Supporting Use Cases Based Requirements Simulation*

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
Email: ssome@site.uottawa.ca

Abstract *In this paper, we present an approach for requirements simulation in the context of requirements engineering. Simulation is an effective technique for requirement elicitation an early validation. This technique involves the derivation of a prototype from requirements. In our approach, requirements are described as use cases. A state machine automatically generated from use cases is used as a prototype. The approach is supported by a tool that creates a graphical environment for use cases simulation.*

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

1 Introduction

Requirements engineering includes the elicitation, understanding and representation of customers needs for a system. It is a critical tasks in software engineering; the source of a great number of software failures. The main reason for these requirements induced failures is a gap 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 others 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. Simulation of a prototype model derived from requirements is one way to bridge the gap between requirements and the development process. Simulation can help requirements validation and elicitation by allowing customers and requirements analysts to animate the original users requirements.

Use Cases [4] that describe possible interactions involving a system and its environment are increasingly being accepted as effective means for requirements elicitation and analysis [3]. An advantage is the intuitive and partial nature of use cases. A use case describes a piece of a system behavior without revealing the internal structure of the system. As such, use cases are useful to capture and document external requirements of systems. Use cases can also be used for requirements validation through prototyping and simulation [3]. In the current practice, however, informal definitions of use cases are used and the prototype derivation process is still manual. In [5], we proposed a formalization and a natural language based notation for use cases description. We also presented an algorithm for the integration of a set of related use cases in a finite state machine using a domain model. This paper presents an extension of the use case composition algorithm allowing to take into account extend relations. We also present an approach for requirement capture, clarification, and simulation built on this previous work. The approach is supported

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

by a tool called UCEd (Use Case Editor [1]) that takes a set of related use cases written in a restricted form of natural language and generates and executable specification that integrates the partial behaviors of the use cases. UCEd uses information contained in an application domain model for syntactical analysis of use cases and specification generation.

This paper is organized as follows. The next section outlines a general approach for use cases based requirements engineering. Section 3 presents an extension of our use cases composition algorithm for extension use cases. We discuss use cases simulation in section 4, and we conclude this paper in section 5.

2 Requirements engineering process

Figure 1 describes a Use Cases based requirement engineering process that starts with

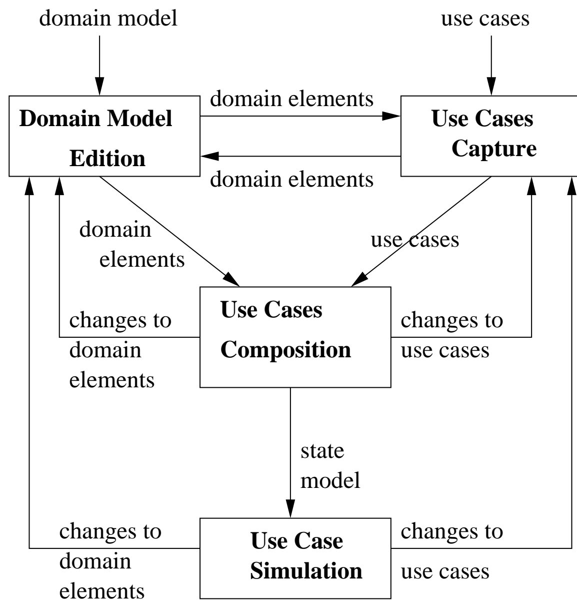
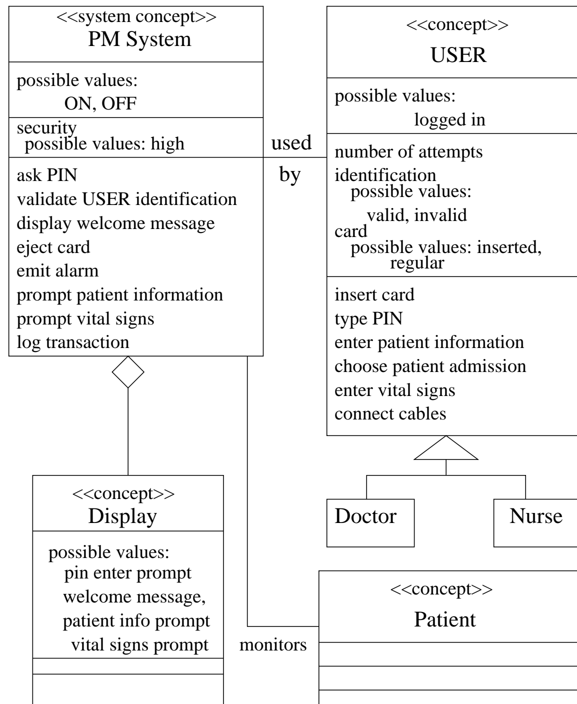


Figure 1: Use Cases based requirement engineering process. The boxes show activities and the arrows data elements passed.

an early view of requirements consisting of "rough" domain model and use cases, and produces a high-level state model specification of the system as well as clarified use cases and domain models. All the activities are supported by UCEd.

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. The domain concepts include the system as a black box with the "things" that exist or events that transpire in the environment in which the system works [3]. We use UML class diagrams extended with *stereotypes* [4] to describe domain models. Figure 2 shows a graphical representation of a domain model in the UML notation. In addition, we specify operation effects as added-conditions and withdrawn-conditions. Added-conditions denote conditions known to become *true* after an operation. Withdrawn-conditions are removed after the operation execution. We use added and withdrawn-conditions for use cases composition into finite state machines.

Use cases are narrative description of interactions involving a system and its environment. Use cases are defined in a *Use Case model* that consists of use cases, actors and relationships. 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*. Include relationships are already supported in the algorithm in [5]. We present an extension for the *extend* relationship in section 3. A use case diagram is a graphical depiction of a use case model that does not include a description of use cases interactions. We developed a restricted form of natural language for use case description [5]. Figure 3 shows an example of use case diagram and Figure 4, the details of a use case in our



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

Figure 2: Example of domain model description for a Patient Monitoring System.

use case description language.

Use Cases composition is based on: the specification of operations effects as withdrawn

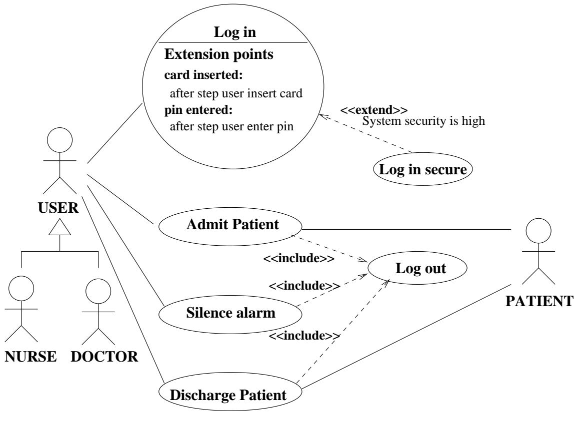


Figure 3: Example of Use Case diagram.

Title: Log in

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

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: PMSystem displays a welcome message to User
6: PMSystem ejects Card

Extensions:
1a: User Card is not regular
1a1: PMSystem emits alarm
1a2: PMSystem ejects Card
4a: User identification is invalid AND User number of attempts is less than 4
4a1 Go back to Step 2
4b: User identification is invalid AND User number of attempts is equal to 4
4b1: PMSystem emits an alarm
4b2: PMSystem ejects Card

Postcondition: User is logged in

Figure 4: Use case "Log in" describing a login procedure in the Patient Monitoring System.

and added-conditions, and a relation between *states* and conditions such that each state is defined by *characteristic conditions* holding in it. Figure 5 shows a state transition machine generated from use case "Log in". The composition algorithm allows the integration of several related use cases in a same state machine.

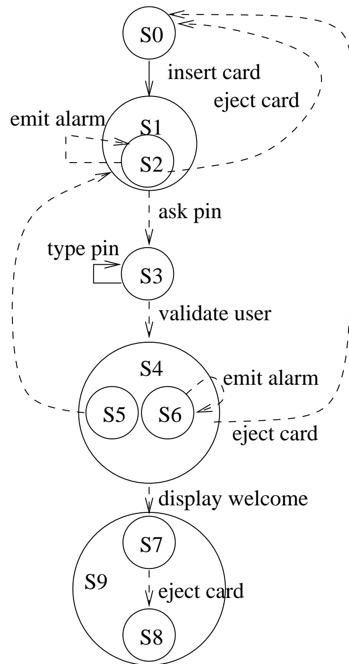


Figure 5: State machine generated from use case "Log in". Plain transitions correspond to external actor actions while dotted transitions correspond to the system responses. The state machine is hierarchical. States may include sub-states and a transition going from a state applies to all its sub-states.

State transition machines can be used as prototypes. UCEd includes a Use Case Simulator that generates a prototype with a graphical user interface from state machines. Using the interface, UCEd allows "playing" the use cases giving an opportunity to validate requirements and uncover possible interactions between use cases. Section 4 describes use case simulation in more detail.

Title: Log in secure

Parts: At extension point card inserted

1: System logs transaction

At extension point pin entered

1: System logs transaction

Figure 6: Extension use case "Log in secure". Extension points are defined in use case "Log in".

3 Extension use cases

An extension use case includes one ore more *parts* that are to be inserted at specific *extension points* in a *base use case*. An extension use case is a tuple [*Title*, *Parts*] with: *Title* a label identifying the use case and *Parts* a set of parts. Each part is a tuple [*ExtPoint*, *Steps*] with *ExtPoint* a reference to an extension point (defined in the UML specification) and *Steps* an ordered sequence of steps. 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 condition under which the extension can take place, and *ExtPoints* are a set of extension points referred to in the extension use case. As an example suppose the extension use case *Log in secure* shown in Figure 6. *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.

For extension use cases composition, we keep a correspondence between extension points and states as follow. In the state machine generation algorithm, when a step referred by an extension point is considered, we associate the resulting state of that step with the extension point. Let *corr_state(extp)* be a function such that given the extension point *extp*, *corr_state* returns the state corresponding to *extp*. The generation of a state transition machine in

presence of an extension proceeds as shown in Figure 7. Figure 8 shows the state machine

Given: $[\Sigma, S, F, S0]$ a state machine generated from *UCbase*, extension use case $UCext = [Title, Parts]$, extend relation $[UCbase, UCext, ExtCond, ExtPoints]$
For each parts $[extp_i, Steps_i]$ in *Parts*,

1 Let s_i be the state $corr_state(sext_i)$

2 Let $sext_i$ be a state such that $pred(sext_i) = pred(s_i) + ExCond$

3 add the steps $Steps_i$ from $sext_i$ (as in [5]).

Figure 7: State machine generation algorithm from extension use cases.

generated from the extension use case *Log in secure*. The composition starts here from the

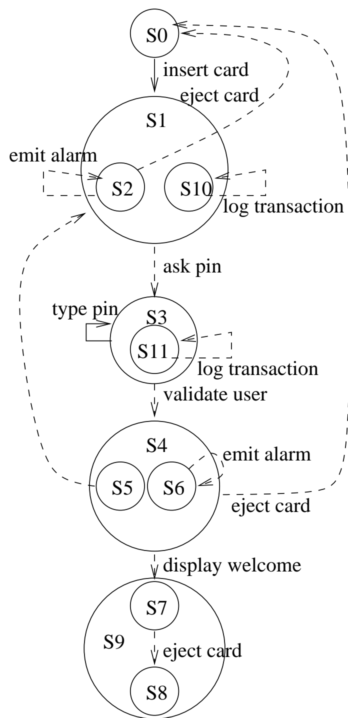


Figure 8: State machine generated from use case "Log in secure".

state machine shown in Figure 5. The extension point "card inserted" corresponds to the state *S1* and the extension point "pin entered" corresponds to the state *S3*. The algorithm creates the sub-states *S10* and *S11* of *S1* and *S3* with the additional characteristic condition "*System security is high*", and adds a transition from these states.

4 Use Cases simulation

Figure 9 shows a view of UCEd simulator. The

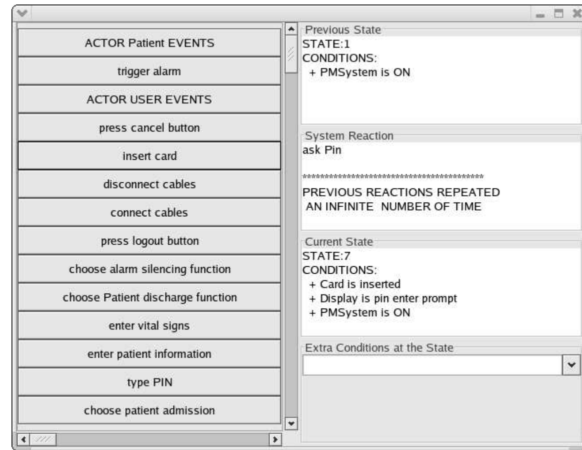


Figure 9: Simulator tool view.

simulator includes an "actor events panel" (left panel) and a "simulation results panel" (right panel). UCEd simulator tool provides an interface for use cases simulation using a generated state model as prototype. 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 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 state model 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 are more than one transition on the selected operation from the current

state, the simulator displays an error message to the fact that the state model (an therefore the use case and/or domain model) is ambiguous. This is a form of non deterministic behavior that might be an indication of erroneous use cases. 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 outgoing transitions to the reactions area. The final state reached then becomes the new current simulation state. When a simulated transition enters a state with *sub-states*, the simulator prompts the user such that the extra conditions of the sub-states might be enabled making thus the simulation switch to one of these sub-states. As an example suppose the simulation of the state machine obtained from use cases *Log in* and *Log in secure* shown in Figure 8. The simulation starts in state *S0* the state machine initial state. Suppose the user chooses operation *insert card*, the simulation moves to state *S1*. Since *S1* includes *S2* and *S10* as sub-states, the simulator prompts the user such that one of the conditions "Card is not regular" or "Security is high" may be enabled. These conditions are respectively states *S2* and *S10* extra conditions in comparison to *S1*. Suppose the user chooses to enable condition "Card is not regular". The simulator would move to state *S2*, add reactions "emit alarm" and "eject card" to the system reactions area, and set the new current state as state *S0*. Suppose now condition "Security is high" is chosen. The simulator would move to state *S10* and add operation "log transaction" to the system reactions area. Because there is no other transition from state *S10*, state *S1* transition to *S3* would then be considered. Operation "ask pin" would be added to the system reactions area and the simulation would continue from state *S3*.

5 Conclusion

We have presented an approach that aims at helping requirements engineering by allowing use cases simulation. Our work shares similarities with the play-in/play-out approach of Harel and Marelly [2]. 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.

References

[1] Use Case Editor (UCEd) toolset. http://www.site.uottawa.ca/s̃some/Use_Case_Editor_UCEd.html.

[2] D. Harel and R. Marelly. *Come, Let's Play.* Springler, 2003.

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

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

[5] 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.