

Use Case Editor (UCEd) User Guide
version 1.6.2

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

August 2007

Contents

1	Objective of UCEd	3
2	UCEd startup window	5
2.1	File Menu	6
2.2	Validate Menu	7
2.3	Generation Menu	7
2.4	State Machine	8
2.4.1	State Machine Viewer	9
2.5	Simulation Menu	9
3	Domain model	11
3.1	Elements of a Domain model	11
3.2	Conditions	13
3.2.1	Simple conditions	13
3.2.2	Complex conditions	17
3.3	Any conditions	18
3.4	Operation declaration	18
3.5	UCEd Domain Model Edition tool	19
3.5.1	Domain Model element types	20
3.5.2	Domain elements edition	20
3.5.3	Editors key combinations	23
3.6	Domain Model validation	24
3.7	Extraction of Domain elements from Use cases	24
4	Use Cases	27
4.1	Use Case Diagrams	27
4.2	Description of Use Cases	28
4.2.1	Normal use cases description	29

4.2.2	Extension use cases description	38
4.3	UCEd Use Cases Edition tool	38
4.3.1	Use Case models edition	39
4.3.2	Use Case descriptions edition	42
4.4	Use Cases validation	46
5	State Models	47
5.1	Control flow based state model	47
5.2	State models synthesis based on operation effects	49
5.2.1	Detailed State Charts	49
5.2.2	State machine synthesis	52
5.2.3	State chart generation	53
6	Simulator tool	55
6.1	Operation of the simulator tool	55
6.2	Simulation History	57
7	Scenario Model	59
7.1	Elements of scenarios	59
7.2	Scenario Model Edition tool	61
7.2.1	Scenario model element types	62
7.2.2	Scenario model edition	62
7.3	Scenario validation rules	64
7.4	Scenario simulation	64

Chapter 1

Objective of UCEd

The objective of the Use Case Editor (UCEd) is to provide automated support to requirements engineering. Requirements in UCEd approach consist of a Domain model, Use Cases, and Scenarios.

- A Domain model describes the *system* under consideration with the pertinent *concepts* in the system's operating environment. The description of each domain element includes *properties*, *operations* and *relationships*. A domain model is conveniently represented as a UML [4] high level class diagram.
- 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” [4]. A use case describes a piece of behavior of a system without revealing the internal structure of the system. As such use cases are effective means for requirements elicitation and analysis, and various software development approaches including the Unified Software Development Process [3] recommend use cases for requirements description.
- A Scenario is sequence of interactions between a system and actors of that system. Scenarios can be considered as instances of use cases [4].

UCEd takes a set of related use cases written in a restricted form of natural language and generates **State Models** that integrates the behavior specified by the use cases. The generation process relies on information contained in an application domain model.

UCEd uses generated state models as prototypes for simulation. Simulation can be manual through a generated graphical user interface, or automated with scenarios.

The end objectives of UCEd are:

1. the production of validated requirements in the form of use cases, a domain model definition, and scenarios,
2. the production of requirements specifications in the form of description of operation effects in the domain model, and generated state models, and,
3. the production of re-usable test scripts in the form of scenarios.

Chapter 2

UCEd startup window

Figure 2.1 shows UCEd startup window. Apart from a menu, the window shows the current project name (by default a project named 'Unnamed Project' is loaded as a current project) and a toolbar. A project consists



Figure 2.1: UCEd startup window

of: a **domain model** (see chapter 3), a **use case model** (see chapter 4), a **state model** (see chapter 5) generated from the use cases, and a **scenario model** (see chapter 7).

UCEd menus are as follow.

File for opening editors are working with projects.

Validate for validating domain and use case models.

Generate for state model generation.

State Machine for state model visualization.

Simulation for use cases simulation.

Help tells about UCed.

Some of the functions are accessible through the Toolbar.

2.1 File Menu

The **File** menu includes the following sub-menus.

Open shows a sub-menu with:

Use Case Editor to open a *Use Case Edition tool* window (see section 4.3).

Domain Editor to open a *Domain Model Edition tool* window (see section 3.5).

Scenario Editor to open a *Scenario Model Edition tool* window (see section 7.2).

Simulator to open a *Simulator tool* window (see chapter 6).

New Project to create a new project. The User is prompted to save the currently loaded project if it has been modified and is unsaved.

Load Project to load a project.

XMI Import Project to import a project use case model and domain model saved in the XMI format. This feature allows using models developed with UML modeling tools with XMI export facilities such as ArgoUML¹.

Save Project to save the current project. The User is prompted for a name if the project is unnamed.

Save Project As to save the current project in new files. The user is prompted for an XML file name. The binary file name is created by appending '.bin' to the XML file name.

¹<http://www.argouml.tigris.org>

Export Project As HTML to generate a HTML representation of the domain model, the use case model and the state model.

Exit to quit UCed. If the current project has been modified and is unsaved, UCed prompts for saving before exiting.

2.2 Validate Menu

The `Validate` menu includes sub-menus

Extract Domain from Use Cases to create or update an existing domain model given a use case model.

Validate Domain to validate the domain model, and

Validate Use Cases to validate the use case model.

We discuss extraction of domain from use cases in section 3.7. Domain validation rules are presented in section 3.6 and use cases validation rules are presented in section 4.4.

2.3 Generation Menu

UCed implements two algorithms for state model generation from use cases: an algorithm based on **control flow** information in use cases and an algorithm based on **operation effects** specified in the Domain.

- Control flow based generation infers a state model solely from use cases structure and from use case sequencing statements.
- Operation effects based generation infers a state model where states are *characterized by conditions*, and these conditions are obtained from the *postconditions of operations*.

The **Generation** menu includes a menu for generation based on control flow and a menu for generation based on operation effects.

Each of these menus in turn includes a sub-menu corresponding to each of the top-level use cases in the use case model.

- In the generation based on control-flow menu, the selection of a use case launches the generation of a state model corresponding to the selected use case. The generated state model includes states models for the use cases linked by use case sequencing statements.

In addition to a sub-menu per use case, the generation based on control-flow menu includes a sub-menu

Generate StateChart Chart to generate a control flow based state model for all the top-level use cases in the use case model. Use case sequencing relations produce [4] flownodes.

- In the generation based on operation effects menu, the selection of a use case incrementally adds the use case behavior to a global operation-effect based state model (if it has not already been added).

In addition to a sub-menu per use case, the generation based on operation effects menu includes sub-menus.

Add: All Use Cases to add all top-level use cases in the use case model to the state model, and

Reset State Machine to blank the state model.

2.4 State Machine

The **State Machine** menu includes a menu for visualization of state models generated based on control flow and a menu for visualization of state models generated based on operation effects.

- The menu for visualization of state models generated based on control flow includes a sub-menu for each of the top-level use cases in the use case model. These sub-menus allow to visualize the control flow state model of their corresponding use case. In addition, a sub-menu titled **View StateChart Chart** allows to visualize the combined control-flow based state model.
- The menu for visualization of state models generated based on operation effects includes the following sub-menus

View StateChart to show the generated state model without internal states and with complex transitions (with trigger, conditions, reactions).

View Detailed State Graph to show the generated state model with internal states and simple transitions (a single operation per transition).

Export StateChart in Graphviz dot format to export the global State Chart based on operation effects to Graphviz² dot format. Different tools can then be used to visualize the graph and export in other formats.

2.4.1 State Machine Viewer

Figure 2.2 shows the state model viewer menu tab.

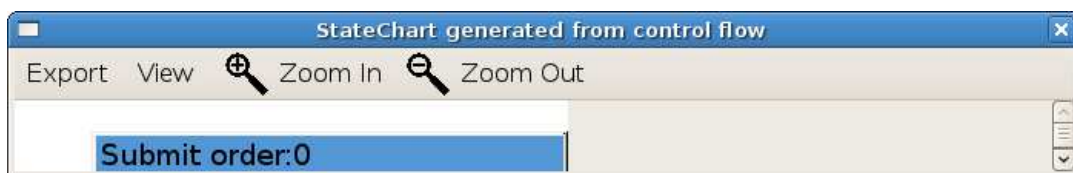


Figure 2.2: State Model Viewer.

Export: allows to export a state machine as a jpg or bmp image.

View: allows switching state machine transition labels on and off.

Zoom In/Zoom out allows to zoom in/out the state model view.

2.5 Simulation Menu

The **Simulation** menu includes a menu for simulation of state models generated based on control flow and a menu for simulation of the state model generated based on operation effects.

²<http://www.graphviz.org/>

- The menu for simulation of state models generated based on control flow includes a sub-menu for each of the top-level use cases in the use case model. Each sub-menu allow to simulate the control flow state model of their corresponding use case. In addition, a sub-menu **Simulate StateChart Chart** allows simulation of the global control-based state model. This sub-menu includes

Set Initial Use Case to specify which use case(s) are initially enabled to execute.

Start New Simulation to initiate a new simulation session. A new scenario is added to the simulation history and a new simulation starts from the current initial state. The added scenario records all simulated events up to the start of the next simulation session.

- The menu for simulation of the state model generated based on operation effects includes

Start New Simulation to initiate a new simulation session.

Set Initial State to set an initial state. By default UCED set the first created state as the initial state. This feature allows setting up a new state as the initial state by choosing among all the state model states.

In addition, the **Simulation** menu includes a menu **View Simulation History** to view a list of **scenarios** generated from previous simulation sessions.

Chapter 3

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 according to [3]. The development of a domain model is an integral part of requirements engineering in the UCEd approach.

3.1 Elements of a Domain model

A domain model must include a concept representing the system under consideration as a black box (a **system concept**). The model may also include one or more concepts representing other classes of objects in the system environment that interact with the system.

Concepts and system concepts can have zero or more concept instances (i.e. objects) defined in the domain model. These instances can be used in conditions and actions.

UCEd uses an extension of UML class diagrams[5] for domain models. The traditional way to extend UML is through *stereotypes*, *tagged values* and *constraints*.

An UCEd domain concept is an instance of a *stereotype* of UML *Class* meta-class called *Concept*. The *Concept* stereotype includes a *tag* called **possibleValues**, used to enumerate a concept *possible values*. These possible values are used in conditions (see section 3.2).

Concepts attributes must be instances of a stereotype called *ConceptAttribute*, and the operations of a concept must be instances of stereotype

ConceptOperation.

The ConceptAttribute stereotype extends the UML meta-class *Attribute* with a tag *possibleValues*. This tag is used to enumerate the *possible values* of the attribute.

The stereotype ConceptOperation extends UML *Operation* meta-class such that some of an operation *postconditions* can be specialized as *withdrawn-conditions*. Withdrawn-conditions denote conditions that are removed after the operation execution. The other postconditions; *added-conditions* are conditions that become true following the execution of the operation.

Figure 3.1 shows a graphical representation of a domain model in the UML notation. The effects of the domain operations are specified in Figure 3.2.

The domain in Figure 3.1 includes a system concept called *PMSystem* as well as environment concepts: USER, Doctor, Nurse and Patient. Doctor and Nurse are sub concepts (specialization) of USER. Display is a concept sub component (aggregate) of PM System.

The possible values of PMSystem are ON and OFF, it is thus possible to express conditions (see section 3.2) such as: 'PMSystem is OFF' or 'PM-System is ON'. USER and Display also have lists of possible values in the model.

Examples of attributes include *security* of PMSystem and *identification* of USER. The possible value of PMSystem attribute 'security' is *high*. USER identification has as possible values *valid* and *invalid*.

PM System and USER have operations. An example of the PM System operation is *validate User identification* this operation postconditions include an added-condition: 'User identification is valid OR User identification is invalid' and a withdrawn-condition: Display is pin enter prompt. Therefore, after operation *validate User identification* is executed,

- if the condition 'Display is pin enter prompt' was verified prior to the operation, this condition is removed (the value of Display becomes *unknown*) and
- the condition 'User identification is valid OR User identification is invalid' is now verified.

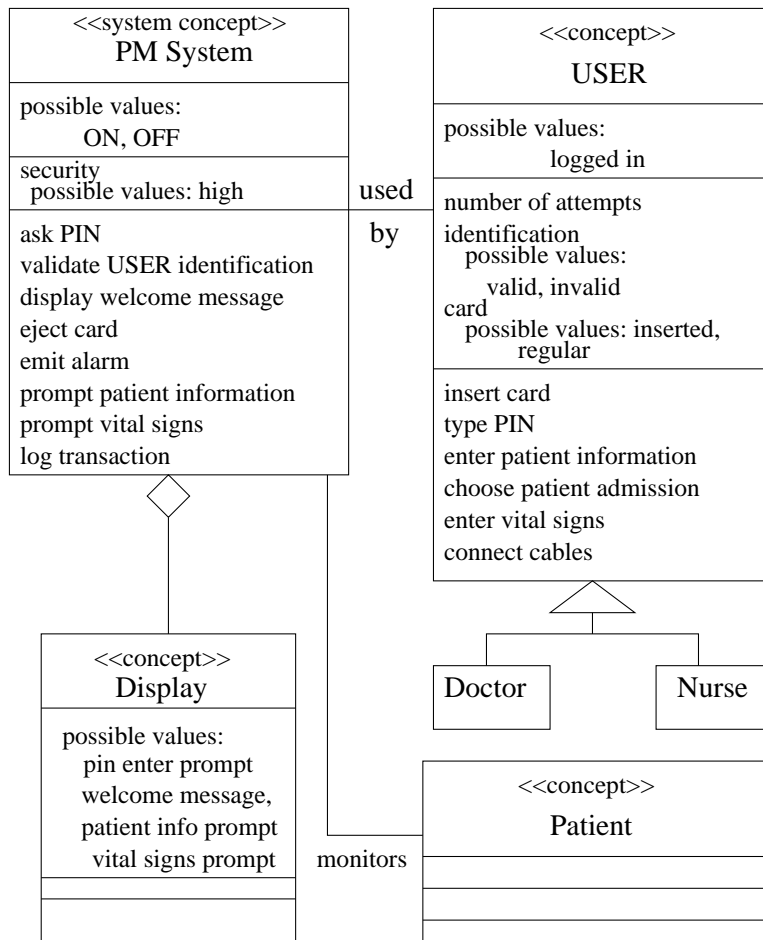


Figure 3.1: Example of Domain model in UML.

3.2 Conditions

Conditions are *predicative sentences* describing *situations* prevailing within a system and environment.

3.2.1 Simple conditions

Entity bound conditions

An entity bound simple condition must adhere to the following syntax.

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

Figure 3.2: Example of Domain model in UML with description of operations.

[determinant] entity verb value

Elements between “[]” are optional.

A *simple condition* starts with an optional *determinant* followed by an *entity*, a *verb*, and a *value*. Notice that UCed grammars are not case sensitive.

- Possible **determinants** are “a”, “an” and “the”.

A condition may or may not start with a determinant with no difference in the meaning. For instance, the conditions ‘User is logged in’, ‘A User is logged in’, ‘The User is logged in’, and ‘An User is logged in’ are all equivalent. Notice that UCed doesn’t check that determinant are correctly used according to the English grammar.

- An **entity** consists of one or more words specified as

$$word_1 \cdots word_n$$

The sequence of words must correspond to a **concept** (an actor or the system under consideration) or an **attribute** of a **concept** in the domain model.

Entities names are specified in extension.

- to refer to a **concept** (or **system concept**),
the sequence used $word_1 \cdots word_n$ needs to correspond to a **concept** as declared in the domain model,
- to refer to an **attribute**
the sequence used $word_1 \cdots word_i \cdots word_n$ needs to be such that $word_1 \cdots word_i$ refers to an entity and $word_{i+1} \cdots word_n$ refers to an **attribute** of the entity as declared in the domain model,
- to refer to the **component** of a **concept** (an **aggregate**)
the sequence used $word_1 \cdots word_i \cdots word_n$ needs to be such that $word_1 \cdots word_i$ refers to an entity and $word_{i+1} \cdots word_n$ refers to an **aggregate** of the entity as declared in the domain model,

As examples,

the sequence “*User number of attempts*” must be used to refer to **attribute** “*number of attempts*” of **concept** “*User*”

the sequence “*User Display*” must be used to refer to **component** “*Display*” of **concept** “*User*”

- The **verb** must be derived from “*to be*” or “*to have*”, and the present tense must be used.

Therefore the only possible **verb** expressions are:

“*is*”

“*isn't*”

“*is not*”

“*are*”

“*aren't*”

“*are not*”

“*has*”

“hasn’t”

“has not”

“have”

“haven’t”

“have not”

“can”

“can’t”

“can not”

As for determinants, UCED does not check the correct English usage of verb expressions. As an example, *“User card is inserted”* and *“User card are inserted”* are equally valid.

- A value is an entity *qualifier*.

A value may be one of the possible values of the condition entity, or specified as a general comparison.

- For possible values, the sequence of words used to refer to the value must be declared as a possible value of the entity in the domain model.

As an example, for *“User identification is valid”*, *“valid”* must be declared as a possible value of entity *“User identification”*.

- For general comparisons, the syntax for value specification must adhere to the following

comparator value

The comparator must be one of the following: *“>”*, *“<”*, *“=”*, *“<=”*, *“>=”*, *“<>”*, *“greater than”*, *“less than”*, *“equal to”*, *“different to”*, *“greater or equal to”*, or *“less or equal to”*.

The value must be a sequence of words referring or not to a numerical value.

When a comparison is used, the entity in the condition must be defined with no possible value listed in the domain model.

As an example attribute *“number of attempts”* of concept *“User”* doesn’t have any possible value in the *PMSystem* domain. This

attribute can therefore be used in a condition such as “*User number of attempts is > 5*” where a value is specified as a general comparison.

On the other hand, suppose attribute “*identification*” has a set of possible values defined, only conditions involving these possible values would be allowed.

Entities with possible values such as “*identification*” are *discrete* entities, while entities without possible values such as “*number of attempts*” are *non discrete*.

Non-Entity bound conditions

A non-entity bound condition is a proposition declared in the domain (see Table 3.1). The condition must appear as declared in the domain.

3.2.2 Complex conditions

A complex condition is a negation, a conjunction or a disjunction of conditions.

- A negation is a condition has one of the forms:

“NO” condition

“NOT” condition

An example of negation is “*Not User identification is valid*”.

Notice that negation may also be introduced in the **verb form** as in “*Not User identification is not valid*”. Both forms are equivalents.

- A conjunction/disjunction of a condition has one of the forms:

condition “AND” condition

condition “OR” condition

The default associativity of “AND” and “OR” is from the left to the right. Parentheses may be used to alter that order.

3.3 Any conditions

An **any condition** refers to a set of conditions on a same entity. Any-conditions may only be used as withdrawn-conditions.

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 an 'individual' condition the same way as an added-condition. However, it is sometime useful to refer to all the conditions affecting 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, 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.

The syntax for an **any condition** is as follow

“ANY” “ON” entity [*]

The condition specification must start with keywords “ANY ON”, followed by an entity name. A wildcard “*” may be used to refer to the entity sub-entities in addition to the entity itself.

As an example “*ANY ON User*” refers to all the conditions with “*User*” as the 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 conditions on “*User*”, as well as all the conditions on attributes of User such as “*User identification*”, “*User number of attempts*” and on sub-components such as “*User Card*”.

3.4 Operation declaration

Concept operations need to be declared in the domain model in the format

action_verb [action_object]

- **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*”).

3.5 UCED Domain Model Edition tool

Figure 3.3 shows UCED Domain Model Edition tool. The tool has the same

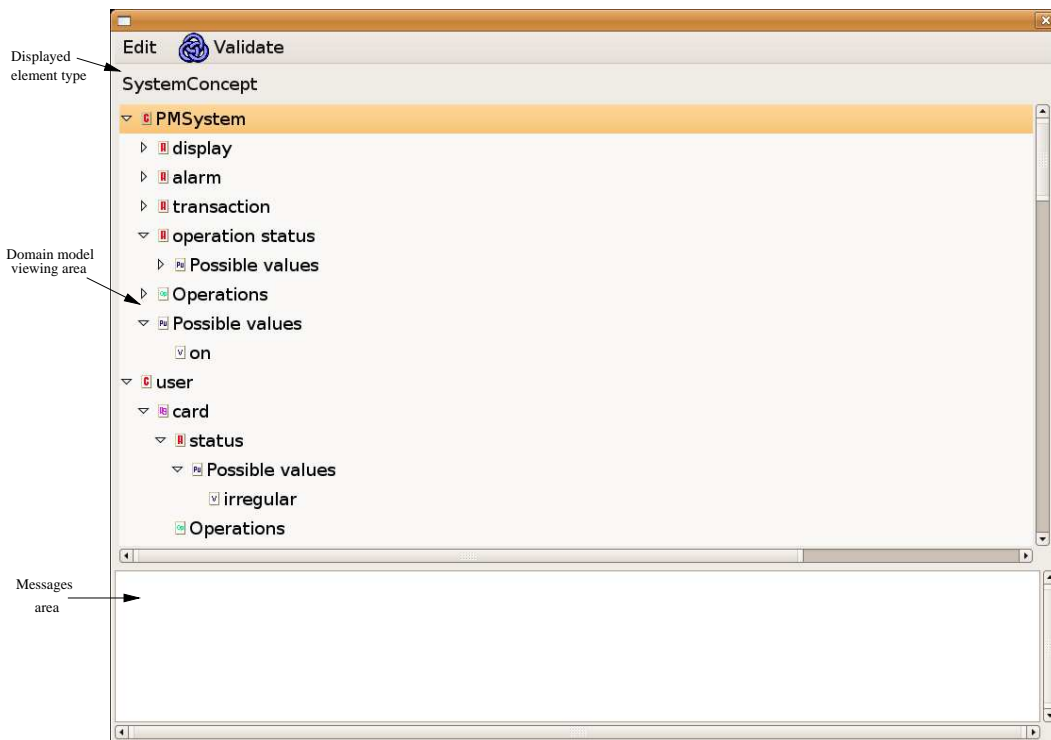


Figure 3.3: UCED Domain Model Edition tool.

look and functionality as the Scenario Edition tool.

The domain model is displayed as a *tree* in a domain model viewing area. Elements are edited by double-clicking or by using the contextual menu. The tool message area displays various messages including validation error messages. The **Edit** menu allows common edition actions, while the **Validate** menu launches domain model validation.

Edit menu

Domain Model Edition tool, Scenario Model Edition tool as well as Use Case Edition tool **edit** allows the following editing operations.

Undo to undo the latest action (Keyboard shortcut **CTRL-z**).

Redo to redo the latest undone action (Keyboard shortcut **CTRL-ALT-z**).

Copy to copy the current line in an editor (Keyboard shortcut **ALT-c**).

Paste to paste a copied line, or the more recently deleted line at the current position (Keyboard shortcut **ALT-v**).

Notice that pasting may not be possible if the copied line cannot be validly inserted at the current position.

3.5.1 Domain Model element types

Table 3.1 shows the UCED representation of domain element types, the icon associated with each of the elements in the editor, and the type of their possible children. An excerpt of domain concept description is shown in Figure 3.4.

3.5.2 Domain elements edition

A domain model can be edited by *left-clicking* on an element in the viewing area to select it, and then *right-clicking*. That will bring a context dependent menu, which allows operation to be performed on the selected element.

The menu displayed for a **Concept** includes:

New System Concept to add a new System Concept to the domain model.

New Concept to add a new Concept to the domain model.

Add Non-Entity bound condition to declare a new Non-Entity bound condition.

Change Concept to System Concept to transform the concept to a system concept if possible.













Domain element types	Icon	Possible children
System Concept		Concept, Aggregate, Attribute, Possible Value Set, Operation Set
Concept		Concept, Aggregate, Attribute, Possible Value Set, Operation Set
Aggregate		Aggregate, Attribute, Possible Value Set, Operation Set
Attribute		Possible Value Set
Operation Set		Operation
Operation		Precondition, Added Condition, Withdrawn Condition
Possible Value Set		Value
Value		
Operation Precondition		
Added Condition		
Withdrawn Condition		
Non Entity Bound Condition		

Table 3.1: UCED representation of domain elements.

Change Concept to Aggregate to transform the concept to an aggregate if possible.

Add Sub-concept to Concept to insert a sub-concept as a child of the concept (inheritance relation). Once inserted, the User may enter the sub-concept name through the **Edition area**.

Add Sub-component to Concept to insert an aggregate as a child of the concept (aggregation relation).

Add Attribute to Concept to insert an attribute as a child of the concept.

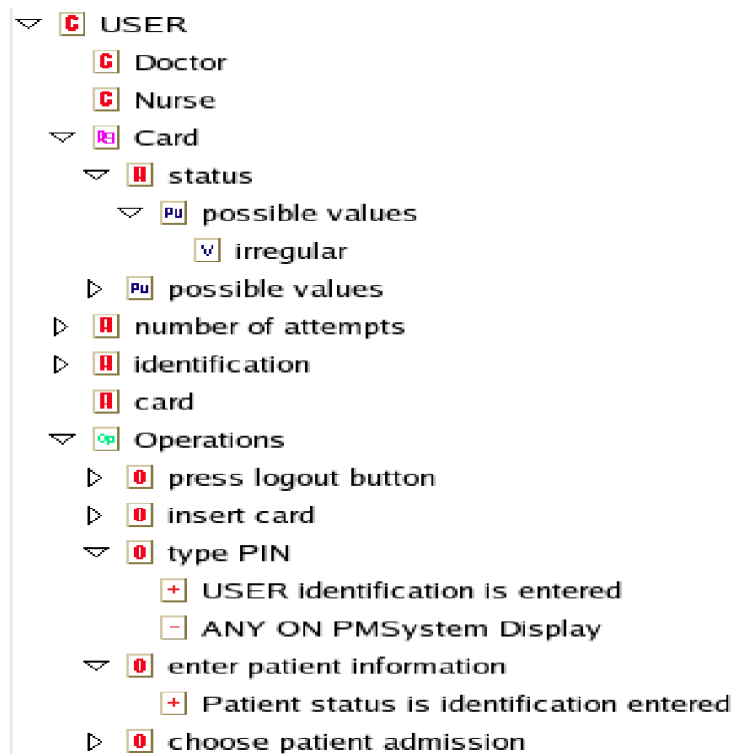


Figure 3.4: Example of UCEd domain concept.

Add OperationSet to Concept to insert an operation set as a child of the concept.

Add ConceptInstance to Concept to insert an instance as a child of the concept (instantiation relation).

Add PossibleValue Set to Concept to insert a possible value set as a child of the concept.

Delete Concept to delete the concept with all its children.

Rename Concept to change the value of the concept.

A domain model may also be edited using key combinations (see Section 3.5.3). For instance if a selected line refers to a **Concept**:

CTRL+ inserts a concept after that concept.

- CTRL-** delete the selected concept with all its sub-elements.
- CTRL→** indent the concept if it is not on the first line. The indented concept becomes a sub-concept (a specialization) of the preceding concept (or system concept).
- CTRL←** outdents the concept if it is a sub-concept of a concept (or system concept). The concept becomes a concept at the same level as the preceding concept.
- CTRL↑** forwards the selected concept by changing its type. The concept is changed to a **System** concept if it is at the top-level of the domain.
- CTRL↓** backwards the selected concept by changing its type. A concept, which is a sub-concept of another concept is transformed to an **Aggregate** of that concept.

3.5.3 Editors key combinations

Domain Model Edition tool, Scenario Model Edition tool as well as Use Case Edition tool allows entering commands by using one of the following key combinations:

CTRL+ to insert a line after the selected line.

CTRL- to delete the selected line.

CTRL→ to indent the selected line.

CTRL← to outdent the selected line.

CTRL↑ to forward the selected line.

CTRL↓ to backward the selected line.

↑ or RETURN to select to the following line.

↑ to select the previous line.

An operation might not be possible depending on the context. The result of the operation performed depends on the selected element.

3.6 Domain Model validation

Selection of `Domain Validation` in the `Validate` menu launches domain model validation. Domain validation checks the following.

1. Names of domain elements must be unique in their scope of definition.
For examples:
 - at the domain level, there shouldn't be any duplicate concept/system concept,
 - all the attributes of a concept must have different names,
 - all the values in a possible values set must be different,
 - etc.
2. There must be at least one system concept at the domain level.
3. Each `added` and `withdrawn` condition must conform to the syntax in section 3.2.
4. *Entities* used in conditions must be present in the domain model (as `System Concepts`, `Concepts` or `Attributes`).
5. *Discrete Values* used in conditions must be defined as possible values of corresponding entities in the conditions.
6. If a condition uses a general comparison, the entity in that condition must have no possible value defined.
7. Any conditions must only be used as withdrawn conditions.

3.7 Extraction of Domain elements from Use cases

Sub-menu `Validate->Extract Domain from Use Cases` brings a `Domain Model Extractor` wizard that helps identify domain elements from use cases text elements.

Figure 3.5 shows the `Domain Model Extractor` wizard. The text element can be accepted as a non-entity bound condition or as an entity bound condition/use case operation.

Domain Model Extractor

Accept as Non-Entity bound Condition or, select Entity parts

Accept as non-entity bound condition pmsystem security is high

Select parts to match string

pmsystem security

Concept

System Concept pmsystem

Aggregation

Attribute

Operation

< Back Next > Finish Cancel

Figure 3.5: Domain Model Extractor

To accept as non-entity bound condition, select the box labelled “Accept as non-entity bound condition”.

Use section “Select parts to match string” to identify entities and operations. The section shows the condition, use case actor or the use case operation which needs to be analyzed in order to extract system concepts, concepts, aggregates, possible values or operations. The following are the parts of the wizard interface.

Concept to select a Concept from the string to be resolved. After selection, a concept name needs to be chosen by right-clicking at the right of the field.

System Concept to select a System Concept from the string to be resolved.

There can't be both a Concept and a System Concept within a string to be resolved.

Aggregation to select an aggregate from the string to be resolved.

Attribute to select an attribute from the string to be resolved.

Operation to select an operation from the string to be resolved.

With the `wordnetdata`¹ directory under the working directory (containing all wordnet data files), the system will detect the operation for you automatically. But this detection is based on extracting the operation under the assumption that its first word is a verb. But this is not guaranteed to work perfectly at all times as there could be words that belongs to both the noun and verb categories. In such a case you have to manually select the correct operation. If you don't have the wordnet data files then the system will require the user to select the operations manually.

A warning is displayed to the fact that the resulting model may be incorrect if the **Next** button is pressed while the string to be matched can't be obtained by concatenating all the selections made.

¹<http://www.cogsci.princeton.edu/~wn/>

Chapter 4

Use Cases

Use cases are narrative description of behaviors. The set of a system use cases with the actors that participate in these use cases, constitutes a *use case model*. A *use case diagram* is used as a representation of a use case model in the UML.

4.1 Use Case Diagrams

A use case diagram depicts use case names, actors, relations between actors and use cases, and relations between use cases.

A relation between an actor and a use case captures the fact that the actor participates in the use case.

Relations between use cases include the *include*, *extend* relations¹.

- The include relation denotes the inclusion of a use case as a sub-process of another use case (the base use case). By default, the control resumes to the including use case only after the successful completion of the included use case (*primary scenario*). Control resumption from *secondary scenarios* can be explicitly specified using [continue statement.
- The extend relation denotes an extension of a use case as addition of 'chunks' of behaviors defined in an extension use case. These chunks of behaviors are added at specific places in a base use case called **extension points**.

¹The current version of UCED doesn't support the UML *specialization* relation.

An extend relation also includes a condition under which the extension can take place.

Figure 4.1 shows a UML use case diagram. The actors are 'Patient' and

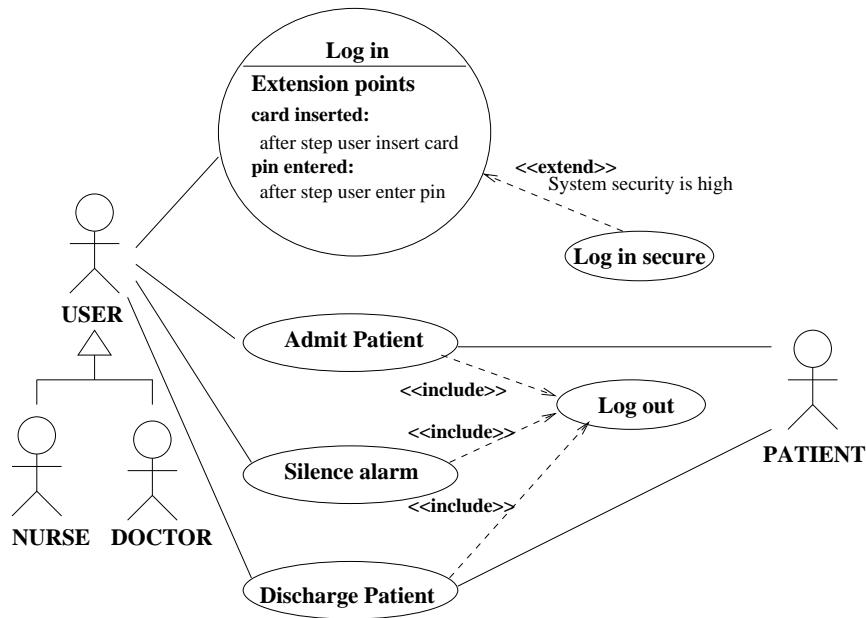


Figure 4.1: Example of Use Case diagram for a PM System.

'USER'. 'Doctor' and 'Nurse' are specializations of USER. The diagram includes use cases: 'Log in', 'Admit Patient', 'Silence alarm', 'Discharge Patient', 'Log out' and 'Log in secure'. 'Log out' is included by use cases: 'Admit Patient', 'Silence alarm' and 'Discharge Patient'. 'Log in secure' is an extension use case that extends use case 'Log in' under condition *System security is high*.

A use case model doesn't provide use cases details. UCED includes an editor for that purpose. The next section describes use case description elements.

4.2 Description of Use Cases

Use case diagrams are abstract high-level view of functionality. They do not describe use cases interactions. 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* interactions between actors and a system. Different *templates* and *guidelines* for use cases edition have been proposed in the literature. UCED template is largely based on [1] where use cases are described using structured text.

We distinguish two types of use case descriptions corresponding to the two types of use cases. *Normal use case description* for *normal* use cases and *extension use case description* for *extension* use cases.

4.2.1 Normal use cases description

A normal use case includes a *Title*, *Description*, *System Under Design* field, *Scope* field, *Level* field, a *Primary Actor* field, a *Participants* field, a *Goal* field, a list of *Following Use Cases* field, a *Precondition*, an *Invariant*, a *Success Postcondition*, a sequence of *Steps*, a set of *Any Extensions*, and a set of *Extension Points*. The description of these elements is as follow.

- **Title:** a label that uniquely identifies the use case within the use case model.
- **Description:** a free form text that summarizes the use case.
- **System Under Design** specifies what is the *system* in the use case. The system is the entity that react to stimuli from *external actors*.
- **Scope:** specifies what system is being considered black box under design (e.g. whole business, system, sub-system, function).

The different possible values for scope are defined in property file `traits.properties`.

- **Level:** is the use case level of detail.

The different possible values for scope are defined in property file `traits.properties`.

- **Primary Actor:** the actor that initiates the use case.

- **Participants:** other actors participating in the use case.
- **Goal:** a statement of the primary actor expectation at the successful completion of the use case.
- **Follows Use Cases:** a list of normal use cases that the use case directly follow.

A *follow list* expression reflects how the listed use cases are synchronized in relation to the described use case. Two operators are used. Operator **AND** expresses *synchronization* (*synchronized follow list*), while operator **OR** expresses *asynchronism* (*unsynchronized follow list*).

Given use cases $uc_0, uc_1, uc_2, \dots, uc_n$, the following interpretation is given.

- If the *follow list* of uc_0 is specified as “ uc_1 **AND** uc_2 **AND** $\dots uc_n$ ”, all of uc_1, uc_2, \dots, uc_n must reach a point from which use case uc_0 is *enabled* before use case uc_0 (synchronism).
- If the *follow list* of uc_0 is specified as “ uc_1 **OR** uc_2 **OR** $\dots uc_n$ ”, use case uc_0 may be executed as soon as any of use cases uc_1, \dots, uc_n reaches a point from which use case uc_0 is *enabled* (asynchronism).

When a use case uc_i refers to a use case uc_j in its *follow list*, use case uc_j must include an **enabling directive** referring to use case uc_i .

- **Invariant:** a condition that must hold through the use case.
- **Precondition:** a condition that must hold before an instance of the use case can be executed.
- **Success Postcondition:** a condition that must be true at the end of a ‘successful’ execution of an instance of the use case.
- **Steps:** a sequence of **repeat blocks** and steps.

A repeat block is a sequence of steps that are supposed to repeat based on time and/or condition.

Each step references a **use case step operation**.

A **use case step operation** may be a *concept operation instance*, a *branching statement* or a *use case sequencing statement*.

A step may also include a set of **step extensions** and an **alternative postcondition**. Each step extension defines an alternative behavior (*secondary scenario*) following the step. Alternative postconditions are conditions that must hold at the end of the *secondary scenario* defined by an step extension.

- **Any Extensions:** a set of **step extensions** that applies to all the steps in the use case.
- **Extension Points:** a set of locations in the use case steps where additional behaviors defined in extension use cases might be inserted.

A use case description can be seen as a two parts description with:

- a **static** part that includes the use case *Title, Description, System Under Design, Scope, Level, Primary Actor, a Participants, Goal, a list of Following Use Cases, Precondition, Invariant, and Success Postcondition* fields,
- a **dynamic or procedural** part that consists of the use case *Steps*.

Repeat Blocks

A repeat block defines a sequence of steps that iterate according to a condition and/or delay. A repeat block is introduced as follow.

“repeat” “while” condition

“repeat” “every” duration_specification

“repeat” “every” duration_specification, “while” condition

The first form introduces repetitive steps controlled by a condition, the second form repetitive steps controlled by time and the third form introduces repetitive steps controlled by time and a condition.

Concept operation instances

A concept operation instance denotes an operation execution by a concept (a concept being an actor or the system under consideration). Concept operation instances must follow the syntax


```
[delay_specification] [condition_statement] [determinant] entity
operation_reference
```

Syntaxes for `determinant` and `entity` are defined in section 3.2.

Delay specification

The two types of delays are `before` and `after` delays. They specify an amount of time relative to the completion of the previous step (or to when the pre-condition became true in case of the first step).

For a `before` delay, the step must be initiated before that delay. For an `after` delay, the delay needs to pass before the step can be executed. None, only one or both delays may be included in an operation instance statement. A `before` delay syntax is

```
“before” duration_specification
```

An `after` delay syntax is

```
“after” duration_specification
```

A `duration_specification` specifies a time amount as follow

```
value unit
```

A `value` is a word that may refer to a number or not.

A `unit` must be one of: `"mmsec"`, `"microsecond"`, `"msec"`, `"millisecond"`, `"sec"`, `"second"`, `"min"`, `"minute"`, `"h"`, `"hour"`, or `"day"`.

Condition statements

A condition statement is used to constraint a use case step.

The following format is used for condition statements

```
“IF” condition “THEN”
```

The `condition` need to hold for the step to be possible.

Notice that no subsequent step is executed when a use case step doesn't hold.

The use case is simply abandoned, unless a step extension can be executed.

Operation references

An operation reference denotes the execution of an operation by an **entity**. The operation must be declared as one of the entity operations according to a specific format (see section 3.4).

An `operation_reference` has the following form:

```
action_specification [action_participant]
```

The `action_specification` has the form

```
conjugated_action_verb [(binding_word)+] action_object
```

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

UCEd recognizes the regular form of present formation; the addition of 's' to the infinitive, as well as exceptional cases ending in 'es'.

- A `binding_word` may be:
 - a possessive adjective "his", "her" or "its" or,
 - an article "a", "an", "the" or,
 - a preposition "to", "for".

None or any number of `binding_words` may appear between the `conjugated_action_verb` and the `action_object`.

As an example, "*validates the user identification*", "*validates her user identification*" are valid `operation_references` given operation "*validate user identification*".

Notice that UCEd doesn't verify the English correctness of sentences therefore "*validates the a its user identification*" would also be accepted.

- The optional `action_participant` is an arbitrary sequence of words that are left uninterpreted by UCEd.

In other words as soon as the concept operation has been recognized, any other following word is irrelevant.

For instance, given the operation declaration "*validate user identification*", "*validates the user identification with the Branch*" is an `operation_reference` with `action_participant` specification "with the Branch".

Branching statement

Branching statements are used to *jump* from a step in a use case to another step.

A branching statement has the form

```
[delay_specification] [condition_statement] "GOTO" ["STEP"] step_reference
```

- A `step_reference` is a number or a label assigned to a step within a use case.

Branching statements can only appear as last in a procedure.

Use case sequencing statements

Use case sequencing statements are: inclusion statement, enabling directives and continue statement.

- A use case inclusion statement has the following syntax

```
[condition_statement] "include" ["use" "case"] use_case_name
```

A use case inclusion statement refers to an included use case. The meaning of a use case inclusion is that the steps of the included use case replace the inclusion statement in the base use case. The remaining steps of the base use case are normally executed after the included use case primary scenario. **Continue statements** may be used to specify continuation after specific secondary scenarios in addition to the primary scenario.

- A use case enabling directive has the following syntax

```
[condition_statement] "enable" ["in" "parallel"] ["use" "case"]
[use_case_names] or
[condition_statement] "resume" ["in" "parallel"] ["use" "case"]
[use_case_names]
```

`use_case_names` is a list of use case names that are *enabled* for execution following the directive.

- If `use_case_names` is not specified, the statement applies to the use case itself and results in its repetition.

- If “in” “parallel” is specified, the use cases in the list referred by *use_case_names* are set to execute in parallel. Otherwise, the next operation forces a choice among these use cases.

For each use case referred to in an enabling directive, the enabled use case must include the enabling use case in its *follow list*.

- A continue statement has the following syntax

```
[condition_statement] “continue” [“use” “case”]
```

Continue statements are meaningful only within included use cases. A continue statement specifies that the including use case execution continue from a scenario even if it is a secondary scenario.

A continue statements must be last in a scenario as any subsequent statement would be unreachable.

Step extension

A **step extension** specifies an alternative behavior that may follow a step. The *enabling condition* of a step extension is specified as follow:

```
delay_specification [condition_statement], or  
condition
```

The *enabling condition* of a step extension may be

- a **before** delay,
- an **after** delay,
- a **condition**,
- a combination of before and after delays, or
- a combination of before, after delays and a condition.

A step extension specifies a sequence of **extension actions**. Each extension action may be a *concept operation instance*, a *branching statement* or a *reference* to an *included use case*.

Example of normal use case

Figure 4.2 shows the details of the use case *Log in*. The use case *Log in* describes a login procedure that must be used by the users of the PM System. Use case *Log in* precondition is condition “*PMSystem is ON*”, and postcondition is condition “*User is logged in*”.

Log in includes six steps listed in the section titled *Steps*. Each step refers to a **use case step operation**.

For instance step 1 references the User’s operation ‘insert card’ and step 2 the PMSystem operation ‘ask pin’. Step 5 includes a step condition ‘User identification is valid’. The execution of step 5 (and that of the following step) is possible only under that condition. Step 6 operation is constrained by an **after delay** ‘After 45 sec’. Means 45 seconds need to pass after the completion of step 5 before this step is possible.

Steps 1, 2 and 4 have **step extensions** described in the section titled *Extensions*. Each extension label starts with the corresponding step label. Extension 1a corresponds to step 1, 2a corresponds to step 2. Extensions 4a and 4b correspond to step 4.

Extensions 1a, 4a and 4b include conditions. For instance, after step 1 is completed the operation 1a1 followed by 1a2 are executed if the condition ‘User Card is not regular’ evaluates to *true*. Extension 2a includes an **after delay**, such that 60 seconds need to pass after step 2 operation completion for the extension to be possible.

All the extension actions in use case *Log in* are **operation instances**. The exception is extension 4a that includes a **branching** statement branching back to step 2.

Use case *Log in* includes one **any extension**. Any extension labels start with ‘*’. *Log in* any extension doesn’t include a condition or a delay. Consequently, the extension is possible anytime during the execution of an instance of the use case.

Use case *Log in* includes the extension points labelled **card inserted** after step 1, and **pin entered** after step 3. An extension use case such as *Log in secure* shown in Figure 4.3 can refer to these extension points.

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

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: PM System is ON

Postcondition: User is logged in

Steps:

- 1: User inserts a Card in the card slot
- 2: PMSystem asks for PIN
- 3: User types her PIN
- 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

Extensions :

*1 :

*1a: USER presses cancel button

*1b: PMSystem ejects Card

1a: USER Card is not regular

1a1: PMSystem emits alarm

1a2: After 20 sec PMSystem ejects Card

2a: after 60 seconds

2a1: PMSystem emits 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 emits alarm

4b2: After 20 sec PMSystem ejects Card

EXTENSION POINTS :

STEP 1. card inserted

STEP 3. pin entered

- Title:** Log in secure
- Parts:** At extension point card inserted
- 1: System logs transaction
- At extension point pin entered
- 1: System logs transaction

Figure 4.3: Extension use case.

an example, use case *Log in* primary scenario is defined by step 1 to 6 operation (sequence 1-2-3-4-5-6). Examples of secondary scenarios are defined by sequences 1-1a1-1a2 and 1-2-3-4-4b1-4b2.

4.2.2 Extension use cases description

An extension use case includes a *Title* as previously defined, and one or more *parts*. These *parts* are to be inserted at specific *extension points* in a *base use case*.

A *part* includes:

- a reference to an extension point (defined in a base use case in reference to a step), and
- a sequence of *steps* defined as previously with the difference that extension use cases cannot include extension points.

As an example suppose the extension use case *Log in secure* shown in Figure 4.3. *Log in secure* extends use case *Log in* such that information provided by a user logging in is recorded. This extension use case includes two parts. One to be included at the extension point “card inserted” and the other at the extension point “pin entered”.

4.3 UCED Use Cases Edition tool

The UCED Use Case Edition tool shown in Figure 4.4 is used for use case models creation and edition. The editor includes two panes: a left pane used for use case models (a use case model is displayed in a tree form), and a right pane that shows the description of the selected use case in the model.

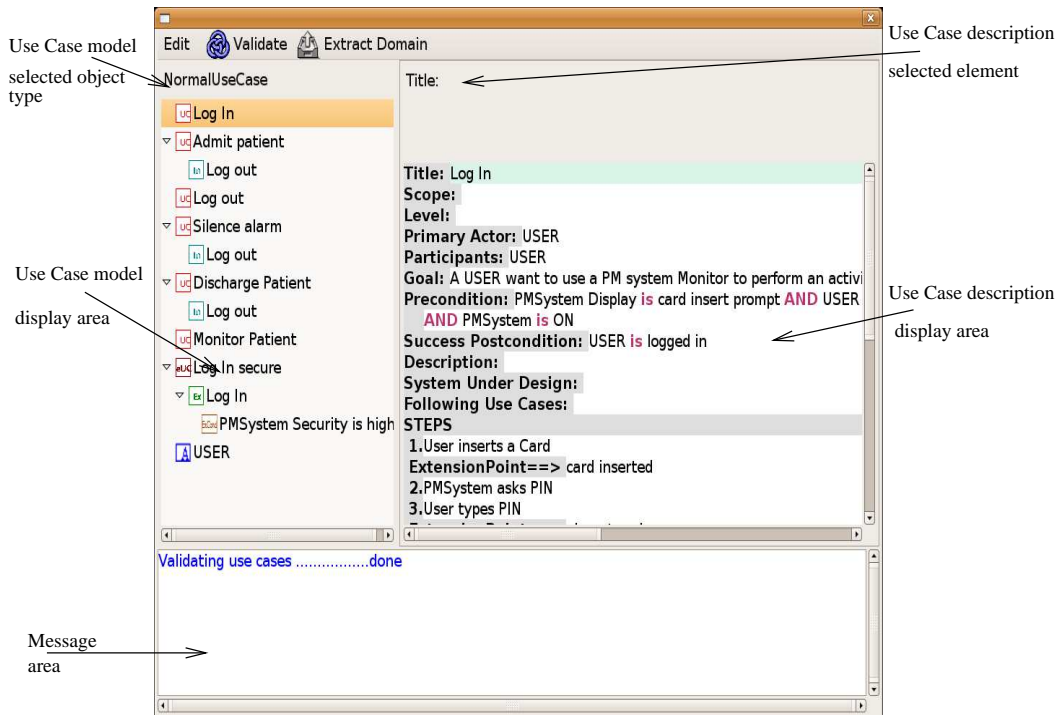


Figure 4.4: UCed Use Cases Edition tool.

The **Edit** menu allows common edition actions. **Validate** menu launches use cases validation and **Extract Domain**, the extraction of domain elements from use cases.

4.3.1 Use Case models edition

Table 4.1 shows the UCed representation of use case model element types, the icon associated with each of the elements in the editor, and the type of their possible children. Figure 4.5 shows an example of use case model.

Use case model edition, domain edition (see section 3.5.2), and scenario model edition (see section 7.2) are similar.

A use case model can be edited by *left-clicking* on an element in the viewing area, and then *right-clicking* to bring a context dependent menu that allows operations on the selected element. As an example, the menu for an Extension Use Case includes:

New Normal Use Case to add a Use Case to the use case model.







Use Case model element types	Icon	Possible children
Normal Use Case		Include Relation
Extend Use Case		Include Relation, Extend Relation
Include Relation		
Extend Relation		Extend Relation Condition
Extend Relation Condition		
Actor		

Table 4.1: UCed representation of use case model elements.

New Extend Use Case to add an Extension Use Case to the use case model.

New Actor to add an Actor to the use case model.

Add Include Relation to Extend Use Case to add an Include relation as a child of the selected use case.

Add Extend Relation to Extend Use Case to add an Extend relation as a child of the selected use case.

Delete Extend Use Case to delete the selected use case from the model.

A use case model may also be edited using key combinations (see Section 3.5.3). For instance if the selected line refers to a **Normal Use Case**:

CTRL+ inserts a new normal use case after that normal use case.

CTRL- deletes the selected normal use case with its description.

CTRL→ indent the use case if it is not on the first line and if it has no description. The use case is transformed to an **Include Relation**, which is the child of the preceding use case.

CTRL← has no effect.

CTRL↑ forwards the selected normal use case by changing its type to an **Extension Use Case**.

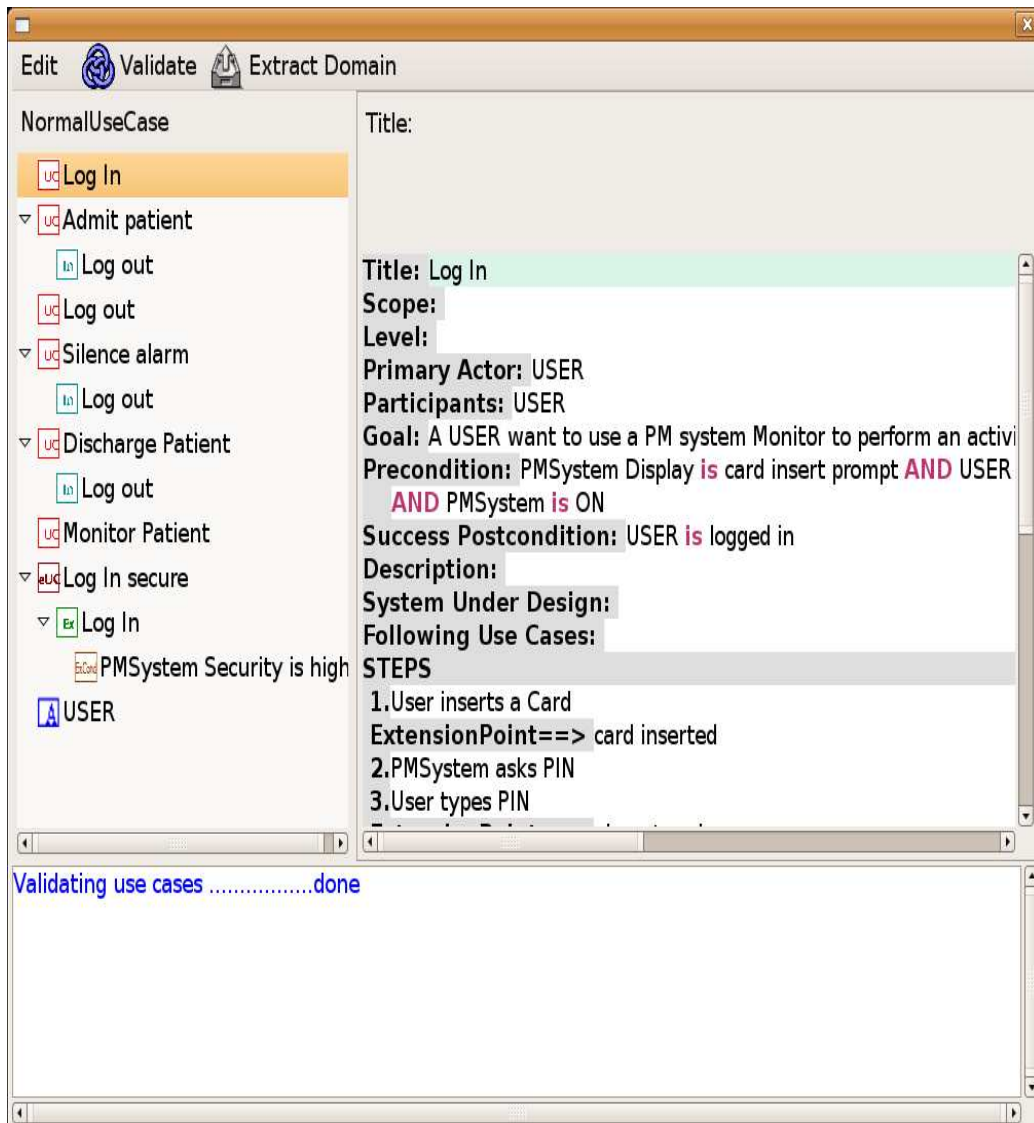


Figure 4.5: Example of UCed use case model.

CTRL↓ backwards the selected use if it has no description, by changing its type to an Actor.

4.3.2 Use Case descriptions edition

Figure 4.6 shows the initial view of a newly created *normal use case*. The

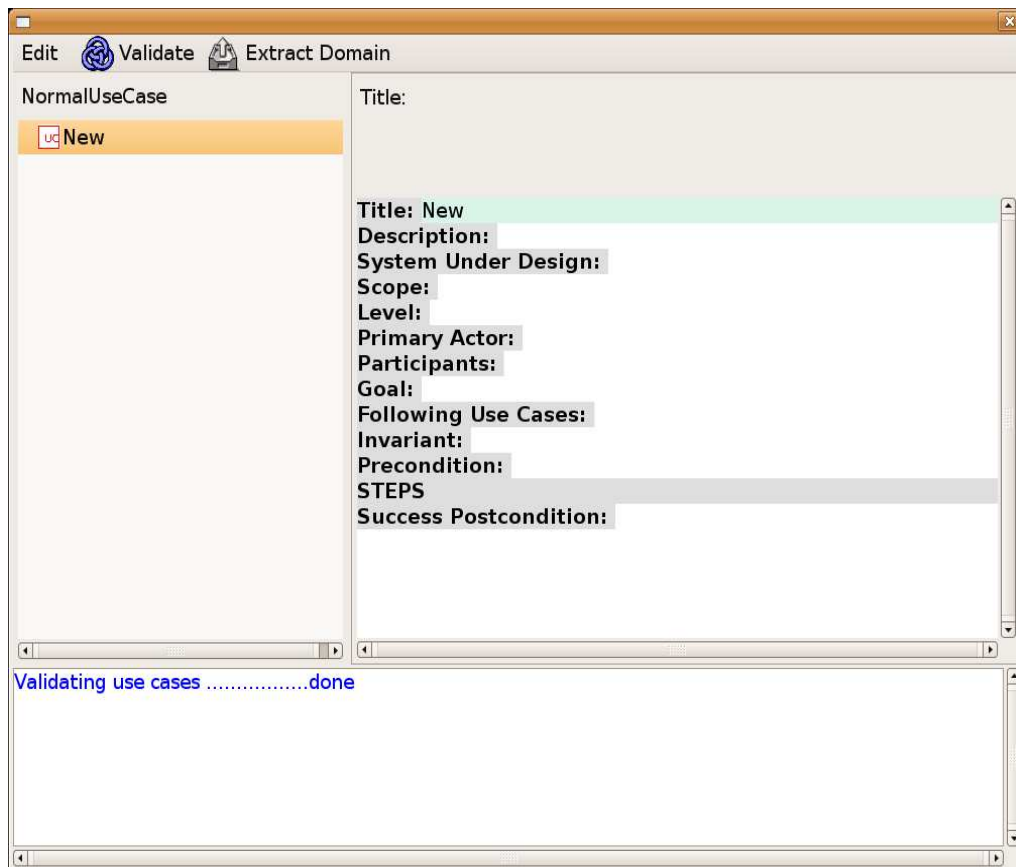


Figure 4.6: Initial description of a new use case in the editor.

description panel shows lines labeled with the use case fields *Title*, *Primary Actor*, *Participants*, *Goal*, *Precondition*, and *Postcondition* (only *Title* would be shown for an *extension use case*). These fields may be edited by selecting the corresponding line and entering a field value.

The first line in a use case **procedural** part is created by:

1. selecting the step separator line (*STEPS*), and
2. using the contextual menu or typing the key combination CTRL+ to insert a new line.

Figure 4.7 shows the editor in Figure 4.2 after insertion of the procedural part first line. For a *normal procedure*, the first line created is a *Step*. A *Part* is

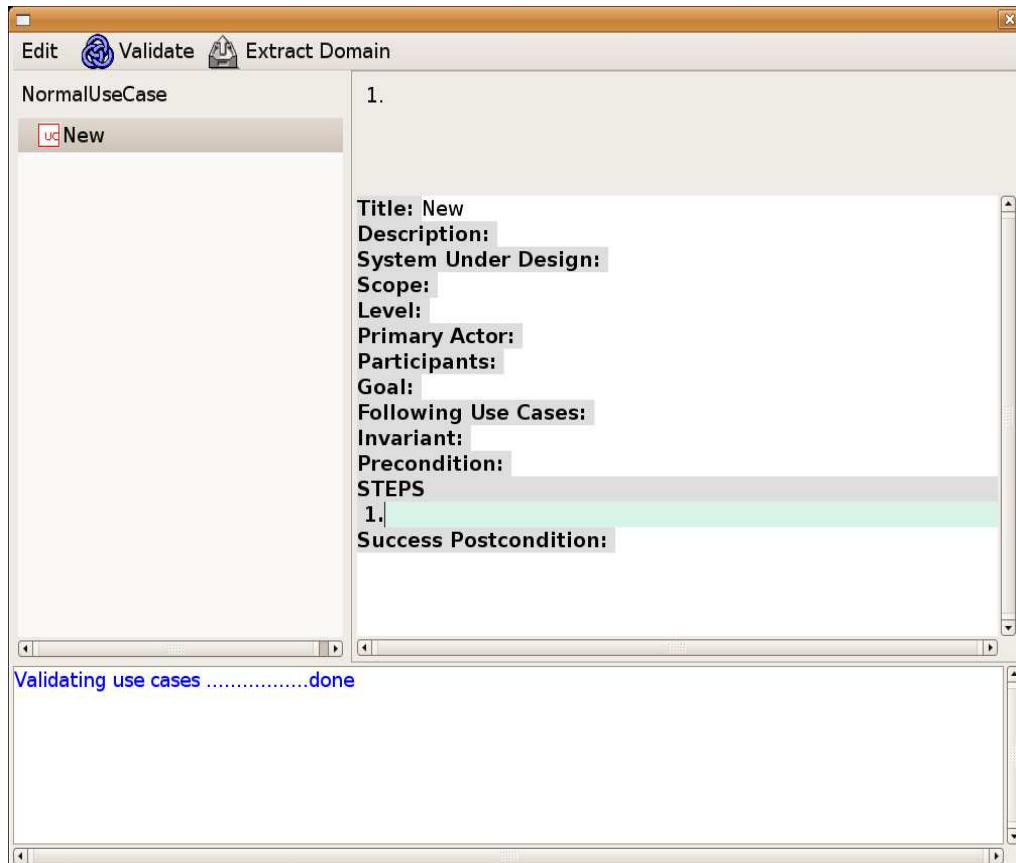


Figure 4.7: Description of a new use case in the editor after the first step creation.

created in the case of an *extension procedure*.

A use case procedural part is edited by selecting a line and using the contextual menu or a key combination (see Section 3.5.3). In addition to the standard keys:

CTRL F1 inserts an extension point after the current line (for *normal use cases* only).

CTRL F2 inserts an any extension to the use case (for *normal use cases* only).

The editor does lines labeling automatically.

- For a Part

CTRL+ inserts a part after the selected part.

CTRL- deletes the selected part.

CTRL→ indents the selected part as a Step of the preceding Part (if the selected line is not the first line).

CTRL← has no effect on parts.

CTRL↑ has no effect on parts.

CTRL↓ transforms the selected part to a Step of the preceding Part (if the selected line is not the first line).

- For a Step

CTRL+ inserts a step after the selected step.

CTRL- deletes the selected step.

CTRL→ indents the selected Step as an Extension of the preceding Step (if the selected line is not the first line).

CTRL← has no effect on steps.

CTRL↑ has no effect on steps.

CTRL↓ transforms the selected Step to an Extension of the preceding Step (if the selected line is not the first line).

- For an Extension

CTRL+ inserts an Operation instance as an action of the Extension.

CTRL- deletes the Extension.

CTRL→ indents the selected Extension as an action Operation instance of the preceding Extension (if the selected line is not the first Extension of a Step).

CTRL← outdents the selected Extension as a Step

CTRL↑ transforms the selected Extension to a Step (if it has no children).

CTRL↓ transforms the selected Extension to an Extension action Operation instance of the preceding Extension (if the selected line is not the first Extension of a Step).

- For an Extension action Operation instance

CTRL+ inserts an Operation instance as an action of the Extension.

CTRL- deletes the action.

CTRL→ has no effect on an Extension action.

CTRL← outdents the selected Extension action as an Extension.

CTRL↑ transforms the selected Extension action to an Extension.

CTRL↓ transforms the selected Extension action Operation instance to a Branching statement.

- For an Extension action Branching statement

CTRL+ inserts an Extension to the selected Branching statement Step.

CTRL- deletes the Branching statement.

CTRL→ has no effect on an Extension action.

CTRL← outdents the selected Branching statement as an Operation instance.

CTRL↑ transforms the selected Branching statement to an Operation instance.

CTRL↓ has no effect on a Branching statement.

- For an Extension Point

CTRL+ inserts a Step after the selected Extension Point.

CTRL- deletes the Extension Point.

CTRL→ has no effect on an Extension Point.

CTRL← has no effect on an Extension Point.

CTRL↑ has no effect on an Extension Point.

CTRL↓ has no effect on an Extension Point.

4.4 Use Cases validation

Selection of **Use Case Validation** in the **Validate** menu launches the current Use Case model validation. Use Case model validation checks the following.

1. Use case model elements must be unique. There shouldn't be any duplicate use case or actor.
2. All *include* and *extend* relation must refer to *normal use cases*
3. There must not be cycles of use cases *inclusion*.
4. A use case *Primary Actor* must be a domain **Concept**.
5. A use case *Precondition* must be a valid condition.
 - the syntax must conform to condition syntax described in section 3.2,
 - the *entities* used must be present in the domain model (as **System Concepts**, **Concepts** or **Attributes**),
 - the *discrete values* used in must be defined as **possible values** of the corresponding entities,
 - if a general comparison is used as value in a condition, the entity in that condition must have no possible value defined.
6. A use case *Postcondition* must be a valid condition (same as preconditions).
7. Use Case steps must follow the use case syntax defined in section 4.2.
 - When an **after delay** and a **before delay** are used in conjunction, the delay value of the **before delay** must be greater than the delay value of the **after delay**.
 - Step conditions must be valid conditions.
 - Operation instances must be present in the domain model.
 - Use case inclusion operations must refer to *normal* use cases.

Chapter 5

State Models

A State Model describes a system's behavior in interaction with its environment. UCED uses state models as frameworks for use cases integration. Two generation approaches are used: state model synthesis based on control flow and state model synthesis based on operation effects. Both approaches allow generation of state models in the StateChart[2] formalism.

Control flow based generation is appropriate at earlier stages when operations haven't been specified. Synthesis based on operation effects is useful to validate contract specification of operations and is therefore more appropriate at the later stage.

5.1 Control flow based state model

Control flow based state model generation relies on the *implicit* flow of control among use case events and the *explicit* flow of control among use cases as specified by *follow lists* coupled with enabling directives.

At the use case level, UCED generates a *StateChart-Chart*. A UML activity diagram[4] with use cases as nodes. Figure 5.1 is a StateChart-Chart generated by UCED as shown by the state model viewer. A StateChart-Chart includes control flow nodes (join, fork, merge, decision) that capture use case sequencing constraints expressed by *follow lists* and enabling directives.

The details of each use case can be shown by **double-clicking** on the use case node. Each use case corresponds to a *StateChart*. Statecharts are useful for reactive behavior model description. A StateChart is defined as a tuple $[Trig_c, Reac_c, G_c, S_c, S0_c, F_c]$.

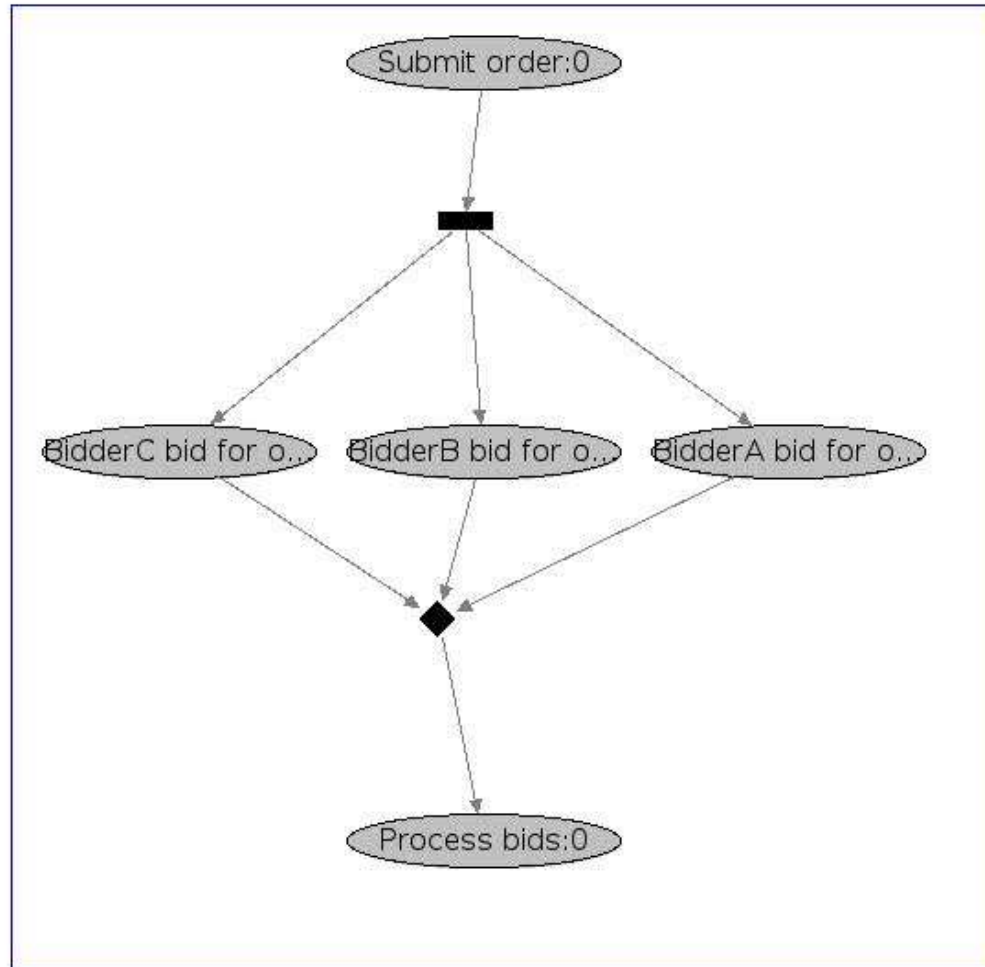


Figure 5.1: Generated StateChart-Chart.

- $Trig_c$ 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. The followings are notable characteristics of StateCharts states.

- Hierarchical states: a state may have *super-states* and *sub-states*. A system is in a state s is concurrently in all the sub-states of s . Any transition starting from a state s also applies to all the sub-states of s .
 - Parallel states: a system may be in several unrelated states at a same time.
- $S0_c \in S_c$ is the initial state of the statechart.
 - 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' .

Figure 5.2 shows the previous *StateChart-Chart* (Fig. 5.1) with all use cases StateCharts displayed. Generated StateCharts includes states and transitions representing use case interactions. A transition to a use case state (or a flow node) corresponds to an enabling directive. Such a transition results in the system entering the use case initial state.

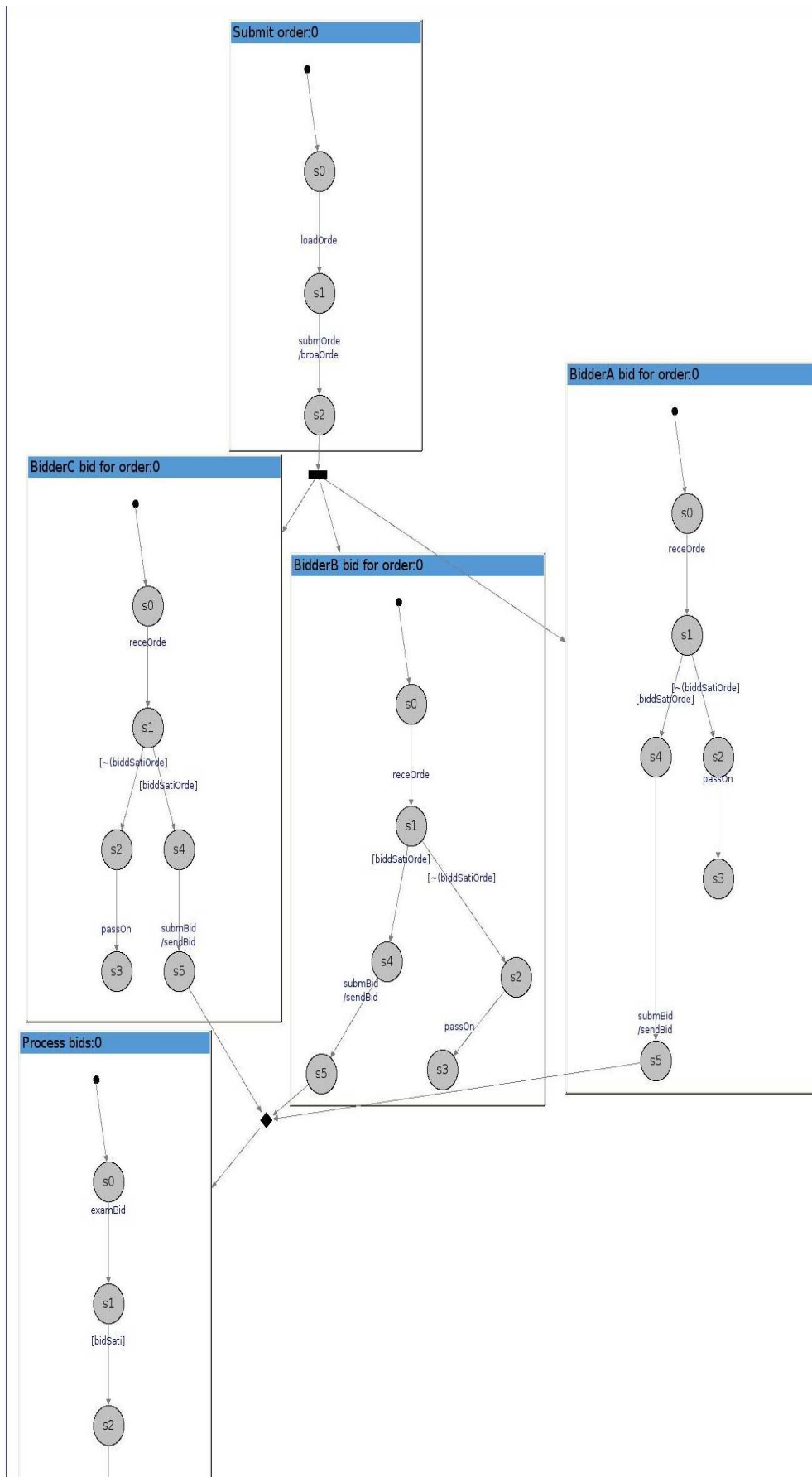
5.2 State models synthesis based on operation effects

State model synthesis based on operation effects proceeds by (1) a synthesis of a detailed version of StateChart and (2) transformation to a compact version if possible.

5.2.1 Detailed State Charts

Figure 5.3 shows a description of a detailed State Chart. In a detailed State Chart:

- Transitions are simple. Each includes a single *event* that can be: a system operation, an actor operation, a timeout or a guard condition.
- States describe *situations* in which the system and environment may be at some point in time.



```

STATE: 1[+ PMSystem is ON]
STATE: 2[+ Card is inserted,+ PMSystem is ON]
STATE: 3[+ Card is inserted,+ status is irregular,+ PMSystem is ON]
STATE: 4[+ Card is inserted,+ Alarm is System Status,+ status is irregular,+ PMSystem is ON]
STATE: 5[+ Card is inserted,+ Alarm is System Status,+ status is irregular,Timer0:20.0 second,+ PMSystem is ON]
STATE: 6[+ status is irregular,- Card is inserted,+ PMSystem is ON]
STATE: 7[+ Card is inserted,+ Display is pin enter prompt,+ PMSystem is ON]
STATE: 8[+ Card is inserted,+ Display is pin enter prompt,Timer1:60.0 second,+ PMSystem is ON]
STATE: 9[+ Card is inserted,+ Alarm is System Status,+ PMSystem is ON]
STATE: 10[+ Card is inserted,+ Alarm is System Status,+ PMSystem is ON,Timer2:20.0 second]
STATE: 11[- Card is inserted,+ PMSystem is ON]
STATE: 12[+ identification is entered,+ Card is inserted,+ PMSystem is ON]
STATE: 13[+ Card is inserted,+ PMSystem is ON,+ identification is invalid]
STATE: 14[+ Card is inserted,+ PMSystem is ON,+ identification is valid]
STATE: 15[+ Card is inserted,number of attempts < 4,+ PMSystem is ON,+ identification is invalid]
STATE: 16[+ Card is inserted,+ PMSystem is ON,number of attempts == 4,+ identification is invalid]
STATE: 17[+ Card is inserted,+ Alarm is System Status,number of attempts == 4,+ PMSystem is ON,+ identification is invalid]
STATE: 18[+ Card is inserted,Timer3:20.0 second,+ Alarm is System Status,number of attempts == 4,+ PMSystem is ON,+ identification is invalid]
STATE: 19[- Card is inserted,number of attempts == 4,+ PMSystem is ON,+ identification is invalid]
STATE: 20[+ Card is inserted,+ Display is welcome message,+ USER is logged in,+ PMSystem is ON]
STATE: 21[+ Card is inserted,+ Display is welcome message,+ USER is logged in,+ PMSystem is ON,Timer4:45.0 second]
STATE: 22[+ Display is welcome message,+ USER is logged in,- Card is inserted,+ PMSystem is ON]
STATE: 23[+ USER is canceling,+ PMSystem is ON]
STATE: 24[+ USER is canceling,- Card is inserted,+ PMSystem is ON]

**** TRANSITIONS ****
5 --[eject Card]-> 6
12 --[validate USER identification]-> 13
21 --[eject Card]-> 22
14 --[display welcome message]-> 20
9 --[TIMEOUT(Timer2:20.0 second)]-> 10
3 --[start System status alarm]-> 4
4 --[TIMEOUT(Timer0:20.0 second)]-> 5
10 --[eject Card]-> 11
7 --[type PIN]-> 12
16 --[start System status alarm]-> 17
17 --[TIMEOUT(Timer3:20.0 second)]-> 18
18 --[eject Card]-> 19
15 --[NULL Event]-> 2
2 --[ask Pin]-> 7
23 --[eject Card]-> 24
1 --[insert card]-> 2
1 --[press cancel button]-> 23
7 --[TIMEOUT(Timer1:60.0 second)]-> 8
12 --[validate USER identification]-> 14
20 --[TIMEOUT(Timer4:45.0 second)]-> 21
8 --[start System status alarm]-> 9

```

Figure 5.3: Example of a detailed state model

A *State* situation is defined as a set of *characteristic conditions*.

A characteristic condition is either a *condition* as defined in section 3.2 (i.e. a valuation of a domain *entity*) or a *timeout condition*.

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

States in Figure 5.3 are listed with their set of characteristic conditions.

As an example, state 4 characteristic conditions are $\{Card\ is\ inserted, Alarm\ is\ System\ Status, status\ is\ irregular, PMSystem\ is\ ON\}$.

Notice that all domain entities always have a value that may be explicit or *unknown*, in a given state. As an example, since attribute `Display` of the `PMSystem` doesn't have an explicitly defined value in state 4, its implicit value is *unknown* value.

A **timeout condition** reflects the 'relevancy' of the fact that a timer expired. As an example, timeout condition `Timer0:20 second` that characterizes state 5 means that timer `Timer0` that was set for *20 seconds* has expired, and that information is 'relevant'.

A state s_b is a *sub-state* of a state s_a (its *super-state*), if its characteristic conditions include those of s_a in the logical sense. As an example, state 1 in Figure 5.3 is a super-state of state 2. State 2 characteristic conditions include state 1 characteristic conditions.

A state may have more than one direct super-states. State 17 which direct super-states are states 9 and 16 is an example of such a state.

- A transition specifies the change from a state (a starting situation) to another state (a resulting situation).

5.2.2 State machine synthesis

UCed implements an algorithm [7] for the generation of a state machine from use cases. The principle of the state machine generation algorithm is as follow. For each use case, we augment an initially empty state transition machine with states and transitions such that each *scenario* in the use case is included as state transition sequences in the state transition machine. We use the operations effects (**added** and **withdrawn** conditions) to determine states.

Suppose “-” is an operator such that C_1 and C_2 being 2 sets of conditions, $C_1 - C_2$ is a set obtained by removing all the conditions in C_2 from C_1 , and $C_1 + C_2$ is a set obtained by adding all the conditions in C_2 to C_1 . Given a state s such that $cond(s)$ are the characteristic conditions 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 $cond(s') = (cond(s) - withdr_conds(op)) + add_conds(op)$.

The finite state machine generation algorithm augments a state model with a use case as follow.

- First we determine a set of states corresponding to the use case preconditions.
- Then, starting from these first states, we follow each scenario by adding transitions corresponding to operations.
- After each transition, the resulting state is used as a starting point for the following operation in a scenario.
- Timeout triggered transitions are created to account for delays.

The state model in Figure 5.3 has been generated from use case *Log in* shown in Figure 4.2. State 1 set of characteristic conditions is the use case set of preconditions {“*PMSystem is ON*”}. State 2 is obtained by considering the use case step 1. This state is characterized by the set of conditions {“*System is ON*”, “*Card is inserted*”} since the operation “*insert Card*” adds the condition “*Card is inserted*” according to the domain model in Figure 3.1. The algorithm generates state 3 when adding step 1 extension 1a. State 3 is a *sub-state* of state 2 because of 1a *extension condition* “*Card status is irregular*”. State 3 characteristic conditions are {“*PMSystem is ON*”, “*Card is inserted*”, “*Card is irregular*”}. Transition 7 ---[**TIMEOUT(Timer1:60.0 second)**]--> 8 is an example of **timeout-triggered transition**. This transition is created to consider extension 2a after delay. Timer1 is considered to be set as the state 7 is entered. Therefore the timeout event is produced if more than 60.0 seconds passes while in that state.

5.2.3 State chart generation

UCed generates compact State Charts from detailed State Charts. A difference between the two forms of State Charts is that a compact State Chart transition may include a *trigger*, a set of *guards* and a set of *reactions*. While there is a single event per transition in a detailed State Chart.

Compact State Chart generation is impossible when a detailed State Chart includes non-deterministic transitions. In order for compact State Chart generation to be possible the set of all transitions starting from a state *s* (including transitions from the super-states of *s*) should:

- consists on transitions on triggers only, or

- consists on transitions on guard conditions only, or
- consists on a single transition on a system reaction.

Chapter 6

Simulator tool

Figure 6.1 shows UCEd simulator tool used to simulate the current state model.

The simulator includes an **actor operations** panel (left panel) and a **simulation results** panel (right panel).

The actor operations panel includes a line for each of the actor operation such that clicking on that line triggers the given operation.

The simulation results panel includes different areas.

Previous State displays the label of the states and characteristics conditions before the latest actor operation.

System Reaction displays all the reactions of the simulated transition. .

Current State displays the current states labels and the current characteristic conditions holding in the system (if an operation effects-based State Chart is being simulated).

At the beginning of a simulation session, the current state is initially the state model initial state and the current characteristic conditions are the initial state characteristic conditions. As the simulation goes on, transitions taken change the set of characteristic conditions and current states.

6.1 Operation of the simulator tool

Simulation menu sub-menu **Start New Simulation** starts a fresh simulation session. Initially, the previous states area is empty and the current states

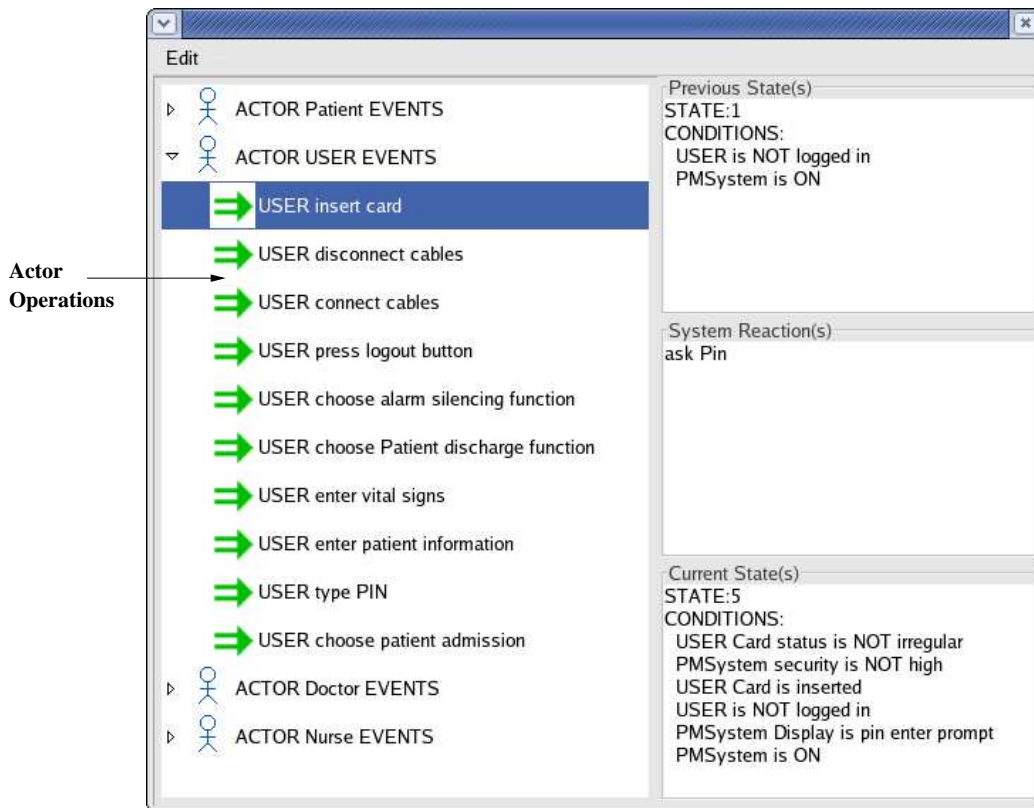


Figure 6.1: Simulator tool view

area shows the label and characteristic conditions of the state model initial state.

Clicking on a trigger in the actors operations panel, selects the corresponding event.

If the state model doesn't include a transition triggered by the selected operation from any of the current states, the simulator displays a message and the current states remain unchanged.

If there is a transition triggered by the selected operation, its reactions are added to the System Reactions area and the current states altered to include the transition resulting state.

When a state from which there are outgoing transitions with *guards* is reached, the simulator prompts the User such that one of the guards is chosen. Figure 6.2 shows an example of interaction for a guard choice.

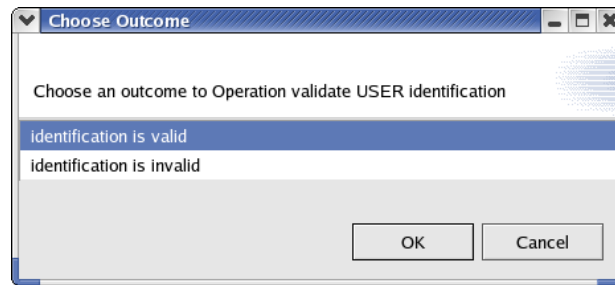


Figure 6.2: Prompt for guard choice

When a timeout triggered transition is encountered, the simulator prompts the User for a choice between letting enough time pass for the timeout or not. Figure 6.3 shows an example of interaction for a time delay choice.



Figure 6.3: Prompt for time delay choice.

6.2 Simulation History

Each event in a simulation session is recorded in a scenario. The simulation history shown in Figure 6.4, is the set of scenarios obtained from simulation.

Scenarios can not be edited in the simulation history viewer. They may however be moved to the scenario model edition tool.

Left-clicking on a line in the history viewer, then *right-clicking*, brings a menu which allows moving the selected scenario to the scenario model edition tool. Once moved, a scenario can be subject to all the editing operations supported by the scenario model edition tool.



Figure 6.4: View of Simulation History.

Chapter 7

Scenario Model

A scenario describes a sequence of interactions involving a system and actors of that system. Use Cases are collection of scenarios. However a scenario is not limited to a single use case. A scenario may cross over several use cases.

Scenarios are useful to document interactions of interest that may be wanted or not and to serve as repeatable *scripts* for simulation.

7.1 Elements of scenarios

A scenario is a sequence of: *triggers*, *system reactions*, *waiting delays*, *guard realizations* and *assertions*. Figure 7.1 shows a scenario that describes a normal login to the PMSystem.

- A *trigger* is an operation of an actor of the system (a **concept** in the domain model).

Line 1 of scenario *Normal login* is a trigger corresponding to operation “insert card” of concept “User”.

- A *system reaction* is an operation of the system under consideration (a **system concept** in the domain model).

Line 3 of scenario *Normal login* is a system reaction.

- A *waiting delay* specifies a point in a scenario where a certain amount of time passes without any trigger or system reaction.

Line 4 of scenario *Normal login* is a waiting delay of 30 seconds.

Scenario: Normal login

1. Trigger: USER insert card
2. Guard: USER Card status is NOT irregular AND PMSystem security is NOT high
3. Reaction: PMSystem ask Pin
4. Wait: 30 sec
5. Trigger: USER type PIN
6. Reaction: PMSystem validate USER identification
7. Guard: USER identification is valid
8. Reaction: PMSystem display welcome message
9. Assertion: User is logged

Figure 7.1: Scenario describing a normal login to the PMSystem.

Timeouts may be enabled by waiting delays during scenario execution.

- A *guard realization* is a condition set to hold at a certain point in a scenario.

Line 7 of scenario *Normal login* is a guard realization.

Guard realizations are used set conditions necessary for choosing among several execution paths.

- A *assertion* is a condition that needs to be true at a certain point in a scenario.

Line 9 of scenario *Normal login* is an assertion.

Assertions serve to check that certain conditions are realized at specific points in an interaction sequence.

A scenario may be “positive” (by default) or “negative”. A positive scenario describes interactions that need to be supported while a negative scenario describes interactions that need to be avoided.

7.2 Scenario Model Edition tool

Figure 7.2 shows UCed Scenario Edition tool. The tool has the same look as

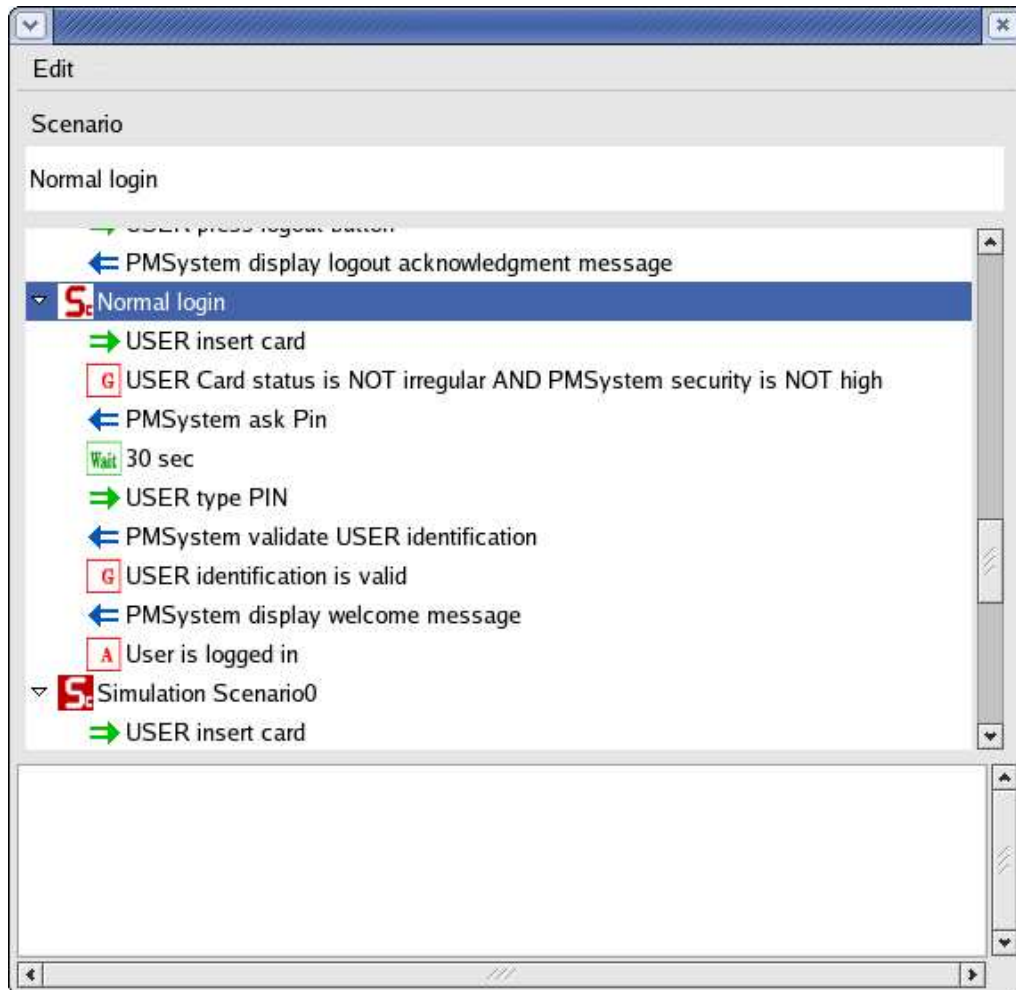


Figure 7.2: Scenario edition tool.

the Domain Model Edition tool. A scenario model is displayed as a *tree* in the scenario viewing area. The tool message area displays various messages including validation error messages and scenario simulation results. The **Edit** menu allows common edition actions.

7.2.1 Scenario model element types

Table 7.1 shows the UCed representation of scenario model element types, the icon associated with each of the elements in the editor, and the type of their possible children.








Scenario model element types	Icon	Possible children
Scenario		Trigger, System Reaction, Guard Realization, Assertion, Waiting Delay
Negative Scenario		Trigger, System Reaction, Guard Realization, Assertion, Waiting Delay
Trigger		
System Reaction		
Guard Realization		
Assertion		
Waiting Delay		

Table 7.1: UCed representation of scenario elements.

7.2.2 Scenario model edition

A scenario model can be edited by *left-clicking* on an element in the viewing area, and then *right-clicking*. That will bring a context dependent menu, which allows operation to be performed on the selected element.

- The menu displayed for a Scenario includes:

New Scenario to add a new “Positive” Scenario to the scenario model.

New Negative Scenario to add a new Negative Scenario to the model.

Change to Negative Scenario to set the Scenario as a Negative Scenario.

For a Negative Scenario, the corresponding menu option would be **Change to Scenario** to set the Scenario as “*Positive*”.

Add Trigger to add a Trigger at the end of the Scenario.

Add System Reaction to add a System Reaction at the end of the Scenario.

Add Delay to add a Waiting Delay at the end of the Scenario.

Add Guard to add a Guard Realization at the end of the Scenario.

Add Assertion to add an Assertion at the end of the Scenario.

Edit Scenario Description to edit a text description associated with the Scenario.

Validate Scenario to check the scenario elements according to scenario validation rules.

Simulate Scenario to simulate the execution of the Scenario.

Notice that a scenario needs to be validated before its simulation.

Export Scenario to produce a HTML output from the scenario.

Delete Scenario to delete the Scenario with all its children.

- The menu displayed for scenario elements includes:

New Scenario to add a new “*Positive*” Scenario to the scenario model.

New Negative Scenario to add a new Negative Scenario to the model.

Insert Trigger to insert a Trigger after the current selection.

Insert System Reaction to insert a System Reaction after the current selection.

Insert Delay to insert a Waiting Delay after the current selection.

Insert Guard to insert a Guard Realization after the current selection.

Insert Assertion to insert an Assertion after the current selection.

Delete Element to delete the current selection.

A scenario model may also be edited using key combinations (see Section 3.5.3).

7.3 Scenario validation rules

To be valid a scenario must satisfy the following rules.

1. Each *trigger* must correspond to a valid Concept operation.

[determinant] entity operation_reference

With *entity* a reference to a Concept.

2. Each *system reaction* must correspond to a valid System Concept operation.

[determinant] entity operation_reference

With *entity* a reference to the System Concept.

3. Each *guard realization* must be a valid condition.
4. Each *assertion* must be a valid condition.
5. Each *waiting delay* must be a valid duration specification (see section 4.2.1).

7.4 Scenario simulation

Scenario simulation consists of running UCED simulator using the scenario *triggers*, *guard realizations* and *waiting delays* to drive the simulation. *System reactions* are compared with the actual reactions produced by simulation; and *assertions* are used to check reached states.

Notice that a scenario need to be validated before its simulation.

Figure 7.3 shows a scenario simulation result. Simulation output is displayed in the scenario model editor tool message area. A scenario simulation proceeds according to the follow.

- The simulation is initiated from the state chart initial state, and the scenario elements considered in sequence.

The conditions corresponding to the current simulation state are displayed before each of the scenario actions.

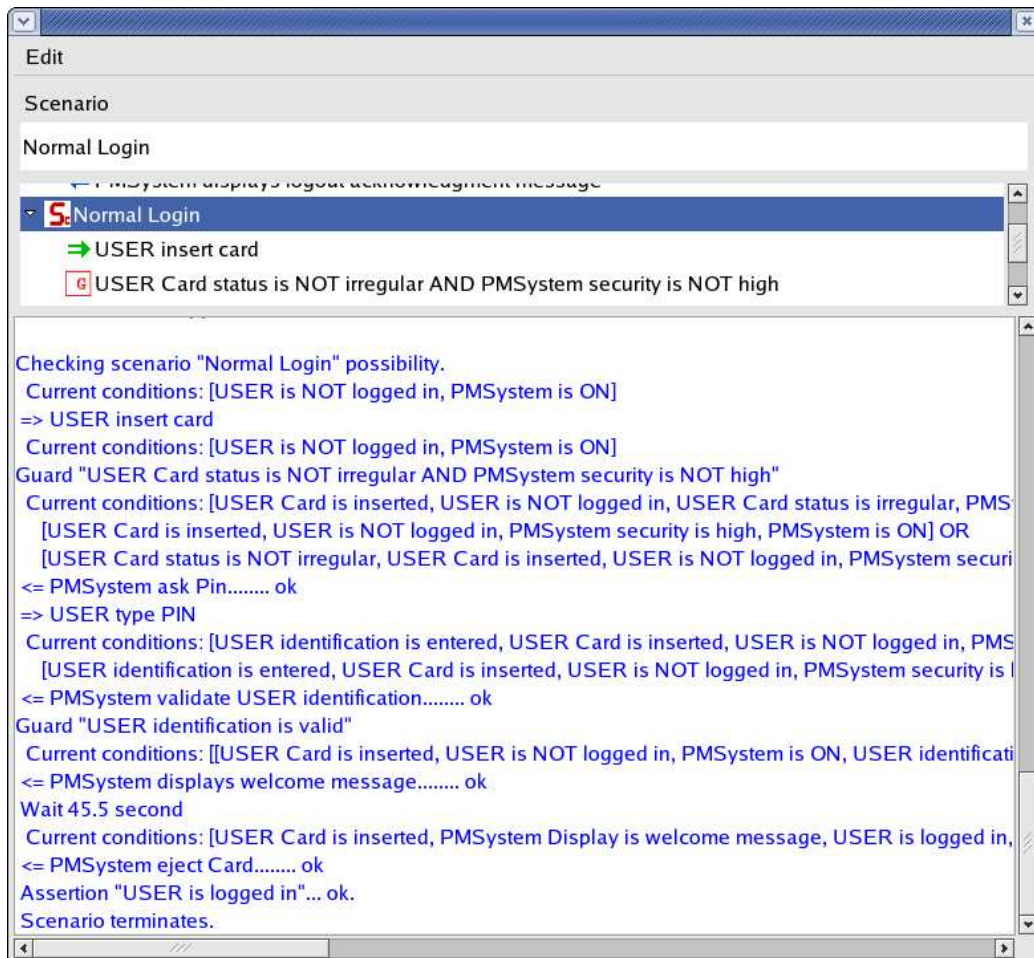


Figure 7.3: Scenario simulation result.

- A *trigger* is simulated by checking if there is a transition on it from the current simulation state. If so, and if the transition guards are satisfied, all the reactions are checked with the *system reactions* that immediately follow in the scenario.
 - A scenario fails and an error message is displayed if a trigger can not be fired.
 - If a scenario doesn't specify system reactions, any actual system reactions following a trigger are just displayed in the simulation output.

- If a scenario includes system reactions different to those obtained from the state model, a warning message is displayed but the simulation continues normally.
- When a state from which there are outgoing transitions with *guards* is reached, there must be a *guard realization* in the scenario at that point in the course of the simulation such that a single outcome is selected. An error message is displayed and the simulation fails when a selection is not possible.
- For a timeout to be simulated, there needs to be a *waiting delay* equal or greater than the timeout delay when the timeout transition is reached.

Bibliography

- [1] A. Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2001.
- [2] D. Harel. STATECHARTS: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
- [3] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1998.
- [4] OMG. UML 2.0 Superstructure, 2003. Object Management Group.
- [5] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [6] Geri Schneider and Jason P. Winters. *Applying Use Cases a practical guide*. Addison-Wesley, 1998.
- [7] S. Somé. An approach for the synthesis of state transition graphs from use cases. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'03)*, volume I, pages 456–462, june 2003.