The Role of Use Cases in Requirements and Analysis Modeling

Hassan Gomaa and Erika Mir Olimpiew

Department of Information and Software Engineering,
George Mason University, Fairfax, VA
hgomaa@gmu.edu, eolimpie@gmu.edu

Abstract. This paper describes the role of use cases in the requirements and analysis modeling phases of a model-driven software engineering process. It builds on previous work by distinguishing between the black box and white box views of a system in the requirements and analysis phases. Furthermore, this paper describes and relates test models to the black box and white box views of a system.

1 Introduction

Use case modeling is widely used in modern software development methods as an approach for describing a system's software requirements [1]. From the use case model, it is possible to determine other representations of the system at the same level of abstraction as well as representations at lower levels of abstraction. Since the use case model is a representation of software requirements, it presents an external view, also referred to as a black box view, of the system. In this paper, we consider other black box views of the system to be at the same level of abstraction as the use case model. Views that address the internals of the system are white box views, which are at lower levels of abstraction.

This work builds on the previous work on use case modeling in model-driven software engineering by distinguishing between black box and white box views of the system in the requirements and analysis phases. It also describes and relates software test models to the black box and white box views of the system.

2 Related Work

Use cases were conceived by Jacobson. The Object Oriented Software Engineering method [1, 2] describe how use cases relate to system analysis and test models. In the analysis phase, use cases relate to classes in the static model, object interaction diagrams in the dynamic model, and test specifications in the test model. The 4+1 view model of software architecture in [3] emphasizes the importance of use case modeling as a driver to determine architectural elements, and as the starting point for testing the

system. Use case diagrams are fully incorporated into the UML notation [3,4]. UML-based software development methods, such as COMET, all start with use case modeling to describe software requirements, The COMET method [4] describes how to develop collaboration models and statecharts from use cases. A collaboration diagram depicts the objects that participate in the use case and the sequence of object interactions. A state-dependent use case may be described with a statechart. The statechart despicts the sequence of inputs and expected outputs, corresponding to all use case scenarios.

3 Black Box Views of the System

The black-box views of a system describe a software system's external behavior. The internal workings of the software system, such as the internal software objects and their interactions, are not shown in the black-box views. Black-box views show how the system interfaces with its actors and other external objects. The approach considers black box views of the software system. Hence, hardware devices and people, which are all outside the software system, are explicitly considered in black box views.

The next sections describe the following black-box views of the system and how they relate to use cases:

1. Use case model of the requirements phase
2. System level sequence diagrams
3. System context class model
4. Statechart for state dependent use case
5. Test model

3.1 Use Case Model

A use case model [1] consists of use case diagrams depicted in UML and use case descriptions. The UML model depicts the use case, actors, communication associations between actors and use cases, and use case relationships, in particular the «extends» and «includes» relationships. The UML notation does not address the use case description, which is in many ways is the most important part of the use case model. The description includes the preconditions, postconditions, the description of the main sequence of the use case and the description of alternative sequences. Thus a use case describes several scenarios. The main scenario describes a successful sequence of events, and the alternative scenarios describe a sequence of events that differ from a typical usage of the system. An atypical usage of the system may be less frequently taken interactions or unsuccessful terminations that can be detected and resolved by the application; alternative system outputs; or alternative ways of entering system input.

3.2 System Interaction Model

The main sequence of the use case is described as a sequence of interactions between the use case actor(s) and the system. From the use case description, the analyst can depict this sequence of interactions on a system interaction model using a system level sequence diagram. This diagram depicts all the actors that participate in the use case and their interactions with the software system, which is depicted as one single aggregate object. Depicting the system as one single object preserves the black box property of this view. This view has the advantage of explicitly describing the sequence of external inputs to the system and external outputs from the system.

3.3 System Context Class Model

A system context class model depicts all the external classes that interact with the software system, depicted as an aggregate class. This model is developed by analyzing each use case and determining all the external objects. This diagram shows all the external classes, which might be external users, external devices or external systems [4]. Actors are intended to be proactive, providing inputs to the system. Since an actor in a use case can interact with the system using several external classes, the system context class diagram often provides additional information compared to a use case diagram. For example, in the use case for a customer withdrawing cash from an ATM machine, the actor is the customer. However, the actor actually uses several external hardware devices (card reader, keyboard, display, cash dispenser, printer) to interact with the system. These hardware devices are depicted as external classes in the system context class diagram. This diagram is particularly useful for systems with several external classes, such as real-time systems [4].

3.4 Statechart for State-dependent Use Case

In state dependent applications such as real-time applications, it is often both more concise and more precise to depict the interactions between the actor and system on a statechart. Once again, this statechart should be a black box view at the same level of detail as the use case, depicting inputs from the actor and state dependent responses from the system. For example, in highly state-dependent applications, such as the cruise control system [4] or microwave system [9], the statechart is more effective at specifying the sequences of external inputs and outputs. However, by relating the statechart to the use case, the scope of the statechart can be clearly determined with respect to the other use cases of the system, some of which might not be state dependent.

The statechart is developed by considering inputs from the external environment to be events that cause state changes on the statechart and the resulting actions execute functions described in the use case and/or provide outputs to the external environments [4]. The sequences of input events on the statechart and action generated outputs must be consistent with the sequence of inputs and outputs described in the use case.

3.5 System Test Model

The use case model is the starting point for the creation of test item specifications. With the use case modeling approach, the functional requirements of the system are described in terms of actors and use cases. A use case contains one or more scenarios, a main scenario and usually several alternative scenarios. A use case main scenario describes a sequence of interactions between one or more actors and the system. The alternative scenarios describe a sequence of inputs and actions that differ from a typical use of the system, such as for error handling. Each use case scenario can in turn be described with an interaction diagram that shows the sequence of interactions between the use case actors and the system.

A system test template is constructed from an interaction diagram. The input and output messages in the sequence diagram become test steps in the system test template. The message parameters and system state variables become test parameters. The use case precondition becomes part of a test condition for the test template. A test condition is more specific than the precondition of a use case. A use case precondition describes a constraint on part of the system state variables, while a test condition specifies additional constraints on the values of the input variables and system state variables. A test condition forces the execution of a specific use case scenario, in terms of a sequence of events and actions. In the test template, the use case precondition is shown in a separate section to maintain traceability to the use case.

A postcondition can also be added to the test template. A postcondition is a constraint or assertion on the expected output values and on the state of the system, which must be true after the test is executed. The precondition, test condition and postcondition of a test template are described in first-order predicate logic using OCL [5].

A system test template can be shown in tabular format (Table 1). The test template is more general than the test item specification document described in the IEEE standard [6], because it needs to be reusable across different application configurations, for different functional testing strategies. Instead of specific data values, the test template may have parameters, constraints on input parameter values in its test condition, and assertions on output parameter values. A test template can be used to generate the specific data values for a test item specification (test case), by substituting the data values that satisfy a test template's test condition into the test parameters. Model-based testing is described in more detail in [7].

Table 1. Test template specification

Function to exercise	<use case scenario name>
Test template id	<unique id of test template>
Test objective	<objective, or goal of test. Refers to a use case scenario that will be exercised with this test>
Precondition	<use case precondition>
Test condition	<constraint on the user input values and on the state of the system that must be true before a test generated from this template can be executed>
Test steps	<an ordered list of test steps, which are user inputs and system outputs. The inputs are tagged <input> and the outputs are tagged <output>>
Postcondition	<constraint on the expected output values and on the state of the system that must be true after a test generated from this template is executed>

4 White Box Views of the System

The white box views describe the internal static and dynamic structures of the system. The white box views represent analysis modeling where analysis of the system takes place. During analysis, each use case is realized by considering the objects required to participate in the use case. Object structuring criteria are provided to assist in identifying the software objects [1, 4], which can then be depicted in UML using stereotypes such as <<control>> or <<entity>>, The software objects will typically include interface objects (also referred to as boundary objects) which receive external inputs from actors and provide external outputs to actors, control objects which are decision making objects, which can be state dependent, and entity objects, which store data [4].

4.1 Object Interaction Model

The Object Interaction Model is part of the dynamic model of the system. Given the objects that realize each use case (see above), the next step is to depict the sequence of interactions among these objects [4]. The sequence of external inputs and external outputs sepicted in each scenario must correspond to the sequence described in the use case (see 3.1) and in the in the system interaction model (see 3.2). Furthermore, if there is a state dependent object, it must execute a statechart. During analysis, any statechart that was developed earlier (see 3.4) is encapsulated inside a state dependent control object [4]. It is important to ensure that the statechart is consistent with the state dependent control object. Thus, each incoming message to the state dependent control object on the interaction diagram must be correspond to and be consistent with an event on the encapsulated statechart and each action of the statechart must correspond to an outgoing message from the object [4].

Each use case scenario can be depicted on a separate interaction diagram, either sequence or communication diagram. The different scenarios of the use case can all be combined and depicted on a generic version of the interaction diagram. It is usually easier to see the realization of individual use case scenarios on individual interaction diagrams, but the generic diagram provides information in a more concise form.

4.2 Entity Class Model

For information intensive systems, there will be several entity classes, whose primary role is to encapsulate data. These classes can be initially determined from the use case model and are depicted on an entity class model, which describes the attributes of these classes and their relationships. Thus the use case descriptions for a hotel reservation system would refer to hotels, rooms, reservations, and customers, all of which would appear on the entity class model. The objects which interact with the entity objects would be depicted on the interaction diagrams. Understanding how the entity object is accessed, as depicted on the interaction diagram, helps with determining the class operations during design.

4.3 Integration Test Model

The white-box sequence diagrams created to describe the sequence of object interactions in a use case scenario can be used to design integration tests. An integration test validates the interfaces between the objects of a system. Integration testing requires call logging traces or assertions to be built in to the objects being tested so that the tester can observe the sequence of object interactions.

A system test template can become a test driver for integration testing. A tester enables the call logging traces for the objects described in the sequence diagram, executes a test case generated from the system test template, and compares the observed calls against the sequence diagram.

5 Relationship between Use Case Model and Other System Views

Consider traceability from the black box views to the white box view. Following on from the black box view, any use case based statechart needs to be encapsulated in a state dependent control object. An instance of each entity class from the entity class model will need to appear in at least one interaction diagram. An instance of each external class from the context class model will also need to appear on at least one interaction diagram. All interactions described in the use case diagram will need to be realized on the interaction diagrams.

The relationship between the use case model and system views is summarized in the meta-model of (Figure 1). This meta-model is based on the meta-modeling approach described in [8], which is adapted to address the views described in this paper and to add the test model views. Furthermore, the meta-model distinguishes between

black-box and white-box views with the use of «black-box view» and «white-box view» stereotypes. The meta-model emphasizes the relationships between the use case model and the other black-box and white-box views with bold associations. Some relationships between views are also shown but others are omitted for conciseness.

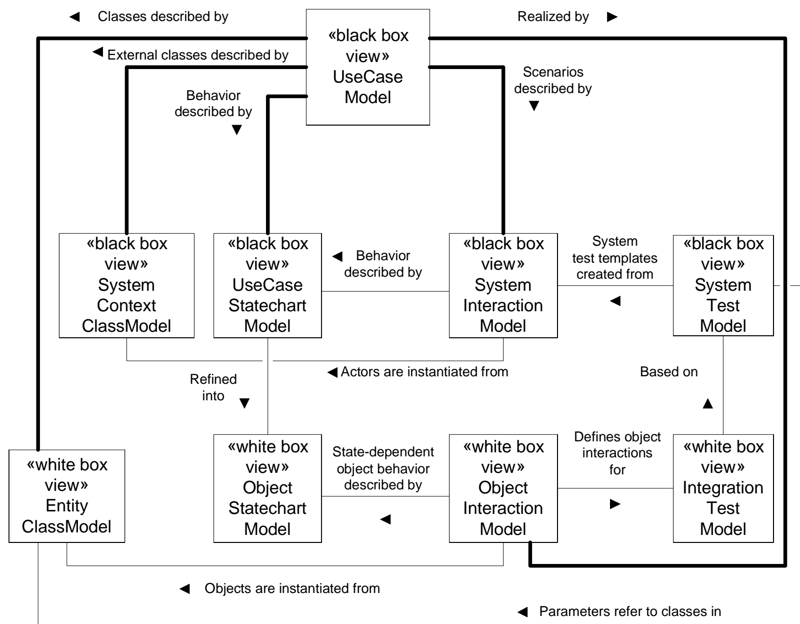

Fig. 1. Relationship between use case model and system views

6 Example

A section of the use case model and some system views are described for an example hotel system. Part of the use case model for a hotel system that contains the Login, Make reservation and Cancel reservation use cases is shown in (Figure 2). A use case description of the Make reservation use case is shown in (Table 2).

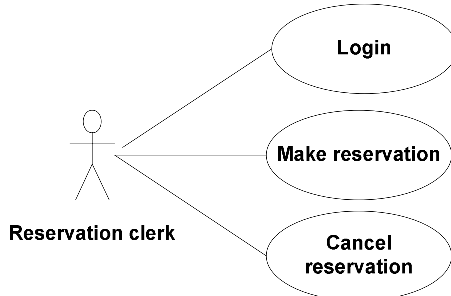

Fig. 2. Section of a use case model for a hotel system

Table 2. Use case description of Make reservation use case

Name	Make reservation
Summary	Reservation clerk reserves a room for a hotel guest
Actor	Reservation clerk
Precondition	Reservation clerk has logged into the Hotel system
Description	
	1. Guest gives personal details to reservation clerk, such as name, room type, number of occupants, and dates and times of arrival and departure 2. Reservation clerk enters information into the system and looks up availability and room reservation rate 3. If room is available, clerk requests the guest to guarantee the reservation with a credit card number 4. The clerk enters the guest's credit card number to guarantee the reservation. 5. If credit card number is valid, the clerk updates the reservation as guaranteed.
Alternatives	Room not available: Line 3: If a room is not available, clerk requests the customer for another date and / or time of arrival and departure Invalid card: Line 5: If the credit card number is not valid, the clerk requests the customer for another credit card number
Postcondition	The reservation is created for the guest.

A system level sequence diagram for the main scenario of the Make reservation use case is shown in (Figure 3). The Reservation Clerk actor and Hotel system in the sequence diagram are instances of the Reservation Clerk actor and Hotel system in the Make reservation use case description. The sequence of interactions is based on the description section of the main scenario of the Make Reservation use. It represents the information that is passed between the actor and the system.

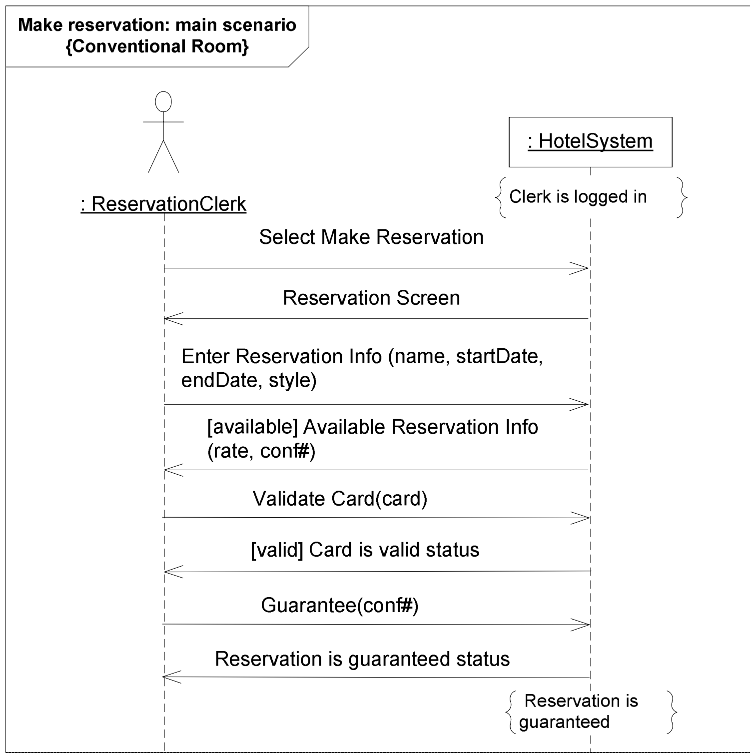

Fig. 3. A system level sequence diagram for the main scenario of the Make reservation use case

Part of a context model for the Hotel system is shown in (Figure 4).

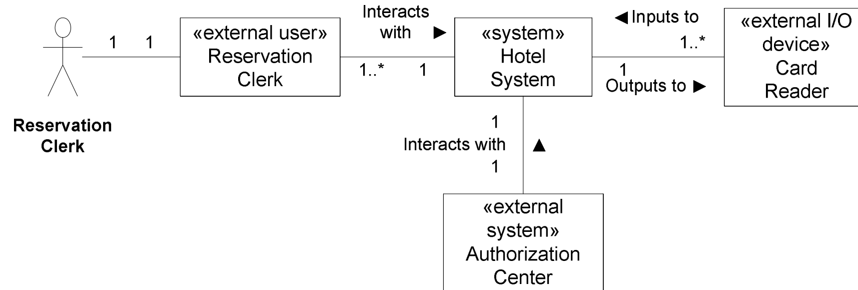

Fig. 4. Section of system context model for hotel

A test template created from the sequence diagram of (Figure 3) is shown in (Table 3) for the system test model. The test condition states that a room is available for the guest in the requested dates (aConventionalRoomIsAvailable) and the guest's credit card is valid (creditCardIsValid). The OCL constraints that correspond to the test condition are shown in (Figure 5). Some of the parameters in the test condition refer to classes in the class entity model of (Figure 6), such as the Calendar, Conventional-Room, and Reservation classes. Objects will be instantiated from these classes to generate the input values for a test item created from the template.

Table 3. Test template for Make reservation: main scenario

Function to exercise	Make Reservation use case, Main scenario
Test template id	MakeReservation_Main
Test objective	Exercise the main scenario of the Make reservation use case for the Conventional room feature
Precondition	**aClerkIsLoggedIn**
Test condition	**aRoomIsAvailable** AND **creditCardIsValid**
Test steps	1 <input> selection = MakeReservation 2 <output> display = ReservationScreen 3 <input> name: String, start: Date, end: Date, style: EConventionalStyle, selection = EnterReservationInfo 4 <output> display = AvailabilityScreen, rate: Real, conf#: String 5 <input> cardId: String, cardExpDate: Date, selection = ValidateCard 6 <output> display = ValidationMessageScreen, card-StatusMsg: String 7 <input> conf#: String, selection = GuaranteeReserva-tion
Post condition	**aReservationExists**

```
context HotelSPL
   def reservationClerks:
      let reservationClerks: Set(ReservationClerk)
   def dates:
      let dates: Set(Date)
   def rooms:
      let rooms: Set(ConventionalRoom)
   def reservations:
      let reservations: Set(Reservation)
   def cal:
      let cal: Calendar


context HotelSPL::makeReservation_Main(start: Date, end: Date,
style: EConventionalStyle, cc: CreditCard)

pre testcondition:

   let requestedDates = dates->select(d: Date | d.getValue() >=
start.getValue() and d.getValue <= end.getValue())

   let aClerkIsLoggedIn = reservationClerks->exists(r: Reserva-
tionClerk | r.isLoggedIn()= true)

   let aRoomIsAvailable = rooms->exists(r: ConventionalRoom |
requestedDates->forAll (d: Date | cal.isRoomAvailable(r, d) =
true and r.getStyle() = style))

   let creditCardIsValid = cc.isCardValid() = true in

   aClerkIsLoggedIn and aRoomIsAvailable and creditCardIsValid

post postcondition:
   reservations->exists(r: Reservation | r.isGuaranteed() =
true)
```

Figure 5 OCL constraints for MakeReservation_Main test template

The classes in the class entity model were identified in the Login, Make Reservation and Cancel Reservation use cases from the use case description and are shown in (Figure 6).

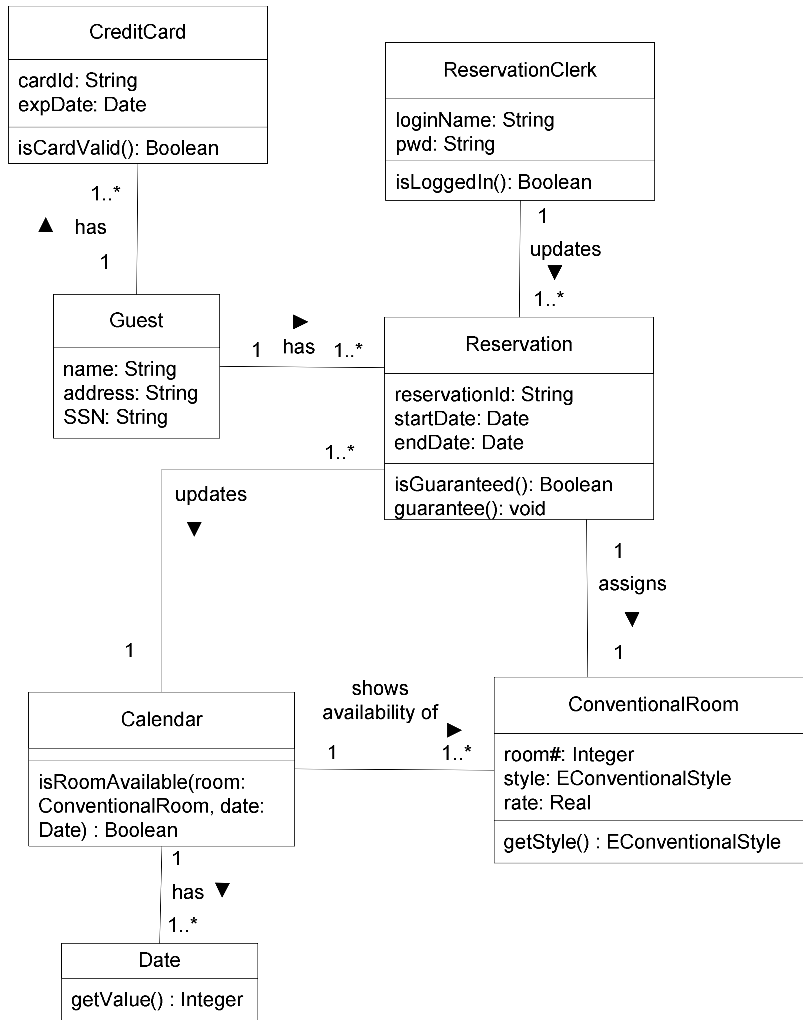

Fig. 6. Class entity model for Hotel System

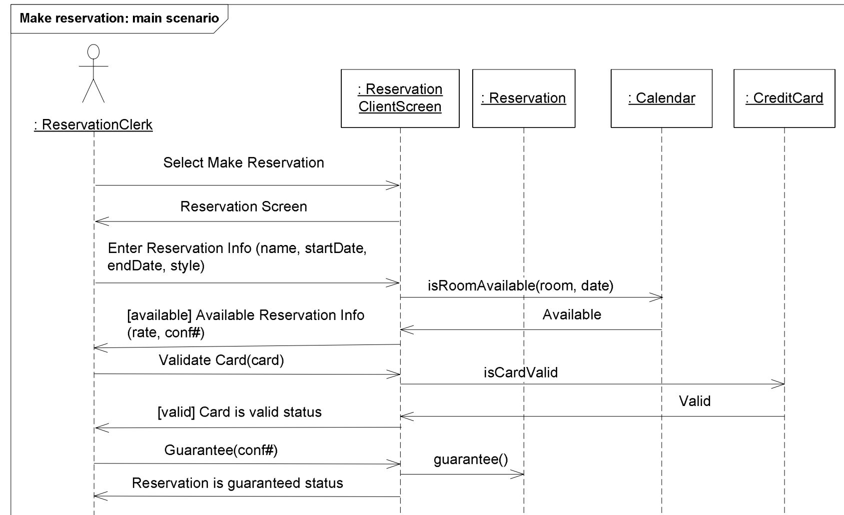

Fig. 7. Object interaction model for the main scenario of the Make Reservation use case

The object interaction model in (Figure 6) describes the message sequence between objects that participate in the main scenario of the Make Reservation use case.

6 Conclusion

This paper has described how the use case model relates to the other black box and white box views in the requirements and analysis phases of a model-driven software engineering process. The black box and white box views were expanded from previous work to identify their role in the software development process. Furthermore, system and integration test models were described as well as how they relate to the other black box and white box views of the system. The multiple view modeling approach described in this paper can be extended to address software product lines, where there is an additional dimension of complexity because it is necessary to model the commonality and variability among the members of the product line [9]

References

1. Jacobson, I., M. Christerson, P. Jonson, and G. Overgaard, *Object-oriented Software Engineering: a Use Case Driven Approach*. 1992, Reading, MA: Addison-Wesley.

2. Jacobson, I., M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*. 1997, Reading, MA: Addison-Wesley.
3. Krutchen, P., *Architectural Blueprints - The "4+1" View Model of Software Architecture*, in *IEEE Software*. 1995. p. 42-50.
4. Gomaa, H., *Designing Concurrent, Distributed and Real-Time Applications with UML*. 2000: Addison-Wesley.
5. OMG, *UML 2.0 OCL Specification*, in *OMG Final Adopted Specification ptc/03-10-14*. 2004, Object Management Group.
6. IEEE, *IEEE standard for software test documentation*, in *IEEE Std 829-1998*. 1998.
7. Olimpiew, E.M. and H. Gomaa. *Model-based Testing For Applications Derived from Software Product Lines*. in *Advances in Model-based Testing*. 2005. St. Louis, Missouri.
8. Shin, E., *Evolution in Multiple-View Models of Software Product Lines*, in *Information and Software Engineering*. 2002, George Mason University: Fairfax, VA.
9. Gomaa, H., *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. The Addison-Wesley Object Technology Series. 2004: Addison-Wesley Professional.