Capturing Business Transaction Requirements in Use Case Models

Patrice Chalin Daniel Sinnig Kianoush Torkzadeh

Dependable Software Research Group
Concordia University
Montreal, Quebec, Canada
{chalin, d_sinnig, k_torkza}@encs.concordia.ca

ABSTRACT

A significant portion of our modern economy is dependent on the reliability and usability of enterprise applications (EAs) of which business transactions and concurrency management are central concepts. The correct orchestration of subordinate system transactions forming a business transaction, as well as proper concurrency conflict resolution strategies are crucial factors. In this paper we argue that modeling business transactions and concurrency management are a domain activity and as such, are to be analyzed and documented during the requirements phase. Failing to do so can have a significant negative effect on the usability of an application. Driven by our own experiences in writing use cases for EAs, we demonstrate how business transactions can be modeled within use case specifications. Concrete examples and usage guidelines are offered as well as a demonstration of how our approach helps contribute to the difficult task of requirements elicitation.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications – *languages, methodologies, tools.*

Keywords

Requirements, Use Cases, Business Transactions, Concurrency.

1. INTRODUCTION

A significant portion of our modern economy is dependent on the reliability and usability of a class of software known as enterprise applications (EAs), in particular web-based EAs. For example, in the third quarter of 2006 alone, US retail *e*-commerce sales totaled over 27 billion dollars [1]. With the increasing complexity of today's business needs, the correct specification and modeling of EAs has become more important than ever. Much of this complexity pertains to the fact that EAs offer users *concurrent* access to (large volumes of) data [2]. Software engineers have tamed concurrency at the detailed design and coding levels by making use of *system transactions* with their well known properties of Atomicity, Consistency, Isolation and Durability (ACID) [3]. Hence, for example, most database management systems have built-in support for transactions and rollback.

SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.

On the other hand, from EA end-user's point of view, system transactions are imperceptible internal details whose sole purpose are to address his or her needs—e.g. transferring funds between bank accounts or making an online purchase. Such end-user interactions are termed business transactions. A *business transaction* is a logical unit of interaction between two business entities such that the ACID properties still hold. The decision as to which concurrency management strategy is most appropriate depends on the domain and hence should be modeled during requirements analysis and documentation [2; 4; 5].

Unfortunately, most literature on the topic deals with business transaction modeling from a design perspective, only [6; 7]. Thus, a false impression is given that business transaction modeling is a responsibility of the software designer with little or no consultation with domain experts and, most importantly, potential end-users. This misconception was part of the motivation behind the research reported in this paper whose two main objectives are as follows:

- Reassert that business transaction modeling and concurrency management are a requirements activity and as such are to be addressed by the software engineer and the customer.
- To demonstrate how business transaction requirements can be captured in use case models in such a way that the intuitive/readable nature of use cases is preserved.

2. MOTIVATING SCENARIO

Using the flight reservation system "Get-You-There", customer Paul wants to book a flight from Montreal to Vancouver. After entering his flight criteria, the system suggests a set of suitable flights. Paul selects a flight and, out of excitement for getting such a great air fare, runs into the kitchen to tell his wife about his good fortune. In the mean time, Frank, who is also interested in flying from Montreal to Vancouver, coincidentally selects the same flight as Paul. Frank lives alone and, without any delays, proceeds to select an appropriate seat and purchases the flight using his credit card. After successful validation of his credit limit, the system issues an electronic ticket. While Frank already enjoys a cool beverage to celebrate his upcoming vacation, Paul returns to his computer in order to finalize his booking. He selects a seat and submits payment information. Unfortunately, in Paul's case, the system indicates that seats in the selected booking class are already fully booked. As it turns out, Frank forestalled Paul and booked the last available seat in the booking class. Paul, frustrated, returns to the kitchen to tell his wife that he will never use the system again.

Apparently the business transaction was implemented using an optimistic concurrency management strategy, and hence, no lock was set on the corresponding flight data upon initiation of a

business transaction. As a result, the system gave the impression to all interested customers that a sufficient number of seats was available, even though the number of potential customers was greater than the number of available seats. If the system had been designed using a pessimistic strategy, Paul's disappointment would have been avoided, as all resources would have been locked in Paul's favor until the business transaction finished. Hence, Frank would not have had the chance to book the same flight. Which approach is best? It is up to all stakeholders, especially, targeted end users to decide. The main benefits of modeling business transactions and associated concurrency management can be summarized as follows:

- Contrary to low-level system transactions, the nature of business transactions and their corresponding compensation actions depend on the application **domain** and the needs of stakeholders. It was demonstrated that concurrency management is, to a great extent, a domain issue and should be analyzed and documented during requirements activities [2].
- **Elicitation of requirements**. Functional requirements and the involved user interaction are dependent on the chosen concurrency management strategy. For example, the functional requirements for optimistic concurrency management focus on conflict resolution whereas the functional requirements for pessimistic concurrency management are based on conflict prevention.

A main goal of this paper is to provide support for the modeling of business transactions at the requirements level, specifically through use cases. Use cases are the vehicle of choice for documenting functional requirements. Before proceeding, we review the key concepts involved in use case modeling.

3. MODELING BUSINESS TRANSACTIONS REQUIREMENTS

3.1 Use Case Models

Use cases were introduced roughly 15 years ago by Jacobson. He defined a use case as a "specific way of using the system by using some part of the functionality" [8]. More recent popularization of use cases is often attributed to Cockburn [9]. Use case modeling is gradually making its way into mainstream practice which sees it as a key activity in software development processes (e.g. Rational Unified Process) and as a result, there is accumulating evidence of significant benefits to customers and developers [10; 11].

A use case captures the interaction between actor(s) and the system under development. A use case starts with a header section containing various properties of the use case (e.g. primary actor, goal level, pre-condition, etc.). The core part of a use case is its main success scenario, which follows immediately after the header. It indicates the most common ways in which the primary actor can reach his/her goal by using the system.

A use case is completed by specifying extensions, i.e. alternative scenarios which may or may not lead to the fulfillment of the use case goal. They represent exceptional and alternative behavior (relative to the main success scenario) and are indispensable to capturing full system behavior. Each extension starts with a condition, which makes the extension relevant and causes the main scenario to "branch" to the alternative scenario.

Unfortunately the current state-of-the-art in use case writing does not provide proper means to model business transactions and the associated concurrency management mechanisms. In order to overcome this limitation, we define a set of new use case primitive constructs to specify business-transaction boundaries together with the involved transactional resources and the chosen conflict prevention (locking) mechanism. As a first step towards addressing these requirements we establish, in the next section, a vocabulary that can be easily understood by any stakeholder.

3.2 Proposed Terminology

The main terms used in our extension to use cases are as follows:

Transaction denotes a use case step, whose substeps are the activities of a business transaction. Each such transaction is attributed a set of transactional resources.

Transactional Resource (TR) denotes a domain entity that is directly or indirectly either created, read, modified or deleted by a use case step within the scope of a transaction. It is important to note that since we are dealing with requirements, a TR refers to a domain concept and not to a design or implementation level class.

Conflict Prevention Stage denotes the interval of time during which a transaction has exclusive access to its TRs. During this time, concurrency conflicts are not possible and hence TRs can be safely read and altered without risking inconsistent reads or lost updates.

Commit and **Abort.** The last substep of a transaction is a commit. In committing a transaction, its effects become permanent and visible to others. For various reasons (some of which will be explored in the following section), a transaction can be aborted in which case any intermediate substeps are "undone" so that the system is left with no trace that the transaction was ever initiated. In our approach, the main success scenario of a use case contains a COMMIT step and a TRANSACTION ABORT can only occur in the corresponding extensions; both lead to the release of the resource locks and hence the end of the conflict prevention stage.

3.3 Business Transaction "Template"

In this section we define a "template" for the integration of business transactions into use cases. As depicted in Figure 1, business transaction modeling is captured in the main success scenario as well as in use case extensions. Note that for sake of readability, we have collapsed sequences of zero or more steps (written as step*) whose details are not relevant to our presentation into a single numbered step—e.g. steps 1 and 3 of the main success scenario.

MAIN SUCCESS SCENARIO. The start of a transaction is indicated by the keyword TRANSACTION followed by the list of TRs used in the transaction. The transaction will contain a sequence of substeps consisting of user interactions and internal system computations including accessing and modifying TRs. There exist two internal system steps which are present in every business transaction. The first is denoted by BEGIN CONFLICT PREVENTION STAGE and it indicates the obtaining of exclusive access to TRs. The second, a COMMIT, marks the successful termination of the transaction. Note that depending on *when* the conflict prevention stage begins, either an optimistic, a pessimistic, or a hybrid concurrency management strategy is defined. Specifically, we have a

- *Pessimistic strategy* when there are no steps in 2.1; i.e. the transaction starts with BEGIN CONFLICT PREVENTION STAGE.
- *Optimistic strategy* when the conflict prevention stage only consists of system steps that directly update the TRs.
- *Hybrid strategy* otherwise.

MAIN SUCCESS SCENARIO
1. step*.
2. TRANSACTION.
   TRANSACTIONAL RESOURCES (TRs): *list of TRs*.
   2.1 step* (*possibly accessing TRs*).
   2.2 BEGIN CONFLICT PREVENTION STAGE
   2.3 step* (*possibly accessing, changing TRs*).
   2.4 COMMIT.
3. step*.

EXTENSIONS

(2.1-2.3) Primary actor indicates that he/she wishes to abort the transaction:
- step*.
- TRANSACTION ABORT.
- step*.

(2.1-2.4) System detects that TRs were changed by another actor:
- step* (*conflict resolution*).
- TRANSACTION ABORT (*optional*).
- step* (*optionally resuming at 2.x in an attempt to retry*).

(2.2) Exclusive access to TRs could not be obtained:
- step* (*conflict resolution*).
- TRANSACTION ABORT (*optional*).
- step*.

(2.3) Response time out:
- step*.
- TRANSACTION ABORT.
- step*.

(2.4) Failure persisting data:
- step* (*recovery*).
- TRANSACTION ABORT (*optional*).
- step* (*optionally resuming at 2.4 in an attempt to retry*).

Figure 1. Use Case Template of a Business Transaction

While general guidelines are applicable, the decision as to the most appropriate choice of conflict prevention strategy can only be made by domain experts [2]. Domain experts are also those who are in the best position to decide what should be done during transaction conflicts or failures as well as on the most suitable resolution and recovery actions—we discuss these next.

EXTENSIONS. As depicted in the template (Figure 1), there also exists a set of extensions associated with a business transaction. As will be clarified below, not all extensions shown in the template are applicable in all cases.

(2.1-2.3) Primary actor indicates that he/she wishes to abort the transaction. At any time between the start of the transaction and the COMMIT step, the user may request that the transaction be aborted. The extension steps that follow often consist of the system asking for confirmation, the user acknowledging, and the transaction being aborted—of course, more complex interactions are possible.

(2.1-2.4) System detects that TRs have been changed by another actor. When conflict prevention is optimistic or hybrid, TRs may be changed concurrently by multiple transactions outside the conflict prevention stage. In order to avoid lost updates and inconsistent reads [2], the system must ensure that TRs have not been updated by another transaction prior to the COMMIT step. When a conflict is detected, conflict resolution measures are taken: in the simplest case, this involves notifying the user and aborting the transaction. Making the right choices with respect to conflict resolution is a good example of key opportunities for requirements elicitation.

(2.2) Exclusive access to TRs could not be obtained. This extension is triggered if the system fails to enter the conflict prevention stage because one or more of the TRs are already bound in the conflict prevention stage of *another* business transaction. Under such circumstances the TRs cannot be modified and the transaction must be aborted or the primary actor given the opportunity to indicate that a resource lock be reattempted.

(2.3) Response time out. A common transaction failure scenario pertinent to the pessimistic concurrency management strategy is a response time out. Such an extension is necessary to ensure that resources are not locked indefinitely, should the user be unable to complete the transaction—e.g. the client is operating from a remote site and their network connection to the application server is lost.

(2.4) Failure persisting data. During the commit step of the transaction it is possible that the system fails to persist the changes made to the TRs, hence recovery actions become necessary. These actions may either lead to an attempted re-execution of the COMMIT step or to the transaction being aborted.

3.4 Example

An example of a use case which follows our template is given in Figure 1. This "Add Student to Class List" use case is a *simplified* version of a use case excerpt from an ongoing research project in which we are developing a *Course Manager* application. We have modified the use case step numbering to conform to the template so as to make reference to the template easier. Moreover, due to space constraints, only the main success scenario and the extensions are shown in full, the header section of the use case is omitted.

In the use case, the trigger for the main success scenario is the primary actor's request to add a student to a class list. The next step defines the business transaction for the use case and declares the Student Registry and all Class Lists as TRs. As a first transaction substep the primary actor identifies a Class List and the Student to be added. This is followed by an input validation step performed by the system. Failure in validating the input results in extensions (**2.1a**) or (**2.1b**), both of which allow the user to repeat the input step. Note that in step 2.1 we use the alias <Alice> to represent the student to be added; we find that this makes the use case less verbose and more readable than "the student to be added".

Next, the system attempts to enter a conflict prevention stage by obtaining exclusive access to the TRs. The case where such an exclusive access cannot be obtained is covered in extension (**2.2**), which, upon acknowledgement of the primary actor, will return to step 2.2 in the main success scenario where the system will re-attempt to gain exclusive access. After entering the conflict prevention stage, the system adds <Alice> to the Class List. On a successful commit, the system notifies the primary actor of the success of the transaction and this marks the end of the main success scenario.

A failure in persisting the data during the commit step leads to an extension of step (**2.4**) which aborts the transaction and ends the use case. At any time during the transaction the system may detect a concurrency conflict—i.e. that a TR has been changed by another actor. In such a case (extension **2.1-2.4**) the conflict is also resolved by notifying the primary actor and giving him/her a chance to try again as of step 2.1.

MAIN SUCCESS SCENARIO

1. (Trigger) Primary actor indicates that he/she wishes to add a Student from the Student Registry to a Class List
2. TRANSACTION
   TRANSACTIONAL RESOURCES: Student Registry, all Class Lists.
   2.1 At the System's prompt, the Primary actor identifies a Class List and the student to be added (who we will call <**Alice**>); System ensures that the provided information is valid.
   2.2 BEGIN CONFLICT PREVENTION STAGE
   2.3 System adds <Alice> to the Class List.
   2.4 COMMIT.
3. System notifies primary actor that <Alice> has been successfully added.

EXTENSIONS

(2.1-2.3) Primary actor indicates to abort the transaction:
– TRANSACTION ABORT.

(2.1-2.4) System detects that TRs were changed by another actor:
– System notifies primary actor that TRs have been changed by another transaction. *Use case resumes* at 2.1.

(2.1)a The given id for <Alice> is not in the Student Registry:
– System informs primary actor that information of <Alice> is invalid—e.g. invalid student id. *Use case resumes* at 2.1.

(2.1)b <Alice> is already in the Class List:
– System informs primary actor that <Alice> is already in the Class List. *Use case resumes* at 2.1.

(2.2) Exclusive access to TRs could not be obtained:
– System notifies primary actor that the conflict prevention stage could not be entered. System asks the primary actor if he/she wishes the system to retry entering the conflict prevention stage. Primary actor acknowledges. [Extensions to this step are not shown.] *Use case resumes at 2.2.*

(2.4) Failure persisting data:
– System notifies primary actor that the transaction did not complete because the changes could not be persisted.
– TRANSACTION ABORT.

Figure 2. Use Case: Add Student to Class List

Finally we point out that the example use case makes use of an optimistic concurrency management strategy. As such, the conflict prevention stage merely encompasses the system update to TRs and the COMMIT step of the transaction. Despite the possibility of a concurrency conflict (ext. **2.1-2.4**), the primary actor is prompted to identify <Alice> and a Class List. Exclusive access to the TRs is obtained only for a minimal period of time, which leaves no room for user interaction. As a direct consequence, there is no need for our sample use case to have a time-out extension. The main reason for the adoption of optimistic concurrency management is the low probability of its occurrence and the relatively low cost in its resolution—the primary actor merely repeats use case step 2.1.

4. EXPERIENCES AND GUIDELINES

The work reported in this paper was conducted in the context of a larger project involving the formalization of use case models and task models [12; 13]. One of the outcomes of our earlier work which supported the research done in the context of this paper was the creation of a meta-model for use case documentation; we adapted this meta-model to support business transactions.

Our meta-model was used during the development of the requirements documentation of the before-mentioned *Course Manager* application, as well as during a third-year undergraduate software engineering course in our department, which involved the development of use case models for student projects. In particular, the template "guided" stakeholders into thinking in terms of business transactions and concurrency management issues. In a sense, the template (especially the section on extensions) acted as a checklist against which the (relative) completeness of the use case could be assessed.

For example, response time outs were often neglected in those situations were pessimistic concurrency management was adopted. Another extension that was often overlooked was the possibility of the concurrent modification of TRs (under optimistic management schemes). Based on our experiences while working with our use case business transaction extension we were able to compile two important guidelines which we describe next.

Determining the boundaries of a transaction, that is when a transaction begins and when it commits, is not always obvious especially when optimistic concurrency management is used. A business transaction does not always begin and end with a system step that modifies TRs. Instead, a business transaction may start with a user interaction. A general heuristic to determine which steps belong to the business transaction is as follows:

(1) Assume (regardless of the envisaged concurrency management strategy) a pessimistic concurrency management strategy and determine the steps for which the TRs need to be locked such that a concurrency conflict is impossible. (2) Add all interaction steps that determine the successful commit of the transaction to the previously determined sequence of steps. (3) Set the transaction boundaries as follows: the transaction starts immediately prior the first identified step and commits immediately after the last one.

In the example use case of Section 3.4 the transaction begins when the primary actor identifies the Class List and the Student, followed by a validation step performed by the system. It is at this point where a concurrency conflict may first occur. The transaction ends with a step where the system adds the Student to the Class list followed by the commit step.

Choosing the right set of TRs for a given transaction can be difficult. The main challenge is in identifying all *relevant* resources while avoiding the inclusion of unnecessary resources (since the latter is likely to lead to reduced system availability).As was mentioned earlier, TRs are domain concepts taken from a system's Domain Model. With a detailed Domain Model at hand, we have found the following heuristic helpful:

1. Identify every domain object that is *directly* manipulated by the business transaction as a TR.
2. If 'A' has already been identified as a TR then every domain entity that is directly related to 'A' by virtue of an aggregation or composition relationship (in a domain model typically signified by <<is part of>>, <<consists of>>, <<has>> stereotypes) is a prime candidate for inclusion as a TR as well.

5. RELATED WORK

The importance of modeling transactions and concurrency management in the early development stages has been identified by various authors. Fowler emphasizes that the choice of concurrency control strategy depends, to a great extent, on the likelihood and the consequences of a concurrency conflict, which in turn are to be determined during requirements specification [2]. Banagala points out that understanding and modeling business transactions is a crucial activity during the analysis phase [4]. In his work, Banagala demonstrates how business transactions can be modeled at the analysis level by proposing an enhancement to Jackson's problem frames approach [14].

Bennett *et al*. propose a framework for the analysis and the design of long running business processes and their entailed business transactions [15]. A business process is modeled by a set of interrelated tasks resulting in a task dependency graph. During the design phase one or more tasks are mapped to a node in the *Unit of Work* tree, where each *Unit of Work* represents a transaction. Unfortunately the proposed framework is not very well integrated with standard functional requirements documentation techniques.

Use case model extensions to increase expressiveness and quality of use cases have been proposed by various authors [16-20]. Ratcliff and Budgen [19] as well as Cox and Phelp [17] noted that there exists a semantic gap between use case models and follow-up design models. In order to bridge the gap the former propose including statecharts into the use case model which, at the design stage, can be further refined into system object behavior models. The latter promote the inclusion of internal system events into the use case specification. The authors argue that such events are typically at a lower level of abstraction and hence can be easier understood and processed by software designers.

In the domain of use case authoring, a number of guidelines have been proposed to enrich the quality of use cases. Most notably are the CREWS Guidelines [20] and the CS Rules [18]. For both it has been empirically proven that their strict application leads to more complete (w.r.t. a given problem statement) use cases which contain less requirements errors [16; 18].

Within the context of transactional business software, Correa and Werner [5] assert that scenario specifications (in the form of use cases) and the specification of business rules and transactions are closely related. As such, an in-depth understanding of the underlying transactions is indispensable for the specification of interaction details. Based on this assertion, the authors propose an approach where preliminary use case specifications are enriched with OCL annotations. These OCL annotations specify business rules as well as pre- and post conditions of business transactions. In this paper, similar to the approach of Correa and Werner, an extension to use case models has been defined. In contrast to their work and the works of others discussed in this section, our approach is not based on an external notation but is integrated with the use case model. Additionally, a novel aspect of our approach is that it incorporates references to domain objects as a means of identifying transactional resources.

6. CONCLUSION

We have presented an extension for use cases that enables analysts to accurately capture business transaction requirements. The extension is relevant to a wide class of applications, particularly enterprise applications for which modeling and processing business transactions is one of the most crucial and complex aspects. The main motivation behind our research was the mere fact that business transaction modeling and associated concurrency management is to great extent a domain activity and hence should be tackled in the requirements phase. We pointed out that functional requirements and the choice of concurrency management are closely related. Yet, the state of the art in use case writing does not provide proper means for an integrated specification of functional and business transaction requirements.

In this paper, we tackled this shortcoming by defining an extension for use cases that covers modeling business transactions. The challenge at hand was to define a well-integrated extension which preserves the intuitive nature of use cases. First, we established a vocabulary that could be easily understood by any stakeholder. The vocabulary captures relevant core concepts for modeling business transaction at the requirements level. Next we devised a template for modeling business transactions in the main success scenario as well as in use case extensions.

Currently our use case extension is only suitable for modeling individual business transactions. As future work, we are aiming to also provide extensions for long running business transactions and sagas. In both cases, due to performance reasons, pure conflict prevention policies are impractical. As a consequence, long running business transactions are often employed with an optimistic locking strategy, which attempts to keep the locking time of the transactional resources to a minimum. If such an optimistic scenario is not possible, the transaction needs to be restructured such that the atomicity and the isolation properties can be relaxed and technically speaking the transaction becomes a saga.

7. REFERENCES

[1] US Census Bureau, 3rd Quarter Retail E-Commerce Sales, www.census.gov/mrts/www/data/html/06Q3.html, 2007.

[2] Fowler, M., Patterns of Enterprise Application Architecture, Addison-Wesley, Boston, MA, 2003.

[3] Ramakrishnan, R. and J. Gehrke, Database Management Systems 3rd edition, McGraw-Hill, 2002.

[4] Banagala, V., Analysis of Transaction Problems Using the Problem Frames Approach, in Proc. of *ICSE 2006*, China, pp. 5-12, 2006.

[5] Correa, A. L. and C. M. L. Werner, Precise specification and validation of transactional business software, in Proc. of *Requirements Engineering 2004*, Kyoto, Japan, pp. 16-25, 2004.

[6] Alrifai, M., P. Dolog and W. Nejdl, Transactions Concurrency Control in Web Service Environment, in Proc. of *ECOWS '06*, 2006.

[7] Butler, M., C. Ferreira and M. Ng, Precise Modelling of Compensating Business Transactions and its Application to BPEL, in Journal of Universal Computer Science, 11, 2005.

[8] Jacobson, I., Object-Oriented Software Engineering : A Use Case Driven Approach, ACM Press, New York, 1992.

[9] Cockburn, A., Writing Effective Use Cases, A-W, Boston, 2001.

[10] Larman, C., Applying UML and Patterns, Prentice Hall PTR, 2002.

[11] Toerner, F., M. Ivarsson, F. Pettersson and P. Oehman, An Empirical Quality Assessment of Automotive Use Cases, Proc. of *RE'06*, 2006.

[12] Sinnig, D., P. Chalin and F. Khendek, Consistency between Task Models and Use Cases, in Proc. of *DSV-IS 2007*, Spain, 2007.

[13] Sinnig, D., P. Chalin and F. Khendek, Towards a Common Semantic Foundation for Use Cases and Task Models, in ENTCS, 2006.

[14] Jackson, M., Problem Frames, Pearson, London, 2001.

[15] Bennett, B., B. Hahm, A. Le, T., A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes, in Proc. of *Middleware*, pp. 331 - 348, 2000.

[16] Achour, B., C. Rolland, N. Maiden and C. Souveyet, Guiding Use Case Authoring: Results of an Empirical Study, in Proc. of *4th IEEE Symposium on Requirements Engineering*, Ireland, 1999.

[17] Cox, K. and K. Phalp, Exploiting Use Case Descriptions for Specifications and Design, in Proceedings of *EASE 2003*, UK, 2003.

[18] Phalp, K., J. Vincent and K. Cox, Improving the Quality of Use Case Descriptions: Empirical Assessment of Writing Guidelines, in Software Quality, 2007.

[19] Ratcliffe, M. and D. Budgen, The Application of Use Cases in Systems Analysis and Design Specification, in Information and Software Technology, 47, 2005.

[20] Salinesi, C., Authoring Use Cases, chapter in *Scenarios, Stories, Use Cases:* I. Alexander and N. Maiden, John Wiley, 2004.