# Enhancement of a Use Cases based Requirements Engineering approach with Scenarios

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

## Abstract

*Use Cases and Scenarios are sometimes considered as analogous. In this paper, we take the position that use cases and scenarios are different but play complementary roles in requirements engineering. A use case is a collection of scenarios. Use cases are appropriate as specification of a system required behavior in interaction with its actors. A scenario is an example of execution involving a system and its actors. A scenario may be defined with the intention that it should be supported or the intention that it should be avoided. Scenarios can thus be used to validate functional as well as non-functional requirements specification. We present an integration of scenarios to a use case based requirements engineering approach and discuss the merits of such integration.*

## 1. Introduction

Requirements engineering is known as a critical task in the software life cycle. According to different studies, a great number of project failures can be traced to poor requirements definition and lack of users involvement in the requirements engineering process. Early validation of requirements in conjunction with users is generally considered as a promising solution to that problem. The objective of requirements validation is to ensure requirements captured from users effectively describe their wishes for a system being developed. Because they are the source of requirements, users are the ultimate judges on the accuracy of what is specified as their requirements. Users involvement is therefore necessary for effective requirements validation. Not surprisingly users simulation of prototypes derived from requirements is one of the most effective requirements validation approaches. A problem with simulation is the time and effort needed to build prototypes. For the approach to be efficient, prototypes should be easy to obtain from requirements. A good prototype should also have a high degree of modifiability in order to quickly adapt to requirements volatility in the early stages of requirements engineering.

We are developing an approach where use cases are used to capture requirements [11]. A use case is the specification of a sequence of actions, including variants, that a system can perform, interacting with actors of the system [10]. Use cases have become one of the favorite approaches for requirements capture more so ever since their adoption by software development approaches such as the Unified Process [7]. We automatically generate executable state machines from use cases using information in a domain model [12]. The generated state machines are used as prototypes for requirements validation by simulation. Because of the automated generation, prototypes in our approach are obtained from requirements in a timely manner and with little effort. However, initial applications of the approach brought some requests for improvement from the users of the approach. One of these requests is to provide a mechanism that would facilitate repeatability of simulation sessions.

In this paper, we introduce *scenarios* as a response to this requirement. The terms "use case" and "scenario" are sometimes wrongly considered synonymous. We see a scenario as a single linear sequence of interactions between external actors and a system, while a use case integrates a set of scenarios (a main scenario and zero or more secondary scenarios). Moreover, use cases and scenarios have different purposes in the development process. As pointed out by Ian Alexander and Neil Maiden, there are two positions on the usage of scenarios/use cases: a *modernist* view and a *traditionalist* view [1]. We consider use cases for the modernist view and scenarios for the traditionalist view. In accordance with the modernist view, use cases are analytical description of how the system should react in interaction with actors. Use cases are part of the system functional specification. Scenarios on the other hand, are used in the traditionalist view. They are samples of interactions between actors and a system. As specifications, use cases needs to be complete. Scenarios are unlikely to be complete. An infinite number of scenario is possible in presence of loops. We consider use cases and scenarios as complimentary; use cases define what the system should do, and scenarios are used to validate use cases. Another interesting distinction is that use cases are limited to description of the functional aspect of a system, while scenarios may be used to capture some non-functional properties such as safety and security.

The main contributions of the paper are a definition of scenarios in relation with use cases and, the integration of scenarios to a use cases based requirements engineering approach. The rest of the paper is organized as follow. In the next section, we describe our use case based requirements engineering approach, and identify some limitations leading to the introduction of scenarios. We define scenarios as well as scenario execution in section 3. Section 4 discusses some related work and finally, section 5 concludes the paper with some future works.

## 2. Requirements engineering process

Figure 1 describes our use cases based requirements engineering process. It is an enhancement of an approach presented in [11]. The process is supported by a tool called Use Case Editor (UCEd) [13]. The following activities are

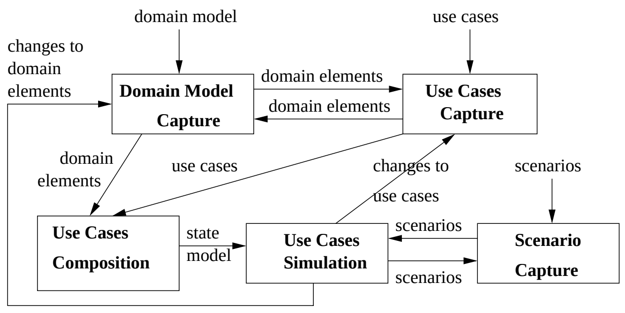


**Figure 1. Use cases based requirements engineering process. The boxes are activities and the arrows show data elements exchanged between these activities.**

included in the approach reported in [11]: domain model capture, use cases capture, use cases composition and use cases simulation. These activities are reviewed and illustrated with a case study in the remainder of this section. In this paper, we introduce scenarios as part of requirements, and add a scenario capture activity to the requirements engineering process. Our motivations are discussed in section 3. They can be summarized as (1) a necessity to make simulation recordable and repeatable, and (2) a necessity to be able to define what behaviors should be avoided in addition to required behaviors.

### 2.1. Use Cases

Use cases are defined in a *Use Case model* along with actors and relationships. Figure 2 shows a UML use case model for a web based Electronic Voting System (EVote System) and Figure 3 shows the description of three of the use cases. The use case template is based on Cock-

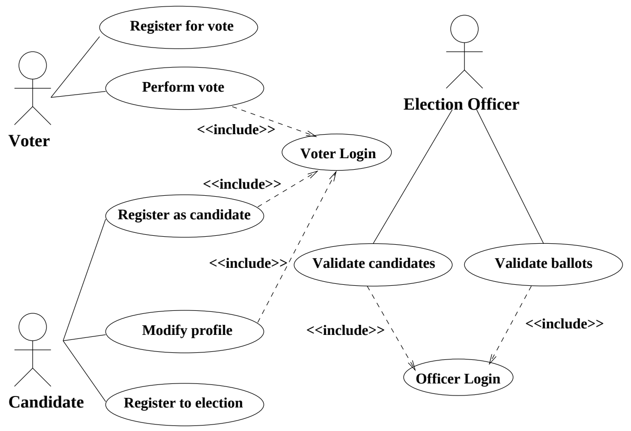


**Figure 2. UML representation of a use case diagram for a web based EVote System.**

burn's [3] and the restricted form of natural language used for use case description is presented in [11]. Each use case includes a *precondition* and a *postcondition*. The precondition must hold before the use case is executed, while the postcondition need to be true after a *successful* completion of the use case. A use case is collection of related *scenarios*. Each being a linear sequence of actor actions (triggers) and system reactions. Each use case includes a *primary scenario* (or main course of events) and 0 or more *secondary scenarios* that are alternative courses of events to the primary scenario. For instance, use case *Register for vote* primary scenario consists of lines *1* to *6*. Secondary scenarios from the same use cases are sequences *1 - 1.a.1*, *1 - 2 - 2.a.1*, *1 - 2 - 3 - 3.a.1* and *1 - 2 - 3 - 4 - 5 - 5.a.1*.

### 2.2. Domain Model

A Domain Model is a high level UML class model that captures *domain concepts* and their relationships. 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 [7]. For instance, actors that interact with the system are part of domain concepts.

```
Title: Register for vote
Primary Actor: Voter
Goal: An unregistered voter want to register
 in order to be able to vote. If successful,
 the system generates a login id and
 password for the voter.
Precondition: EVote System is online
Postcondition: Voter is registered
1. Voter loads EVote System web page
2. Voter selects voter registration option
3. EVote System asks for name, social
   security number, date of birth, adress
4. Voter provides name, social security
   number, date of birth, adress
5. EVote System checks Voter status
6. EVote System generates Voter login id
   and password
1.a. After 60 sec
1.a.1. EVote System displays time out page
2.a. After 60 sec
2.a.1. EVote System displays time out page
3.a. After 60 sec
3.a.1. EVote System displays time out page
5.a. Voter data is not in record
5.a.1. EVote System displays incorrect
    information error page
```

```
Title: Perform vote
Primary Actor: Voter
Precondition: EVote System is online
1. INCLUDE Voter log in
2. Voter selects register vote operation
3. EVote System displays open ballots page
4. Voter selects a ballot
5. EVote System displays selected ballot page
6. Voter marks his vote
7. EVote System prompts Voter for more votes
8. IF Voter prompt response is more votes
   THEN Go back to Step 3
3.a. After 60 sec
3.a.1. EVote System displays time out page
Title: Voter log in
Primary Actor: Voter
Precondition: EVote System is online
Postcondition: Voter is logged in
1. Voter loads login page
2. Voter enters login id and password
3. EVote System displays voter operation page
1.a. After 60 sec
1.a. 1. EVote System displays time out page
2.a. Voter is not registered
2.a. 1. EVote System displays incorrect
    information error page
```

**Figure 3. EVote System use cases "Register for Vote", "Perform Vote" and "Voter log in".**

We use definitions in the domain model as a knowledge base for the syntactical analysis of the natural language descriptions in use cases. Another important role of the domain model is the specification of concept operations effects as *preconditions* and *postconditions*. We refine operation postconditions into *added-conditions* and *withdrawn-conditions*. An added-condition becomes true after an operation. Withdrawn-conditions denote conditions that are removed after the operation execution. We consider a domain model as an integral part of requirements. Discovery of domain elements, particularly operation effects is an important part of the requirement engineering process.

Figure 4 shows a domain model for the EVote System. This model includes enough definitions for the syntactical analysis of use cases in Figure 3. Domain concepts are *EVote System*, the system under consideration, and *Voter* an actor. Specification of a concept includes attributes and operations. Operations in turn may be specified using preconditions, withdrawn-conditions and added-conditions. As an example, operation *select register vote operation* of concept *Voter* precondition is "Voter is logged in". That condition needs to be true for the operation to be possible. After completion of the operation, condition "Voter transaction status is vote operation selected" becomes true.

## 2.3. State Model

A state model includes *states* and *transitions*. States are defined by *characteristic conditions* holding in them. The principle of state model generation [12] is for each use case, to augment an initially empty state model with states and transitions such that all the scenarios in the use case are included as state transition sequences in the state transition machine. We use the operations' added and withdrawn-conditions to determine states. Suppose "-" is an operator such that $C_1$ and $C_2$ being 2 sets of predicates, $C_1 - C_2$ is a set obtained by removing all the predicates in $C_2$ from $C_1$, and $C_1 + C_2$ is a set obtained by adding all the predicates in $C_2$ to $C_1$. Given a state *s* such that *pred(s)* are the characteristic 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
*pred($s'$)* = (*pred(s)* - *withdr_conds(op)*) + *add_conds(op)*.

Figure 5 shows a state model generated from use cases in Figure 3 based on operation effects in Figure 4. The state model description includes a definition of states in term of characteristic conditions followed by state transitions. Characteristic conditions define a hierarchy of state inclusion. A state *s* is a *substate* of a state $s'$ if $s'$ characteristic conditions include those of *s*. For instance, state *2* is a substate of state *1*. Transitions are in the format `sd---trigger/reactions-->sa`, `sd---guards/reactions-->sa`, or `sd---TIMEOUT(delay)/reactions-->sa`. *sd* is a transition *departing state*, *sa* an *arrival* state, *trigger* an actor operation, *reactions* a sequence of system operations, and *guards* a set of conditions that need to evaluate to true for the transition to be possible. *TIMEOUT(delay)*

```
System Concept:EVote System
 Attribute:display
 Operation:ask name social security number
 Operation:check Voter status
  AddedCondition:Voter data is not in record
                 OR Voter data  is in record
 Operation:generate Voter login id password
  AddedCondition:Voter is registered
 Operation:display incorrect information error
  AddedCondition:EVote System display is
                 incorrect information page
 Operation:display voter operation page
  AddedCondition:EVote System display is
                 voter operation page
  AddedCondition:Voter is logged in
 Operation:display open ballots page
  AddedCondition:EVote System display is open ballots page
 Operation:display selected ballot page
  AddedCondition:EVote System display is ballot page
 Operation:prompt Voter more votes
  AddedCondition:Voter prompt response is more votes
   OR Voter prompt response is not more votes
 Operation:display time out page
  AddedCondition:EVote System display is timeout
Concept:Voter
 Attributes:transaction status, prompt response, data
 Operation:load EVote System web page
  AddedCondition:Voter transaction status is
                  main page selected
 Operation:select voter registration option
  AddedCondition:Voter transaction status is
                 registration selected
 Operation:provide name social security number
  AddedCondition:Voter transaction status is info entered
 Operation:load login page
  AddedCondition:Voter transaction status is
                 login page selected
 Operation:enter login id and password
  AddedCondition:Voter transaction status is login entered
 Operation:select register vote operation
  Precondition:Voter is logged in
  AddedCondition:Voter transaction status is
                 vote operation selected
 Operation:select ballot
  AddedCondition:Voter transaction status is
                 ballot selected
 Operation:mark vote
  AddedCondition:Voter transaction status is vote marked
```

**Figure 4. Domain model for the EVote System.**

```
State:1[System is online]
State:2[trans status is login page,System online]
State:3[System is online,trans status is main page]
State:4[System is online,trans status is main page,
        display is timeout]
State:5[System online,trans status is registration]
State:6[System online, display is timeout,
        trans status is registration]
State:8[data is in record, Voter is registered,
 trans status is information entered,System online]
State:9[data is NOT in record,System is online,
   trans status is information entered,
   display is incorrect information page]
State:11[trans status is login page,System online,
   display is timeout]
State:12[Voter is logged in, System is online, display is
  voter operation page, trans status is login entered]
State:13[Voter logged in,display is open ballots
 page,System online,trans status is vote operation]
State:14[System is online,display is timeout,
 Voter logged in, trans status is vote operation]
State:15[Voter is logged in, System is online,
 display is ballot page, trans status is ballot]
State:17[trans status is vote marked,
 prompt response is NOT more votes,System online,
 Voter is logged in,display is ballot page]
State:18[Voter NOT registered, System online,
 trans status is login entered,
 display is incorrect information page]
**** TRANSITIONS ***
 1---load EVote System web page/-->3
 1---load login page/-->2
 2---TIMEOUT(60)/display time out page-->11
 2---enter login id and password/-->10
 3---TIMEOUT(60)/display time out page-->4
 3---select voter registration option/
     ask name social security number -->5
 5---TIMEOUT(60)/display time out page-->6
 5---provide name social security number/check status-->7
 7---[data is NOT in record]/
      display incorrect information error page-->9
 7---[data in record]/generate login password-->8
 10---[Voter is NOT registered]/
     display incorrect information error page-->18
 10---[Voter is registered]/
     display voter operation page-->12
 12---select register vote operation/
     display open ballots page-->13
 13---TIMEOUT(60)/display time out page-->14
 13---select ballot/display selected ballot page-->15
 15---mark vote/prompt more votes-->16
 16---[prompt response is NOT more votes]/-->17
 16---[prompt response is more votes]/
     display open ballots page-->13
```

**Figure 5. State model generated from use cases “Register for Vote”, “Perform Vote” and “Voter log in”.**

is a *timeout condition* that indicates the expiry of a timer set when the departing state was last entered after a given delay. Any transition going from a state *s* also applies to all substates of *s*. A particular state is chosen as an *initial state*. In Figure 5, state *1* is the initial state.

## 2.4. Simulation

We use the state models obtained from use cases composition as prototypes for requirements validation by simulation. UCEd includes a state model simulation engine and a graphical user interface that allows simulation by selecting actors operations. Simulation is basically a run through a state model. A simulation session starts with the state model initial state as a *current state*. When a selected actor operation is a trigger to a transition going from the current state, UCEd displays the system reactions of that transition and changes the current state to the transition arrival state. The tool user is prompted for guards and timeout conditions.

Simulation helps validate requirements in collaboration with users by ensuring that the combination of use cases and domain description is satisfactory. Given a set of use cases, the generated state model and hence use cases simulation results are strongly dependent on the operation effects defined in the domain model. In the extreme case that no operation effect is defined, the resulting system would be modeless. The generated state model would include a single state with looping transitions and every possible input would be accepted at any moment. Although, such a system would allow the sequence of events defined in the use cases, many other additional sequences would also be permitted. In the EVote example, state *4* is a substate of state *3*. Therefore the sequence of triggers *Voter load EVote System web page - TIMEOUT - select voter registration option* is valid and result in the system operation *ask name social security number*. Notice that such extra sequences are not necessarily bad. We consider use cases as possible behavior descriptions from which more behavior can be inferred. However, in a situation where an extra sequence is deemed unacceptable, some operation effects in the domain model must be altered to remove it. In the previous example, the sequence *Voter load EVote System web page - TIMEOUT - select voter registration option* would be prevented by changing *added* and *withdrawn* conditions such that state *4* is not a substate of state *3* anymore. In that case it is sufficient to add withdrawn-condition "*ANY ON Voter transaction status*" to operation *display time out page*. This withdrawn condition states that any condition on attribute *transaction status* is to be withdrawn-ed after operation *display time out page*. Figure 6 shows changes to the generated state model after the modification to the domain model. The

```
State:4[System is online,display is timeout]
State:14[Voter is logged in,System is online,
 display is timeout]
**** TRANSITIONS ***
2---TIMEOUT(60)/display time out page-->4
5---TIMEOUT(60)/display time out page-->4
```

**Figure 6. Changes to the state model in Figure 5 after a modification to the domain model.**

characteristic conditions of states *4* and *14* have been modified. In addition, transitions from state *2* to *11* and from state *5* to *6* have been replaced by transitions to state *4*. Because state *4* is not anymore a substate of state *3*, trigger *select voter registration option* is not acceptable from state *4* in the modified state model. The offending sequence *Voter load EVote System web page - TIMEOUT - select voter registration option* is no more possible.

The previous example is a snapshot of our approach for use cases elaboration in conjunction with the domain model. A detailed presentation can be found in [11]. The approach involves several iterations of use cases and domain specification followed by simulation. It is being used in combination with UCEd to teach software engineering students how to better describe use cases and domain model elements. The following limitations became apparent while applying the approach. Once a sequence that should not be accepted is encountered and corrections attempted, it is not always possible to remember the original sequence in order to ensure the effectiveness of the corrections made. A related problem is that valid sequences may also need to be rechecked after cycles of iteration to ensure the specification still support them. Finally, it is sometimes useful to define a sequence beforehand even as simulation is not yet possible. These shortcoming are similar to what software testers were faced with before the advent of automated testing. The solution for testing was the introduction of *test cases*. We use scenarios to address these shortcomings.

## 3. Scenarios

Scenarios describe interactions between systems and actors. A scenario is a sequence of: *triggers*, *system reactions*, *waiting delays*, *guard realizations* and *assertions*. More formally, we define a scenario $\mathcal{S}$ as a sequence of elements $[e_0, \cdots, e_n]$ such that each $e_i \in \mathcal{T} \cup \mathcal{R} \cup \mathcal{D} \cup \mathcal{G}r \cup \mathcal{A}$ with $\mathcal{T}$ a set of triggers, $\mathcal{R}$ a set of system reactions, $\mathcal{D}$ a set of waiting delays, $\mathcal{G}r$ a set of guard realizations, and $\mathcal{A}$ a set of assertions. We defined triggers and system reactions in section 2.3. A *waiting delay* specifies a point in a scenario where a certain amount of time passes without any trigger or system reaction. A *guard realization* is a condition set to hold at a certain point in a scenario. An *assertion* is a condition that needs to be true at a certain point in a scenario.

Figure 7 shows and example of scenario that includes all these types of elements. Assertion "*Voter is not logged in*"

```
Scenario: example 1
1.  Assertion:Voter is not logged in
2.  Trigger:load EVote System web page
3.  Trigger:select voter registration option
4.  Reaction:ask name social security number
5.  Trigger:provide name social security number
6.  Reaction:check Voter status
7.  Guard:Voter data is in record
8.  Reaction:generate Voter login id and password
9.  Assertion:Voter is registered AND
10. Trigger:loads login page
11. Wait:60.0 second
12. Reaction:display time out page
13. Assertion:Voter is not logged in
```

**Figure 7. Example of scenario.**

must be verified at the beginning of the scenario. The scenario then specifies a sequence of triggers and reactions on lines *2* to *6*. The guard on line *7* means that condition "*Voter data is in record*" is set to hold at this point in the scenario if possible. Line *11* includes a waiting delay such that 60.0 seconds passes with the system waiting at that point in the scenario. Line *13* specifies an assertion stating that condition "*Voter is not logged in*" must hold at the end of the scenario. This assertion can be seen as a statement of a safety property.

Scenarios and use cases are related. A use case consists of a collection of scenarios. However, we define scenarios independently from use cases as they may cross over several use cases. Scenarios are assumed to start from the system's initial state, while use cases are guarded by a precondition.

We use scenarios as persistent artifacts for simulation. Given a state model, the execution of a scenario is similar to a manual simulation discussed in section 2.4. However, all selections are taken from the executed scenario as follow. A triggers in a scenario corresponds to an actor operation selection. A guard realization corresponds to a choice among several possible guards. A waiting delay may enable a timeout condition. Finally, an assertion specifies a property that need to hold at a specific moment during a system run. Execution of a scenario **passes** if all its assertions are verified, all its triggers accepted by the system and all its reactions are produced whenever they appear. The execution **fails** otherwise.

### 3.1. Positive and negative scenarios

A scenario may be "positive" or "negative". A positive scenario describes interactions that must be supported. For instance, scenario "example 1" in Figure 7 is positive. Its execution against a state model generated from use cases in Figure 3 and the domain model in Figure 4 must **pass** for these requirements to be valid.

A negative scenario describes interactions that need to be avoided. Scenario "Multiple registrations" in Figure 8 is negative its execution against the state model generated from use cases in Figure 3 and the domain model in Figure 4 must **fail**. An interesting feature of negative scenarios,

```
Scenario:Multiple registrations
1.  Trigger:load EVote System web page
2.  Trigger:select voter registration option
3.  Reaction:ask name social security number
4.  Trigger:provide name social security number
5.  Reaction:check Voter status
6.  Guard:Voter data is in record
7.  Reaction:generate Voter login id and password
8.  Trigger:load EVote System web page
9.  Trigger:select voter registration option
10. Trigger:provide name social security number
11. Guard:Voter data is in record
12. Reaction:generate Voter login id and password
```

**Figure 8. Example of scenario.**

is that they can be related to properties such as safety and security [4]. For instance, suppose the following statement of a security requirement for the EVote System:

NFR1: The Evote system should prevent multiple registrations by a same Voter.

Scenario "Multiple registrations" describes a sequence where a Voter successfully registers twice. A system supporting this scenario would not satisfy requirement *NFR1*. "Multiple registrations" may therefore be defined as a negative scenario attached to requirement *NFR1* such that a valid system in regard to the requirement should be such that the scenario is not accepted.

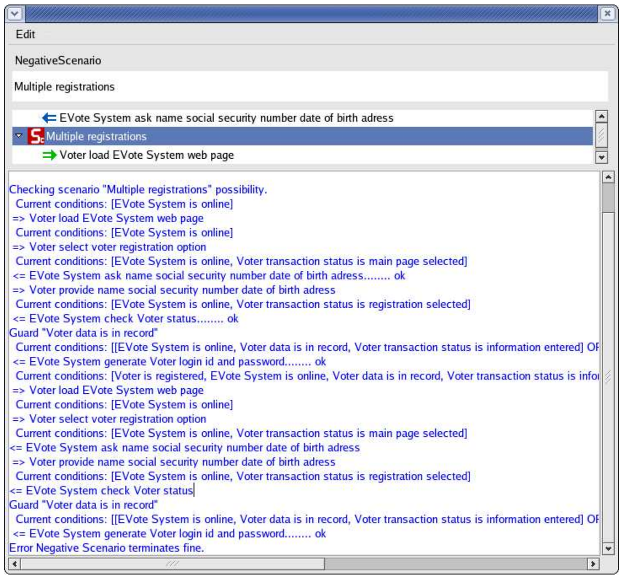
Notice that in the above example, we realize that scenario "Multiple registrations" may be acceptable if the two registration attempts are made by different Voters. However, the example is still pertinent as nothing specifies that it can not be a same Voter.

### 3.2. Requirements elaboration with scenarios

In order to support scenarios in our approach, we added a scenario editor to the UCEd toolset for scenario editing. We also altered UCEd simulator such that interaction sequences during simulation are recorded as scenarios. These scenarios can then be moved to the scenario editor for editing. The scenario editor allows different operations including changing a scenario from positive to negative (and conversely from negative to positive), adding a description to scenarios, and simulation of scenarios.

An important goal of scenario simulation is to give feedback to help eventually correct unsatisfactory requirements. For that purpose, scenario simulation results include a description of conditions holding before and after each operation. These conditions are characteristic conditions of states before and after the operation. The requirements analyst needs to identify a *deviation point* in a scenario where the behavior resulting from simulation deviates from the expected behavior. For a positive scenario that fails, the deviation point is the event at which the simulation stopped. For a negative scenario that passes, the deviation point is the first event (trigger, assertion) in the scenario that shouldn't have been accepted, or the first system reaction that shouldn't have been produced by the system. Based on conditions shown as part of a simulation results, correction generally consists of: modification of the operation at or just before the deviation point pre/postconditions, and/or addition of guards to use cases such that the operation at or just before the deviation point is better constrained.

As an example, Figure 9 shows the simulation results of negative scenario "Multiple registrations". The scenario passes, which is an un-desired outcome. The use cases and/or the domain model need to be modified such that this scenario rather fails. The deviation point here is the system reaction at line *12*. Operation *generate Voter login id and password* should be prevented when dealing with a previously registered Voter. Looking at the simulation result, we can realize that Voter registration status is never checked. Therefore, a possible correction consists of adding an alternative to step *5* of use case *Register for vote* in Figure 3 such that the use case become as shown in Figure 10. Alternative *5.b* covers the situation where condition "*Voter status is registered*" hold. Consequently, step *6* will only be possible when a Voter is not already registered. The introduction


Edit
NegativeScenario
Multiple registrations
EVote System ask name social security number date of birth adress
Multiple registrations
Voter load EVote System web page

```
Checking scenario "Multiple registrations" possibility.
 Current conditions: [EVote System is online]
=> Voter load EVote System web page
 Current conditions: [EVote System is online]
=> Voter select voter registration option
 Current conditions: [EVote System is online, Voter transaction status is main page selected]
<= EVote System ask name social security number date of birth adress........ ok
=> Voter provide name social security number date of birth adress
 Current conditions: [EVote System is online, Voter transaction status is registration selected]
<= EVote System check Voter status........ ok
Guard "Voter data is in record"
 Current conditions: [[EVote System is online, Voter data is in record, Voter transaction status is information entered] OF
<= EVote System generate Voter login id and password........ ok
 Current conditions: [Voter is registered, EVote System is online, Voter data is in record, Voter transaction status is infor
=> Voter load EVote System web page
 Current conditions: [EVote System is online]
=> Voter select voter registration option
 Current conditions: [EVote System is online, Voter transaction status is main page selected]
<= EVote System ask name social security number date of birth adress
=> Voter provide name social security number date of birth adress
 Current conditions: [EVote System is online, Voter transaction status is registration selected]
<= EVote System check Voter status
Guard "Voter data is in record"
 Current conditions: [[EVote System is online, Voter data is in record, Voter transaction status is information entered] OF
<= EVote System generate Voter login id and password........ ok
Error Negative Scenario terminates fine.
```

**Figure 9. UCEd Scenario Editor showing results of a scenario simulation.**

```
Title: Register for vote
Primary Actor: Voter
Goal: An unregistered voter want to register
 in order to be able to vote. If successful,
 the system generates a login id and
 password for the voter.
Precondition: EVote System is online
Postcondition: Voter is registered
1. Voter loads EVote System web page
2. Voter selects voter registration option
3. EVote System asks for name, social
   security number, date of birth, adress
4. Voter provides name, social security
   number, date of birth, adress
5. EVote System checks Voter status
6. EVote System generates Voter login id
   and password
1. a. After 60 sec
1. a. 1. EVote System displays time out page
2. a. After 60 sec
2. a. 1. EVote System displays time out page
3. a. After 60 sec
3. a. 1. EVote System displays time out page
5. a. Voter data is not in record
5. a. 1. EVote System displays incorrect
    information error page
5. b   Voter status is registered
5. b. 1  EVote System displays already
    registered message
```

**Figure 10. New version of use case “Register for Vote” to prevent scenario “Multiple registrations”.**

of alternative *5.b* might also trigger modification of operation *check Voter status* effects in the domain model shown in Figure 4, such that “Voter status is registered OR Voter status is not registered” is an added-condition.

## 4. Related work

A work related to ours is the play-in/play-out approach developed by Harel and Marelly [6]. 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. The LSC language also allows expression of anti-scenarios similar to our negative scenarios. Differences between the two approaches include the use of textual use cases and a domain model as a basis for requirement capture in our approach. One of our objectives is to support textual use cases definition. Another of our objectives is to help capture of domain elements including a specification of operations.

Negative scenarios can be related to misuse cases introduced by Guttorm Sindre and Andreas Opdahl [4, 5]. A misuse case is a reverse use case where an actor (a misactor) intentionally or accidentally uses a system in a way that is harmful to some of the system’s stakeholders. We do not consider negative scenarios at the same level as use cases mainly because we see scenarios in general as examples of execution sequences, and use cases as part of the specification. However, we believe the concept of misuse cases could be integrated to our approach as an additional way of expressing non-functional properties. Threat and hazard analysis are used in [1] to systematically discover mis-actors and mis-use cases, and investigate mitigation strategies. Similar techniques could be used to discover negative scenarios in our approach.

Another work that deals with negative scenario is presented in [14]. Requirements in this approach are defined as Message Sequence Charts (MSCs). Specifications expressed as Labeled Transition Systems (LTSs) are automatically generated from the MSCs. A LTS analyzer is used to detect implied scenarios [2] from generated specifications. An implied scenario is an extra behavior not defined by the MSCs. Detected implied scenarios may then be accepted as new MSCs or rejected as negative scenarios. The iterative approach for requirements elaboration in [14] is similar to ours. However, the level of abstraction in the description of systems is different. We see a system as a black box completely abstract from internal details. The approach in [14] deals with intra objects interactions at the architectural level. In fact, an implied scenario results from a mismatch between architecture and MSC description of behavior.

The same difference in abstraction level can be established between this work and different scenario based approaches [8, 9, 15]. Scenarios are described in these approaches as MSCs or sequence diagrams and, state machines are generated from the scenarios. In addition to this difference, our position is that a requirements specification needs a fair degree of completion and structure. A pure scenario approach for requirements specification suffers from problems such as scenario explosion and redundancy, since several separate scenarios are needed to cover all the exceptional cases. Use cases provide a framework for grouping and organizing related scenarios. Completeness can be achieved by ensuring through a careful analysis that exceptional cases are covered. An advantage is that these exceptional cases are integrated in a single description. Use cases modeling also includes structuring mechanisms such as "include" and "extends" relations useful for dealing with large descriptions. Although these elements add more power of expression to use cases, a drawback is that use cases lose some of the simplicity and intuitiveness that made scenario approaches so successful. Another common criticism of use cases is their inability to capture non-functional properties. We believe scenarios can be used in conjunction with use cases as a solution to these problems. Scenarios describe linear interaction sequences. They can span over several use cases and be used to capture some non-functional requirements.

## 5. Conclusions

In this paper, we discussed scenarios integration to a use cases based requirements engineering approach. Use cases are requirements artifacts used to specify how a system should behave from its users point of view. Our approach allows definition of domain operations required pre and postconditions such that use cases can be effectively realized without undesirable side effects. Simulation is used for validation.

The integration of scenarios to our use cases based requirements engineering approach aims at improving simulation. It is a response to a requirement from the approach users. According to our preliminary observations, scenarios effectively solve the initial problem of being able to represent simulation sequences in a persistent form. Because scenarios allow expression of some non-functional properties, an additional benefit is an expansion of requirements expressiveness.

We made an analogy between our work and testing. Scenarios can be seen in the context of use cases based requirements engineering as corresponding to test cases in the context of testing. The parallel can be pushed further by defining a notion corresponding to coverage for use cases validation. More precisely and as future works, we would like to be able to define degrees of satisfaction regarding use cases validation, how much simulation is enough for a specific degree of satisfaction, and which scenarios need to be tried. A related question is helping generate these scenarios.

## References

[1] I. Alexander and N. Maiden. *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*. Wiley, 2004.

[2] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *22nd IEEE International Conference on Software Engineering (ICSE'00)*, 2000.

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

[4] S. Guttorm and A. L. Opdahl. Eliciting Security Requirements by Misuse Cases. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-37'00)*, pages 120–131, 2000.

[5] S. Guttorm and A. L. Opdahl. Templates for misuse case description. In *Proc. Seventh International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ'2001)*, June 2001.

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

[7] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unifi ed Software Development Process*. Addison Wesley, 1998.

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

[9] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing software architecture descriptions from message sequence chart specifications. In *ASE*, pages 192–195, 1998.

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

[11] S. Somé. Supporting use cases based requirements engineering. Information and Software Technology (article in press). available at doi:10.1016/j.infsof.2005.02.006.

[12] 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.

[13] S. Somé. An Environment for Use Cases based Requirements Engineering. In *12th IEEE International Conference on Requirements Engineering (RE'04), Kyoto, Japan*, pages 364–365. IEEE Computer Society, 2004.

[14] S. Uchitel, J. Kramer, and J. Magee. Negative Scenarios for Implied Scenario Elicitation. In *10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'02)*, 2002.

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