Formalizing and Automating Use Case Model Development

Marinos G. Georgiades[1,*] and Andreas S. Andreou[2]

[1]*Department of Computer Science, University of Cyprus, Nicosia, Cyprus*

[2]*Department of Electrical Engineering & Information Technology, Cyprus University of Technology, Limassol, Cyprus*

Abstract: This paper proposes an approach that formalizes specific elements and activities of the use case modeling process in order to overcome problematic issues common to the conventional use case methods, namely the lack of systematic elicitation support in the identification of use case elements, the vagueness introduced by the use of informal natural language to define use case specifications, and the limited support of dedicated software tools that makes UCDA a time-consuming and error-prone activity. In particular, with the use of our approach, formalization of the stage for identifying the use case elements is achieved with the use of predefined types of use cases and actors, specific guidelines to define associations, relationships and business rules, and formalized sentential patterns. Formalization and clarity of the use case specification is achieved with the use of specific types of actions and guidelines, on one hand, and natural language-based authoring rules, on the other. A dedicated software tool supports the automation of the proposed approach including the automated generation of use case diagrams and specifications. Preliminary empirical evaluation of the proposed approach indicated its effectiveness and efficiency.

Keywords: Use case modeling, formalization, automation, natural language.

1. INTRODUCTION

Use case driven analysis (UCDA) has gained a wide acceptance among the many methods in requirements engineering [1], principally because the UC model—resulting from UCDA-allows functional requirements to be represented in an informal, easy-to-use style which appeals to technical as well as non-technical stakeholders of the software under development [2]. UCDA helps cope with the complexity of the requirements analysis process. By identifying and then independently analyzing different use cases, the analysts may focus on one narrow aspect of the system usage at a time [3]. Since the idea of UCDA is straightforward and use case specifications are usually compact, textual documents written in natural language (NL), the customers and the end users are expected to easily understand and actively participate in requirements analysis.

The use case model is composed of use case diagrams and specifications. Specifically, the UC model comprises actors, use cases and associations, which are depicted in a use case diagram. Each use case, according to Cockburn [4], represents a major piece of functionality that is complete from beginning to end and is described with a UC specification including: the basic flow of the use case, the alternative flows, involved actors and stakeholders, conditions, and reference to other related use cases. Finally, the business rules associated with the use case interactions must be specified or, at least, referenced [1].

*Address correspondence to these authors at the Department of Electrical Engineering and Information Technologies, Cyprus University of Technology, Limassol, Cyprus; Tel: +357 25002204; Fax: +357 25002750; E-mail: colognet@ucy.ac.cy

Although UCDA offers a more compact framework for analyzing requirements in contrast to the classical approach, which is basically performed with the use of a generic SRS template (e.g., IEEE SRS template [5]), building the UC model and especially writing use case textual specifications is still a difficult and time-consuming activity. The structured, though unrestricted, form of the UC specifications suffers less from ambiguities than the specifications derived from the use of the classical approach. However, because UC specifications remain essentially textual, ambiguities are inevitable [6]. According to Denney [7] and Jagielska [8], the produced descriptions usually suffer from problems such as ambiguities, redundancies, inconsistencies, conflicts with domain terminology and implementation details (jargon-contaminated use cases), which make them difficult to be maintained and understood by customers. El-Attar and Miller [9] state that these problems produce low quality information systems (ISs).

The unrestricted textual nature of UC specifications, as mentioned above, is one of the reasons for ill-defined UC specifications. In addition, the major difficulties in producing high quality use case models originate from the elicitation process; as Kim *et al*. [3] state, the lack of support for a systematic requirements elicitation process is probably one of the main drawbacks of UCDA. This lack of elicitation guidance in UCDA sometimes results in an ad hoc set of use cases without a consistent underlying rationale. Some existing approaches use NL parsing techniques to retrieve the UC elements from pre-existing requirements documents—either written based on a predefined SRS template (e.g., IEEE) or a UC template—but this method is not reliable, because of ambiguities, redundancies and inconsistencies present in such documents. In other approaches, the analyst tries manually, based on his/her own expertise and again from existing informal textual requirements, to derive the use cases and their elements. Due to the aforementioned problems in existing documents, the analyst must be an expert to derive the UC elements correctly and completely and, irrespective of the analyst's experience, this procedure is extremely time-consuming. A third category of approaches concerns the development of the UC model from scratch, by using the classical approach of open-ended questions that lack specificity and formality [10] and thus again result in a document with ill-defined requirements that need to be re-organized and re-adjusted, in order for the analyst to derive efficiently the UC elements[1]. Additionally, the informality in such documents is the principal hindrance to the use of automated tools for UC modeling, since informal NL is inherently complex, vague and ambiguous, and so UC elements are difficult to identify completely and correctly. Therefore, there is a lack of approaches that automatically generate UC models.

The current work focuses on three objectives, pertaining to the three problems mentioned above: (i) to formalize the elicitation process of UCDA; (ii) to provide an understandable and semi-formal way to write use case specifications; and (iii) to provide CASE-tool support and achieve time-saving and error-free UCDA. Objective (i) is achieved with the use of predefined types of use cases and actors, as well as guidelines to derive their associations, relationships and business rules. The development of this formalization is guided and derived from corresponding ideas and formalization provided by the NLSSRE (Natural Language Syntax and Semantics Requirements Engineering) methodology [11-15] described later. The elicitation process is also facilitated by the use of formalized sentential patterns—provided by NLSSRE—which provide a structured and expressive way to write the UC elements. Objective (ii) is achieved with the application of adaptation and authoring rules on the identified UC elements and formalized sentences, in order to easily construct a semi-formal NL use case specification. The basic and alternative flows sections of the UC specification are also formalized with the use of specific types of actions performed with a specific sequence. Finally, objective (iii) is achieved with the development of a dedicated case tool that covers all the stages of the UC model development and results to the construction of the UC diagram and specifications. In this paper, we will not expand on the demonstration of the tool, but we will show its main aspects underlining its application and usefulness through a few indicative examples.

This paper is organized as follows: Section 2 outlines related work and describes how our approach differs from others. Section 3 illustrates how the UC model is developed through our approach. Section 4 describes the experimental evaluation of the methodology, while section 5 provides conclusions and recommendations for future work.

2. RELATED WORK

Most of the existing approaches attempt to elicit various UC elements, such as UCs and actors, from existing requirements documents or textual descriptions written in informal NL. Then, using some rules or patterns and with the involvement of the analyst, these approaches utilize the extracted elements to feed the UC specification templates and form the UC diagram. NIBA (Natural Language Requirements Analysis in German) [16] is a project that parses requirements documents in German, interprets and transforms the output of the parser to conceptual pre-design schemas, validates the schemas and finally generates a conceptual model in UML. Another approach introduced by Dias *et al.* [1] uses fragments to describe different types of interactions that could form a use case. In this approach, the analyst must first identify the use cases and the actors by using an initial pre-existing UC model of the IS, and then try to match a set of interactions, guided by the given fragments, to each use case. Another approach introduced by Liu et al. [17] uses an NL parser on a document written in informal NL including stakeholders' requests, to identify use cases and actors and write UC elements as specific NL statements. The analyst has to be involved in the identification process because the parser cannot be considered reliable, due to the nature of the initial requirements document. Then, based on specific NL use case schemas, the NL statements feed a predefined use case specification template. All these approaches do not provide:

1) a reliable outcome, since NL requirements documents are full of ambiguity, vagueness as well as inconsistency, and therefore the identification of the UC elements from such documents often results in a poorly defined UC model;
2) the capability for complete automation of the procedure from the stage of UC elements identification to the creation of the UC model, since the analyst's involvement is required to identify or clarify the final set of UCs and Actors. Therefore, the informality often present in the initial requirements documents hinders the use of automated tools for system modeling, since informal NL is inherently complex, vague, and ambiguous; and
3) a time-saving process for identifying the UC elements and developing the UC model, again due to the difficulties resulting from the existing requirements documents.

Other approaches that do not use pre-existing requirements documents but instead apply a manual, labor-intensive task, with the use of open-question interviews which lack specificity and formality, lead also to answers and requirements documents with ambiguities and redundancies [10]; these approaches rely on the analyst's expertise to organize the requirements correctly and match them to the various UC elements of a UC specification template.

In working towards the second objective of this work, we see many drawbacks to describing a use case using informal natural language, as recommended by Jacobson [18] and Booch *et al.* [19]. Although the use of natural language facilitates communication between the analyst and the domain expert, natural language, used in its free, informal style, increases the risks of ambiguity, inconsistency and incompleteness of the use case description/specification. In order to avoid these typical problems with natural language, it is

[1] Elicitation approaches, such as NL parsing techniques and open questions, are applied generally in RE to mainly derive textual requirements. These approaches result also to ill-defined requirements, since they have the same weaknesses already mentioned.

important to use a more structured or formal technique for such a description. In the relevant literature, some structured techniques for the description of use cases have been proposed. In [20] a tabular representation is used, and in [21] a structured natural language is presented to describe the use cases. These structured representations provide a generic formalization of the UC specification template, hence not a clear formalism of the use case specification elements, and especially the transaction flow actions. Ochodek and Nawrocki [22] provide a semi-formal NL representation of transaction flow actions, however this formalism is still generic and does not cover completely all the possible transaction flow actions and the use case elements (e.g., actors) involved in each action. Some formal techniques such as grammars [23] or statecharts [24, 25] have also been introduced for the description of use cases. Although such formal representations facilitate formal analysis, they are difficult for analysts and users to understand and use. In our opinion, use cases must be described using a semi-formal form of NL, because such a form may be (a) understandable by both users and analysts, (b) semantically rich enough so that all pertinent description of the use case can be taken into account without any ambiguity, and (c) implementable.

The formalization of the process of identifying the UC elements and the formalization of the use case specification template with the main focus on its transactions flow sections are the major steps covered by our approach as part of a series of steps for the development of the UC model, which will be described in the next sections. Formalization is mainly achieved with the use of predefined types of use cases and actors, formalized sentential patterns, formalized types of transaction flow actions, and specific guidelines and NL authoring rules. The latter also helps in providing a clear and understandable semi-formal UC specification. The automation of the UC model development is supported by our dedicated CASE tool called NALASS (Natural Language Syntax and Semantics) which is also described through indicative examples.

3. USE CASE MODEL DEVELOPMENT

For the identification and development of use cases, actors, associations[2], relationships, use case modules and use case subsystems, we utilize specific elements provided by the NLSSRE methodology [11-13]. NLSSRE focuses on formalizing and automating the discovery, analysis and specification of user requirements for the development of Information Systems. In particular, the methodology handles user requirements concerned with the operational aspect of an IS[3] and builds these requirements with the use of the following IS elements: predefined types of functions, specific categories of data, user roles, business rules, and functional conditions (i.e. the circumstances within which each function is performed), as well as specific patterns for writing requirements as structured, semi-formal NL sentences. In addition to the identification of the main UC elements and their development into use case diagrams (steps 1–6 described below), our approach uses specific types of actions to formalize the transactions flow sections of the UC specification template, as well as adaptation and NL authoring guidelines to make the development of the template content easier on one hand and more understandable on the other. In particular, the steps of our approach for the development of the use case model are as follows:

1. Identify UC modules
2. Define use cases of each UC module
3. Identify the actors of each use case, associations, relationships and complementary use cases
4. Structure identified UC elements as formalized sentences
5. Define UC subsystems
6. Relate business rules with use cases and actors
7. For each use case, write the use case specification (UCS)

Below we explain each step, including also relevant explanatory references to the NLSSRE elements used each time.

Step 1: Identify UC modules

A use case module can be conceived as a small UC model—actually the smallest model of the entire information system. A UC module is created for each information object (IO) of the system and contains, in addition to relevant actors, specific types of use cases that correspond to specific types of functions related to an IO. According to NLSSRE, an Information Object (IO) is a digital representation of a tangible or intangible entity-described by a set of attributes—which the users need to manage through Creating, Altering, Reading, and Erasing its instances, and be Notified by the messages each instance (IOi[4]) can trigger. In the methodology, the Create, Alter, Read, Erase and Notify functions are called CAREN functions.

The process of identifying the IOs is a critical step in defining the UC modules, and IO identification is implemented by the NLSSRE methodology. In particular, NLSSRE provides techniques and guides for the identification of the IOs, such as an information flow table, a data flow questionnaire, IO categories (including business roles, inanimate objects, procedures, documents, events, and other animate entities) and specific rules [11, 12]. A detailed description of this identification step is outside the context of this paper. However, as a basic example, an indicative list of IOs for a Hospital Information system (HIS)[5] includes: the doctor, pharmacist and patient, as business roles; a drug as an inanimate object; examination, treatment, diagnosis, user authentication, and payment, as procedures; a patient record, insurance, x-ray, invoice, receipt, and prescription, as documents; an

[2] We make a distinction between 'association' and 'relationship': the former occurs between actors and use cases, and the latter between use cases (denotes include, extend and generalization relationships) or between actors.

[3] According to Ellison and Moore [26], an Information System is any combination of information technology and people's activities using that technology to support operations, management, and decision-making. The application domain of NLSSRE is mostly concerned with the operational aspect of an IS (also known as transaction processing – dealing with day-to-day transactions).

[4] An IO is conceived and processed at an abstraction level, while an IOi is conceived and processed at a factual level. Instances of the same IO differ only in the values of their attributes.

[5] Through the paper, to support clearly our arguments, we provide examples taken from the application of our approach in a real-life project, that is, the development of the information system of the general hospital of Nicosia, in Cyprus.

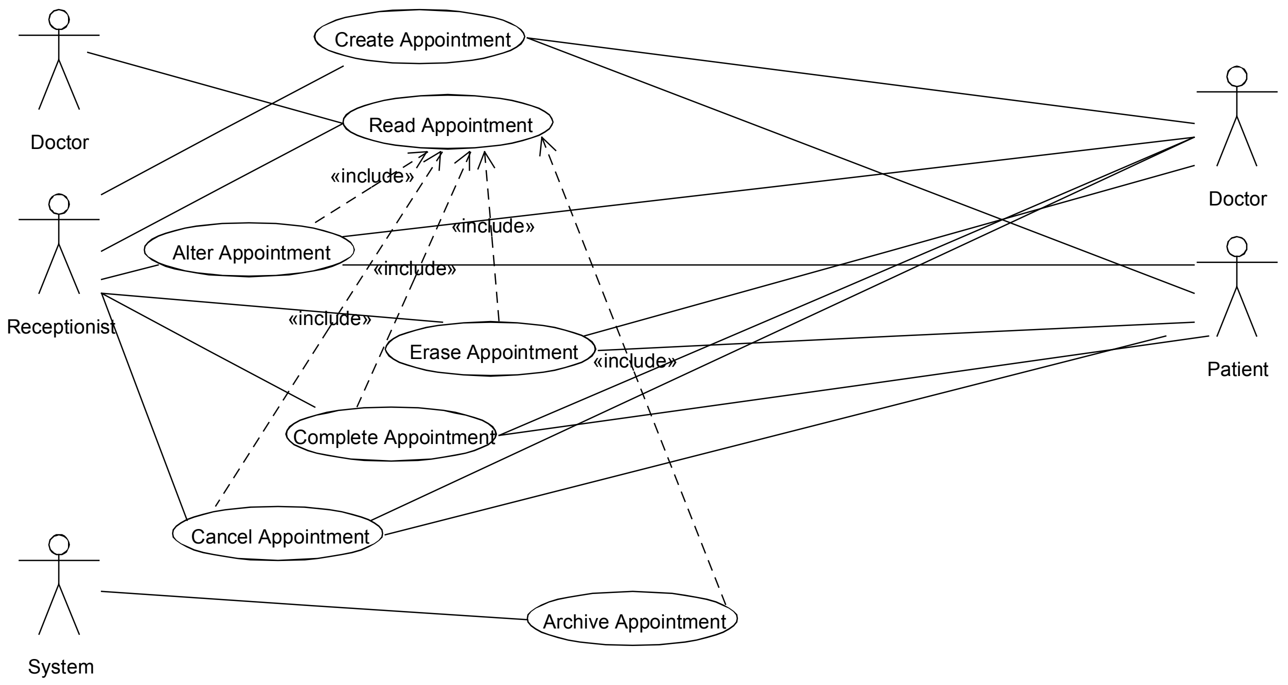


Fig. (1). The use case diagram of the Appointment module, as created by NALASS.

appointment as an event; and blood as an animate entity. Documents which are collections of attributes of different IOs, such as a *report* or a *notification*, which are created automatically by the system, are not considered to be IOs. However, there could be, for example, the rare case where a business sells its reports. In this case the *report* would be considered as an IO. As mentioned before, each IO corresponds to a UC module, therefore for each identified IO, a UC module needs to be defined. Each UC module will include specific use cases, actors, associations, relationships, a use case diagram, and a UC specification for each use case. Additionally, different UC modules may be grouped together and compose UC subsystems; and subsystems are then grouped together to compose the entire IS UC model. All these issues are described in the next sections of this paper. Fig. (**1**) shows an example of a use case diagram (UCD) corresponding to the Appointment UC module of an HIS.

Step 2. Define use cases of each UC Module

The principal aim of our approach is to formalize the identification of UC elements, including use cases and actors. This step handles formalization of use cases. Use cases of a UC module are derived from the CAREN functions provided by NLSSRE. As mentioned in step 1, **C**reate, **A**lter, **R**ead, and **E**rase are the main functions of the IO, while **N**otify is applied (triggered) after the creation, alteration, reading or erasure of an IO instance Fig. (**2**).

Our focus is on system functions at the user's level, that is, we are interested in what the system will do to fulfill the users' requirements. User-level system functions are represented by system use cases. Specifically, a system use case is conceived at the system's functionality level, and describes the function or the service that the system provides for the actors. The system use case specifies what the system will do in response to an actor's actions. System use case names should begin with a verb (e.g., *create* appointment, *select* payments, *cancel* appointment) [27]. In contrast, we do not focus on programmer's level requirements, that is, how system functions will be designed and programmed. The programmer's requirements will be defined at a later stage of the software development cycle[6]. We neither focus on abstract-level requirements, represented by business use cases. According to Podeswa [27] and de Cesare [28], business use cases focus on the business processes that the business actors (people or systems external to the process) use to achieve their goals (e.g. manual payment processing). Business use cases may involve both manual and automated processes. Often business use cases are free of technological terminology and treat the system as a "black box".

The formalization concept is more easily applicable to the system use cases, because they are applied on electronic information, while it is hardly applicable to the business level use cases, due to the complexity of the business environment, in both size and terminology.

For example, the use case *Enroll in Seminar,* which may be represented or implemented as a system or a business use case by conventional approaches such as Cockburn's (2000), is formalized and represented in our approach through the system UC modules *Enrollment* and *Seminar*, which are both IOs. The UC module *Enrollment* includes the system use cases *Create, Alter, Cancel, Erase and Read Enrollment*. The UC module *Seminar* includes the system use cases *Create, Alter, Cancel, Erase and Read Seminar*. Information about *Seminar* will be part of the UCs specifications of the *Enrollment* module (e.g., *seminar id* is used when creating or altering an enrollment) similarly also to information about the student who initiates the enrollment. *Student* will be also

[6] For example, a user-level system requirement is to allow the user to alter or read/view some particular data. However, the way with which these functions/tasks will be implemented, including retrieval and search methods/functions, is outside the users' requirements.

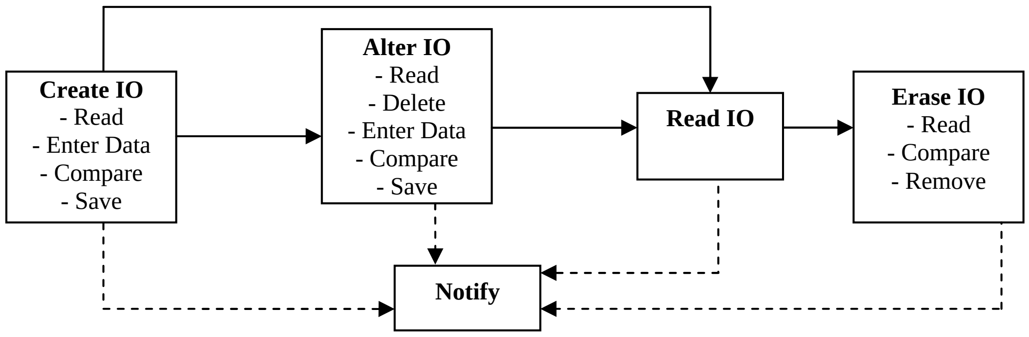


Fig. (2). CAREN - A recommended set of functions and sub-functions applied on an IO, and the notifications produced.

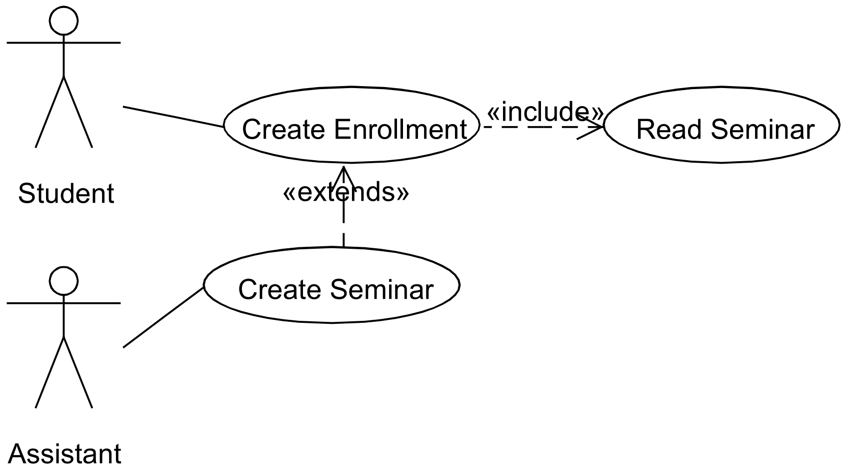


Fig. (3). Conceptual representation of use cases through the proposed approach.

a different UC module, as it is a different IO. Fig. (**3**) depicts how *Enroll in Seminar* is conceived and represented by our approach.

Below we give a description of each UC type, as well as relationships between use cases. We will also show what actions can be derived for each basic use case.

Create IO is the most significant use case, because during its execution the attributes of an IOi take their initial values; these values are then processed by other use cases. The sub-functions *Read, Enter data values, Compare* and *Save,* of the *Create* CAREN function correspond to actions of the *Create* UC specification. This will be elaborated in step 7 (constructing the UC specifications) later on, where we will see how the sub-functions, data constraints and business rules of the UC *Create IO* are used to form its transaction flow.

Alter IO: During the execution of this UC, the actor can change the existing values of the attributes of an IOi. A significant attribute that changes during alteration of an IOi is the attribute *State*. When the IO corresponds to a procedure (e.g., examination) or event (e.g., appointment), the *State* value may change from *Start* to *Ongoing/ Pending* to *Finished/ Completed* or *Cancelled, or even Expired or Archived*; when the IO is an inanimate physical object (e.g., book, drug) then *State* may change from *InStock* to *Sold/Lent*, and when the IO is an animate object *State* usually takes values according to the IOs business role (e.g., *Student* IOi *State* may be *new*, *studying*, *graduated*, *suspended*, or *Patient* IOi *State* may be *ill*, *under treatment*, *cured*); and when the IO corresponds to a document (usually in electronic form, e.g., prescription, voucher), *State* may take values such as *stored*, *archived*, *cancelled*, *edited* or *retrieved*. The change from one state to another (e.g., from *Pending* to *Complete*), for a particular IO, may lead to the creation of a new *expanded* alteration use case, such as *Cancel IO*, *Complete IO*, etc. However, if the change of state does not justify the existence of a new use case, it should be represented through additional actions in the transaction flow of the specifications of the basic alteration use case *Alter IO*. When a change of state occurs, we should check what new pre-conditions, post-conditions and actors are involved in the execution of the new derived use case or in the case of representing the change of states as actions the existing basic *Alter* use cases. Usually when the change of state of an IO results in significantly different pre-conditions or post-conditions, or results in a new group of actions than those provided by basic[7] *Alter* UC, we recommend to represent this self-contained information (pre-conditions, post-conditions, actions) as a new expanded alteration use case. For example, *cancelling an appointment*, results in a different post-condition than the post-condition resulting from the normal transaction flow of the UC *Create Appointment*, which is to complete the appointment. In particular, by cancelling an appointment, the *State*

[7] To distinguish the *Alter* UC from its related use cases derived as a result of change/alteration of state, we sometimes call it "basic *Alter* UC". Additionally, for simplicity, we call the related use cases (e.g., *Cancel IO, Complete IO*) "*Alter-related*" use cases. In some situations when we refer to the *Alter* UC, we also mean the alter-related use cases.

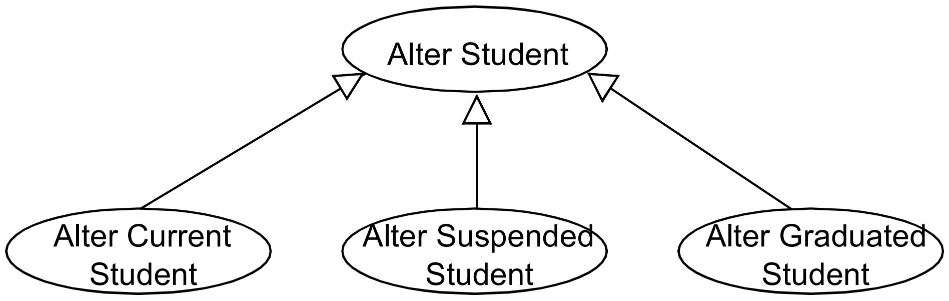


Fig. (4). Generalization relationships.

Table 1. Basic flow pattern for UC *Create IO*.

1.	<Creator> selects to create <IO>.	
2.	System displays new <IO> creation form, including required and optional fields.	
3.	<Creator>, <Accompaniments> enter(s) <IO> <IO.attribute.value>.	Repeated
4.	System must check <IO> <IO.attribute.value>.	
5.	<Creator> selects submit the new <IO>.	
6.	System saves the new <IO> in the database.	
7.	<System> notifies <Actor>, <Accompaniments>, <Intended Recipients> that <IO> is created *via UC <UC id>*.	

Table 2. Basic flow pattern for UC *Alter IO*

1.	<Alterer> selects to alter <IO>.	
2.	System displays existing <IO> *via UC <UC id> "Read <IO>".*	
3.	<Alterer>, <Accompaniments> deletes <IO> <IO.attribute.value$_x$>.	
4.	System must check <IO> <IO.attribute.value$_x$>.	Repeated
5.	<Alterer>, <Accompaniments> enter(s) <IO> <IO.attribute.value$_y$>	
6.	System must check <IO> <IO.attribute.value$_y$>	
7.	<Alterer> selects to submit the altered <IO>	
8.	System saves the altered <IO> in the database	
9.	<System> notifies <Alterer>, <Accompaniments>, <Intended Recipients> that <IO> is altered *via UC <UC id>*.	

attribute of the IO *Appointment* will change to 'cancelled', and this cancellation should create the post-condition "new empty schedule time slot". Therefore, we should consider *Cancel Appointment* as a new use case. Similarly, *completing a prescription* derives the pre-condition "Drug is given to patient" comparing to the basic UC *Alter Prescription* which has the precondition "Prescription is created". Completing a prescription is also performed by a different actor (pharmacist) at a different place (drug store) than the actor (doctor) that initiates the *Create* and *Alter* use cases of the prescription module, at the hospital or clinic. Therefore, we should consider *Complete Prescription* as a new use case. We may also conceive *Erase IO*, described below, as a new use case, where new post-conditions might be "IOi is archived" or "IOi is removed completely from the system's databases". The *State* attribute may also result in generalization relationships[8] between use cases, such as those depicted in the example of Fig. (**4**) where the student, due to the nature of his/her role, can move to different states during his/her studies.

Read: There are different types of information to be read, and this information is represented based on its type, as follows:

a Information to be read only by end-users. Usually, information is confidential, and the system users need authentication to read it. We distinguish two types of information:

i. Forms: IO forms usually need to be read when an end-user primary actor[9] creates a new IOi or changes the state of an existing IOi. The reading process should be represented as an "include"[10] use case *Read IO* for the use cases *Alter IO*, *Cancel IO*, *Complete IO*, etc., as depicted in Fig. (**5**) and Table **2** action 2 (in step 7), because it is composed of several actions, including retrieving and checking the existing information about an IOi, from the database, in contrast to the reading procedure for the

[8] Generalization relationship: If two or more use cases are similar, we can extract similarities into the base use case. Derived use cases can add behavior and modify behavior defined in the base use case [29].

[9] Primary and secondary actors, as well as actor functional roles, such as *notifiee* and *intended recipient* are defined and explained in step 3.

[10] An include relationship between two use cases means that the sequence of behavior described in the included use case is included in the sequence of the base (including) use case [30]. Include is used when the same behavior is *duplicated* in multiple use cases. A base use case is dependent on the included use case(s); without it/them the base use case is incomplete. Additionally, the included use case should be self-contained and cannot make any assumptions about which use case is including it.

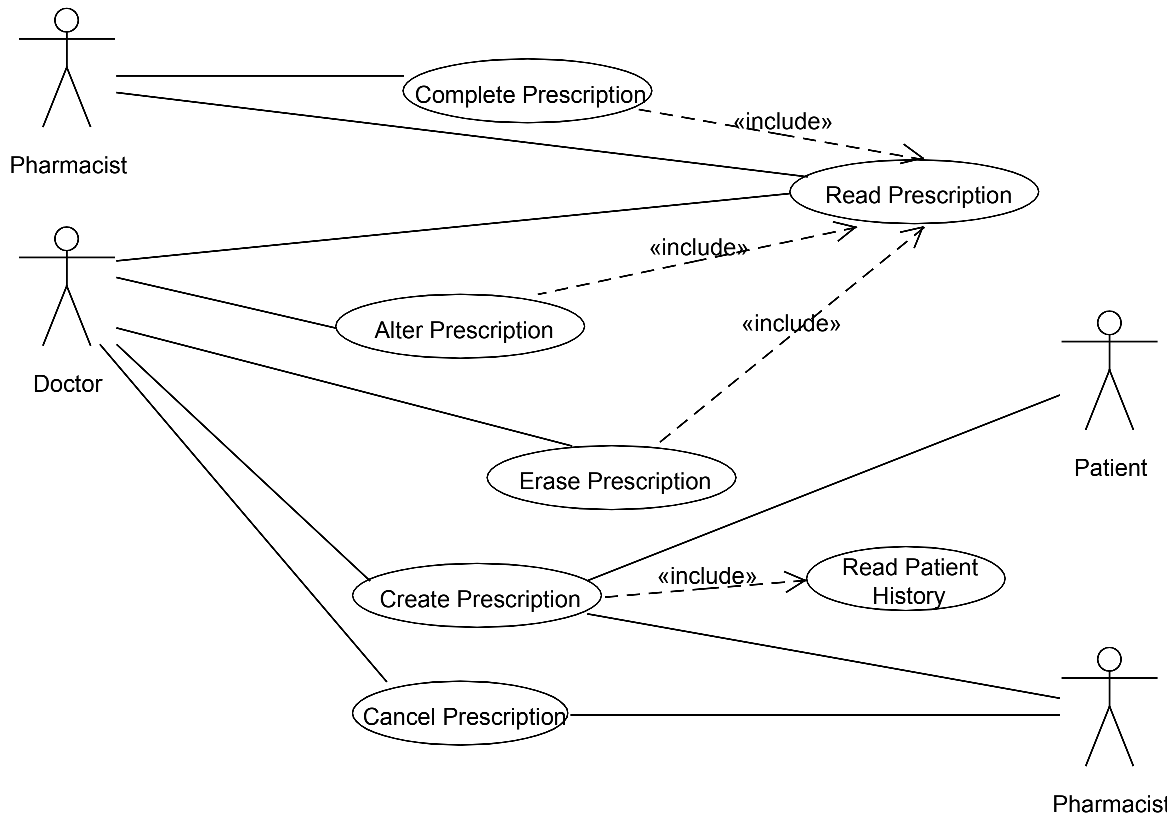


Fig. (5). Part of the use case diagram of the Prescription module, which is created automatically by NALASS.

UC *Create IO*, which only concerns building a form of required and optional empty fields, and thus represented as one or more simple action(s) in the *Create IO* UC specification, as illustrated in Table **1** action 2 (in step 7). This issue is discussed further in step 7, on constructing the UC specifications.

ii. Reports: reports of the IO per se (intra-reports) or of the IO in relation with other entities (inter-reports). Examples of such reports may be about appointments completed over a specific period (an intra-report, since it involves only the IO Appointment—*time* is an attribute of the IO *Appointment*), and appointments for a particular patient (an inter-report, since it involves two IOs, *Appointment* and *Patient*). The use case *Read IO intra-report* is part of the IO module, while the use case *Read IO inter-report* may be part of the IO module (e.g., *Read Patient History*, which is a report involving information related to the IO *Patient* from various IOs, such as *Examination*, *Diagnosis*, *Prescription* and *Treatment*, may be considered part of the *Patient* module) or of a more general *Report* module, because inter-reports may be used by (i.e., "included in") different use cases of different UC modules. Usually *Read* use cases about reports, and especially the inter-report type, are useful for the execution of use cases of other modules, and so they are represented as "include" use cases. This relationship usually occurs when an actor creates or alters an IOi, and so the actor may need to read information about instances of other IOs, related to the IOi the actor creates or alters. For example, when a doctor (actor) creates or alters a prescription (IOi), s/he may need to read information about the patient related to the prescription. If the information is large and involves other IOs, then it should be a different UC, such as *Read Patient History* (Fig. **5**), which it involves information about examination, treatment, prescription, etc., for the patient. *Patient history* is a report and not considered as a different IO. Reports are created automatically by the system. Since they will not be altered throughout time, but they are only to be read, we consider that their creation is embedded in the *Read* UC. Reports do not need to be stored.

b Information which is usually not important enough to be processed by or stored in the system. This information refers usually to notifications produced by use cases to notify actors or stakeholders of the system. For example, the UC *Create Prescription* produces notifications for the patient and the doctor (creator of the prescription) that the prescription is created. Reading a notification is part of *Notify* or *Send Notification* which is represented as a group of actions or as a separate use case, as described below.

Erase IO: Erasure of an IOi means that the IOi is permanently deleted. All of the particular information in that IO instance regarding attributes and functions that exist in the context of the IS is deleted. Erasure usually occurs when the user does not need to keep an IOi in the system anymore. However, at system/database level, the erased IOi may be stored at a separate place/database server.

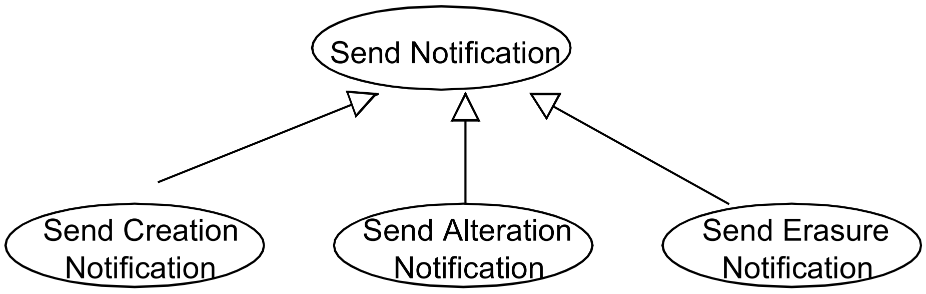

Send Notification
Send Creation Notification
Send Alteration Notification
Send Erasure Notification

Fig. (6). UC *Send Notification* may be specialized according to the type of use case which invokes it (e.g., UC *Create IO* invokes UC *Send Create Notification*).

Notify: In a computerized IS, transmission exists at the messaging level, which we call *Notification*. In particular, when an IOi is created or altered (or even read), then a notification should be sent to the interested parties which are classified into two groups: the *Intended Recipients* (IR) who will have to take an action within the IS as a consequence of the creation or alteration of the IOi (e.g., a *Pharmacist* is the IR of a *Prescription* IOi, because, after its creation, s/he will utilize it to create a *Drug* IOi), and other entities who just need to be informed about the creation or alteration of the IOi, these are, *Notifiees* (e.g., *patient* in the *Prescription* IOi example) or primary actors who created or altered the IOi.

The end of a *Create, Alter, Alter-related* and *Erase* use case specification should include specific actions about sending a notification to the actors or stakeholders interested in the creation or alteration of an IOi. If sending notifications involves different actions for the different types of UCs (*Create, Alter,* etc.), then *Send Notification* may be a separate UC with specialized UCs (Fig. **6**) included in and invoked by their including UC.

Step. 3 Identify the Actors of Each UC, Associations and Complementary Use Cases

For each basic use case identified in step 2, we need to identify the actors and other stakeholders involved in its execution. Actors usually refer to system end-users, customers, or trusted external users (e.g., suppliers). In contrast, other stakeholders refer to business users, managers, information users (e.g., a patient's relative in a hospital IS is an example of such a user), and shareholders [31]. In NLSSRE, each user has a business role in the system, which is involved in each CAREN function of a particular IO. Accordingly, in UCDA, each actor—in the place of a business role is involved in each use case of a particular UC module.

According to Marsic [32] and Sybase [33], an actor can be a primary actor for a use case if it triggers the actions performed by the use case; the primary actor is the one who asks for an action to be performed by the use case. Primary actors are located on the left of the use case in the UCD. On the contrary, an actor can be a secondary actor for a use case if the actor assists the use case in completing the actions but does not trigger the actions (i.e., a secondary actor is someone who participates in the use case but does not initiate it.) An actor is also considered as secondary when the actor receives information (e.g., results, reports, documents) produced by the execution of a use case. Secondary actors are located on the right of the use case. In a UC subsystem, as will be illustrated later, a secondary actor can also be a primary actor in another use case, in the same diagram. Finally, other stakeholders, as described above, can be represented by a third category of actors, namely *Offstage actors* which are stakeholders with interest in the outcome of the use case, but not playing an active role in the use case.

To identify the actors involved in each use case, we take into account the type of the use case—*Create, Alter, Alter-related, Read, Erase*—and the functional roles involved in each UC type. By making questions about the functional roles, we can identify the actors. A *Create* use case involves the functional roles *Creator, Accompaniment, Intended Recipient,* and *Notifiee*. An *Alter UC,* an *Alter-related* UC and an *Erase* UC involve the functional roles *Alterer, Accompaniment, Intended Recipient,* and *Notifiee*. A *Read* UC involves the functional roles *Experiencer, Accompaniment, Intended Recipient,* and *Notifiee*. The *Creator*, *Alterer* and *Experiencer* are played by primary actors, while *Accompaniment* and *Intended Recipient* are played by secondary actors. The *Notifiee* concerns offstage actors. Since primary actors initiate the use cases, they are usually required to have authorization to do it. Therefore, a use case "*Authorize <Actor>* " should be executed for each primary actor and link the primary actor to the use cases s/he can execute. Below we explain four of the functional roles actors can play and some indicative questions derived from these roles, in order to identify the actors[11].

a *Creator*: the Creator is responsible for setting the values of a number of particular attributes (required and optional) of the IO of the use case (e.g., *Doctor* is the creator of prescription in the UC *Create Prescription*.) To identify the creator in a *Create* UC, we may ask the following questions (question patterns and pattern instances follow instances are taken from the HIS case study). Questions about notifiees also help to indentify intended recipients.

– Pattern: Who should create an <IO>[12]?

– Instance: Who should create a *Prescription*?

– Pattern: Who has the responsibility for the creation of a(n) <IO>?"

– Instance: Who has the responsibility for the creation of a *Prescription*?

[11] A detailed presentation of all the actors (business roles) and different question sets provided to derive the actors is outside the scope and size of this paper. These details are presented in [11,12]

[12] Terms inside the "< >" should be replaced by the corresponding values

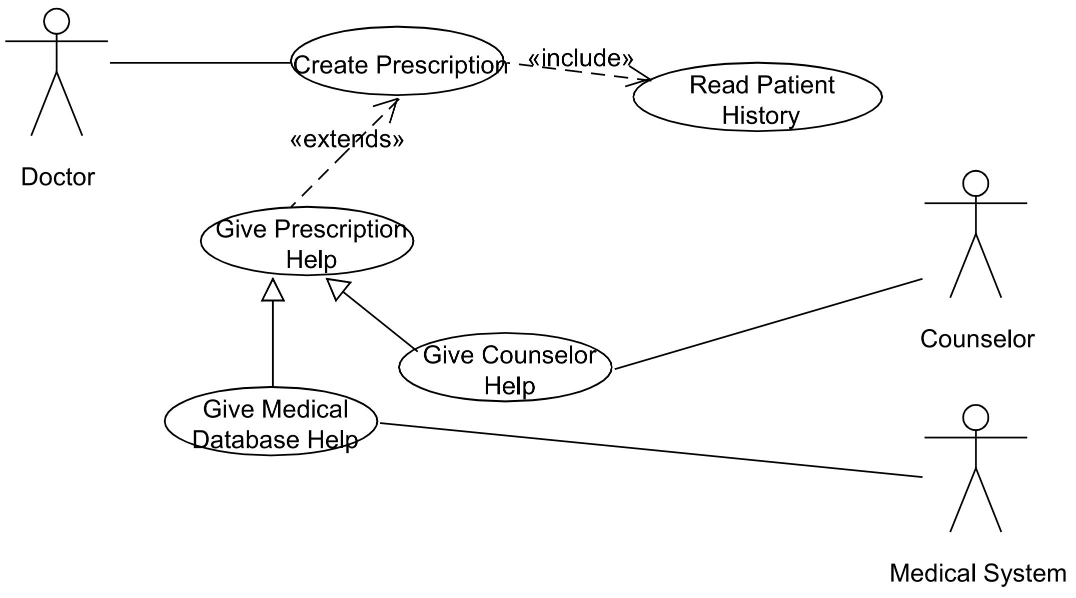


Fig. (7). Complementary use cases derived from relationships between actors (this is the other part of the *Prescription* module depicted in Fig. **5**).

b *Accompaniment*: Accompaniment participates in close association with the *Creator*, *Alterer* or *Experiencer*, depending on the type of use case, to help them in the creation, alteration (including both *alter* and *alter-related* use cases) or reading of an instance of the IO (e.g., *Patient* provides to the *Receptionist* his/her personal and other information to create an appointment for the UC *Create Appointment*.) The collaboration between a primary actor and an accompaniment can derive both include and extend[13] relationships, where extending or included use cases are invoked by their base use cases and triggered by the accompaniments. These use cases are called complementary. For example, as illustrated in Fig. (**7**), during the creation of a prescription, the doctor may need to ask for the assistance of another doctor/counselor or of a medical database system in order, for example, to choose between two drugs for the treatment of a patient. In this case the counselor and the medical system are accompaniments that provide feedback, and *Give Prescription Help* extends the behavior of *Create Prescription*. *Give Counselor Help* and *Give Medical Database Help* are specialized UCs of *Give Prescription Help*, and they occur based on the decision of the doctor. If the doctor does not need any extra knowledge to create the prescription, then the extending UC will not be executed, but the extended (base) UC will be fully completed. In the case where the complementary use cases are not considered to be large, complicated or worth reusing, then they can be described in the transaction flow of the UC *Create Prescription* specification and so they are not defined as separate use cases. To identify the accompaniments in a use case, we should ask questions related to that UC type (*Create*, *Alter*, etc.) and the functional role of the primary actor. For example, to identify the accompaniment in a *Create* UC, we may ask questions based on the following patterns:

– Who should assist the <Creator> to create an <IO>?

– How does the <Accompaniment> help the <Creator> during the creation of an <IO>?

– Does/Should any human or computer system help the <Creator> to record a new <IO>?

c *Intended Recipient*: *Intended Recipient* (IR) takes action within the IS after being notified about the creation, alteration (including both *alter* and *alter* use cases) or erasure of an instance of the IO. The action to be taken needs to fulfill the purpose of the IO within the IS, and the fulfillment is achieved by creating or altering instances of other related IOs. For example, in the UC *Create Patient*, *Doctor* is an IR of the *Patient* IO, because after the creation of a *Patient* IOi, the doctor will fulfill the purpose of the patient within the hospital IS (the purpose of a patient is to receive examination, diagnosis, etc.) by creating an *Examination* IOi, a *Prescription* IOi, etc. Similarly, in the example of the UC *Create Prescription*, *Pharmacist* is an IR of the *Prescription* IO, because after the creation of a *Prescription* IOi, the pharmacist will fulfill the purpose of the prescription (the purpose of a prescription is to provide drugs to the patient) by altering a *Drug* IOi (the drug provided to the patient must be removed electronically from the IS). Furthermore, the IR helps in deriving new use cases (e.g., *Create Drug*) and new UC modules (e.g., *Drug* module) in which the IR plays the role of the primary actor in a use case of the new module, which is linked to a use case of the previous module, where the IR was a secondary actor (e.g., *Pharmacist* who was an IR in the UC *Create Prescription* of the *Prescription* module, is an alterer in the UC *Alter Drug* of the *Drug*

[13] The extending use case is dependent on the base use case; it literally extends the behavior described by the base use case. The base use case should be a fully functional use case in its own right without the extending use case's additional functionality. The "extends" relationship includes the condition that must be satisfied if the extension is to take place, and references to the extension points which define the locations in the base (extended) use case where the extensions are to be made [34].

module). We will see in step 5 how this sequence helps in constructing subsystems. To identify the IR in a use case, we may ask the following relevant questions, based on the purpose of the IO, which is fulfilled by the actor who plays the IR (examples below are taken from the HIS case study):

– What is the purpose of the *Patient*?

– Answer: To receive examinations, diagnoses, prescriptions, treatments.

– Who provides the examination?

– Answer: Doctor

Therefore, as aforementioned, *Doctor* is the IR of the UC *Create Patient*, since s/he fulfills the purpose of the patient (to receive examinations, etc.) through executing new use cases (e.g. UC *Create Examination*).

d *Notifiee*: Notifiee includes the entities that only need to be notified about the function applied on an instance of the IO (these entities will not use the IOi or related information in any way that will cause any interaction within the system). *Notifiees* may include the business roles of business users, managers, information users (e.g., a *relative* of a patient in an HIS) and shareholders who generally do not have a direct interaction with the system; these business roles are considered as *Offstage Actors*. Notifiees of a use case may also include the primary and secondary actors involved in the use case, who need to receive a notification about the creation, alteration, erasure or reading of an instance of the IO they interact with. Additionally, the notification (its content or layout) sent to each notifiee may be different, based on the preferences of each notifiee, thus resulting to separate notification use cases (Fig. **6**). To identify the notifiees, but also IRs, in a use case, we may ask the following relevant questions (examples below are taken from the HIS case study, for the UC *Create Prescription*):

– Who receives notification about the creation of a *Prescription* in the IS?

– Answer: Patient, Pharmacist, Doctor

– What is the action of the *Patient* after being notified about his/her *Prescription*?

– Answer: To go to the pharmacy (that means *Patient* is just a *Notifiee*, since s/he does not affect the operation of the system directly)

– What is the action of the *Pharmacist* after being notified about the creation of a *Prescription*?

– Answer: To provide the *Drug* (in this way the *Pharmacist* needs to change the status/state of the *Prescription* IOi from *Pending* to *Complete*, therefore s/he is an IR).

Step 4. Structure UC Elements as Formalized Sentences

In the previous steps, the analyst identified and defined the UC modules, the use cases of each module, actors, associations, "include" and "extend" relationships between use cases, as well as generalization relationships. Additionally, during these three first steps, the analyst uses the identified UC elements to develop the UCDs which s/he finally completes after the application of steps 4-6. Step 4 involves writing the UC elements as formalized sentences. Such formalization not only helps to make expression of requirements more disciplined, understandable and organized, but it also makes easier their conversion into the UC diagrams and specifications. Additionally, formalization also helps to identify more easily new UC elements, such as complementary UCs, as illustrated in step 3 with the use of the accompaniment, and subsystems, as mentioned later in step 5. A formalized sentential use case pattern FSUC is a structured, semi-formal way of writing a use case of an IS, based on the basic syntactic form for writing a sentence in natural language, that is, <Subject> <Verb> <Object><Adverbial>. An FSUC is defined as follows:

$FSUC_F^{IO}$

=<A><F><IO><FC>::SendNotification<IR><No><FC>

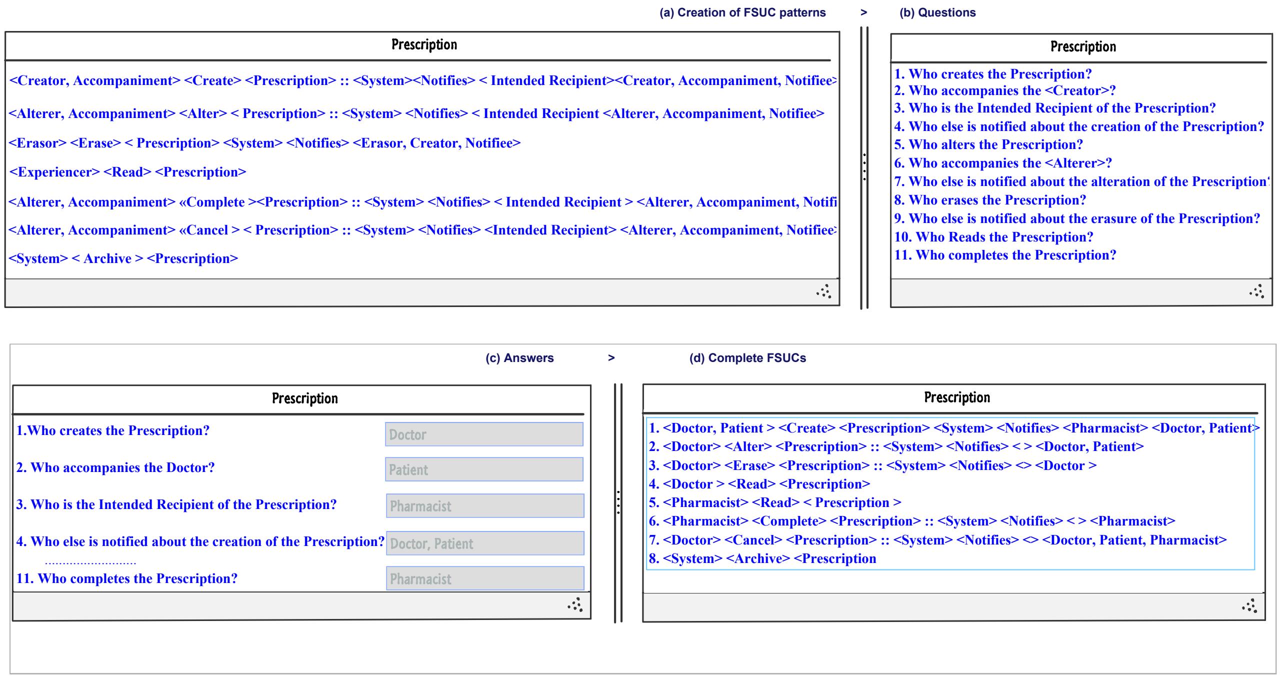
Where

UC function type *F* acts on the *Information Object IO*; the Actor group *A* refers to the primary actor and its accompaniments (secondary actors) if any, *IR* refers to the intended recipients, which are secondary actors, (*No*)tifiees are offstage actors, and *Functional Condition FC* is a clause that adds further information about the function, commonly by establishing the circumstances (temporal, locative, instrumental, and others) within which the function takes place. The syntax of the notification function, which is triggered after the execution of F, is placed after the symbol "::". Finally, the accompaniments' involvement is elaborated through separate complementary sentences. Below we present the FSUC *Create Prescription* example with a complementary sentence.

$$FSUC_{Cre}^{\mathrm{Pr}e} = \langle Doctor, Patient\rangle\langle Create\rangle\langle \mathrm{Pr}\,escription\rangle\left\langle \begin{array}{l} 8:00-14:00, 5\,\mathrm{min}, 10\,prescription/day, \\ DoctOfficeComputer, keyboard \vee stylus \end{array}\right\rangle ::$$
$$SystemNotifies\langle Pharmacist\rangle\langle Doctor, Patient\rangle$$

$$\langle Counselor\rangle\langle Give\rangle\langle \mathrm{Pr}\,escriptionHelp\rangle\langle ByPhone \vee ByEmail \vee ByForm\rangle$$

The FSUC Create Prescription above indicates the primary actor Doctor who executes the use case Create Prescription with the help of the accompaniment Patient who provides relevant data. While the doctor creates the prescription, s/he may need help from a counselor (accompaniment in this case, too) in filling some specific data values, such as



(a) Creation of FSUC patterns > (b) Questions

```
Prescription
<Creator, Accompaniment> <Create> <Prescription> :: <System><Notifies> < Intended Recipient><Creator, Accompaniment, Notifiee>
<Alterer, Accompaniment> <Alter> < Prescription> :: <System> <Notifies> < Intended Recipient <Alterer, Accompaniment, Notifiee>
<Erasor> <Erase> < Prescription> <System> <Notifies> <Erasor, Creator, Notifiee>
<Experiencer> <Read> <Prescription>
<Alterer, Accompaniment> «Complete ><Prescription> :: <System> <Notifies> < Intended Recipient > <Alterer, Accompaniment, Notifi
<Alterer, Accompaniment> «Cancel > < Prescription> :: <System> <Notifies> <Intended Recipient> <Alterer, Accompaniment, Notifiee>
<System> < Archive > <Prescription>
```

```
Prescription
1. Who creates the Prescription?
2. Who accompanies the <Creator>?
3. Who is the Intended Recipient of the Prescription?
4. Who else is notified about the creation of the Prescription?
5. Who alters the Prescription?
6. Who accompanies the <Alterer>?
7. Who else is notified about the alteration of the Prescription
8. Who erases the Prescription?
9. Who else is notified about the erasure of the Prescription?
10. Who Reads the Prescription?
11. Who completes the Prescription?
```

(c) Answers > (d) Complete FSUCs

Prescription	
1.Who creates the Prescription?	Doctor
2. Who accompanies the Doctor?	Patient
3. Who is the Intended Recipient of the Prescription?	Pharmacist
4. Who else is notified about the creation of the Prescription?	Doctor, Patient
....	
11. Who completes the Prescription?	Pharmacist

```
Prescription
1. <Doctor, Patient > <Create> <Prescription> <System> <Notifies> <Pharmacist> <Doctor, Patient>
2. <Doctor> <Alter> <Prescription> :: <System> <Notifies> < > <Doctor, Patient>
3. <Doctor> <Erase> <Prescription> :: <System> <Notifies> <> <Doctor >
4. <Doctor > <Read> <Prescription>
5. <Pharmacist> <Read> < Prescription >
6. <Pharmacist> <Complete> <Prescription> :: <System> <Notifies> < > <Pharmacist>
7. <Doctor> <Cancel> <Prescription> :: <System> <Notifies> <> <Doctor, Patient, Pharmacist>
8. <System> <Archive> <Prescription
```

Fig. (8). Based on the FSUC patterns for each IO (**a**), a number of predefined questions are created (**b**), and based on the answers to the questions (**c**) the complete FSUCs are finally produced. (**d**) Screenshots are taken from our software tool—for the IO *Prescription* of the HIS case study—which automates and supports the proposed approach.

drug dosage; therefore the UC Create Prescription may be extended by the UC Give Prescription Help.

Functional conditions may derive business rules that influence the actions of the UC specifications (e.g., the FC type time point may derive the business rule Doctor can create a prescription from 8:00-14:00) or they may also derive "include", "extend" or "generalization" relationships (e.g., Counselor Gives E-mail Help and Counselor Gives Form Help are specialized use cases of Counselor Gives Prescription Help).

Below we provide two indicative rules that illustrate how the FSUC can assist in creating the UC diagrams:

1. In a *Create*, *Alter*, *Alter-related*, *Erase*, or *Read* FSUC, the first actor in the *Actor* group is the primary actor (*Creator*, *Alterer*, or *Experiencer*) and should be positioned on the left of the use cases of the UCD.
2. The actors on the right of the first actor (primary), in the *Actor* group are accompaniments and are therefore secondary actors and should be positioned on the right of the use cases of the UCD.

Fig. (**8**) shows a screenshot which depicts the procedure for developing the UC requirements as formalized sentences using the NALASS tool. The tool [14] automatically creates the FSUC patterns for each UC module (Fig. **8a**). The UC modules correspond to the IOs provided to the tool by the analyst, as a part of an earlier activity. For the UC elements of each UC pattern, the tool creates relevant questions (Fig. **8b**), and the answers (Fig. **8c**) to these questions form the final complete sentences (Fig. **8d**). Then the tool uses conversion rules, such as the ones explained above, to read the complete formalized sentences and produce their corresponding UC diagrams and modules, as depicted in Fig. (**5**) earlier above, with the *Prescription* module and its UCD.

Step 5. Define UC Subsystems

UC modules of IOs created by the same actor may be possibly related and thus compose a UC subsystem which facilitates better organization and understanding of the UC elements and model. Such a subsystem supports related duties and responsibilities of mainly the same actor. Usually, the different modules of a subsystem are linked with an "extend" or an "include" relationship, but in some cases they may not be linked at all. Fig. (**9**) below shows a part of the subsystem *Hospital Reception* composed of the UC modules *Patient* and *Appointment*[14]. The Hospital Reception subsystem supports the duties of the hospital receptionist. The receptionist is the IS primary actor in creating patient appointments and recording new patients, which are two of her/his duties we indicatively present for the purpose of this paper. The receptionist is also involved in the other use cases—apart from creating appointments and patients, e.g., *Cancel Appointment*, *Read Patient*—of the UC modules *Appointment* and *Patient*. Patient and Doctor are secondary actors; the former provides his personal and other information to the receptionist, upon arrival and/or by phone, in order to create or alter an appointment. The latter provides information to the receptionist, such as confirming his/her availability for an appointment, so as to create or alter an appointment. Additionally, the doctor, as a primary actor, is authorized to read the appointment on his/her computer screen.

The grouping of different UC modules into a UC subsystem drives the analyst to investigate if this grouping derives any extend, include, or generalization relationships. For ex-

[14] For simplification, we haven't included the UCs *Erase IO* and other possible UCs, such as *Cancel IO*, *Complete IO*, and *Archive IO*. Furthermore *Send Notification* is conceived as a small sequence of actions at the end of each UC specification, and therefore it is not conceived as an "include" UC.

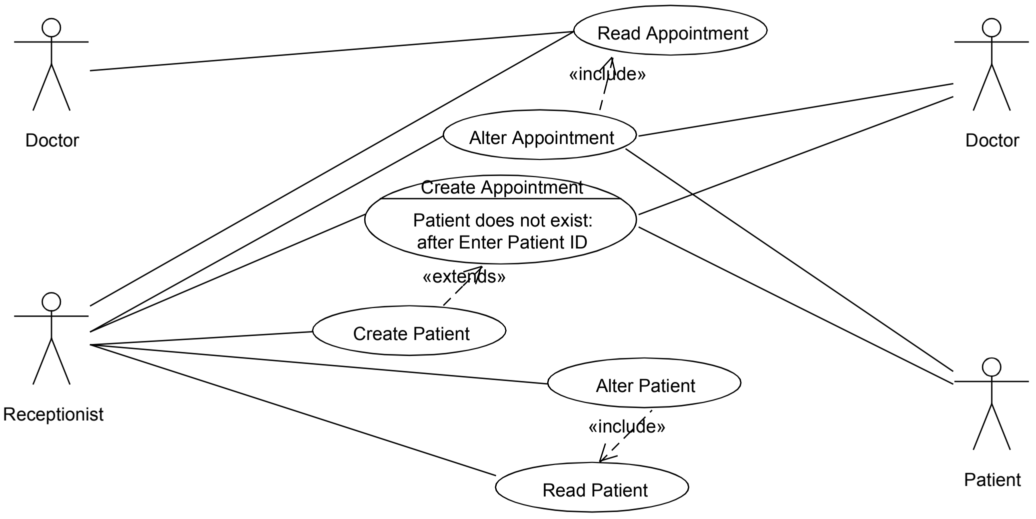


Fig. (9). Hospital Reception Subsystem UCD developed from 2 different modules: Patient and Appointment.

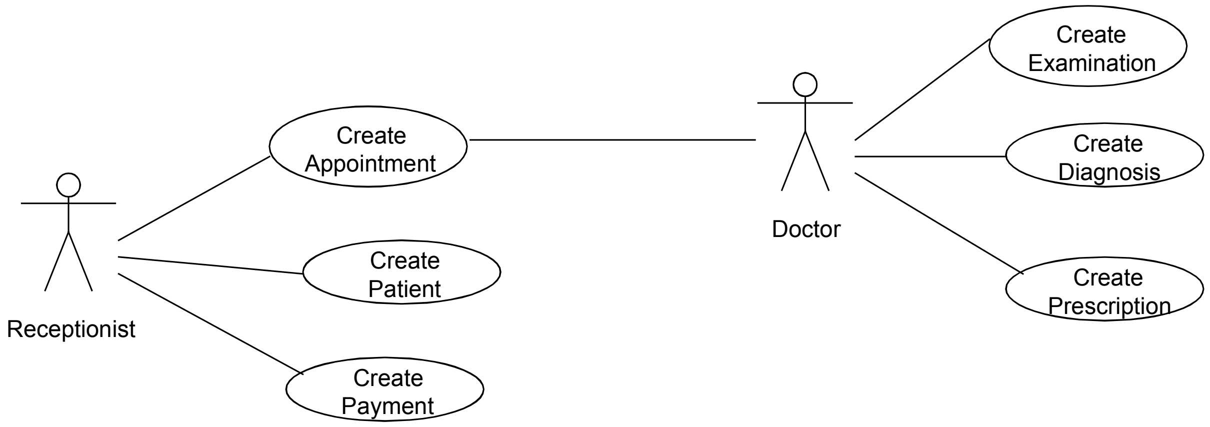


Fig. (10). Different subsystems are linked together to construct the entire system's UCD.

ample, in the Hospital Reception subsystem, the UC *Create Patient* extends the UC *Create Appointment*. This occurs when the receptionist is creating an appointment for a new patient who will be registered for the first time in the system. When the patient is already stored in the system, the extending use case will not be executed.

Different subsystems can be linked together. As mentioned in step 3, a good way to link subsystems is through an actor who plays the role of an IR (secondary actor) in module A of subsystem A and the role of a creator or alterer (primary actor) in module B of subsystem B, which results from module A. In this case, subsystem A may be linked with subsystem B. This is illustrated with the example of the *Prescription* module of the subsystem Hospital Practice[15] and the *Drug* module of the Pharmacy subsystem, in which the *Pharmacist* plays the role of IR in the *Prescription* module (in UC *Create Prescription*) and alterer in the *Drug* module (in UC *Alter Drug*). Another example related to the Hospital Reception subsystem is the relationship of its *Appointment* module with the *Examination* module of the Hospital Practice subsystem, where *Doctor* is an IR in UC *Create Appointment* of the former module, and doctor is a creator in UC *Create Examination* of the latter module. Therefore the Hospital Practice Subsystem is linked with the Hospital Reception subsystem, as shown in Fig. (**10**) above.

Step 6. Relate business rules with use cases and actors

Business rules associated to the use case interactions must be specified or, at least, referenced [1]. Business rules are never "owned" by a use case, since a business rule may be implemented by more than one use case. On the other hand, a business rule can be incorporated in a use case. As illustrated later in step 7 on use case specifications, some UC specification actions may need to comply with business rules. Failure to comply may lead to the termination of a use case or to alternative flows. Business rules can also determine new extend or generalization relationships. There are different types of business rules, such as general policies of an organization about data compliance standards (e.g., coding of clinical elements must comply with specific clinical data standards) or business rules derived from the functional conditions, as mentioned previously in step 4. Here we focus on two major types of business rules, as provided by NLSSRE:

(i) *Inter-related business rules*. These rules are created from combinations of two or more attributes between different interrelated participants (actors and IO). In particular, the

[15] Hospital Practice is composed of the modules *Prescription*, *Examination*, and *Diagnosis*, since doctor, as a primary actor, is the creator of all three of them

values of one or more attributes of one or more participants determine the values of one or more attributes of one or more related participants. The interrelated participants can be identified easily through their co-involvement in the same use case, and of course, in the same FSUC. For example, using the UC *Create Admission*:

FSUC=<Ward Clerk, Patient> <Create> <Admission> <Doctor><Ward Clerk>

we should examine if there are any special relationships between the actors involved during the execution of *Create Admission*. This examination takes place by checking combinations of attributes of the actors and the IO. For example, if *Admission.time* (where *time* is an attribute of the IO *Admission*) is more than one night, then *Patient* will be allotted a bed, whereas if *Admission.time* is zero nights, then *Patient* will not be allotted a bed (except only temporarily). In this case, two new specialized UCs (*Create Outpatient Admission*, *Create Inpatient Admission*) may be created, or the relevant business rule may be incorporated in the specification of UC *Create Admission*.

(ii) *Intra-related* business rules. These rules refer only to a particular IO, where the value of one attribute of an instance of the IO determines the value of another attribute of the same instance of the IO. An example of intra-related business rules in the form of questions, which may apply to the UCs *Create Doctor* and *Create Schedule*, are the following:

–How does the rank of a doctor affect his/her schedule?

Possible answer: If Doctor = Consultant (First) then Doctor's Work Time is no less than 18 mornings/month.

Else: If Doctor = Specialty Registrar (Second) then Doctor's Work Time is no less than 24 mornings/month.

Intra-related business rules are usually incorporated in the transaction flow of the UC specification. For example, when the UC *Create Schedule* is executed, one of its actions will be to check the doctor's rank and based on it to determine the doctor's schedule.

Intra-related business rules may also lead to the development of generalization relationships between actors, like in the example of the *Doctor.rank* attribute, which, when taking different values such as consultant or registrar, may lead to the specialized actors *Doctor Consultant* and *Doctor Specialty Registrar*.

If a business rule applies to a single use case, it may be attached as a note in the use case itself in both the use case diagram and its specification; if a business rule applies to multiple use cases, it may be written only once as a global note linked to every relevant use case in the UCD and UC specification [35].

Step 7. For Each Use Case, Write the Use Case Specification

Previous steps have illustrated how the UC elements are identified through formalization of use case types and actor roles, and how the UC modules, subsystems and the entire UC model is constructed, including UCDs. We have also presented screenshots and description of our CASE tool. Within this step, our approach also intends to formalize and automate the process of completing the UC specification template, and to provide clear and precise specifications. To achieve these aims, our approach applies (i) adaptation guidelines on the identified UC elements or/and on the formalized sentences, and (ii) NL authoring guidelines.

The UC specification template contains entries such as use case *name*, *identifier*, *description* (a couple of sentences or a paragraph describing the basic idea of the use case), *preconditions* (list of the state(s) the system is into before the use case starts), *basic flow* of actions (description of the "normal" processing path), *alternate flow* of actions or *exception conditions*, *post-conditions* (list of the state(s) the system can enter when this use case ends), *actors* (list of primary and secondary actors that participate in the use case), *stakeholders (offstage actors)*, *included use cases* (list of use cases that the template use case includes), *extending use cases* (the use case(s) that extend the template use case), and any *business rules* which concern the template use case.

The UC specification, similar to the construction of the UCD, may be developed incrementally, through the application of the steps of the proposed approach. However, complete UCDs and FSUCs are useful to facilitate the construction; therefore a significant part of UC specifications is constructed after the completion of the previous steps. Here we present the adaptation and authoring guidelines of our approach, and we mainly focus on the most significant parts of the use case specification, which are the basic flow and alternative flow/exception conditions of actions:

Guideline 1. The name of the use case consists of a verb followed by a noun phrase. Our approach provides specific use cases with specific names, such as Create IO, Alter IO, Read <IO report> (e.g., Create Prescription, Read Patient Record).

Guideline 2. Preconditions refer to the list of the state(s) the system is into before the use case starts. A good way to identify preconditions is to check if the primary actor A of the template use case is an intended recipient in another use case, described as essentially preceding use case (EPUC). EPUC is normally about the creation or alteration of an IO_{EPUC} which is used by actor A to execute the template UC. The state of IO_{EPUC}, defined after the execution of EPUC, determines this type of precondition. The syntax of this type of precondition is as follows:

"<IO_{EPUC}> is in < IO_{EPUC}.state> state (from UC <EPUC>)."

For example, a precondition of the UC Create Prescription is "Examination is in Complete state (from UC Create Examination)." as shown in Table **1**.

Regarding the automation part of detecting the preconditions from the elements identified in the previous steps, our CASE tool reads the FSUCs and matches the actors that both play the role of IR in one use case and primary actor (usually by reading the IO_{EPUC}) in another use case, and then it provides to the analyst the possible cases of preconditions to select from.

Another type of precondition refers to the primary actor that initiates the use case, that is, the creator, alterer or experiencer. Normally, the system must check that the primary

Table 3. Basic flow pattern for UC *Read IO*

1. System receives the <IO> identification from <Actor>.
2. System checks its data store for the <IO> based on the identifiers.
3. System converts the <IO> into the relevant format for viewing.
4. System displays <IO> mandatory and optional fields.
5. System notifies < **Actor**> that <**IO**> is displayed on screen.

actor has the access rights/credentials to initiate the use case. For example, for the UC *Create Prescription*, "*Doctor is authenticated*" is a pre-condition. The syntax of this precondition type is as follows:

"<Actor> is authenticated (*from* UC <Actor> Creates Authentication)"

For example, a precondition of the UC *Create Prescription* is "Doctor is authenticated (*from* UC Doctor Creates Authentication)." as shown in Table **4**.

Guideline 3. A post-condition usually refers to the resulting state of the IO after the execution of its use case. For example, for the UC *Create Prescription*, the result will be the prescription in a *Pending* state, which should have the following syntax:

"<IO> is in <IO.state> state."

For example, the post-condition of the UC *Create Prescription* is "Prescription is in Pending state." as shown in Table **4**.

Guideline 4. Actors that participate in the use case include at least one primary and zero or more secondary actors. From an FSUC, as shown in step 4, we can derive the primary actor, which is a creator, alterer or experiencer, and the secondary actors which play the roles of accompaniment or intended recipient. Each actor should be named with a singular noun; if actors are specializations of a general actor or if they refer to a system, they may be represented by a noun phrase, e.g., eye-doctor, medical system.

Guideline 5. According to Meyer *et al.* [36], a typical use case is described as a sequence of actions, and each action is expressed in natural language (if needed, one can extend a given action with an alternative behavior). That makes use cases readable for end-users. To maintain a high-degree of readability and understandability and to minimize ambiguity, our approach intends to formalize the use case actions by providing specific types of actions, written in a structured form of NL, as well as to automate their specification. The formalization is achieved by utilizing the sub-functions of each CAREN function, the attributes of each IO[16], functional conditions, data constraints and business rules. The automation is facilitated by our CASE tool.

In particular:

– The sub-functions *Enter Data, Check,* and *Save* are used as main actions in the basic flow of the *Create* UC specification (e.g., Table **1** actions 3, 4, and 6; Table **4** actions 3–12.3).

– The sub-functions *Delete*, *Enter Data*, *Check*, and *Save* are used as main actions in the basic flow of the *Alter* UC specification (also for any form of alteration, such as *cancel, complete,* etc.) (e.g., Table **2** actions 3–6 and 8).

– The sub-functions *Delete,* and *Check* are used as main actions in the basic flow of the *Erase* UC specification.

– *Read IO* is a basic use case which is included in the use cases *Alter IO, Erase IO,* and *Alter-related* UCs (*cancel*, *complete*, etc.), and it is invoked by the first action in the basic flow of the above UCs (e.g., Table **2** action 2).

– *Read* is decomposed to a sequence of actions in the UC *Create IO*. It has to do with reading a form with empty fields (required, optional) to be filled (e.g., Table **1** action 2).

– *Send Notification* is normally executed as the last action in the flow of the use cases *Create IO*, *Alter IO* (also *Cancel IO, Complete IO, etc.*), and *Erase IO*. It can be decomposed to small actions or defined as a separate use case (Table **1** action 7; Table **2** action 9; Table **4** action 13).

– *Select* or *Click* are secondary actions.

Normally, request actions are executed by an actor (including any involved accompaniments too), and respond actions are executed by the system. Usually an actor's action is followed by a system's action. In Tables **1**-**3** below, we present the sequence of actions for the basic flows of the UCs *Create IO*, UCs *Alter IO* and *Read IO,* whereas in Table **4** (actions 1-14, *flow of events* section) we can see the basic flow of the UC *Create Prescription*. NALASS reads each IO, its attributes and its FSUCs and creates the UC specifications flows based on the below patterns and by replacing the elements in "< >" with their corresponding values.

Alternative flows or exception conditions are easily defined by the use of data constraints. For each IO attribute entry in the UCs *Create IO* or *Alter IO*, or for each IO attribute deletion in the UCs *Alter IO* or *Erase IO*, an exception condition is applied with reference to its possible triggering point in the basic flow, after a system check is applied. The syntax of this kind of exception condition is as follows:

The system displays 'Invalid <IO> <IO.attribute>' message, if <IO> <IO.attribute> is incorrect. <IO> cannot be saved.

Table four shows examples of implementing various exception conditions, regarding the UC *Create Prescription* (actions 1.1-5.1, *exception conditions* section).

[16] The NLSSRE methodology provides different types of IOs and attributes that help in the identification of the attributes of each IO.

Table 4. Use Case Specification Example for UC *Create Prescription*

Use Case Name	Create Prescription
ID	UC 4
Description	The doctor fills out the form for a new prescrip-tion.
Preconditions	Examination is at Complete state (*from* UC *Create Examination*). Doctor is authenticated (*from* UC *Doctor Creates Authentication*).
Actors	Doctor (Primary), Patient (Secondary), Pharmacist (Secondary)
Stakeholders	Patient's Relative
Post-Conditions	Prescription is in *Pending* state
Flow of Events	Doctor selects create Prescription by clicking on 'create prescription' button. System displays new prescription creation form, including required and optional fields. Doctor Patient enter(s) Patient ID. The System checks Patient ID. Doctor enter(s) Drug Name. The System checks Drug Name. [Extension point: UC 22 Get prescription help] Doctor enter(s) Drug Dosage. 5.1 The System checks Drug Dosage. [Extension point: UC 22 Get prescription help] Doctor clicks on the Submit button. The Doctor adds Doctor's digital signature to the Prescription (BRU.001) The System adds a unique identifier to the Pre-scription. (BRU.002) The System saves the Prescription in the database. The System notifies the Doctor, Pharmacist, and Patient that Prescription is created *via UC 15*. Use case ends.
Exception condi-tion	3.1. The System displays 'Invalid Patient ID' mes-sage, if patient ID is incorrect. Prescription cannot be saved. 4.1. The System displays 'Invalid Drug Name' message, if Drug Name is incorrect. Prescription cannot be saved. 5.1. The System displays 'Invalid Drug Dosage' message, if Drug Dosage is incorrect. Prescription cannot be saved. 12. The System does not take any action if Doctor clicks on the Cancel button. Use case ends.
Includes	UC 15: Send Notification
Extended by	UC 22: Get Prescription Help
Extending other UCs	
Business rules	BRU.001: The Doctor signature follows NEHTA specifications. BRU.002: The Prescription identification number must comply with the format specified by NEHTA.

Guideline 6. Extension Points of a Use Case show exactly where in the basic flow an extending use case is allowed to add functionality. Extension points can be derived easily from the UCD. The extends relationship, as shown in Fig. (**9**), includes the condition that must be satisfied if the extension is to take place, and references to the extension points which define the locations in the base (extended) use case where the additions are to be made. For example, as shown in Fig. (**9**), UC *Create Appointment* is extended by UC *Create Patient*, under the condition "Patient does not exist in the system", at the extension point "Enter Patient ID". Our CASE tool reads the extension point "Enter Patient ID" of the UCD and matches it with the corresponding action of the extended UC (UC *Create Appointment*, in this example). On the right of the corresponding action, a relevant message is written, with the following syntax:

[Extension point: UC <UC id> <UC name>]

Table **4** includes two extension points at actions 6.1 and 7.1 of the basic flow.

4. EVALUATION

In order to prove the usefulness of our approach, we compared it to the Cockburn's widely-used use case approach [4], by applying both in 3 different real settings. A preliminary evaluation of the methodology preceded the second and third evaluations, the latter of which is presented in this paper. The first evaluation concerned the application of the approach for the RE task for the development of a Banking IS in Cyprus, and it was conducted by a postgraduate student with extensive knowledge in the field of software engineering. Relevant training was given to her to become familiar with the proposed approach and the dedicated software tool provided at the time of the evaluation. The evaluation assisted in clarifying and establishing several concepts of the approach, such as the overall application framework, the identification of primary and secondary actors, the alter-related use cases, and other issues. The second evaluation concerned the application of our approach for a Dentistry IS, and it was conducted by a novice software engineer, currently working in the software industry, at the same time with carrying out the evaluation described in this paper; the same engineer examined both approaches to investigate the experience of the same person using two different methods and tools. The second evaluation produced rather similar results to those obtained from the third evaluation presented in this paper[17].

As aforementioned, the methodological guidelines by Cockburn which we shall call *classical approach* for simplicity were chosen, because, besides their wide acceptance, they provide detailed and straightforward guidance for identifying the use case elements and constructing the use case specification. Two novice software engineers (for evaluation purposes we will call them SE1 and SE2) were assigned to test the two approaches. They were both graduate students of the University of Cyprus who attended several courses in software development over a period of three years prior to the experiment. Additionally, a specific one-week training and lecture were given to SE1 regarding the use of our approach and the NALASS tool, and a similar one-week course was given to SE2 in the form of a knowledge refresher on the classical UC approach—SE2 was already familiar with UML classical methods from corresponding courses in his studies. We assigned SE1 and SE2 the requirements engineering task for the development of a subsystem of the Library Information System (LIS) of the University of Cyprus.

[17] All case studies are available upon request from the corresponding author.

Table 5. Objective Quality Metrics Used to Determine the Effectiveness of the Approach

Quality Factors	Metrics	Our Approach	Classical Approach
Completeness[18]	Percentage of missing Use Cases (UC)	5% (3/60 UC)	20% (12/60 UC)
	Percentage of superfluous Use Cases	7% (4/57 UC)	17% (8/48 UC)
	Percentage of missing Primary Actors (PA)	0% (0/68 PA)	10% (6/58 PA)
	Percentage of missing Secondary Actors (SA)	2% (4/188 SA)	14% (22/158 SA)
	Percentage of missing use-case specification actions (Ac)	2% (18/855 Ac)	32% (232/720 Ac)
	Percentage of missing pre-conditions (Pre)	0% (0/66 Pre)	9% (5/55 Pr)
	Percentage of missing post-conditions (Pos)	6% (4/61 Pos)	10% (5/51 Po)
	Percentage of superfluous pre-conditions	0% (0/57UC)	6% (3/48UC)
	Percentage of superfluous post-conditions	0% (0/57UC)	6% (3/48UC)
	Percentage of superfluous Actors	0% (0/57UC)	8% (4/48UC)
	Percentage of superfluous use-case specification actions	0% (0/855)	20% (90/720)
	Number of missing associations and relationships	6 (for all 57 UC)	79 (for all 48 UC)
	Percentage of superfluous associations and relationships	0% (for all 57 UC)	12 (for all 48 UC)
	Percentage of missing business rules (BR)	14% (5/35 BR)	41% (12/29 BR)
Correctness[19]	Percentage of use cases with no identifier	0% (0/57 UC)	5% (3/55 (UC)
	Percentage of use cases and actors with no names	0%	1%
	Percentage of incorrect associations and relationships	0%	3%
	Number of other violations to Use Case modeling standards	0	6
	Number of Spelling errors	16[20]	38
Consistency	Percentage of redundant use cases	0%	12% (7/55 UCs)
	Percentage of redundant actions (per use case)	0%	10%
	Occurrences using words from more than one language	0	2
	Percentage of redundant business rules	0%	10% (3/29)
	Number of requirements referring to elements which are not present (e.g., use cases, use case diagrams)	0	6

Our evaluation, based on the method proposed by Geisser *et al.* [37] for evaluating RE methodologies, tested the following for each approach (including its underlying tools):

• The efficiency in terms of the output/effort ratio. Study of efficiency is beyond the scope of this paper. However it is worth noting that preliminary results indicated the application of our formalized approach with the automated support of NALASS to be much faster than the application of the classical approach, mainly due to the automatic generation of use case diagrams and specifications, examples of which were illustrated previously in this paper, as well as small references to the automation provided, later in this section.

• The effectiveness in terms of the achieved quality of the use case model produced by the application of each approach. The use case model included use-case elements (use cases, actors, associations and relationships), use-case diagrams and use-case specifications.

In order to objectively evaluate the output, we used a high-quality UC model as the reference. This UC model included UC diagrams and specifications and was derived from an existing, high-quality object-oriented SRS document, created by the analysts of the fully functional (currently in daily operation) LIS. The SRS document had been refined several times during the LIS initial development and implementation, and was known to reflect the desired performance of the existing LIS and the high satisfaction of its users. The UC model therefore served as a benchmark for the quality assessment of the specification developed by each student. We performed additional processing on the benchmark to achieve a clearer focus on atomic requirements, and

[18] This metric is applied to comparable use cases between each approach and the reference model. The denominator of the fraction in parenthesis refers to the number of elements existed in the reference model for the comparable use cases. For example, 68 is the number of primary actors existed in the reference model for the comparable use cases (57) between the reference model and our approach. Accordingly, for the classical approach this number is 58 (for 48 comparable use cases)

[19] For correctness we took into account the non-missing and redundant use cases and their elements provided by each approach, that is, 57 use cases from our approach and 55 use cases from the classical approach (7 use cases were defined twice). We did not take into count superfluous use cases.

[20] With 14 appearances of the same mistake, not different mistakes.

especially on UC actions of each UC specification, with the aim of ensuring a high degree of comparability. The existing UC elements, diagrams and specifications were compared with those derived from both our approach and the classical UC approach. SE1 used the proposed approach and its corresponding CASE tool, while SE2 employed the classical UC approach.

4.1. Evaluation Criteria and Analysis of Results

To evaluate the quality (and therefore, the effectiveness) of each produced UC model, we formed quality factors based on the quality frameworks of Moody [38], Moody and Shanks [39], and Sharma [40], as well as on the IEEE Recommended Practice for SRS [5]. Each factor was objectively measured against quality metrics. In this paper we make reference to what we and others [41, 42] consider the most significant quality factors, namely completeness, correctness, and consistency. Table **5** shows the summarized results for the quality of each produced specification, followed by discussion only on completeness, due to space limitation. To achieve a higher level of objectivity in the comparison, especially for measuring completeness, the comparison metrics were applied on equivalent elements. We found this equivalent-element comparison more meaningful when, for example, determining the percentage of missing actors from all the comparable use cases of each approach rather than the total number of missing actors from all the use cases of each approach and thus also including superfluous and redundant use cases. The comparison would not have been that objective if we compared the actors identified by each approach to the actors of the 60 use cases of the reference model, which include a number of incomparable use cases since some of them were not identified by the two approaches. Similarly, we are interested in the average percentage of missing actions in each comparable use case rather than the total number of missing actions in all use cases which also include superfluous and redundant use cases. With the term *comparable* use case, we mean the use case of our approach or the classical approach that has equivalent functionality with a use case of the reference model. For example, the comparable use case *Create Book* in our approach was *Add Book Details* in the classical approach and *Record New Book* in the reference use case model. Superfluous or redundant use cases are not included in the set of comparable use cases of each approach. The number of comparable use cases was 57 for our approach and 48 for the classical approach, compared to the 60 use cases proved by the reference model.

a. Completeness. Completeness refers to the extent to which the requirements model contains all necessary requirements [39] which for a UC model are use cases, actors, associations and relationships, use case diagrams, use case specifications including actions (in both normal and exception flows), pre-conditions, post-conditions, and business rules related to use cases. To assess the completeness of each UC model, we check for necessary information which is missing or information which is superfluous. Completeness is mainly focused on the content of the use-case model and not in the way it is written. In our experiment, we observed that the use-case model of the classical approach included several missing and superfluous use cases, actors, associations and relationships, actions, pre-conditions, post-conditions, and business rules. One of the major problems that arose from the use of the classical UC specification template was the omission of system response actions (e.g., a notification sent by the system to the librarian about a purchase of a new book was omitted). Another problem was the grouping of atomic actions in one transaction (e.g., *Librarian fills authorization form* instead of *Librarian adds username / Librarian adds password /* etc.) which also led to omissions of system response actions. All these problems mainly occurred due to the lack of formalized methods for identifying and specifying the UC elements. In contrast, our approach produced significantly better results. Our approach missed many fewer use cases. Also significantly, our approach tended to include all the elements inside each use case, including actions, pre-conditions, post-conditions, actors, references to "included" and "extending" use cases and business rules. These better results are due to the formalization provided by our approach for identifying the UC elements, with the use of predefined use case types and actors and guidelines to identify related associations and relationships, as well as due to the formalization of the UC specification actions of the transactions flow, rules for identifying preconditions and post-conditions, and an understandable way of expressing the content of UC specifications. However, although the error rate was lower, our approach was not 100% complete. The very small number of superfluous use cases resulted from the inclusion of one superfluous IO; in our approach the identification of IOs (and therefore, UC modules) is the first step and is performed manually by the analyst with the help of a relevant guide (provided by the NLSSRE methodology), which, although providing specific steps for the identification of IOs, could be enriched further. Moreover, the use case specification actions missed by our approach concerned notifications to four secondary actors the analyst failed to identify, therefore this issue did not occur because of a weakness in the proposed method of formalizing the actions of the UC specification transaction flows. Additionally, the application of our approach omitted three lower priority use cases, related to the reading of reports. Such use case types are not provided directly by our approach—although we give specific guidance as illustrated in step 2—and it is up to the analyst to identify them. Furthermore, the use of NALASS helped avoid missing the UC elements. As the results show, the analyst who applied the classical approach missed considerably more (as a percentage) preconditions and post-conditions than the analyst who applied our approach. The use of NALASS helped to minimize missing these elements because it automatically provided the preconditions of each use case and also different options for each use case regarding the new state (post-condition) of the IO, such as *Pending*, *Completed*, etc., so the analyst could decide accordingly. The two missing post conditions occurred, because the identification of post-conditions is not yet a fully automated process.

b. *Correctness*. Correctness refers to the extent to which the model conforms to the rules and conventions of the writing/modeling technique [41], which are, in our case, the naming rules, definition rules, diagrammatic conventions, etc. for the creation of the use case elements, use-case diagrams and use-case specifications. Since the NALASS tool automatically provides the use case types and also uses specific conversion and authoring rules for writing and drawing the use-case model, very minor problems appeared when using our approach, most of which were spelling mistakes from the analyst's input. The spelling mistakes mainly occurred from the manual entry of elements such as IOs, actors and IO attributes indicate that dictionary verification of the input data is an important future step. When compared with the error rate of the model produced by our approach, many more errors were found in the model produced by the classical approach, including spelling and grammatical mistakes as well as use cases with no identifiers. The language knowledge level of SE2 (as well as of SE1) was checked before the experiment and proved to be good. Therefore most of the mistakes are considered to be due to the analyst's oversight.

c. *Consistency*. Consistency assessment involves finding contradictions/conflicts between requirements, such as two or more use cases describing the same functionality with different terms, or two use cases with the same identifier, or missing use cases specification for use cases depicted in a use-case diagram and vice-versa. In our approach, contrary to the classical one, for each IO identified by the analyst, the NALASS tool provides clearly and automatically the use cases for each IO identified by the analyst and also guides the analyst in defining additional use cases, such as Alter-related (e.g., Cancel IO, Complete IO) or complementary use cases (e.g., the extending use case Give Prescription Help, as illustrated in one of our previous examples, in step 3). The NALASS tool also provides the specification template for each identified use case, with specific types of actions as defined earlier in step 7. In contrast, the use-case model of the classical approach used different wording/terminology for the same type of use case elements; for example, for the creation use case of each UC module, different verbs were used, such as 'create', 'record', 'fill', and 'complete'.

4.2. Threats to External Validity

Two main threats to external validity are relevant to our experiment, and are typical when running controlled experiments within time constraints: i) Are the subjects representative of software professionals? ii) Is the experiment material representative of industrial practice?

In our context, the main difference between students and professional requirements engineers is that the latter have more experience, and therefore we assume that they would apply the approaches more effectively than students given the same amount of training. Nevertheless, we consider valid the evidence that, given the same level of training and experience of the analysts, our approach produced more complete, correct and consistent results than the conventional approach. Additionally, the evaluation shows that one week of training with our tool (and approach) is sufficient to produce moderate- to high-quality results.

As for the second validity threat mentioned above, the application of the two approaches to larger scale systems seems likely to demonstrate at least a proportional increase in the differences between the two approaches. The involvement of more actors, information objects, use cases, relationships, use-case specification actions, pre-conditions, post-conditions, and business roles would be more easily handled by a structured, formal and understandable approach, such as ours, than from the classical approach. An experiment on larger scale systems is in our future work plans.

5. CONCLUSIONS AND FUTURE WORK

Use case driven analysis (UCDA) is one of the most common approaches in requirements engineering. However, existing UCDA approaches often result in poorly defined use case models due to the following reasons: (i) lack of specific support in identifying the use case elements, including use cases, actors, relationships, associations and business rules; (ii) use of generic use case specification templates that do not guide the analyst clearly how to identify each element of the template; (iii) use of free natural language to express the content of use case specification, which leads to inevitable ambiguity; (v) finally, the lack of a software tool makes UCDA a time-consuming and error-prone activity.

This paper presented an approach that is intended to solve the aforementioned problems through (i) formalizing the elicitation stage of UCDA by providing specific types of use cases and actors, specific guidelines to define associations, relationships and business rules, and formalized sentential patterns which provide a structured and expressive way to write the UC elements. (ii) formalizing the UC specification by providing specific types of actions performed with a specific sequence, for the normal and exception flows of the UC specification, and guidelines to complete the other parts of the UC specification template, such as pre-conditions and post-conditions. Additionally, authoring rules are applied on the identified UC elements and formalized sentences, in order to easily construct a semi-formal NL UC specification; (iii) a dedicated software tool that supports the automation of the proposed approach including the automated generation of use case diagrams and specifications.

To evaluate the effectiveness and efficiency of our approach, we performed a short-scale experimental study through which we compared our approach to the classical UCDA approach, by applying both of them in a real-life setting. The results showed that the proposed approach performed much better than the classical approach in the various objective quality metrics established, proving superior in terms of completeness, correctness and consistency. Additionally, the evaluation showed that our approach and tool can easily be learnt and applied in practice. The difference was also significant in regard to efficiency, although examined only briefly in this paper, where our approach performed much faster than the classical one.

It is our belief that this novel work has achieved significant steps toward providing straightforward and automated support for the development of the use case model. However, it remains to be tested on large scale projects, and such an experiment will be part of future work. Other issues to be addressed in future research will include enriching the guidelines for facilitating more precise and straightforward identification of alter-related use cases including *Cancel IO*, *Complete IO*, etc., as well as extending the approach and its CASE tool in order to support the requirements design phase, with easy creation of interaction and state diagrams from the use case model. Additionally, the use of a dictionary to check user's input for spelling will be a future feature of the NALASS tool.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflicts of interest.

ACKNOWLEDGEMENT

Declared none.

REFERENCES

[1] F. Dias, A. Schmitz, M. Campos, A. Correa, A. Alencar, "Elaboration of use case specifications: an approach based on use case fragments" In: *ACM Symposium on Applied Computing (SAC)*, Fortaleza, Ceara, Brazil, 2008, pp. 614-618.

[2] R. Pooley, and P. Stevens, *Using UML - Software Engineering with Objects and Components*. Addison Wesley Longman: Harlow 1999.

[3] J. Kim, P. Sooyong, and S. Vijayan, "A Linguistics-Based Approach for Use Case Driven Analysis Using Goal and Scenario Authoring", In: *Proceedings of Applications of Natural Language to Data Bases,* 2004, pp. 159-170.

[4] A. Cockburn, *Writing Effective Use Cases*. *Reading*, Addison Wesley: Massachusetts 2000.

[5] IEEE *IEEE Recommended Practice for Software Requirements Specifications*, *ANSI/IEEE Standard 830-1998*. Institute of Electrical and Electronics Engineering: New York 1998.

[6] B. Dano, H.Briand, and F.Barbier, "A Use Case Driven Requirements Engineering Process", *Requirements Engineering*, vol. 2, No. 2, 1997, pp. 79-91.

[7] R. Denney, *Succeeding with Use Cases Working Smart to Deliver Quality*. Addison-Wesley Professional: Boston 2005.

[8] D. Jagielska, P. Wernick, M. Wood, and S. Bennett, "How natural is natural language? how well do computer science students write use cases?" In: *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'06),* Portland, Oregon, USA, 2006.

[9] M. El-Attar and J. Miller, "Matching antipatterns to improve the quality of use case models", In: *Proceeding of the 14th IEEE International Requirements Engineering Conference (RE'06)"*, pp. 99-108, 2006.

[10] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, 2005, pp. 277-330.

[11] M. Georgiades and A. Andreou, "A novel methodology to formalize the requirements engineering process with the use of natural language", In: *Proceedings of the IADIS Conference on Applied Computing,* Timisoara, Romania, IADIS Digital Library, 2010, pp. 11-18.

[12] M. Georgiades and A. Andreou, "A methodology to formalize and automate the requirements engineering process with the use of natural language" [Under preparation for journal submission. Extended version of [11]].

[13] M. Georgiades, A. Andreou, and C. Pattichis, "A requirements engineering methodology based on natural language syntax and semantics", In: *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05) (Paris, France, August),* IEEE Computer Society, Washington 2005; pp. 73-74.

[14] M. Georgiades and A. Andreou, "A novel software tool for supporting and automating the requirements engineering with the use of natural language", In: *Proceedings of the ICSCT International Conference on Software and Computing Technology,* Kunming, China, IEEE Press, , 2010, pp. 256-263.

[15] M. Georgiades and A. Andreou, "Automatic generation of a software requirements specification (SRS) document", In: *Proceedings of the Intelligent Systems Design and Applications Conference,* Cairo, Egypt, IEEE Press, 2010, pp.1095-1100.

[16] G. Fliedl, C. Kop, W. Mayerthaler, H. Mayr, and C. Winkler, "The NIBA workflow: From textual requirements specifications to UML-schemata", In: *ICSSEA '2002 - International Conference 'Software & Systems Engineering and their Applications'*, Paris, France, 2002.

[17] D. Liu, S. Kalaivani, E. Armin, and F. Behrouz, "Natural language requirements analysis and class model generation using UCDA", *Lecture Notes in Computer Science, Innovations in Applied Artificial Intelligence: 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE,* Springer: Berlin, vol. 3029, pp. 295-304, May 2004.

[18] I. Jacobson, "Use cases - Yesterday, today, and tomorrow", *Software and System Modeling*, vol. 3, No.3, pp. 210-220, 2004.

[19] G. Booch, J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language User Guide*. 2nd ed. Addison-Wesley: Massachusetts 2005.

[20] M. Eriksson, K. Börstler, and K. Borg, "Marrying features and use cases for product line requirements modeling of embedded systems", In: *Proceedings of the Fourth Conference on Software Engineering Research and Practice (SERPS'04),* Sweden, 2004, pp.73-82.

[21] J. Leite, G. Rossi, M. Balaguer, G. Kaplan, G. Hadad, and A. Oliveros, "Enhancing a requirements baseline with scenarios", In: *Proceedings of Requirements Engineering,* Annapolis, USA, 1997.

[22] M. Ochodek, J. Nawrocki, "Automatic transactions identification in use cases", In: *Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007,* Poznan, Poland, October 2007, pp. 55-68.

[23] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen, "Formal approach to scenario analysis", *IEEE Software*, vol. 11, No.2, March 1994.

[24] M. Glinz, "An integrated formal model of scenarios based on statecharts", In: *Proceedings of 5th European Software Engineering Conference,* Sitges, Spain, Springer (Lecture Notes in Computer Science 989), Sept 1995, pp. 254-271.

[25] C. Seybold, S. Meier, and M. Glinz, "Scenario-driven modeling and validation of requirements models", In: *5th ICSE International Workshop on Scenarios and State Machines: Models, Algorithms and Tools,* Shanghai, May 2006, pp. 83-89.

[26] J. Ellison and P. Moore. "*Trustworthy Refinement Through Intrusion-Aware Design (CMU/SEI-2003-TR-002)"*. Technical Report. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. Available: http://www.sei.cmu.edu/publications-/documents/03.reports/03tr002.html .

[27] H. Podeswa. "*UML for the IT Business Analyst: A Practical Guide to Object-Oriented Requirements Gathering"*. Course Technology PTR: USA, 2005.

[28] S. de Cesare, M. Lycett, and R. Paul, "Actor perception in business use case modeling", *Communications of the AIS,* vol 12, pp. 223-241, 2003.

[29] P. Zielczynski, "*Requirements Management Using IBM. Rational RequisitePro"*. IBM Press: USA, 2007.

[30] D. Coleman, "A use case template: draft for discussion", *Fusion Newsletter*, April 1998. [Online]. Available: http://www.hpl.hp.-com/fusion/md_newsletters.html

[31] D. Avison and G. Fitzgerald, *Information Systems Development: Methodologies, Techniques and Tools*, 3rd ed. McGraw-Hill: UK2003.

[32] I. Marsic, "*Software Engineering*". Rutgers, The State University of New Jersey, 2009. [E-book]. Available: http://www.ece.rutgers.-edu/~marsic/books/SE/

[33] Sybase, *PowerDesigner Object Oriented User's Guide*. Sybase: Dublin, 2002.

[34] R. Malan and D. Bredemeyer, *Functional Requirements and Use Cases,* June 1999. [Online]. Available: http://www.bredemeyer.com/use_cases.htm

[35] S. Alhir, *Guide to Applying the UML*. Springer: Berlin, 2002.

[36] B. Meyer, J. Nawrocki, B. Walter, "Balancing agility and formalism in software engineering", In: *Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007,* Poznan, Poland, October 2007, ["Revised Selected Papers", in Proceedings of CEE-SET, 2008].

[37] M. Geisser, T. Hildenbrand, F. Rothlauf, and A. Atkinson, "An evaluation method for requirements engineering approaches in distributed software development projects" In: *Proceedings of the Second International Conference on Software Engineering Ad-*

vances, Cap Esterel, French Riviera, France, IEEE Computer Society Press, 2007, pp. 39-39.

[38] D. Moody, “Measuring the quality of data models: an empirical evaluation of the use of quality metrics in practice”, In: *Proceedings of the Eleventh European Conference on Information Systems*, Naples, Italy, June 2003.

[39] D. Moody and G. Shanks, “What makes a good data model? evaluating the quality of data models”, *Australian Computer Journal*, pp. 97-110, 1998.

[40] A. Sharma, *Requirements Quality Assessment for Outsourcing*. Master’s thesis, Eindhoven University of Technology, Eindhoven, Netherlands, 2009.

[41] S. Espana, N. Condori-Fernández, A. González, and O. Pastor, “An empirical comparative evaluation of requirements engineering methods”, *Journal of Brazilian Computer Socity*, vol 16, No. 1, pp. 3-19, 2010.

[42] I. Menzel, M. Müller, A. Groß, and J. Dörr, “An experimental comparison regarding the completeness of functional requirements specifications”, In: *Proceedings of the 18th IEEE international Requirements Engineering Conference*, Sydney, Australia, September 2010.

Received: April 07, 2012 Revised: May 20, 2012 Accepted: May 21, 2012