also check that that value is declared as a possible value of the entity.

A contradiction between an *after* and a *before* delay is an example of inconsistency detected simply by syntactical analysis of use cases. When both types of delays constraint an operation, the *before-delay* value must be greater than the *after-delay* value. Otherwise the timeout value used in step *4.4'* of our delays consideration algorithm would be negative. We also perform use cases verification against the domain model during use case syntactical analysis. The verification is based on the following rules: each operation in a use case must refer to a concept operation in the domain model, each condition must refer to an entity declared in the domain model and any atomic value must be a possible value of that entity.

During use case composition, we check for *inconsistent states*, use cases *post-conditions* and *preconditions* of operations.
We define an inconsistent state as a state with a logically "unsatisfiable" set of characteristic predicates. Our composition algorithm avoids inconsistent states. Before creating a state characterized by a predicate set *PSet*, we ensure that *PSet* doesn't include contradictions. Suppose for example adding an alternative from a state with predicate c when the alternative condition is *not c*. The resulting set {c, *not* c} is unsatisfiable. Therefore the composition algorithm would not create a state, but would rather report an inconsistency. In general, two predicates $p_1 =< E_1, V_1 >$ and $p_2 =< E_2, V_2 >$ are contradictory if (1) they refer to a same entity ($E_1 = E_2$), and (2) the values V_1 and V_2 are not consistent one with the other. Values consistency is defined as follow. Two atomic values are not consistent if different. Two set values are not consistent if their intersection is empty. For instance, set values V_1 defined as "> 10" and V_2 defined as "< 5" are not consistent while "> 10" and "> 20"

are consistent.

A type of use cases inconsistencies results from contradictions between use cases. Typically two use cases are inconsistent when following a same actor action (trigger event), in a same *situation*, and given similar timing constraints; the two use cases specify different system reactions. That type of inconsistency constitutes a class of the *feature interaction* problem in the telecommunication industry (9). We detect this type of inconsistency by analyzing the *reactive state model* (presented in section 7) resulting from use cases composition. As in (10) and (11), the inconsistencies produce *non-deterministic* transitions that go from a same state, with same stimuli.

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. Similarly, we check the precondition of operations by looking at *characteristic conditions* of the states from which these operations are added. An operation precondition specifies necessary conditions for the operation to be applicable.

7 Use cases simulation

Simulation is an effective technique used for requirements elicitation, requirements validation and requirements completion. A drawback with simulation is in the necessity of developing a prototype. Manual derivation of prototypes from requirements can be error prone and costly. State machines have the property of being executable and used as prototypes. Consequently, simulation can be efficiently applied within our approach. UCEd includes a simulator tool that allows use cases simulation using generated state machines as prototypes. The objective of simulation is to reproduce the reactive behavior described in use cases and to exhibit the global behavior resulting from their integration.

Simulation with UCEd is conducted through a graphical user interface generated from the specification of actors. Figure 12 shows a view of UCEd simulator tool for the PMSystem. The simulator includes an "actor events panel" (left panel) and a "simulation results panel" (right panel). UCEd generates a button corresponding to each actor operation in the "actor events panel" such that clicking on the button simulates the given operation. The operations are obtained from the *domain model*. The "simulation results panel" includes areas for the state prior to the latest actor operation, the system reactions in

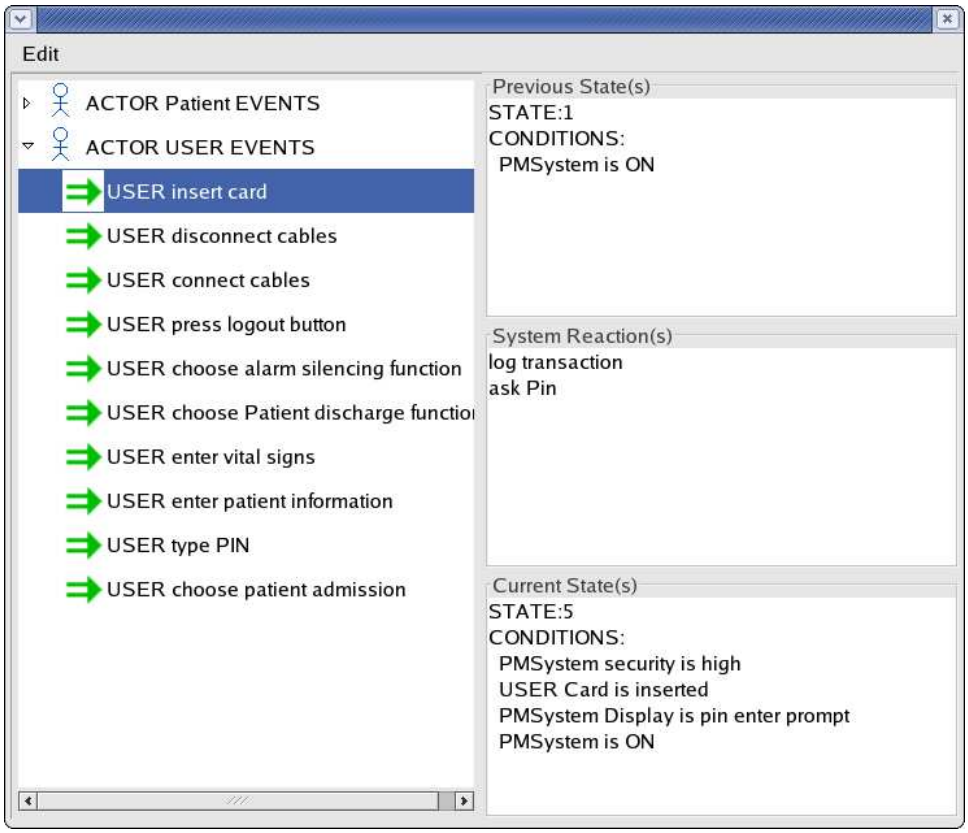

Fig. 12. Simulator tool view

response and the new state reached. Initially, the previous state area is empty and the current state area shows the label and characteristic conditions of the initial state.

The simulator handles selected actor operations according to the underlying state machine. If the state machine doesn't include a transition triggered by the selected operation from the current state, the simulator displays a message and the current state remains unchanged. If there is a single transition triggered by the actor operation from the current state, the simulator moves to the resulting state of that transition and adds all the system operations on the outgoing transition to the reactions area. The transition ending state then becomes the new current simulation state. When a simulated transition includes *guards* or is triggered by a *timeout event*, the simulator prompts the user for selection.

As an example suppose the simulation of the state machine shown in Figure 11. The simulation starts in state *1* the state machine initial state. Suppose the user chooses operation *insert card*. Three transitions are possible from state *1* on operation *insert card*. The choice of transition depends on *guards*. Therefore, the simulator will prompts the user such that one of the conditions "Card status is irregular" or "Card status is not irregular", and one of the

conditions "Security is high" or "Security is not high" is selected. Suppose the user chooses to enable the two conditions: "Security is not high" and "Card status is not irregular", the simulator would select transition
`2---[USER Card status is NOT irregular, PMSystem security is NOT high]/ask Pin-->3`, display operation *ask Pin* in the system reactions area and state *3* characteristic conditions in the new state area. Since there are timeout event based transitions from state *3*, the simulator would prompt again the user to select whether a delay of 20 or 60 seconds has passed while waiting in state *3* or not. Depending on the choice, the system reaction *eject Card* would be displayed and state 4 selected.

8 Use Cases based Requirements Elaboration

Figure 13 describes our use case based requirement engineering process. The process is supported by a Use Case Editor (UCEd). It starts with an early view

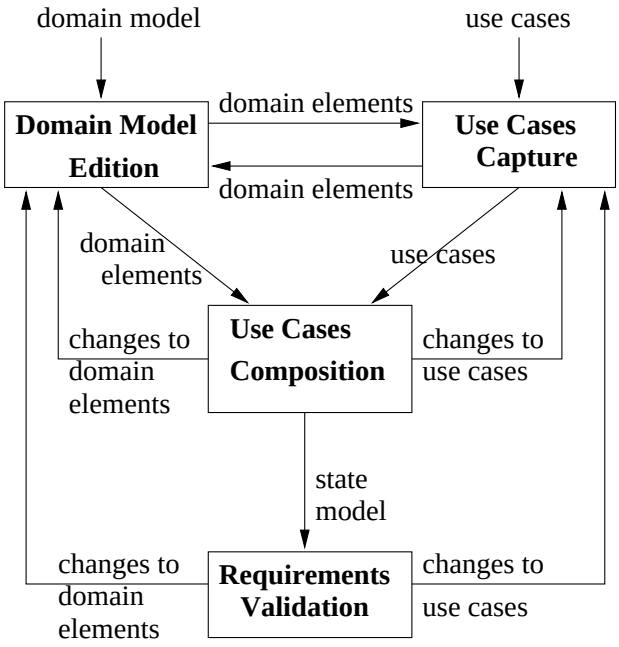

Fig. 13. Use Case based requirements engineering process. The boxes are activities and the arrows show data elements exchanged between these activities.

of requirements consisting of "rough" domain model and use cases. It produces a high-level state model specification of the system as well as clarified use cases and domain model. Several iterations are involved. Each iteration includes the following activities.

- Capture of use cases.
  Use case capture is done through a Use Case Writing module of UCEd. This module provides an interface for the edition of use case models and

details. Figure 14 shows a view of UCEd Use Case Writing module with the use case model described in Figure 1. Use cases are captured through a

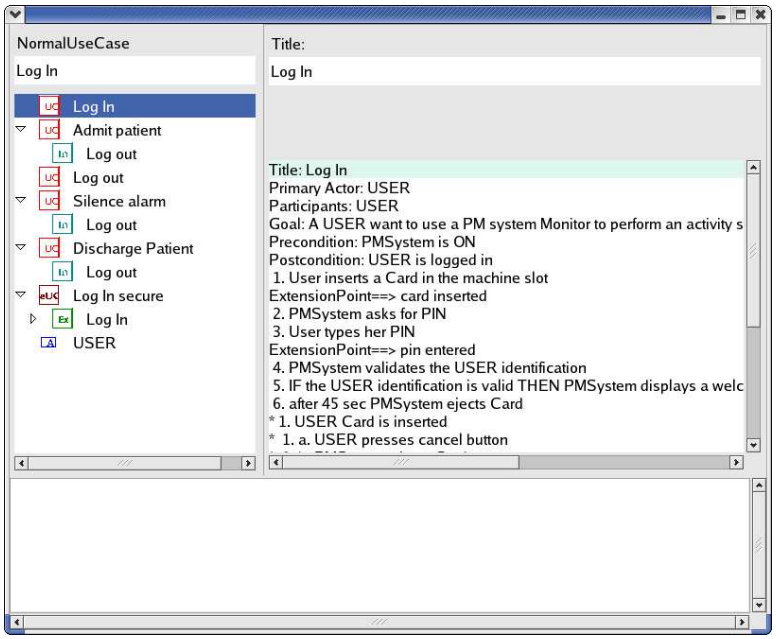

Fig. 14. UCEd Use Case Writing module.

field-oriented editor in a restricted form of natural language. A field-oriented form offers the advantage that writers do not have to worry about delimiting the different parts of their use cases. UCEd checks use cases and the domain model against each other and reports inconsistencies and omissions.

- Capture of domain elements.
  This activity aims at defining enough elements to allow use cases syntactical analysis. Another objective is to define operations effects for state model generation. UCEd allows an automated extraction of domain entities names from use cases.
- Generation of a state model.
  State model generation is done automatically with UCEd based on our state model generation algorithm. Use cases composition is incremental. Each use case is analyzed and its partial behavior merged in a state model obtained from previously composed use cases. The composition process is driven by information in the domain model. Some inconsistencies in the use cases and in the domain can be found and reported during use cases composition.
  Use cases composition results in a state model specification that combines all the use cases partial behaviors.
- Requirements validation.
  The objective of requirements validation is to ensure that the combination of use cases and domain model satisfies the intended behavior for the system. Validation consists of a manual inspection of generated state model, as well as simulation with UCEd. Validation may uncover possible extra sequences of events that should not be supported. In that case, an alteration of use

cases and/or the domain model is needed for correction.
For instance, suppose use cases "Log in" shown in Figure 3, "Log in secure" shown in Figure 4, "Admit patient" shown in Figure 10 and the postconditions shown in Figure 9. The state model generated from these use cases based on the specified postconditions is shown in Figure 11. The sequence: *User insert card - PMSystem ask pin - User insert card* is possible according to the generated state machine. State *3* is a sub-state of state *1* and therefore any operation possible from state *1* such as *User insert card* is also possible from state *3*. This extra sequence is invalid. It should not be possible to insert a card while there is already another card inserted. In a situation such as this one, some use cases or some operation effects in the domain model must be altered to remove the unwanted sequence. In the present case, a correction could be made by simply adding condition "User Card is not inserted" to the preconditions of use case "Log in". Another possible sequence is: *User insert card* - [*USER Card status is irregular*] *PMSystem start System status alarm - User type pin.* In this sequence, a User insert an *irregular* card, the system rightfully starts an alarm. However, the user could still enter a pin an continue with a normal interaction. The sequence is caused by the fact that state *6* is a sub-state of state *3*. In order to avoid that sequence, we need to add postconditions such that state *6* is not a sub-state of state *3* anymore. A version of use case "Log in" and the domain model with the necessary corrections is shown in appendix A. The state model generated from this use case and domain model does not allow sequence *User insert card* - [*USER Card status is irregular*] *PMSystem start System status alarm - User type pin.*

9 Conclusions

In this paper, we have presented an approach that aims at helping requirements engineering with a framework as well as a tool for use cases edition, clarification, and early simulation. By using a domain model, the approach allows capture of the relevant domain concepts, as well as the definition of operations *pre* and *postconditions* in parallel with use cases. Pre and postconditions may serve as *contract* definitions of operations going to the design phase. Our experience with the approach has mainly been in an academic setting. We are using the approach in combination with UCEd to teach software engineering students how to better describe use cases and domain model elements. Examples of projects for which students used UCEd include a "Patient monitoring system", an "Automated teller machine", a "Library system" and a "Telephone PBX system". Experimental results show that the notation is intuitive and easy to master. The approach helps production and communication of use cases and clarification of the domain. We still need to validate

the approach in an industrial setting. A more expressive logic such as OCL (1) may also be needed to capture and reason about some elaborate situation such as allowing postcondition to be expressed in term of preconditions.

Differently to several scenario-based approaches, we take a black-box view where the system under consideration is seen as a single component. This is motivated by the desire to focus on the system's external requirements. Scenario approaches because of the use of formal graphical notations and their gray-box view of the system are geared more toward software design than requirement engineering. The use of these approaches follows a necessary stage of capturing and understanding the high-level goal of the system abstract from its architectural decomposition. Use cases and scenario approaches are complementary. An interesting question is how to make the transition from a use case point of view to a scenario point of view. We plan to look at that question as part of our future work.

We propose a natural language notation for use cases. Because there is no standard description of use case contents, several ways of writing use cases in natural language exist in practice. Some of which are organization specific. We defined an abstract syntax for use cases such that any concrete syntax could be used as long as use cases can be unambiguously mapped to the abstract syntax. One of our plans is to improve UCEd ability to be adapted to *custom* notations. We also plan to integrate non-functional requirements such as performance and security constraints, to use cases.

References

[1] OMG, OMG Unified Modeling Language Specification version 1.4 (2001).

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

[3] Use Case Editor (UCEd) toolset, http://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCEd.html.

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

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

[6] G. Schneider, J. 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), Vol. I, 2003, pp. 456–462.

[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, 2004.

[9] P. Zave, Feature Interactions and Formal Specifications in Telecommunications, Computer 26 (8) (1993) 20–30.

[10] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, C. Chen, Formal approach to scenario analysis, IEEE Software (1994) 33–41.

[11] M. P. Heimdahl, N. G. Leveson, Completeness and Consistency Analysis of State-Based Requirements, in: Proceedings of the 17th International Conference on Software Engineering, 1995, pp. 3–14.

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

[13] Glinz, Improving the quality of requirements with scenarios, in: Proceedings of the Second World Congress on Software Quality, 2000, pp. 55–60.

[14] K. Böttger, R. Schwitter, D. Richards, O. Aguilera, D. Mollá, Reconciling use cases via controlled languages and graphical models, in: INAP 2001, Proceedings of the 14th International Conference on Applications of Prolog, 2001, pp. 186–195.

[15] A. Fantechi, S. Gnesi, G. Lami, A. Maccari, Application of linguistic techniques for use case analysis, in: RE'02, Proceedings of the 10th Requirements Engineering Conference, 2002.

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

[17] S. Somé, R. Dssouli, 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, 1995.

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

[19] S. Leue, L. Mehrmann, M. Rezai, Synthesizing ROOM Models from Message Sequence Chart Specifications, in: 13th IEEE Conference on Automated Software Engineering, 1998.

[20] J. Whittle, J. Schumann, Generating statechart designs from scenarios, in: International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, 2000.

[21] D. Harel, H. Kugler, Synthesizing state-based object systems from LSC specifications, Lecture Notes in Computer Science 2088.
URL `citeseer.nj.nec.com/harel00synthesizing.html`

[22] S. Uchitel, J. Kramer, J. Magee, Synthesis of Behavorial Models from Scenarios, IEEE Transactions on Software Engineering 29 (2).

[23] R. Plösch, Contracts,Scnenarios and Prototypes An Integrated Approach to High Quality Software, Springer-Verlag, 2004.

[24] R. Dssouli, S. Somé, J. Vaucher, A. Salah, A service creation environment based on scenarios, Information and Software Technology 41 (11-12) (1999) 697–713.

[25] D. Harel, R. Marelly, Come, Let's Play, Springler, 2003.

A Improved use cases and domain model

Title: Log in

Primary Actor: User

Participants:

Goal: A User wants to identify herself in order to be able to use the PM system to perform a task such as admitting a patient or changing silencing an alarm.

Precondition: PMSystem is ON **AND User Card is not inserted**

Postcondition: User is logged in

Steps:1: User inserts a Card in the card slot
`Extension Point ==>` card inserted
2: PMSystem asks for PIN
3: User types her PIN
`Extension Point ==>` pin entered
4: PMSystem validates the User identification
5: IF the User identification is valid THEN PMSystem displays a welcome message to User
6: AFTER 45 sec PMSystem ejects the User Card

Alternatives:

1a: User Card status is irregular
1a1: PMSystem starts System status alarm
1a2: AFTER 20 sec PMSystem ejects Card
2a: AFTER 60 seconds
2a1: PMSystem starts System status alarm
2a2: AFTER 20 sec PMSystem ejects Card
4a: User identification is invalid AND User number of attempts is less than 4
4a1 GO TO Step 2
4b: User identification is invalid AND User number of attempts is equal to 4
4b1: PMSystem starts System status alarm
4b2: AFTER 20 sec PMSystem ejects Card

Fig. A.1. Modified version of use case log in with a pre-condition such that sequence *User insert card - PMSystem ask pin - User insert card is not allowed*.

```
CONCEPT: PMSystem
  Operation:display logout acknowledgment message
    WithdrawConditions:ANY ON USER*
  Operation:ask Pin
    AddedConditions:PMSystem display is pin enter prompt
  Operation:eject Card
    AddedCondition:Not USER card is inserted
    WithdrawCondition:ANY ON PMSystem Alarm
  Operation:validate USER identification
    AddedConditions:User identification is valid OR
                    User identification is invalid
    WithdrawConditions:ANY ON PMSystem display
  Operation:display welcome message
    AddedConditions:PMSystem display is welcome message, User is logged in
    WithdrawConditions:ANY ON USER identification
  Operation:start System status alarm
    AddedConditions:PMSystem Alarm is System Status
    WithdrawConditions:ANY ON PMSystem Display
  Operation:prompt patient information
    AddedConditions:PMSystem display is patient info prompt
  Operation:prompt vital signs
    AddedConditions:PMSystem display is vital signs prompt
  Operation:log transaction
    AddedConditions:PMSystem log status is logged
  Operation:start patient monitoring
    AddedConditions:Patient status is monitoring
CONCEPT: User
  Operation:press logout button
    AddedConditions:USER is loging out
  Operation:insert card
    AddedConditions:USER Card is inserted
  Operation:type PIN
    AddedConditions:USER identification is entered
    WithdrawConditions:ANY ON PMSystem Display
  Operation:enter patient information
    AddedConditions:Patient status is identification entered
  Operation:choose patient admission
    AddedConditions:Patient status is admission initiated
    WithdrawConditions:ANY ON PMSystem Display
  Operation:enter vital signs
    AddedConditions:Patient status is vital signs entered
    WithdrawConditions:PMSystem Display is vital signs prompt
  Operation:connect cables
    AddedConditions:Patient status is connected
```

Fig. A.2. Added and withdrawn conditions for domain operations to avoid sequence: *User insert card - [USER Card status is irregular] - PMSystem start System status alarm - User type pin* (see section 8). Postconditions have been added to operations "*ask Pin*" and "*start System status alarm*".

```
**** STATES ****
1:  [USER Card is NOT inserted, PMSystem is ON]
3:  [USER Card is inserted, PMSystem Alarm is System Status,
     USER Card status is irregular, PMSystem is ON]
4:  [USER Card status is irregular, USER Card is NOT inserted, PMSystem is ON]
5:  [USER Card is inserted, PMSystem Display is pin enter prompt, PMSystem is ON]
6:  [USER Card is inserted, PMSystem Alarm is System Status, PMSystem is ON]
10: [USER Card is inserted, PMSystem Display is welcome message,
     USER is logged in, PMSystem is ON]
11: [USER Card is inserted, PMSystem Alarm is System Status, PMSystem is ON,
     USER number of attempts == 4, USER identification is invalid]
12: [USER Card is NOT inserted, PMSystem is ON, USER number of attempts == 4,
     USER identification is invalid]
13: [PMSystem Display is welcome message, USER is logged in,
     USER Card is NOT inserted, PMSystem is ON]
14: [PMSystem Display is welcome message, USER is logged in, PMSystem is ON]
15: [Patient status is admission initiated, PMSystem Display is patient info prompt,
     USER is logged in, PMSystem is ON]
16: [PMSystem Display is vital signs prompt, USER is logged in,
     Patient status is identification entered,
     PMSystem is ON]
17: [Patient status is vital signs entered, USER is logged in, PMSystem is ON]
18: [USER is logged in, Patient status is monitoring, PMSystem is ON]
19: [Patient status is monitoring, PMSystem is ON]
20: [USER Card is inserted, PMSystem is ON, USER identification is invalid]
**** SCTRANSITIONS ***
   1---insert card/-->2
   2---[PMSystem security is high]/log transaction, ask Pin-->5
   2---[USER Card status is NOT irregular, PMSystem security is NOT high]/ask Pin-->5
   2---[USER Card status is irregular]/start System status alarm-->3
   3---TIMEOUT(Timer4:20.0 second)/eject Card-->4
   5---TIMEOUT(Timer7:60.0 second)/start System status alarm-->6
   5---type PIN/-->7
   6---TIMEOUT(Timer9:20.0 second)/eject Card-->1
   7---[PMSystem security is NOT high]/validate USER identification-->8
   7---[PMSystem security is high]/log transaction, validate USER identification-->8
   8---[USER identification is invalid]/-->9
   8---[USER identification is valid]/display welcome message-->10
   9---[USER number of attempts < 4]/-->2
   9---[USER number of attempts == 4]/start System status alarm-->11
   9---[USER number of attempts > 4]/-->20
   10---TIMEOUT(Timer20:45.0 second)/eject Card-->13
   11---TIMEOUT(Timer16:20.0 second)/eject Card-->12
   14---choose patient admission/prompt patient information-->15
   15---enter patient information/prompt vital signs-->16
   16---enter vital signs/-->17
   17---connect cables/start patient monitoring-->18
   18---press logout button/display logout acknowledgment message-->19
```

Fig. A.3. State machine obtained from the new version of use cases "Log in", use cases "Admit patient" and use case "Log in secure", with operations specified in Figure A.2. This state model does not allow sequence: *User insert card - [USER Card status is irregular] - PMSystem start System status alarm - User type pin* (see section 8).