# A NATURAL LANGUAGE-BASED METHODOLOGY TO FORMALIZE AND AUTOMATE THE REQUIREMENTS ENGINEERING PROCESS

Georgiades Marinos Georgiou

University of Cyprus, 2011

Existing Requirements Engineering (RE) approaches often result in poorly defined requirements due to the lack of appropriate methods for discovering and documenting user needs. This dissertation describes Natural Language Syntax and Semantics Requirements Engineering (NLSSRE), a compact and clear-cut methodology that intends to formalize and automate a large part of the Requirements Engineering (RE) process, including discovery, analysis, and specification of user requirements for the development of information systems. The formalization is mainly achieved by utilizing elements of natural language syntax and semantics, with the focus on keeping ambiguities low and expressiveness high, while the automation is realized with the use of a dedicated CASE tool to support NLSSRE. In particular, RE is converted to a series of predefined steps, through which the analyst is guided in advance what specific types of data, functions, business rules and conditions to use and search for, how to form and document them using formalized sentential patterns, and what specific questions to ask the users in order to correctly elicit their needs. Finally specific rules are utilized to build diagrammatic notations and semi-formal specifications. Particular focus and elaboration is given on how NLSSRE is adapted for formalizing and automating use case model development.

Preliminary empirical evaluation demonstrated the effectiveness and efficiency of the proposed methodology.

Georgiades Marinos Georgiou - University of Cyprus, 2011

# A NATURAL LANGUAGE-BASED METHODOLOGY TO FORMALIZE AND AUTOMATE THE REQUIREMENTS ENGINEERING PROCESS

Georgiades Marinos Georgiou

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

August, 2011

# APPROVAL PAGE

Doctor of Philosophy Dissertation

## A NATURAL LANGUAGE-BASED METHODOLOGY TO FORMALIZE AND AUTOMATE THE REQUIREMENTS ENGINEERING PROCESS

Presented by
Georgiades Marinos Georgiou

| | |
|---|---|
| Research Supervisor | Andreas S. Andreou, supervisor 9/2001-8/2010, co-supervisor 9/2010-8/2011<br>……<br>Research Supervisor's Name |
| Committee Member | Constantinos S. Pattichis, co-supervisor 9/2010-8/2011<br>……<br>Committee Member's Name |
| Committee Member | Christos N. Schizas<br>……<br>Committee Member's Name |
| Committee Member | Vasos Vassiliou<br>……<br>Committee Member's Name |
| Committee Member | Nuria Castell<br>……<br>Committee Member's Name |
| Committee Member | Nikos Karacapilidis<br>……<br>Committee Member's Name |

University of Cyprus

August, 2011

# ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr Andreas Andreou for being advisor and friend. Without his guidance and support, this dissertation would not have been accomplished. No matter how busy he was, Andreas always found time to answer my questions, and review my papers and dissertation. I am grateful to him for providing time, ideas, funding, as well as encouragement and psychological support to make my Ph.D. experience productive and invigorating.

I'd also like to give special thanks to Dr Constantinos Pattichis for his courteous and caring attitude, from the initial stages of my Ph.D. His academic support, funding and personal boost are greatly appreciated.

Thirdly, I am very grateful to the remaining members of my dissertation committee, Dr Christos Schizas, Dr Vasos Vassiliou, Dr Nikos Karacapilidis and Dr Nuria Castell. I appreciate their time, interest, helpful comments and insightful questions.

My gratitude is also extended to Damon Ericsson who proof-read this dissertation and provided stylistic and substantive corrections, and suggestions for improvement.

I thank Christiana Hadjikyriakou and Irene Kyriacou for their significant contributions to the empirical evaluation of the proposed methodology. I am also thankful to Tasos Klitou, Nicoletta Nicolaidou and Sofia Hadjidemetriou for their contribution to the development of the CASE tool supporting the proposed methodology.

I want to express my gratitude to Savvoula Efstathiou who helped me with every procedural study matter, such as completing required paperwork and deliver it to the right place.

Lastly, but not least, I would like to thank my beloved parents for their love and encouragement. For always being there when I needed them and supported me with patience and faith.

## LIST OF FIGURES

# LIST OF TABLES

## ACRONYMS AND ABBREVIATIONS

CAD = Computer-Aided Design

CAREN = Create-Alter-Read-Erase-Notify

CASE = Computer-Aided Software Engineering

CIRCE = Cooperative Interactive Requirements-Centered Environment

COLOR-X = COnceptual Linguistically-based Object-oriented Representation language for Information and Communication Systems (ICS, expressed as X)

DFD = Data Flow Diagram

DOORS = Dynamic Object-Oriented Requirements System

EPUC = Essentially Preceded Use Case

ER = Entity Relationship

FC = Functional Condition

FSR = Formalized Sentential Requirement

FSUC = Formalized Sentential Use Case

HIS = Hospital Information System

HTML = Hypertext Markup Language

ID = Identity

IEEE = Institute of Electrical and Electronics Engineers

IO = Information Object

IOi = Information Object instance

IR = Intended Recipient

IS = Information System

IT = Information Technology

LIS = Library Information System

NALASS = Natural Language Syntax and Semantics

NEHTA = National E-Health Transition Authority

NL = Natural Language

NL-SRS = Natural Language Software Requirements Specification

NLSSRE = Natural Language Syntax and Semantics Requirements Engineering

OO = Object Oriented

PC = Personal Computer

QuARS = Quality Analyzer of Requirements Specification

RA = Requirements Analysis

RD = Requirements Discovery

RE = Requirements Engineering

RS = Requirements Specification

RTF = Rich-Text Format

SDLC = System Development Life Cycle

SE1, SE2 = Software Engineer 1, Software Engineer 2

SRS = Software Requirements Specification

STORM = Software Tool for the Organization of Requirements Modeling

UC = Use Case

UCD = Use Case Diagram

UCS = Use Case Specification

UML = Unified Modeling Language

# 1 Introduction

Requirements Engineering (RE) is of vital importance for the development of information systems (Sommerville, 2005). Several studies have shown that a large proportion of errors detected in the implementation and later stages of the systems development life cycle (SDLC) can be traced back to incomplete, ambiguous, incorrect or omitted requirements defined during RE, early in the SDLC (The Standish Group, 2003, 2009; Hall et al., 2002). A substantial percentage of software projects continue to fail, often because requirements are ill-defined, ambiguous, incomplete with respect to the users' needs, or are not managed correctly as the project unfolds. The Standish Group's periodically released "Chaos Report" indicates that the software crisis remains a challenging issue. Table 1.1 illustrates the percentage of software projects that failed or were over budget in different years between 1994 and 2008.

TABLE 1.1 STANDISH FINDINGS BY YEAR UPDATED FOR 2008

| | 1994 | 1996 | 1998 | 2000 | 2002 | 2004 | 2008 |
|---|---|---|---|---|---|---|---|
| Succeeded | 16% | 27% | 26% | 28% | 34% | 29% | 32% |
| Failed | 31% | 40% | 28% | 23% | 15% | 18% | 24% |
| Challenged | 53% | 33% | 46% | 49% | 51% | 53% | 44% |

Further results indicate 52.7% of projects cost 189% of the original estimates. On the basis of this research, the Standish Group estimates that American companies and Government Agencies spend $55 billion annually for cancelled software projects. The

Standish group study, after having contacts with respondents, noted that the three most commonly cited factors that caused projects to be late and not meet expectations were (i) lack of user input, (ii) incomplete requirements and specification, and (iii) changing requirements and specifications. Similarly, the three major reasons that a project will succeed are user involvement, executive management support, and a clear statement of requirements. Without them, the chance of failure increases dramatically. Van Vliet (2008) demonstrates that requirements errors are likely to be the most common class of errors, and requirements errors are likely to be the most expensive errors to fix if not tracked early in the SDLC process. Requirements errors are likely to consume 25-40% of the total project budget. Requirements errors account for 70 percent to 85 percent of the rework costs on a software project. Wiegers (2006) states that if one finds a requirements defect during the requirements phase and it costs one unit to fix (for example, three engineering hours, $500), the cost of fixing that same defect will typically increase if it is found later in the project's life cycle. In fact, studies show that it can cost more than 100 times more to fix a requirements defect if it is not found until after the software is released to the field.

It is also evident that the least understood parts of RE are the activities of requirements discovery, analysis and specification. The problem observed is that there is an enormous gap between the clients' needs and the software engineers' understanding of the clients' needs (Goldin and Berry, 1997; Mich et al., 2004). Clients often speak with vague sentences and/or cannot express their functional needs or, even worse, they do not know what these needs really are. This problem is amplified further when the analyst does not provide the right questions, as s/he essentially does not know precisely what to ask.

Additionally, the ambiguous and incomplete requirements gathered during requirements discovery are insufficiently classified and organized during the analysis stage, due to lack of current approaches to provide specific predefined types of functions and data. As a result, the poorly-organized gathered requirements result to ill-defined and ambiguous specifications, usually written in free natural language—which is also ambiguous, inherently—and based on a generic requirements specification template.

An understandable and straightforward Requirements Engineering methodology is therefore necessary for the successful development of an Information System (IS), a methodology which will formalize and automate the critical stages of requirements discovery (RD), analysis (RA), and specification (RS). Such a methodology is greatly enhanced when it involves the use of natural language (NL), because NL makes the entire RE process and its outcomes easy to understand for both users and analysts. Current approaches in RE, both those based on NL and those not based on NL, fail to provide a specific, easily understood formalization of the major parts of requirements discovery, analysis and specification. The problem originates from the weakness of existing approaches to formalize the RD process. On the one hand, some approaches use NL parsing techniques to retrieve requirements from pre-existing requirements documents, but this method is not reliable, because of ambiguities, redundancies and inconsistencies present in such documents. On the other hand, other approaches avoid formalization and use open-ended questions that lack specificity and formality (Gervasi and Zowghi, 2005). In either case, often the result is a requirements document with ambiguities, redundancies and inconsistencies. In the subsequent RA and RS stages, current approaches are also weak in providing a specific and easily understood formalization of the elements of an IS,

so that the analyst will know specifically what data, functions, business rules, and functional conditions to use and search for, as well as how to define them. Instead, they use general guides and templates, such as the traditional IEEE SRS document template or templates related to the more contemporary Use Case specification. The use of such templates also results in requirements documents written in a free, informal version of NL which promotes ambiguity and redundancy. The informality in such documents hinders also the use of automated tools for system modeling, since informal NL is inherently complex, vague and ambiguous.

The RE process varies considerably depending on its context type and the application being developed. RE for information systems is significantly different from RE for embedded control systems, or from RE for generic software services such as networking and operating systems (Sommerville, 2005; Nuseibeh and Easterbrook, 2000). This dissertation proposes the Natural Language Syntax and Semantics Requirements Engineering (NLSSRE) Methodology, which intends to engineer the correct requirements in a clear-cut, time-saving, understandable and reliable way. NLSSRE aims to formalize and automate the discovery, analysis and specification of user requirements for the development of Information Systems. In particular, the methodology strives towards handling user requirements concerned with the operational aspect of an IS[1] and building these requirements with the use of the following IS elements: people (usually end-users,

[1] According to Ellison and Moore [2002], 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 of a business). Examples of ISs encompassing the operational aspect to a large degree are a Hospital IS or a Library IS.

clients and trusted external users), processes related to the creation, modification, transmission, storage and presentation of information along with the circumstances within which these processes are performed, as well as data, constraints, and business rules. The formalization of RE, as performed with NLSSRE, intends to make easier the transformation of requirements into specification and design models, and finally into their software implementation. Formalization can be achieved with the aid of NL elements such as verbs, nouns, genitive case, adjectives and adverbials, and it will be automated with the support of a dedicated CASE tool. Specifically, the methodology is expected to achieve its aim by providing the analyst with several critical elements in advance of each main task involved in RE (in other words, the following are the specific contributions of this research):

- Specific questions and guidance for identifying requirements including business roles, information objects, data attributes, business rules and functional conditions (Georgiades and Andreou, 2010b, 2011a, 2011b; Georgiades et al., 2005)..
- Predefined types of functions, specific categories of data, and specific types and methods to identify and define business rules and functional conditions (the circumstances within which each function is performed) (Georgiades and Andreou, 2010b, 2011a, 2011b; Georgiades et al., 2005).
- Specific patterns for writing requirements as structured, semi-formal NL sentences (Georgiades and Andreou, 2010b, 2011a, 2011b; Georgiades et al., 2005).
- Specific rules to transform the abovementioned identified requirements into diagrammatic notations, including class diagrams, data flow diagrams and use-case diagrams, as well as use-case and textual specifications, the latter following a certain

IEEE SRS template (IEEE, 1998) (Georgiades and Andreou, 2010a, 2010b, 2010c, 2011a, 2011b, 2011c; Georgiades et al., 2005).

- A CASE tool that automates the entire procedure (Georgiades and Andreou, 2011c, 2010c, 2010a).

Finally, another objective of the NLSSRE methodology is its adaptation for formalizing use case model development (Georgiades and Andreou, 2011a), since use case driven analysis is a popular, user-friendly approach which attempts to cover the three RE activities (elicitation, analysis, specification). We will show how the entire NLSSRE methodology can be adjusted for use case model development and how it concludes with the construction of use case diagrams and use case specifications.

Our decision to adopt NL for RE is based on the following significant reasons: (i) language, by its nature, is the most powerful medium of expression—through the analysis of specific linguistic elements, we expect to facilitate the identification and formalization of specific IS requirement elements, including functions, data, functional conditions and business rules; (ii) linguistic terminology can facilitate a common terminology of requirements and eliminate ambiguities and redundancies in specifying them; and (iii) it is simpler and more reliable to construct requirements in some form of formal NL than in any other language, because requirements are initially conceived and expressed in NL. NLSSRE gives requirements an NL-like description which is also very understandable and useful as a communication medium between the users, analysts and programmers of the IS. NL-like description corresponds to the use of a structured, semi-formal NL for describing requirements, and lies between formal languages that require high expertise, and free NL that is inherently ambiguous. Semi-formal NL provides, on the one hand,

requirements precision and transformability to models, and, on the other hand, understandability.

The remainder of this dissertation is structured as follows:

Chapter 2 explains the application domain of RE with particular reference to the application domain of the proposed methodology. The main part of this chapter elaborates on the main components and concepts of RE, which are necessary to be taken into account for building a solid RE methodology. Existing RE approaches and techniques, most of them with particular focus on the use of natural language, are also discussed and compared to the NLSSRE methodology.

Chapter 3 presents NLSSRE, a novel methodology that is intended to formalize and automate the major activities of RE, namely requirements discovery, analysis and specification, with the aid of natural language. The chapter discusses in detail the underpinning of the methodology, as well as each of its steps with some illustrative examples, many of them taken from the software tool that provides automation support to the methodology.

Chapter 4 describes the adaptation of the methodology for formalizing use case model development. In particular, elements of the NLSSRE methodology are used to formalize use case elements, such as use case types, actors and use case specification actions.

Chapter 5 describes NALASS, the dedicated software tool that automates the NLSSRE methodology.

Chapter 6 describes an experimental evaluation of the methodology, through which we compared the NLSSRE methodology to the classical Object Oriented (OO) RE approach, by applying both of them in a real-life setting.

Chapter 7 discusses how the proposed methodology is different from other closely related approaches and techniques, and how it achieves its aim and objectives.

Chapter 8 draws the conclusions of this research and summarizes its contributions, while chapter 9 provides recommendations for future work.

# 2 Background and Related Work

In this chapter, an initial understanding of *Requirements Engineering* is developed through a number of definitions, and the application domain of RE is discussed with particular reference to the application domain of the proposed methodology. The main part of the chapter elaborates on the main components and concepts of RE, which need to be taken into account when building a solid RE methodology. Existing RE approaches and techniques, most of them with particular focus on the use of natural language, are also discussed and compared to the NLSSRE methodology.

## *2.1 Definitions*

The current literature has several definitions of Requirements Engineering. The following definitions are widely used and serve to develop an initial understanding of the constituent parts of RE and their context.

Requirements engineering defined by Pohl (2010) as "the process of eliciting individual stakeholder requirements and needs and developing them into detailed, agreed requirements documented and specified in such a way that they can serve as the basis for all other system development activities."

A second definition is given by Thayer and Dorfman (1997) "Software requirements engineering is the science and discipline concerned with establishing and documenting software requirements."

Loukopoulos and Kavakli (1995) defines RE as "the systematic process of developing requirements through an iterative co-operative process of analyzing the problem,

documenting the resulting observations in a variety of representation formats, and checking the accuracy of the understanding gained."

One of the most commonly used definitions in the literature is given by Zave and Jackson (1997):

"Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families."

All the definitions agree that Requirements Engineering is a process which produces the software requirements for a system. This process involves the activities of elicitation, analysis and specification, and results in a document including functional and non-functional requirements. RE is a multidimensional discipline, because it is not only related to technical issues and problems but also to managerial, organizational, economic and social issues. Requirements Engineering, as part of software engineering, is a systematic approach intended to support professional software development**,** rather than individual programming. It includes techniques that support requirements elicitation, analysis, and specification, none of which are normally relevant for personal software development (Sommerville, 2010).

## *2.2 Application Domain*

The way RE is implemented varies dramatically depending on the organization developing the software, the type of software application, and the people involved in the development process. According to Sommerville (2010), perhaps the most significant factor in determining which software engineering methods and techniques are most important is the type of application that is being developed.

There are many different types of applications including stand-alone applications (e.g. office applications on a PC, CAD programs, photo manipulation software), embedded control systems which use their software system to control and manage hardware systems (e.g., software in a mobile phone), simulation systems intending to model physical processes or situations, entertainment systems intended mostly for personal use and entertainment of the user (e.g., games), data collection systems intended to collecting data from the environment using sensors, and information systems; this last type is the focus of our work.

According to Ellison and Moore (2002), 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 (fig. 2.1). This is the most significant part—it is actually an information system by itself—of the entire IS of an organization, because it deals with the day-to-day transactions of a business. It involves significant interaction between employees with operational roles, who use data and processes to produce, change and store information. The automation of an existing manual system allows for faster processing, reduced clerical costs and

improved customer service. Examples of ISs encompassing the operational domain are a Hospital IS or a Library IS, both of which could contain sub-systems such as a billing IS, a salary payment IS, and others.

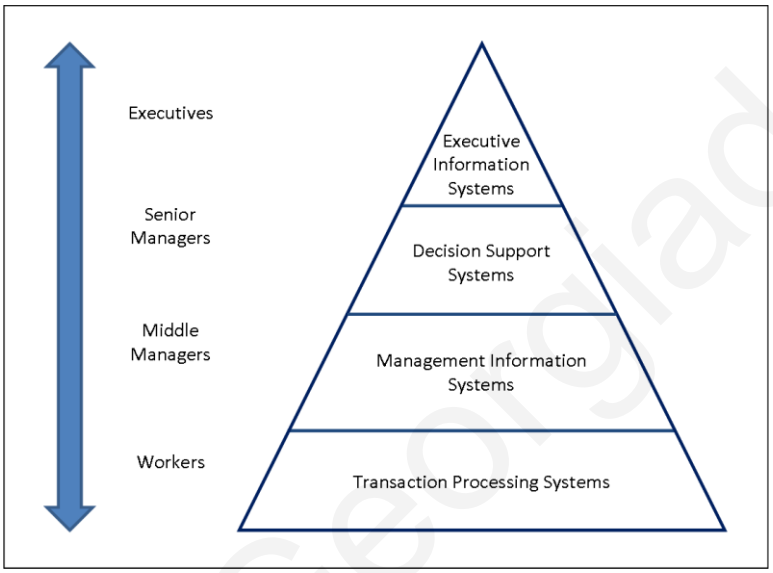


FIGURE 2.1 FOUR LEVEL PYRAMID MODEL BASED ON THE DIFFERENT LEVELS OF HIERARCHY IN THE ORGANIZATION (SOURCE: WIKIPEDIA, 2010)

Regardless the type of the application and the people involved, there are a number of generic activities common to all RE process models, which are: requirements elicitation, requirements analysis, requirements specification, requirements validation and requirements management. The proposed methodology focuses on the first three activities. According to Westfall (2006), eliciting, analyzing, and writing good requirements are the most difficult and important parts of software engineering. Of course, the RE models that utilize these activities, the techniques applied during these activities, the types of requirements handled by these activities, the people involved, and tools that automate the

application of the RE activities vary according to the type of application. This dissertation proposes a methodology that is intended to formalize the requirements elicitation, analysis and specification activities for the development of information systems, and the aforementioned RE elements need to be taken into account for developing a solid RE methodology.

## *2.3 Requirements*

A number of problems that occur during the requirements engineering process result from failing to make a clear distinction between the different levels and types of requirements (Sommerville, 2005). By recognizing these different levels and types of requirements, requirements engineers gain a better understanding of what information they need to elicit, analyze, specify, and validate when they define their software requirements. Sommerville focuses on two levels of requirements, the user-level and system-level requirements, while Wiegers (2004) distinguishes an additional more abstract level, which is the business level.

Wiegers (2004) states that business requirements define the business problems to be solved or the business opportunities to be addressed by the software product. In general, the business requirements define why the software product is being developed. Business requirements are typically stated in terms of the objectives of the customer or organization requesting the development of the software.

User-level requirements look at the functionality of the software product from the user's perspective. They define what the software has to do in order for the users to fulfill

their objectives. Multiple user level requirements may be needed in order to fulfill a single business requirement. For example, the business requirement to allow recording appointments might translate to multiple user requirements including:

- Customer makes a call
- Receptionist records date and time of the appointment

System requirements set out the system's functions, data, constraints and non-functional requirements with specificity and detail. The system requirements document (sometimes called a functional specification) should be precise. It should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers. Software system requirements are often classified as functional requirements, nonfunctional requirements or domain requirements:

1. The system's functional requirements that define the software functionality must be built into the product to enable users to accomplish their tasks, thereby satisfying the business requirements. Multiple functional level requirements may be needed to fulfill a user requirement. For example, the user-level requirement that the users can record time and date of the appointment might translate into multiple functional software requirements, such as:
   - "Receptionist selects to see appointment form on her computer screen
   - System shows appointment form
   - Receptionist enters customer's ID
   - System checks if customer is already in the database
   - System returns relevant message

- If the customer does not exist in the database, Receptionist selects the New Customer form, otherwise…"

2. Non-functional requirements. These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.
3. Domain requirements. These are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain. They may be functional or non-functional requirements (Sommerville, 2010). Business rules are a kind of domain requirement; they are the specific policies, standards, practices, regulations and guidelines that define how the users do business (and are therefore considered user-level requirements). The software product must adhere to these rules in order to function appropriately within the user's domain. Therefore business rules need to be formalized too.

The focus of the NLSSRE methodology is on system-level requirements, that is, we are interested in what the system will do to fulfill the users' requirements. As we will see in the next chapter, the formalization concept is more easily applicable to the system-level functions, because they are applied on electronic information. The formalization concept is not directly applicable to the user-level functions, due to the complexity and ambiguity of terminology as well as the complexity in size, which exists in the business environment. In particular, the methodology builds and formalizes the system requirements with the use of the following Information System elements: people (usually end-users, clients and trusted external users), processes related to the creation,

modification, transmission, storage and presentation of information along with the circumstances within which these processes are performed, as well as data, constraints and business rules.

## *2.4 Requirements Engineering Process Models*

A model is an abstract representation of how the activities of the RE process are put together. Macaulay (1996) finds that "no model is suitable for all situations" and recognizes that many issues related to the personnel and even the structure of the organization need to be considered. Additionally, the size and scope of the project both affect which model is best suited. Four widely-known RE process models with different structures and used in information system development are: linear, linear with iterations between activities, iterative and cyclical.

A linear and incremental model is often used to describe the RE process, with RE activities (e.g., elicitation, analysis) following linear transitions. As an addition to this common linear process, Kotonya and Sommerville (1998) propose a "conceptual linear RE process model, which indicates iterations between activities" (Figure 2.2). They show that the RE process steps overlap and are often performed iteratively. Such a model is nearer to the way our methodology could be applied.

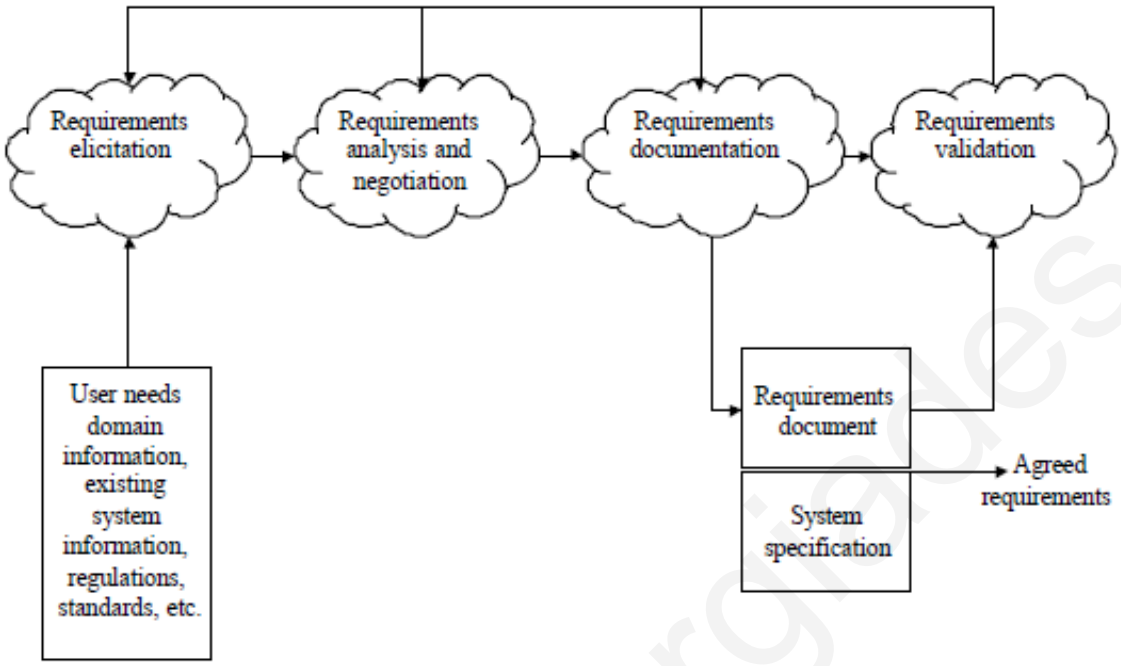


FIGURE 2.2 KOTONYA AND SOMMERVILLE (1998) LINEAR REQUIREMENTS ENGINEERING PROCESS MODEL

Macaulay (1996) provides a purely linear RE process model (Figure 2.3). It does not indicate the overlapping or iterations of activities, suggested by the Kotonya and Sommerville (1998) model. The RE activities are categorized under different headings, however the linear progression resulting in documentation is common to both models. Macaulay (1996) acknowledges that the RE process is situation dependent and discusses seven different customer-supplier relationships and their corresponding RE processes.

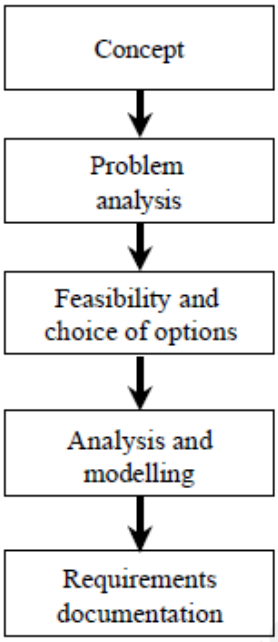


FIGURE 2.3 MACAULAY (1996) LINEAR REQUIREMENTS ENGINEERING PROCESS MODEL

While much of the literature on the RE is based on linear models, non-linear models have also been proposed. The model proposed by Loucopoulos and Karakostas (1995) shows an iterative and cyclical RE process (Figure 2.4). This model shows how the elicitation, validation and specification stages interact with both the user(s) and the IS itself, with each aspect influencing the others through iterative developments. We see that the stages and RE activities in Loucopoulos and Karakostas have similar properties to the linear models, but because the order in which these activities occur is non-linear, what happens in one stage can have a direct effect on the next stage as the cycle moves forward or on a previous stage as the iterative cycle repeats.

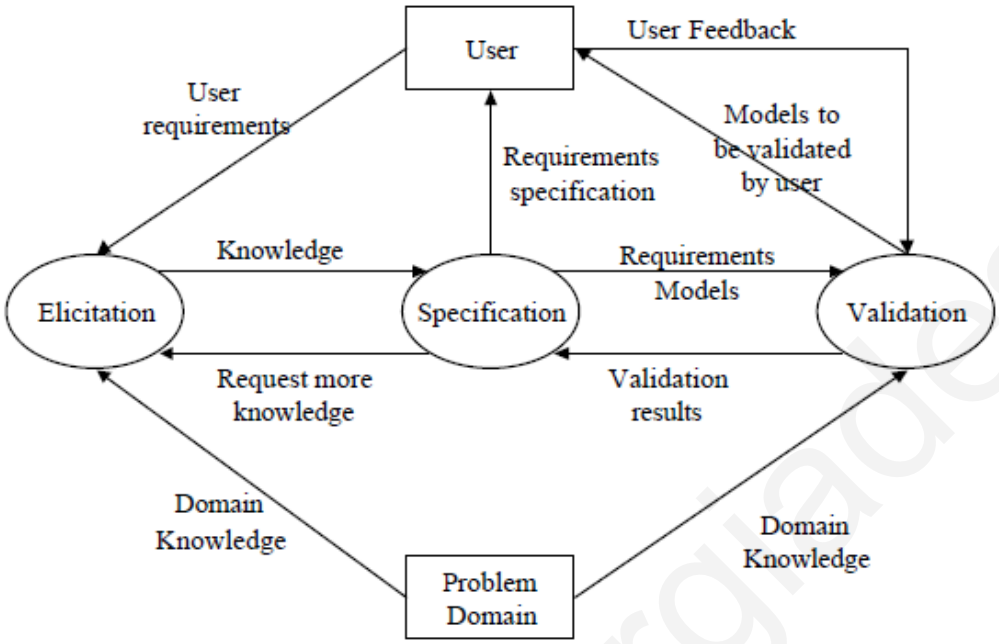


FIGURE 2.4 LOUCOPOULOS AND KARAKOSTAS (1995) ITERATIVE REQUIREMENTS ENGINEERING PROCESS MODEL

The spiral model (fig. 2.5) proposed by Sommerville (2010) is another non-linear model. Sommerville divides the RE process into three major categories of requirements elicitation, requirements specification, and requirements validation. The model views the process as a spiral, passing several times through each category starting from early stages of business specification through user specifications and system requirements specification.

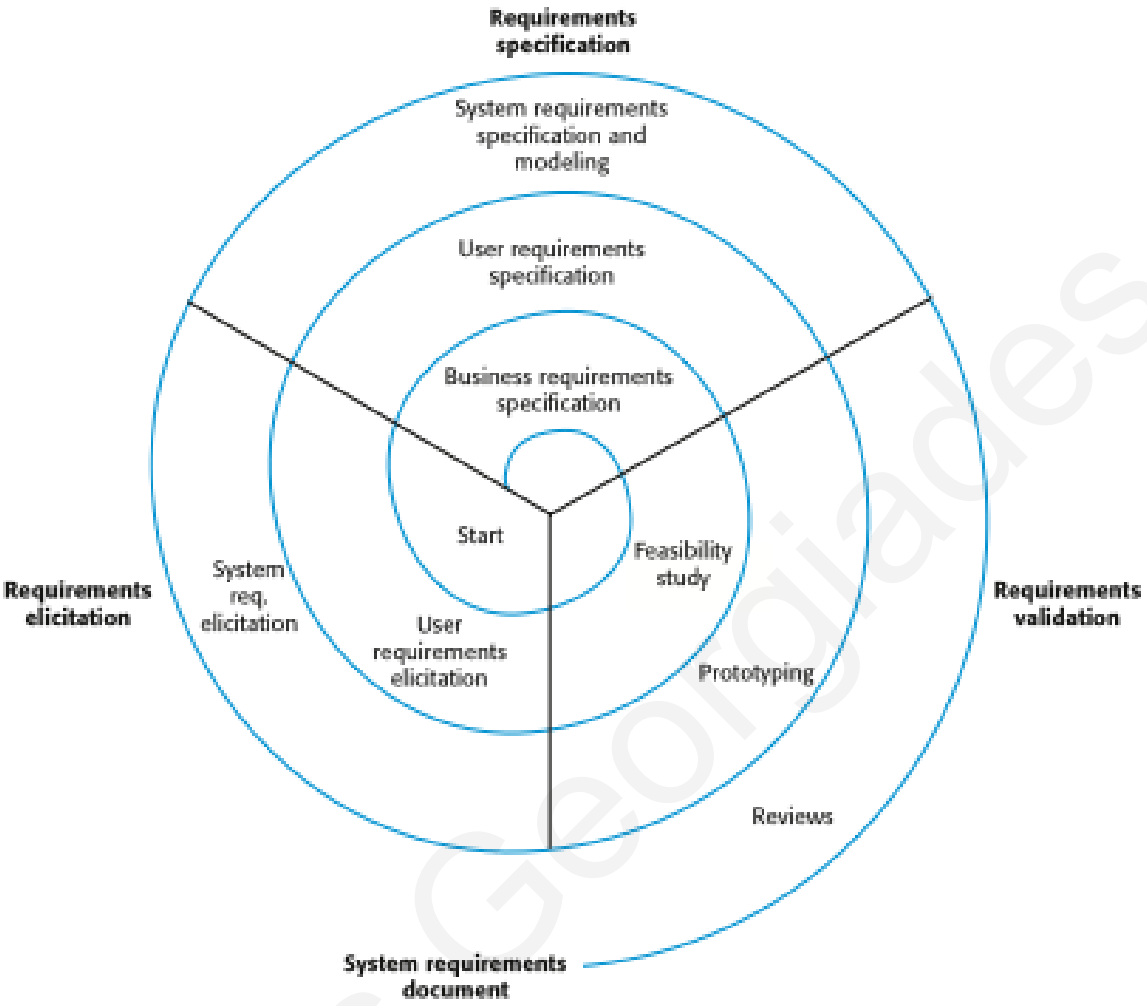


FIGURE 2.5 THE SPIRAL MODEL OF THE RE PROCESS

The spiral model allows for varying levels of detail at each requirements stage, and also shows how the spiral need not be traversed all the way to the end; the spiral can be exited as soon as the required level of specification is completed. For example, if the project needed only user requirements, then the later system requirements stages need not be completed.

Some researchers consider the application of structured analysis methods such as object-oriented analysis to the RE problem (Larman, 2002). An object-oriented approach

often involves graphical system models and use-case models which later serve as system specifications. Clearly, the success of object-oriented methods has shown that such methods are useful to RE, but such methods do not encompass all of RE. In particular, human-centred nature of the elicitation phase makes it hard to apply the often stricter structure of an object-oriented approach.

Unfortunately, in spite of several linear and non-linear models describing the RE process, studies show that such models in the literature do not reflect the current state of RE process in practice. For example, Nguyen and Swatmann (2000) found that the RE process in their case study did not occur in a systematic, smooth and incremental way, but was opportunistic, with sporadic simplification and restructuring of the requirements model when it reached points of high complexity. Hofmann and Lehner (2001) examined the 15 RE processes in industry and found that most participants saw RE as an ad hoc process, with only some using an explicitly defined RE process or customizing a company standard RE process.

## *2.5 The keystone of NLSSRE*

We believe that one of the main reasons existing RE approaches and models are not sufficient is that they focus on the way the RE process activities are interrelated and organized only during their application. The difference between the existing approaches and the proposed methodology does not lie on the way they are applied, but on the way they are built. Specifically, in the existing approaches, we find no direct link at an architectural level between the analysis and elicitation or between specification and

elicitation. The lack of a direct connection between elicitation and the later stages of RE means the way elicitation is built is not actually related with the way analysis is built. If a methodology were designed such that both elicitation and analysis were built on the same language and framework then the elicitation data collected from the users would feed easily into the analysis and specification activities, without any need for clarifying ambiguities, redundancies and searching for missing requirements as occurs with current approaches. Since the problem starts from the elicitation stage where analysts cannot clearly discover user needs, a different approach is necessary that will provide this underpinning link between analysis, specification and elicitation. The most widely used approach for RE, which involves the use of NL, is the Use Case-driven Analysis (UCDA), which is mainly used for object oriented analysis and design (Dias et al., 2008). Goal-oriented requirements engineering (Saeki, 2010; Lamsweerde, 2001) is another approach that conceives requirements as goals, and it can use NL to define them, as well as questions to refine goals and sub-goals. The conceptual models of both approaches, as well as relevant techniques used will be discussed in the following sections, as well as a detailed discussion about use case-driven analysis which is also used for the comparative evaluation of the proposed methodology.

Figure 2.6 illustrates the main concept of the methodology's architecture, which starts with the principle that if the analysts know, in advance, specifically what types of functions, data, business rules and conditions (RA) they should search for and write down, then they will be able to ask specific questions (RD) regarding that particular information (during the application of NLSSRE, the answers to these questions will facilitate the progress of the RA and RS processes). Additionally, the way requirements are written

(RS) is based on utilizing RA elements and specific formalized sentential patterns. Such patterns also contribute to the process of developing the RD questions, as well as the identification of the RA elements, as we will see in detail in chapter 3. We will also show that the way the methodology is built together with the formalization provided for the IS elements allows the methodology to be complete and self-verified.

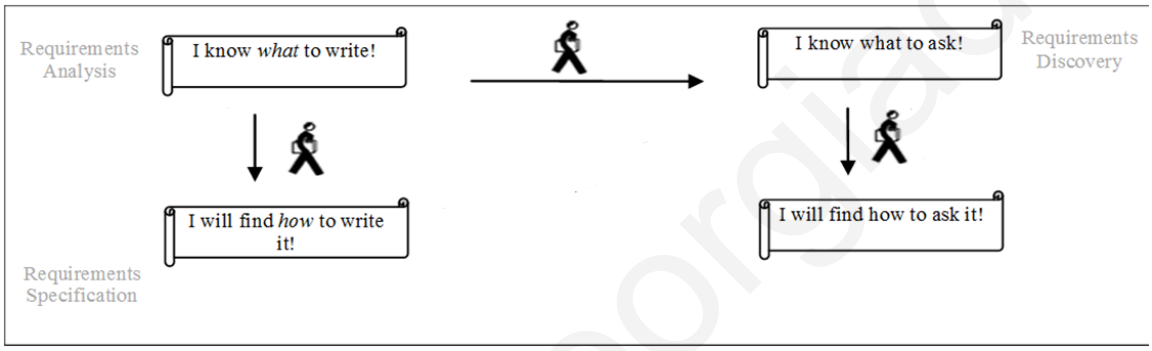


FIGURE 2.6 GENERAL OVERVIEW OF THE METHODOLOGY'S ARCHITECTURE.

Below we describe the three major activities of the RE process – elicitation, analysis and specification – and the most widely used methods and techniques with particular focus on the use of natural language, as well as common problems. Finally we make a comparative reference to how our approach intends to provide solutions.

## *2.6 Requirements Elicitation*

Requirements elicitation is the stage where the software or systems engineer discovers and collects the requirements. Elicitation is most commonly a human-centered activity (i.e., gathering requirements information from users and other stakeholders). Typically,

elicitation is the first stage of developing a software system, where the development team and customer begin building an understanding of the problem the software will be designed to solve and also begin building a relationship for solving that problem together. Elicitation is also known as "requirements capture," "requirements discovery," and "requirements acquisition." (Pfleeger, 2001)

Studies have observed that that the majority of people involved in software requirements elicitation prefer to use free, common natural language (NL), as the means to discover requirements (Mich et al., 2004; Gervasi and Zowghi, 2005). Natural language is more understandable to both users and analysts, on the one hand, and on the other, it is easier to move from one type of natural language (informal – during elicitation) to another (formal – during analysis and specification). In this section, we examine other approaches used in RD, especially those which use natural language, either formalized or free, as a comparison with our methodology.

The first step in Requirements Discovery involves identifying the requirements sources which mainly are the stakeholders and secondly documentation and existing or legacy systems. The stakeholders include people such as users, customers, suppliers and decision-makers, and also include the environment where the system will be used (Zhang, 2007). Additionally, the roles of the people who participate in the requirements process need to be taken into account. Step 1 of the NLSSRE methodology defines in detail the people involved in the RE process for the development of information systems.

After identifying the stakeholders, the analyst can begin the elicitation (RD). The principle difficulty of RD is that the human stakeholders find it difficult to accurately express their requirements and in fact may have difficulty describing their own tasks.

People often forget to state obvious but nevertheless extremely important information, or they may be simply unwilling to cooperate. The analyst must be sensitive to the human aspect of the process. Because elicitation is not a passive activity, the analyst must recognize that even when the stakeholders are cooperative and articulate, the analyst still must work hard to capture the correct information. The principal techniques for obtaining this RD information are listed below (Zhang, 2007):

- Interviews are the most commonly used method of eliciting requirements because communication in free natural language is an easy way to express needs and ideas, and ask and answer questions. Usually interviews use open-ended questions and are used to elicit non-tacit requirements. Interview is handy and commonly used throughout the requirements development process. In nearly every case, an interview can be used when the stakeholder is a person. Unstructured interviews, mostly with the use of open-ended questions, are conversational in nature where the interviewer enforces only limited control over the direction of discussions. Because they do not follow a predetermined agenda or list of questions, there is the risk that some topics may be completely neglected. It is also a common problem with unstructured interviews to focus in too much detail on some areas, and not enough in others (Zowghi and Coolin, 2005). Unstructured interviews and open-ended questions are best applied for exploration when there is a limited understanding of the domain, or as a start for more focused and detailed structured interviews which are a tactic followed by our approach, with firstly the use of a general description of each user's everyday work (accompanied with some other techniques described later) and subsequently with the use of **specific questions** the answers of which are guided by the methodology's predefined artifacts. Structured

interviews are conducted using a predetermined set of questions, mainly closed-ended in nature, to gather specific information. The success of structured interviews and closed-ended questions depends on knowing what are the right questions to ask, when should they be asked, and who should answer them. This is a basic principle of our methodology. Templates such as IEEE Std 830 Software Requirements Specification (IEEE, 1998) and Volere Requirements Specification Template (Robertson, 2001) represent the most basic method used by analysts to support the process of requirements elicitation. In addition to them, Da Silva and Leite (2006) and Leite and Gilvaz (1996) developed an interview-driven requirements elicitation support system, based on the idea of having a general interview assistant. The system's knowledge database is developed based on Business System Planning, Critical Success Factors, and End Means Analysis. Twenty two kinds of questions are automatically generated. These question sentences consist of a fixed part and a variable part. The variable part is generated by incorporating answers already obtained from other questions, and a chain of questions is established by generating the variable part. In addition, a heuristic is triggered at the end of the interview, or for specific questions during the interview, so that a question is presented. When a user answers it, the accuracy of the answer, the relationship between two answers, and the need for further questions are checked with the use of the aforementioned heuristic. The objective of this work is to automate the interview-driven requirements elicitation process and check the accuracy of requirements specification. PREview (Sommerville et al., 1997) is another approach that refers to a model of viewpoints that is intended to be used to organize system requirements derived from radically different sources. It uses the notion of 'concerns', which are key business drivers of the requirements elicitation process, and

they may be used to elicit and validate system requirements. They are decomposed into questions which must be answered by system stakeholders.

- **Scenarios** are narrative and specific descriptions of current and future processes including actions and interactions between the users and the system. They are a valuable means for providing context to the elicitation of user requirements. Scenarios allow the software engineer to provide a framework for questions about user tasks by permitting "what if" and "how is this done" (open-ended) questions to be asked. The most common type of scenario is the use case scenario which is mainly used in object oriented analysis. However scenarios focus on user requirements, not system requirements. A substantial amount of work from both the research and practice communities has been dedicated to developing structured and rigorous approaches to requirements elicitation using scenarios including CREWS (CREWS) and Scenario Plus (Scenario Plus). Scenarios are additionally very useful for understanding and validating requirements, as well as test case development.

- **Observation** is a well-used method where the software engineer or engineers learn about the user requirements by observing how the stakeholders use their current software and how they interact with other stakeholders. On the one hand, observation can reveal tasks and business processes that are too subtle (too obvious) or too complex for the stakeholders to describe accurately, but on the other hand such direct observation techniques are expensive, both in terms of the longer time it may take to observe (compared to a direct interview) and in manpower cost. Tight system development deadlines often preclude the lengthy observations needed for good results. Social analysis and ethnography are examples of observational methods.

- **Analytic Methods**. Analytic methods provide ways to explore the existing documentation or knowledge and acquire requirements from a series of deductions. A variety of documentation may shed light on requirements of the desired product. It includes problem analysis, organizational charts, standards, user manuals of existing systems, survey report of competitive systems in market, and so on. By studying it, engineers capture the information about the application domain, the workflow, the product features, and map it to the requirements specification. Also, they identify and reuse requirements from the specification of the legacy or similar products. It is always worth probing and rummaging for reports and recorded information relevant to the desired product.

The deduced information from experts' knowledge and experience form another source of requirements in analytic methods. Requirements can be dug up from domain experts' knowledge. Repertory grid is a technique that provides ways to elicit attributes that are not immediately and easily articulated by the expert. In general, analytic methods are not vital to requirements elicitation, since requirements are captured indirectly from other sources, rather than end users and customers. However, they form complementary specifications to improve the efficiency and effectiveness of requirements elicitation, especially when the information from legacy or related products is reusable. Analytic methods provide effective support of requirements elicitation in application domains where the domain related documentation and experts are available.

- **Goals.** A technique for finding out sub-goals and requirements is to keep asking HOW questions about the goals already identified (Lamsweerde, 2000b). Formal goal refinement patterns may also prove effective when goal specifications are formalized;

typically, they help finding out subgoals that were overlooked but are needed to establish the parent goal. To find out more abstract, parent goals is recommended to keep asking WHY questions about operational descriptions already available. More sophisticated techniques have been devised to elicit goals from scenarios. Based on a bidirectional coupling between type-level scenarios and goal verb templates, Rolland et al. propose heuristic rules to find out alternative goals covering a scenario (corresponding to alternative values for the verb parameters), missing companion goals, or subgoals of the goal under consideration. The Requirements Elicitation Guide (REG – Fuentes et. al, 2005) is another method to analyze the key intentional and social features of an information system and its context. Based on them, REG contains *questions* which represent the information that activity theory (which has been developed in the fields of sociology and psychology) considers important to elicit about activities. The answers to REG *questions* are the requirements of the software system and they are provided by customers and developers. For example, for the activity theory aspect *Goals of the new component* and the question *Is there any inconvenience for the organization or groups in it about building the new component?*, REG replaces the aspect with the more specific goal *Give Support to Customers*. This method, like most of the goal oriented elicitation methods, is mostly applicable to non-functional requirements, or agent-based systems, or in deriving more general information from customers.

**Natural language processing** (NLP) techniques deal with requirements retrieval from pre-existing requirements documents, by using either rule-based (Goldin and Berry 1997; Rolland and Proix 1992; Li et al. 2005; Gervasi and Zowghi, 2005) or probabilistic techniques (Rayson et al. 2000; Sawyer et al. 2002). In this category also fall a few

approaches (Tjong et al. 2006; Videira and da Silva 2005; Videira et al. 2006) which suggest that users should write a paragraph describing their job tasks in free text on which similar retrieval rules to elicit the requirements are applied.

In Goldin and Berry's work (1997), an approach and a prototype tool are presented, for suggesting requirement abstractions to the human elicitor. Their method compares sentences using a sliding window approach on a character-by-character basis and extracts matching fragments that are above a certain threshold in length. The approach tries to handle arbitrary lengths, gaps and permutations and avoids some specific weaknesses in confidence and precision when using only parsers or counting isolated words. Rayson et al. present two experiments in probabilistic NLP using tools they have developed (part-of-speech and semantic taggers integrated into an end user tool). The results suggest that the tools are effective in helping to identify and analyze domain abstractions.

In summary, although existing methods and techniques provide a good guide on their use in RD, still the major problem in RE is that there is difficulty in understanding user needs. On one end of the spectrum there is the classical, most commonly used approach based on open-ended questions which usually produce vague answers written in free, informal natural language, with inconsistencies and redundancies, because neither the analyst nor the user know exactly in advance what to ask or answer regarding specific functions and data of the IS. On the other end we have requirements retrieval from pre-existing requirements documents. However, the retrieval approach is not particularly reliable, since requirements are often not written syntactically, grammatically and semantically correctly from the beginning, and the rules applied to retrieve them cannot work well to produce reliable and complete results; additionally, there is a good chance

that the original texts do not cover all the requirements of the IS under development and also that they include ambiguities, redundancies and disorganized material.

In contrast, our approach differs from the aforementioned ones, since it guides the analyst how to define specific sets of questions from predefined patterns of functions and specific types of data, business rules and functional conditions. The answers to these questions feed and complete the analysis and specification stages. Therefore, the way we discover the requirements is clearly connected to the analysis and specification of requirements. In the current literature—to the best of our knowledge—this link does not exist, and therefore the resulting requirements documents produced from current approaches need to be re-organized, re-validated and re-adjusted. Additionally, NLSSRE uses documentation (expert) techniques, as complementary to interviews, mainly to identify data attributes.

## *2.7 Requirements Analysis*

Requirements Analysis (RA) is concerned with taking the unstructured collection of requirements, and, based on different conceptual models and classifications, it groups related requirements and organizes them into coherent clusters. RA also detects and resolves conflicts between requirements.

Existing approaches classify requirements based on a number of dimensions such as whether the requirement is functional or non-functional, or whether one requirement is more essential than another by giving different levels of priority to requirements, etc. Classifications are also influenced by the conceptual model used. The development of

conceptual models of a real-world problem is key to software requirements analysis. They usually comprise the core of developing a methodology and aim to aid in understanding and decompose the problem (a very common term used is the 'separation of concerns'), rather than to initiate design of the solution. Several kinds of models can be developed. These include data and control flows, state models, event traces, user interactions, object models, data models, and many others (Sommerville, 2010):

- Activity Oriented Meta-Models. Activity oriented meta-models allow modeling a system as a set of activities related by data or by execution dependencies. These meta-models are well suited to model systems where data are affected by a sequence of transformations at a constant rate. Data flow diagrams (DFDs) and flowcharts are two examples of activity oriented meta-models .

- Structure Oriented Meta-Models. Structure oriented meta-models allow the description of system physical modules and their interconnections.

- Data Oriented Meta-Models. Data oriented meta-models allow modeling a system as a collection of data related by some kind of attribute.

- Multiple-View Approach. Multiple view modeling, as defined by several authors (Sommerville et al., 1997, Leite et al., 1996) can adopt orthogonal views: (1) the function view is responsible for representing the processes of the system and UML's activity diagrams can be used to support this view; (2) the data view defines system information, that can be supported by UML's class diagrams; (3) the control view characterizes the system dynamic behavior that can be described by UML's state diagrams. Object Oriented modeling is based on the multiple view concept, and Use Case Driven Analysis

(UCDA), which is discussed in 2.9) is the most widely used structured approach for eliciting, analyzing and specifying the requirements for the development of the OO model.

- State Oriented Models. State oriented meta-models allow modeling a system as a set of states and a set of transitions. The transitions between states evolve according to some external stimulus. These meta-models are adequate to model systems in which temporal behavior is the most important aspect to be captured. Finite state machines (FSMs), finite state machines with data paths (FSMDs), StateCharts and Petri nets are examples of state oriented meta-models.

- Goal-driven analysis for clarifying requirement. Requirements are often unclear when first elicited from clients and stakeholders. The introduction of goals offers one way of clarifying requirements (van Lamsweerde, 2000b). Analyzing requirements in terms of goal decomposition and refinement can be seen as teasing out many levels of requirements statements, each level addressing the demands of the next level. This approach to the clarification of requirements is especially appropriate in the case of non-functional requirements (such as flexibility, robustness, reusability, maintainability), where initial requirements can be difficult to make precise. A goal-oriented approach would allow the requirements to be refined and clarified through an incremental process. Chung et al.'s Non-Functional Requirements framework (1999) is a goal- and process-oriented approach for dealing with non-functional requirements.

Despite the different conceptual analysis models, analysts usually follow the easy, general structure of an (IEEE) SRS template or a use case specification template, where they try to organize the discovered requirements according to the general guides and structure provided in such templates. Since such templates do not provide specific types

of functions and data, or any specific methods to form business rules, the RA process lacks significant specificity. In other approaches, where the main focus is on the use of NL formalism in IS, Videira and da Silva (2005), Videira et al. (2006) and Rolland and Proix (1992) mention a few specific types of actions based on verbal types, but they do not expand on them or match them adequately with IS elements. Also these authors do not provide categorization of data and functional conditions and how they are connected to functions.

Our approach differs from the aforementioned approaches by providing predefined types of functions and specific categories of data and functional conditions to guide the analyst to analyze adequately the functions, data and functional conditions of an IS. Our approach also provides heuristics to help the analyst define the business rules derived from different combinations among data in relation to particular functions. The functions and data are also grouped and decomposed based on predefined patterns. The proposed approach is facilitated by the use of NL semantics of verbs, genitive case, adjectives and adverbials.

We will see in chapter 4 that NLSSRE conceptualizes requirements under the notion of information object and, based on it, can develop a use-case model, and object model or a process model.

## *2.8 Requirements Specification*

Requirements Specification is the activity of translating the information organized during the analysis activity into a document that defines a set of requirements. The software requirements specification establishes the basis for agreement between customers and

contractors or suppliers on what the software product is to do. Two types of requirements may be included in this document (or in two separate documents). User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.

The user requirements document should be written in natural language because it must be understood by people who are not technical experts. However, the system requirements should be expressed in a more technical way, so as to be more precise. Nevertheless, it is evident that natural language is often used to write system requirements specifications as well as user requirements. Specifically, according to a survey (Mich et al., 2004), 95% of the requirements documents found in industrial practice are written in common (79%) or structured (16%) natural language. Similar findings have been reported in an independent survey (Neill and Laplante, 2003), showing that only 7% of the respondents used some kind of formal language to express requirements. NL is also the only language that can be assumed to be common to all the stakeholders. Its use encourages expression and experimentation, which are of paramount importance in the early stages of the evolution of a specification. The traditional vehicle of writing system requirements is the IEEE software requirements specification template (fig. 2.7). The IEEE template is a generic template, with general guidelines to define different types of requirements through the use of informal natural language. The use case specification template is also another way of writing system requirements in free natural language. However, as already mentioned, natural language is inherently ambiguous and results in poorly defined requirements. Different formal methods, such as

structured English (fig. 2.8) or even formal mathematical specifications are much less preferred and much less frequently used. Writing and analyzing a formal specification requires high expertise (van Lamsweerde, 2000a). Often, such expertise is not readily available, and this contributes to the limited use of formal methods in industrial context. Moreover, even when an expert in formal methods is available, the stakeholders cannot be expected to be or become experts themselves. Some translation between formal and informal languages is thus needed, in order to transfer system requirements from a formal version (e.g., structured English, use case specifications) to another (informal) for user requirements. This translation itself introduces in the process a new source of potential errors and delays that may actually make matters worse than they were in the first place.

**4. System Features**

*<This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product. You may prefer to organize this section by use case, mode of operation, user class, object class, functional hierarchy, or combinations of these, whatever makes the most logical sense for your product.>*

**4.1 System Feature 1**

*<Don't really say "System Feature 1." State the feature name in just a few words.>*

4.1.1 Description and Priority

*<Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>*

4.1.2 Stimulus/Response Sequences

*<List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>*

4.1.3 Functional Requirements

*<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate when necessary information is not yet available.>*

*<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>*

REQ-1:
REQ-2:

**4.2 System Feature 2 (and so on)**

FIGURE 2.7 HOW TO DEFINE FUNCTIONAL REQUIREMENTS IN THE IEEE SRS TEMPLATE (SOURCE: IEEE, 1998).

**Insulin Pump/Control Software/SRS/3.3.2**

| | |
|---|---|
| **Function** | Compute insulin dose: Safe sugar level |
| **Description** | Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units |
| **Inputs** | Current sugar reading (r2), the previous two readings (r0 and r1) |
| **Source** | Current sugar reading from sensor. Other readings from memory. |
| **Outputs** | CompDose—the dose in insulin to be delivered |
| **Destination** | Main control loop |

**Action:** CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

| | |
|---|---|
| **Requires** | Two previous readings so that the rate of change of sugar level can be computed. |
| **Pre-condition** | The insulin reservoir contains at least the maximum allowed single dose of insulin. |
| **Post-condition** | r0 is replaced by r1 then r1 is replaced by r2 |
| **Side effects** | None |

FIGURE 2.8 STRUCTURED ENGLISH (SOURCE: SOMMERVILLE, 2010).

As a solution to this problem, the research community has worked on methods based on semi-formal (or controlled) language requirements. We consider semi-formal natural language as the most appropriate combination of formality and understandability. Some approaches, such as Conger's (1994), use a basic syntax (<Subject> <Verb> <Object>) to specify requirements, while others use an additional syntactic element (<Subject> <Verb> <Object> <Complement>), such as in Rolland and Proix (1992). Our methodology goes a step further involving additional linguistic elements, such as the adverbial adjunct which

is related to functional conditions, or the genitive case and the adjective, which are used to assist in the identification and specification of business rules and data; it also provides different patterns for writing requirements, according to predefined types of functions (*Create, Alter, Read, Erase, Notify*) and the functional roles involved. Furthermore, in NLSSRE, contrary to other approaches, requirements can be grouped and specified under one comprehensive function or data object. There are also approaches, such as the one presented in Ben Achour (1998) that aim to reduce the level of imprecision in requirements by using a limited number of sentence patterns to specify requirements for a particular domain. Denger et al. (2002) have also identified natural language patterns used to specify functional requirements of embedded systems, from which they developed a metamodel for requirements statements.

## *2.9 Use case driven analysis*

Use case driven analysis (UCDA) has gained a wide acceptance among the many methods in requirements engineering (Dias et al., 2008), 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 (Pooley and Stevens, 1999). 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 (Kim et al., 2004). 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.

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) (Fliedl et al., 2002) 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. (2008) 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. (2004) 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. Comparing to the proposed methodology, all these approaches do not provide:

(i) 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.

(ii) 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

(iii) 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, as already mentioned in previous sections, lead also to answers and requirements documents with ambiguities and redundancies (Gervasi and D. Zowghi, 2005); 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 using UCDA for requirements specification, we see many drawbacks to describing a use case using informal natural language, as recommended by Jacobson (2004) and Booch et al. (2005). 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 important to use a structured analysis model or a formal technique for such a description. In the relevant literature, some structured techniques for the description of use cases have been proposed. In Eriksson et al.'s work (2004), a tabular representation is used, and in Leite et al.'s (1997), 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 (2007) 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 (Hsia et al., 1994) or statecharts (Glinz, 1995; Seybold et al, 2006)) 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.

## *2.10 Requirements Engineering CASE Tools -NALASS*

While it is hard to automate requirements written in free NL, due to NL's inherent complexity, vagueness and ambiguity, it is much easier and straightforward to automate requirements written in formalized NL. Therefore, we have developed a CASE tool that

automates a large part of the RE process by implementing the steps of the NLSSRE methodology. Current software tools, both in general and in the context of Natural Language Requirements Engineering (NLRE), are mainly limited to document parsers that can be used in various activities such as traceability, verification and prioritization of requirements, or even automated extraction of requirements from NL requirements documents. Abstfinder (Goldin and Berry, 1997) is based on the use of pattern matching techniques to extract abstractions (stakeholders, roles, tasks, domain objects, etc.) The frequency with which the abstractions occur within the text is taken as an indication of the abstractions' relevance. Fabbrini et al. (2001) and Lami et al. (2005) propose an automatic evaluation method called Quality Analyzer of Requirements Specification (QuARS) to evaluate quality in software requirements specification. This work developed a tool that parses sentential requirements written in Natural Language (NL) to detect potential sources of errors. COLOR-X [Burg 1997; Moreno 2001] and Circe (Ambriola and Gervasi 1997, 2006) parse a set of structured requirements in natural language to generate specific models (ER, DFD, OO design, etc.) The common characteristic of these and other related parsing tools is that they are mostly used and applied to pre-existing documents with disorganized text, redundancies and ambiguities. As a result, the retrieval approach is not particularly reliable, as explained earlier in this section. Other tools, such as the one reported by Kassel and Malloy [2003], are not parsers and offer the user the capability to enter the requirements from scratch, but they also lack specific types of questions (for RD) linked to the identification of data, functions and business rules of an IS. More generic tools such as Rational Rose (IBM Rational Rose) and MagicDraw (MagicDraw) provide significant capabilities for drawing diagrams and even generate

code from software models but not adequate facilities for generating textual specifications – hence, the analysts need to write their project's SRS using regular text editors and templates, such as Microsoft Word or LaTex, and the IEEE SRS template (1998), respectively. There are also some tools that can produce Use Case descriptions and scenarios, such as DOORS (DOORS) and STORM (Dascalu et al., 2007). These tools do not generate sufficient plain or semi-formal natural language descriptions, and they are only applicable to Use Case modeling. Their input also has to be processed first by the analyst (the analyst has to create the use cases). In a similar way requirements management tools like CaliberRM (Borland Software Corporation) and RequisitPro (Rational Software Corporation) provide format based support for the elicitation of requirements. Many analysts also utilize specific modeling tools to assist the process of requirements elicitation. These typically have an easy to use graphical or tabular notation. A number of tools have been developed to support specific requirements elicitation approaches, however, so far the mainstream software engineering community has largely not adopted these. Examples include Objectiver (Respect_IT) for goal based modeling and ART-SCENE for scenario elicitation (Maiden, 2004).

In contrast, our NAtural LAnguage Syntax and Semantics (NALASS) tool implements the NLSSRE methodology and provides specific predefined requirement patterns, specific categories of data, functional conditions and business rules, from which a specific set of questions is automatically derived. The answers to these questions feed the analysis and specification stages. Additionally, NALASS may be conceived as a complete toolset that can generate DFDs, Class Diagrams, Use Case specifications and

diagrams, as well as a well-structured NL-SRS document that covers the essential parts outlined by the IEEE SRS template (IEEE 1998).

## *2.11 Summary and Proposed Methodology*

In this chapter we examined requirements engineering and its critical importance as the first step of the software process. We noted that the RE process varies considerably depending on its context type and the application being developed. RE for information systems is significantly different from RE for embedded control systems, or from RE for generic software services such as networking and operating systems. We also examined a number of RE process models which provided different perspectives on the application of RE, and we elaborated on different techniques used for elicitation, analysis and specification of requirements, most of which use natural language. We also showed how the NLSSRE methodology compares to the various RE elements. It is observed that the existing models and techniques do not provide a specific, easily understood formalization of the major parts of the stages of requirements discovery, analysis and specification, and these methodologies usually result in requirements documents with ambiguities, redundancies and inconsistencies. In contrast, the proposed methodology intends to engineer the correct requirements in a clear-cut, time-saving, understandable and reliable way. The next chapter will show how NLSSRE formalizes and automates the discovery, analysis and specification of user requirements for the development of Information Systems.

# 3 The NLSSRE Methodology

The goal of NLSSRE is a clear-cut formalization and automation of the major activities of RE, including requirements discovery, analysis and specification. NLSSRE is designed so that the analyst is guided in advance, through a step-by-step approach, what specific types of data, functions, business rules and conditions to use and search for, what questions to ask, in what specific way to analyze the answers to the questions, and how to write them in a specific formalized way. In this chapter we explain the constituent parts of the methodology [2], which are its architecture (underpinning background), its application steps and techniques, a modeling language for representing requirements and the software tool that automates the entire process. The latter is also presented separately in chapter 5.

## *3.1 Architecture*

The structure and the formalization of the methodology draw from two elements. First, as illustrated earlier, if we know what kind of functions, data, functional conditions and business rules we look for (RA process), as well as what syntax we will use to specify them (RS), then we will be able to derive efficiently the relevant questions to ask (RD process). The answers to the questions will subsequently feed the processes of RA and RS. Second, the other element that facilitates formalization is the use of semantics and

[2]Chalmeta and Grangel [2008] define the constituent parts of a good methodology in a relevant domain.

syntax of NL. Figure 3.1 shows how the two aforementioned elements are used to formalize and build the methodology[3].

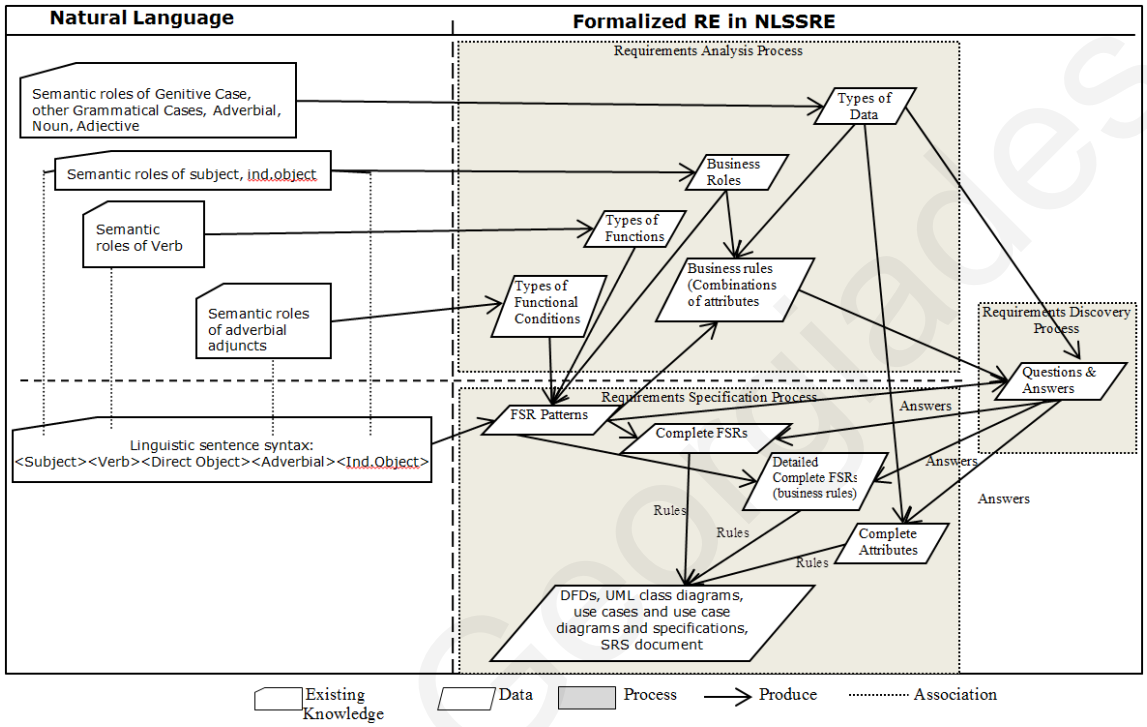


FIGURE 3.1 ARCHITECTURE OF THE NLSSRE METHODOLOGY, IN TERMS OF STRUCTURE AND FORMALIZATION.

For requirements analysis, data analysis is facilitated by the use of semantic types of the genitive case, other grammatical cases, nouns, adjectives and adverbials[4]; the types of functions are determined with the use of semantic types of the verb of the linguistic sentence; business roles are mainly linked to the use of semantic roles of subject and

[3]Additional elements which facilitate the RE process such as relevant principles, methods and techniques are incorporated into NLSSRE as illustrated in Table 3.1.

[4]Adverbials are also used for the prioritization of requirements, which constitutes a non-functional requirement. Non-functional requirements are not yet covered completely by the NLSSRE methodology.

indirect object of the sentence; analysis of functional conditions (also called functional adjuncts), which refer to the circumstances within which a function is performed, is facilitated by the use of adverbial adjuncts; and finally, business rules, which define or constrain some aspects of the business by describing the behavior/reaction of people and data through their relationships, are derived from relations (combinations) between attributes of data entities (business roles and data objects). For requirements specification, functions, data, functional conditions and business rules are written as formalized sentences (hereafter denoted as Formalized Sentential Requirements or FSRs), according to predefined patterns which are derived from the syntax of the linguistic sentence. FSR patterns, business rules and types of attributes are used to derive questions (RD process), the answers to which produce the complete FSRs (including also detailed FSRs which incorporate the business rules). Finally, specific transformation rules are utilized to process the complete FSRs and attributes to derive diagrammatic notations such as DFDs, UML class and use case diagrams, as well as use case specifications and the SRS document. All these are illustrated in detail in each step of the methodology in the next sections.

Before proceeding to the description of the application steps, it is necessary to introduce and describe the meaning of the Information Object (IO), which is a fundamental element of NLSSRE.

## *3.2 Information Object*

Our world consists of either tangible objects (those we can feel by using our five senses, e.g., book, chocolate) or intangible objects (e.g., examination, order).

*An Information Object (IO) is defined as 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 (CAREN) by the messages each instance (IOi[5]) can trigger.*

*Therefore <u>each IO is composed of the CAREN functions and a set of attributes</u>, which are discussed in steps 3 and 4 of the* methodology, respectively. NLSSRE also provides a number of categories of information objects: *business role* (as animate entity, e.g., doctor), *inanimate entity* (e.g., car), *procedure* (e.g., translation), *document* (e.g., book), *event* (e.g., appointment), *site* (e.g., country, hospital),and *state* (e.g., disease). These categories guide the analyst how to identify and manage the IOs, as will be described in steps 1 and 4 later on. Distinguishing IOs is a critical issue in requirements analysis and has not been examined yet adequately in the literature[6]. By making this distinction, we will be able to better organize the elements of the IS and their relationships. Additionally, for the identification and analysis of the IO attributes, NLSSRE provides a number of categories of attributes that must be linked to each category of IO, as will also be discussed in step 4.

[5] 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.

[6] It is out of the scope of this dissertation to discuss the existing literature on identifying objects. The general conclusion is that the few approaches [Bailin, 2002; Coad & Yourdon, 1990; Shlaer& Mellor, 1992,Song et al., 2005] dealing with this issue examine object identification by relying on concepts and principles at the OO programming level and not at the requirements analysis level. A good endeavor is provided by Iivari[1991], however, like the other approaches, it does not provide adequate guidelines for the identification of objects during requirements analysis.

IOs and their correct identification are fundamental concepts in our methodology, since, as already briefly illustrated and as we discuss later on, functions (CAREN), attributes (data), functional conditions and business rules of the system are grouped and formalized in relation to the IOs.

## *3.3 Methodology Steps*

The NLSSRE methodology is divided into a series of steps. Table 3.1 below presents the methodology application framework including the steps of the methodology, and the activities, means and techniques, tool support and expected results of each step.

TABLE 3.1 SUMMARY OF THE METHODOLOGY STEPS, ACTIVITIES, METHODS AND TECHNIQUES USED IN EACH STEP, THE EXPECTED RESULTS OF EACH STEP, AND TOOL SUPPORT.

<table>
<tr><th colspan="3">METHODOLOGY APPLICATION FRAMEWORK</th></tr>
<tr><td colspan="3">STEP 1. COLLECT THE CANDIDATE INFORMATION OBJECTS</td></tr>
<tr><td colspan="3">Activities<br>a. Identify roles and users<br>b. Collect information about items exchanged between users of each role<br>c. Categorize information/items according to the IO categories</td></tr>
<tr><td>Means and Techniques<br>· Summarized descriptions of each user’s work<br>· Data flow questionnaire<br>· Data flow table<br>· Document sampling, brief discussions<br>· IO Categories</td><td>Expected results<br>· List of candidate Information Objects</td><td>NALASS Tool support<br>√</td></tr>
<tr><td colspan="3">STEP 2. IDENTIFY THE INFORMATION OBJECTS</td></tr>
<tr><td colspan="3">Activities<br>For each Candidate Information Object:<br>a. Apply a set of given rules to determine if it is an Information Object<br>b. Categorize Information Objects according to the IO categories</td></tr>
<tr><td>Means and Techniques<br>· IO identification guide<br>· IO categories<br>· Questions</td><td>Expected results<br>· List of Information Objects</td><td>NALASS Tool support<br>√</td></tr>
<tr><td colspan="3">STEP 3. DEVELOP FSRs FOR EACH INFORMATION OBJECT</td></tr>
</table>

**Activities**

For each IO:

**a.** Apply CAREN functions and their sub-functions

**b.** Specify the IO, its CAREN functions, the involved functional roles and functional condition types, in the form of formalized sentential requirement (FSR) patterns

**c.** Make questions, derived from FSRs, to find the business roles and values for functional conditions

**d.** Specify complete FSRs, based on the answers received for each FSR pattern element

| Means and Techniques | Expected results | NALASS Tool support |
|---|---|---|
| · CAREN guide<br>· FSR patterns<br>· FSR questions | · 4 complete FSRs for each IO (incl. CAREN functions, business roles, functional conditions) | √ |

**STEP 4. DEFINE ATTRIBUTES FOR EACH INFORMATION OBJECT**

**Activities**

· Identify attributes for each IO, based on their category and the category of the IO

· Make specific questions to the user to confirm the attributes and derive new ones

| Means and Techniques | Expected results | NALASS Tool support |
|---|---|---|
| · Categories of attributes<br>· Categories of IOs<br>· Questions | · Attributes for each IO | √ |

**STEP 5. DEFINE BUSINESS RULES AND DEVELOP DETAILED FSRs**

**Activities**

For each FSR of an IO:

**a.** Make combinations with attributes of the involved business roles and of the IO

**b.** Make questions based on the abovementioned combinations

**c.** Specify business rules in the form of detailed FSRs

For each IO:

**a.** Make combinations between attributes of the IO per se

**b.** Make questions based on the abovementioned combinations

| Means and Techniques | Expected results | NALASS Tool support |
|---|---|---|
| · Detailed FSR patterns<br>· Questions | · A number of business rules for each FSR of each IO | Under development |

**STEP 6. CREATE SRS DOCUMENT AND SEMIFORMAL MODELS (DFDs CLASS & USE-CASE DIAGRAMS)**

**Description**

Use specific transformation rules to transform the IOs, complete FSRs, complete detailed FSRs and business rules, and attributes into an SRS document, Use Case model, Class diagrams, and Data flow diagrams.

| Means and Techniques | Expected results | NALASS Tool support |
|---|---|---|
| · SRS template<br>· Use Case (UC) Spec. template<br>· Transformation rules | · SRS document<br>· UC diagrams<br>· UC specifications<br>· Class diagrams<br>· Data flow diagrams | √ |

### 3.3.1 Collect the candidate Information Objects

The main aim of the first step is to collect the candidate IOs through a short (preferably in a single meeting) interaction with the users. The user group includes the following types: (i) *end-users* who use the system in an operational sense and interact directly with it; (ii) *business users* who are interested in the system's functions and output, as support for achieving their business objectives; (iii) *managers* who are responsible for the strategic use of IT in their business unit and for the overall strategy of the organization and the way information systems can both support and enable the strategy; (iv) *customers* who are external users that use the system to buy/utilize products and services, or search for information related to products; (v) *information users* who are external users that use the system not to buy anything but mainly to be informed or provide information about other system users or entities (a patient's relative in a hospital IS is an example of such a user) ; (vi) *trusted external users* who have a particular relationship with the organization and may be given specific privileges in the system (suppliers are examples of such users); and (vi) *Shareholders* who are external users that have invested in the organization and have financial interest (Avison and Fitzgerald, 2003). In NLSSRE all users need to be involved in the requirements engineering process. Specifically and in order of importance, end-users, customers, trusted external users and information users are involved in functional requirements, data requirements, constraints and business rules (steps 1–5), whereas business users, managers and shareholders are mostly involved in the business rules and quality attributes such as non-functional requirements, as well as in the approval of the SRS document (steps 5 and 6).

We have previously mentioned that IOs usually fall in the categories of business roles[7], physical entities, procedures and documents (also sites and states). In a manual or semi-automated IS, information describing entities which fall in the abovementioned IO categories—therefore such entities are candidate IOs—can be found in items exchanged between system users, such as documents (e.g., forms, reports, orders), queries/feedback (e.g., advice request), and physical items (e.g., book). Therefore, to collect the information about the exchanged items, we introduce and propose the use of the Data Flow Table (as shown in the Hospital Information System (HIS) example[8] of Table 3.2—for simplification the table shows only the candidate IOs, and the other related data has been removed), each cell of which includes the data, denoted by noun phrases, describing the items sent and received between any two users of the system. Each user has a business role in the system, and the analysts should receive information from a sufficient number of users of each business role. Therefore, we need to first identify the business roles and then utilize them to create and fill the data flow table; the identification of business roles is done by examining descriptions written down by the users about their everyday work, duties and responsibilities. As shown in the table, business roles in the leftmost column are the senders and business roles in the uppermost row are the receivers. The analyst must facilitate the process of eliciting the information describing the exchanged items, with the use of specific questions to the users, which contain verbs denoting the different forms of the transmission, such as *send to* (could respond to

[7] The role each user has in regard to the business context the user is involved.

[8] Through the dissertation, to support clearly our arguments, we provide examples taken from the application of our approach on a sample case study: the development of a hospital information system.

transmitting electronic or paper information), *give to* (physical/material data transmission), *say to* (vocal data transmission), and *show to* (visual data transmission). Table 3.3 shows the first part of the questionnaire, which concerns the transmission of documents (the other two parts concern the transmission of physical objects and queries – see Appendix A). As previously mentioned the information describing the exchanged items is denoted by noun phrases, because it concerns entities, either tangible or intangible. However, there are a few cases where verbs can also lead to candidate IOs. For example, the verb *order* derives the noun *order* which can finally become an intangible IO that denotes a procedure. Similarly, the verb *examine* in the sentence "*The doctor examines the patient"* can lead to the candidate IO *Examination*. Hence, in order to receive the richest information possible, beyond the use of the summarized everyday work descriptions and the data flow table, the analyst may also use complementary techniques, such as small discussions targeting the creation of a broad view of the users' work, as well as document sampling.

The list of candidate IOs for our indicative example of table 3.2 includes: the doctor, pharmacist and patient, as business roles; the drug as an inanimate object; the examination, treatment, diagnosis, and payment, as procedures; the patient record, insurance, x-ray, invoice, receipt, and prescription, as documents; the appointment as an event, and the blood as an animate entity.

TABLE 3.2 DATA FLOW DESCRIBING EXCHANGE OF ITEMS BETWEEN USERS. THE TABLE SHOWS INDICATIVE DATA COLLECTED DURING THE DEVELOPMENT OF THE HIS.

| Sender \ Receiver | | Doctor | | | Pharmacist | | | Patient | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $D_1$ | $D_2$ | … | $P_1$ | $P_2$ | … | $Pa_1$ | $Pa_2$ | … |
| Doctor | $D_1$ | | Knowledge request, Patient | | Drug stock check request | Drug stock check request | | Prescription, Diagnosis, Treatment | Examination, Filled Insurance Patient Record | |
| | $D_2$ | Knowledge feedback, Patient | | | Drug stock check request | | | Examination, Diagnosis, Treatment | Examination, X-ray, Appointment | |
| | … | | | | | | | | | |
| Pharmacist | $P_1$ | Drug stock check update | Drug stock check update | | | Knowledge request | | Invoice, Receipt, Drug | | |
| | $P_2$ | | Patient drug report | | Drug | | | | | |
| | … | | | | | | | | | |
| Patient | $Pa_1$ | Personal and medical info Patient record | | | Patient ID card, Prescription, Payment | | | | Blood | |
| | $Pa_2$ | Insurance | Patient Record, Examination | | | | | | | |
| | … | | | | | | | | | |

TABLE 3.3 A PORTION OF THE FIRST PART OF THE REQUIREMENTS DISCOVERY QUESTIONNAIRE ADDRESSED TO A REPRESENTATIVE NUMBER OF USERS OF EACH ROLE OF THE IS.

Part I. Documents

1. What documents, in electronic or paper form (e.g., forms, receipts, reports), which you create from scratch or change/complete after the recipient or someone else created them, do you send/ give/ show to $Role_rUser_{u,r}$?*
   a. in person
   b. through another person or service
   c. electronically (e.g., e-mail, internet)

   Auxiliary questions:
   - Do you give / send / show any documents to $Role_rUser_{u,r}$?
   - Do you write any documents for $Role_rUser_{u,r}$?
   - Do you sign any documents for $Role_rUser_{u,r}$?
2. What documents, in electronic or paper form (e.g. forms, receipts, reports), which you modify/ complete after the recipient or someone else created them, and/or asked you to modify/ complete them, do you send/ give/ show

to $Role_r User_{u,r}$?

a. in person
b. through another person or service
c. electronically (e.g. e-mail, internet)

Auxiliary questions:

- Do you modify any documents for $User_{1..n}$ with $Role_{1..m}$?

3. What data are or should be included about the document's
   a. Creator?
   b. Author?
   c. Purpose?
   d. Recipient(s)?
   e. Communication channel?
   f. Form?
   g. Other people (e.g. users) that should be notified about the creation or modification of the document?
   h. Procedure (s) mentioned in the document or related to the ones mentioned in the document?
4. What feedback (vocal or written) do you receive from the recipient, after you send the document to him / her?
   a. Does s/he make any change and send it back to you?
5. What initiates the sending procedure?
   a. A request (written or vocal) from the recipient?
   b. A request (written or vocal) from another user?
      i. What is the role of this user?
6. Can you provide a copy of each aforementioned document?

Note for the Requirements Engineer: The above set of questions should be addressed to the same user but from the perspective (role) of him/her being a receiver.

*$1 \leq r \leq p$where p is the number of roles, and $1 \leq u \leq {}_{n_{role_r}}$ where $n_{role_r}$ is the representative number of users of the Role r, excluding the user being asked.

### 3.3.2 **Identify the Information Objects of the new IS**

Within this step specific guiding rules are used to define the actual Information Objects of the new IS, from the list of candidate IOs collected during step 1. The guiding rules, such as the ones indicated below, are related to the definition of the IO, and so they use the notions of CAREN functions and the IO categories, and they need to answer the question "When do we need to create, alter and store this candidate IO?"

(1) When the tasks of *Creation*, *Alteration* (mainly), or *Transfer of Possession*[9] of an instance of this candidate IO cause the creation or change of instances of other IOs. Some examples follow:

(a) The creation of a new *Patient* IO instance in the IS causes the creation of instances of other IOs, such as a *Prescription* IOi (IO instance) and a *Payment* IOi. Therefore *Patient* is an IO.

(b) For a *Patient* IOi, the alteration of the value of its attribute *temperature* causes the creation of instances of other IOs, such as an *Examination* IOi, a *Diagnosis* IOi, and a *Prescription* IOi. Therefore *Patient* is an IO.

(c) In a *Bookstore* IS, the action of *transfer of possession* of a *book* (it actually corresponds to the sale of a book, and to the alteration of the attribute *Status* of a *Book* IOi, from *in stock* to *sold*), from the *Bookstore* (sender) to the *Client* (receiver), causes the creation of instances of other IOs, such as a *Payment* IOi, a *Delivery* IOi, and a *Book Order* IOi from the supplier. Thus *Book* is an IO.

(2) When the Candidate IO has the role of *Creator* / *Modifier* / *Sender* or *Receiver* in a task of *Creation*, *Alteration* or *Transfer of Possession*[10], respectively, which (the task) causes change (including creation) of the same IO or other IOs. Examples follow:

---

[9]Refers to objects that must keep their physical nature (beyond the computerized one) after the implementation of the new computerized IS. Examples include a book and a car, and counter-examples include a prescription and a receipt. Transfer of Possession is conceived electronically as an action of Alteration, since it changes the State of an IO (e.g., for Book: from *In Stock* to *Sold*).

[10]In this case, each role has a responsibility according to the nature of the activity it is part of, e.g., the *Creator* has the responsibility of the creation of an IO.

(a) For the *Bookstore* IS, a *Supplier*, in the role of the *Sender*, sells books to the *Bookstore Stock-keeper* who is the *Receiver*. This transaction causes instances of other IOs to be created such as a Supplier's *Payment* IOi and one or more *Book* IOi corresponding to new books acquired by the bookstore. Therefore *Supplier* and *Stock-keeper* are IOs.

(b) For the Hospital IS, a *Doctor*, in the role of the *Creator*, creates a *Prescription* IOi. This creation also triggers the activity of giving drugs to a patient, and so it causes a change of a Drug IOi. Therefore *Doctor* is an IO.

A counter-example to the above occurs between the *Bookstore* IS and the *Maker-Company* of the *Pen*. Since the latter does not supply (send) the pens to the *Bookstore* directly (they are supplied through a particular *Supplier*), it does not have any direct interaction with the IS, and so it is not considered an IO of that IS.

Documents which are collections of attributes of other IOs, such as a *report* or a *notification*, which are created automatically by the system, are not considered to be IOs, as a rule of thumb. However, there could be, for example, the rare case where a business sells its reports. In this case the *report* falls under rule (1) example (c) and thus would be considered an IO.

### 3.3.3 Develop FSRs for each Information Object

The previous step was concerned with the identification of the Information Objects. This step involves the application of specific (CAREN) functions on every IO, as well as the written specification of requirements in the form of formalized sentences (Formalized Sentential Requirements—FSRs) composed of the IO, its CAREN functions, the involved

business roles and the functional conditions. NLSSRE provides specific FSR patterns, from which our methodology guides the analyst to derive specific questions to find the business roles and functional conditions; the answers to these questions help to form the complete FSRs. Therefore, we note that writing the requirements as formalized sentences not only helps to make expression of requirements more disciplined, understandable and organized, but it also leads to the identification of entities—the business roles and functional conditions—that are involved during the application of a CAREN function on an IO. Furthermore, such formalization makes easier the transformation of requirements into diagrammatic notations and specifications. Figure 3.2 below shows **CAREN**, the proposed set of functions with their sub-functions, which are part of every IO[11].

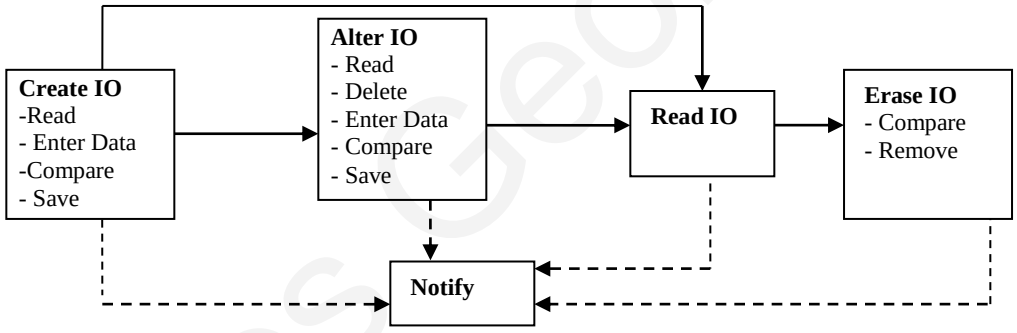


FIGURE 3.2 CAREN - A RECOMMENDED SET OF FUNCTIONS AND SUB-FUNCTIONS APPLIED ON AN IO, AND THE NOTIFICATIONS PRODUCED.

**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. We need to clarify that we use functions at the system level, which are the functions the

[11]The functions are derived from the study of semantic roles of NL verbs by relating them to electronic information and existing IS literature.

system provides to the users to fulfill their needs (what the system will do). 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, based on the users' requirements from the perspective of the system (product). For example, a system-level user's requirement is to be able 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. We neither focus on abstract-level requirements, which include the users' perspective about the functionality of the system.

The formalization concept is more easily applicable to the system functions, because they are applied on electronic information, while it is hardly applicable to the user level functions, due to the complexity of the business environment, in both size and terminology. For example, the function *Enroll in Seminar*, which may be implemented as a system or a business function, is formalized and represented in our approach through the system Information Objects *Enrollment* and *Seminar*. The IO *Enrollment* includes the system functions *Create, Alter, Cancel, Erase and Read Enrollment*. The IO *Seminar* includes the system functions *Create, Alter, Cancel, Erase and Read Seminar*. Information about *Seminar* will be part of the IO *Enrollment* (e.g., *seminar id* is used when creating or altering an enrollment) similarly also to provided information about the student who participates in the enrollment. *Student* will be also a different IO.

What the analyst has to do within this step is to write, in the form of an FSR, each IO, its CAREN function that corresponds to the class of that particular FSR, the business roles (that replace the functional roles—explained later in this section—of the FSR

pattern) involved during the application of each CAREN function on the IO, and the functional conditions that denote the circumstances within which a function is performed. In particular, NLSSRE provides four FSR patterns which include all the aforementioned elements, and each pattern corresponds to a CAREN function. The Notify function is specified as a supplementary formalized sentence of each FSR. Figures 3.3 (a) and 3.3(d) provide examples of FSRs, in their pattern and complete forms[12] respectively, stemming from a real case example. Before we expand on the FSR syntax and its constituent parts by providing relevant definitions and description, we will first give a short description of each FSR's CAREN function which is the core of the FSR pattern.

**Create** is the most significant function, because during its execution the attributes of an IOi take their initial values; these values are then processed by the other CAREN functions. *Create* is decomposed to the sub-functions *Read, Enter data values, Compare* and *Save*. *Read* is the sub-function that presents, through a particular layout, the attributes—required and optional—of the IO which need to be initiated. After the entry of the data values, the *compare* sub-function will check if each value to be assigned to each IO attribute satisfies the relevant data constraints and possible business rules (discussed in step 5) pertaining to that attribute; *Save* then stores the approved values (comparison and saving also produce notifications regarding the success or failure of their action). For example, for the creation of a *Translation* IOi, first the user will read (view) the attributes that constitute this IOi (e.g., *number of words*, *source language*, *target language*, *time of delivery*), then s/he will enter the values for these attributes, and then the system will

[12]A complete form of an FSR is when the constituent elements of the FSR take their values (e.g., *Creator* takes the value *Doctor*)

compare the values entered with relevant constraints and possible business rules. An example may be: "For each entered value for the attribute *number of words*, the system will compare it with the standards indicated by the data constraint '*Number of words is a positive integer'*, and with the business rules '*Comma is the thousands separator and period is the decimal separator'*, and '*A translator can translate 2,000 words per day'*. If both the constraint and the business rules are satisfied (e.g., *number of words = 5,230* and *time of delivery = 3 days*), then the new *Translation* IOi will be saved, otherwise a relevant warning will be given by the system."

**Alter**: During the execution of this function, the existing values of the attributes of an IOi are changed. 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/reviewed* or *retrieved*. The change from one state to another (e.g., from *Pending* to *Complete*), for a particular IO, often derives a new alteration function, such as *Cancel*, *Complete*, etc. However, if the change of state does not justify the existence of a new alteration function, it should be represented through additional sub-functions under the CAREN functions *Alter* or *Create*.

When a change of state occurs, we should check what new pre-conditions, post-conditions are created and what new end users are involved in the execution of the new derived function or—in the case of representing the change of states as sub-functions—the existing *Alter* and *Create* functions. Usually when the change of state of an IO results in significantly different pre-conditions or post-conditions, or results in new sub-functions than those provided by the *Create* or the basic[13]*Alter* function, we recommend to represent this self-contained information (conditions, sub-functions) as a new function. For example, *cancelling an appointment*, results in a different post-condition than the post-condition resulting from the normal execution of the function *Create Appointment*, which is to complete the appointment. In particular, by cancelling an appointment, the *State* 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 function. Similarly, *completing a prescription* derives the precondition that "Drug is given to patient" comparing to the basic function *Alter Prescription* which has the precondition "Prescription is created". Completing a prescription is also performed by a different end user (pharmacist) at a different place (drug store) than the end user (doctor) that initiates the *Create* and *Alter* functions of the prescription IO, at the hospital or clinic. Therefore, we should consider *Complete Prescription* as a new function. We may also conceive *Erase*, described below, as a new

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

function, where new post-conditions might be “IOi is archived” or “IOi is removed completely from the system’s databases”.

**Read**: The meaning of this function may be conceived in two ways: the first, which is the one that concerns requirements analysis, is about what a user wants to read regarding a particular IO per se or from its relations with other objects. It mainly concerns the presentation (optical or acoustical) of notifications and forms regarding the IO per se (e.g., Appointment form), or the presentation of reports of the IO with related objects (e.g., report of a patient’s monthly appointments). The second concept for *Read* concerns the way the data will be presented, including drawings, graphics, video, multimedia, etc.; the first meaning of this concept falls in RE, but the detailed procedures of implementing methods of presentation concerns the Design which is outside of the scope of RE.

In particular, 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 creates a new IOi or changes the state of an existing IOi. The reading procedure for the alteration functions (*Alter*, *Cancel*, *Complete*, etc.) includes retrieving and checking the existing information about an IOi, from the database, in contrast to the reading procedure for the *Create* function, which only concerns building a form of required and optional empty fields, and thus it

is much simpler in structure. *Read* may also be executed independently, when the end user needs to read information from an existing IOi form, e.g., a pharmacist wants to read a prescription.

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 function *Read IO intra-report* is part of the IO, while the function *Read IO inter-report* may still be part of the IO (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* IO) or of a more general *Report* IO, because inter-reports may be used by different functions of different IOs.

Usually *Read* functions about reports, and especially the inter-report type, are useful for the execution of functions of other IOs. This relationship usually occurs when an end-user creates or alters an IOi, and so the end-user may need to read information about instances of other IOs, related to the IOi the end-user creates or alters. For example, when a doctor (end-user) 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 function, such as *Read Patient History*, 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. 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 functions (e.g., *Create Prescription* sends notification to the patient that the prescription is created) for the end-users (e.g., *Doctor* for the function *Create Prescription*) who executed the specific, or for notifiee users (explained in detail later) who simply need to be informed to maybe make an action outside the system (e.g., patient goes to pharmacy after receiving notification that the prescription is created), or for intended recipients (which are also explained later) that need to be notified in order to make an action within the system, as a result of the execution of the function that triggers the notification (e.g., pharmacist who needs to be notified about *Creation* of a *Prescription* in order to provide the drug to the patient and thus to complete the prescription).

**Erase:** 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 end-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.

**Notify**: At the user level, in a manual, paper-based IS, we encounter the *transmission* function (from the linguistic verb of *transfer of possession*), where data is sent from one entity to another. For example, the *Doctor* gives the *Prescription* to the *Patient*, and the *Patient* gives the *Prescription* to the *Pharmacist*. In a computerized IS the *transmission of prescription* is replaced by the *Read* function, since the IO (*Prescription*, in this case) is already stored (after its creation or alteration) in the IS. Therefore, the *Pharmacist* can *Read* the *Prescription* IOi by simply retrieving it from the database. However, 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, as explained earlier: 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 the end-users who created or altered the IOi.

After describing each CAREN function, we now proceed to illustrate how we will write together, in the form of FSRs, each identified IO, its CAREN functions, the business roles and functional conditions involved.

*Definition* **1.** A formalized sentential requirement pattern

$$\boldsymbol{FSR_F^{IO}} = \langle S\rangle\langle F\rangle\langle IO\rangle\langle FC\rangle :: SystemNotifies\langle R\rangle\langle FC_R\rangle$$

is a structured, semi-formal way of writing a requirement of an IS, based on the basic syntactic form for writing a sentence in natural language, that is, <Subject> <Verb> <Object><Adverbial>. It contains a *CAREN* function *F* which acts on the *Information Object IO*; the animate (or system) entity group *Subject S* that refers to either the doer(s) or the experiencer(s) of the function, the animate (or system) entity group *Receiver R* who refers to the entities need to be notified about the application of the involved function on an instance of the IO, and the *Functional Condition FC* which is a clause that adds further information about the function, and especially it normally establishes the circumstances within which the function takes place. The syntax of the notification the execution of *F*, is placed after the symbol "::" (a complete example of FSR is given at the end of this section).

*Definition* **2.** $\boldsymbol{F} \in \{Create, Alter, Read, Erase\}$

is the CAREN function of the FSR, which creates, alters, reads or erases an instance of the IO. NLSSRE uses four FSR classes; those are *Creation*, *Alteration*, *Reading*, and *Erasure*, which correspond to the *Create*, *Alter*, *Read*, and *Erase* CAREN functions, accordingly. Therefore, the FSR pattern of each class contains the corresponding CAREN function. For example, the pattern of the *Creation* FSR class is written

$$FSR_{Cre}^{IO} = \langle S \rangle \langle Create \rangle \langle IO \rangle \langle FC \rangle :: SystemNotifies \langle R \rangle \langle FC_R \rangle.$$

As described previously, each CAREN function decomposes to a set of second level functions, the most interesting of which is the *Compare* sub-function, which we present in step 5 through the discussion on the development of business rules.

*Definition 3. S(ubject)* represents one or more business roles of the IS, which create, alter, read or erase an instance of the IO in an FSR, depending on the FSR class they participate. According to the FSR class, each business role of *S* corresponds to a different functional role in the FSR pattern. A functional role is the role a business role plays with respect to the CAREN function of the FSR. For example, in the simplified

$$FSR_{Cre}^{App} = \langle Assistant \rangle \langle Create \rangle \langle Appointment \rangle$$

*Assistant* is the business role who plays the *Creator* of an instance of the *Appointment* IO. Functional roles can help us derive questions to find the different business roles they correspond to, and also to find relevant attributes. In particular:

For the Creation FSR pattern: $S_{cr} = Cr_i \cup Ac_{i,j}$,

$$i = 1, \dots, n, j = 1, \dots, m, \quad \bigcap_{i=1}^{n} Cr_i = \varnothing, \left|Cr_i\right| = 1, \left|Ac_{i,j}\right| = 0, \dots, m$$

For the Alteration and Erasure FSR patterns: $S_{al} = Al_i \cup Ac_{i,j}$,

$$i = 1, \dots, n, j = 1 \dots m, \bigcap_{i=1}^{n} Al_i, \left|Al_i\right| = 1, \left|Ac_{i,j}\right| = 0, \dots, m$$

For the Reading FSR pattern: $S_{ex} = Ex_i \cup Ac_{i,j}$,

$$i = 1, \dots, n, j = 1, \dots, m, \bigcap_{i=1}^{n} Ex_i, \left|Ex_i\right| = 1, \left|Ac_{i,j}\right| = 0, \dots, m$$

where:

i. *Cr* is the business role who plays the *Creator* functional role by creating an instance of the IO in a *Creation* FSR. That means that the *Creator* is responsible for setting the values of a number of particular attributes (required and optional) of the IOi. To identify

the creator business role in a Create FSR, we may ask the following questions (question patterns and pattern instances follow[14]):

- Pattern: Who should create an <IO> ?
- 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?

$S_{cr}$ must include at least one creator (only rare cases would have more than one creators), and it may or may not include one or more accompaniments for each creator [*n* and *m* are positive integers denoting the numbers of creators and accompaniments, respectively.]

ii. *Al* is the business role who plays the *Alterer* functional role by altering an instance of the IO in an *Alteration* or *Erasure* FSR; in particular the Alterer replaces the old values of the attributes of the IOi with new ones, or erases an IOi, or the alterer adds new values to the optional fields of the IO which were left empty during the creation of an IOi (e.g., Doctor is the alterer in the FSRs *Alter Prescription* or *Cancel Prescription*, when the doctor alters the value of the attribute *dosage* in the former FSR, and when the doctor alters the value of prescription *state* from *pending* to *cancelled*, in the latter FSR). In another example, the pharmacist is the alterer in the FSR *Complete Prescription*, since the pharmacist alters the value of prescription *state* from *pending* to *completed*. To

[14] instances are taken from the HIS case study

identify the alterer business role in an *Alter* FSR, we may ask questions based on the following patterns:

- Pattern: When do you need to change an <IO>?
- Pattern: What changes do you need to make?
- Pattern: What states can this <IO> have? (different types of IOs have different types of states, as mentioned in step 3)
- Pattern: Who changes an <IO> from the <state A> state to the <state B> state?

$S_{al}$ must include at least one alterer, and it may or may not include one or more accompaniments for each alterer *[n* is the positive integer number of alterers, and *m* is the positive integer number of accompaniments].

iii. *Ex* is the business role who plays the *Experiencer* by receiving a sensory impression from viewing or listening to structured or collected information (e.g., reports) about the IO per se or about the IO and other related entities (e.g., *Doctor* or *Pharmacist* are experiencers in an FSR *Read Prescription*.) To identify the experiencer business role in a *Read* FSR, we may ask the following questions (instances are taken from the HIS case study):

- Instance: Who needs to read the daily appointments report?
- Answer: Doctor (experiencer) and Secretary.
- Pattern: Who needs to read an <IO> after its creation?
- Instance: Who needs to read a prescription after its creation?
- Answer: Pharmacist.

An experiencer (as also a creator or an alterer) initiates the *Read* FSR. S/he usually needs authorization to initiate the *Read* FSR (the Read function actually) because this type of FSR mainly handles information within the system. On the other hand, stakeholders who are only notified about the result of an FSR (e.g., when a prescription is created, the patient should receive a notification), they only read a relevant notification, without usually being asked for any authorization. $S_{ex}$, in a *Reading* FSR, must include at least one experiencer, and it may or may not include one or more accompaniments for each experiencer [*n* and m are positive integers, denoting the numbers of experiencers and accompaniments, respectively].

iv. *Ac* is the (animate or system) business role who plays the *Accompaniment* functional role by participating in close association with the *Creator*, *Alterer* or *Experiencer*, depending on the FSR class, to help them in the creation, alteration (including both *alter* and *alter-related* FSRs) 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 FSR *Create Appointment*.) While *Creator* normally corresponds to an internal business role, *Accompaniment* can by assigned to both internal and external business roles. In the case of the internal business role, the *Accompaniment* is usually involved in confirmation of the values ascribed to some attributes of an instance of the IO. For example, when an assistant creates an appointment (*Appointment* IOi), s/he may ask the doctor (internal business role who plays the accompaniment) to confirm that the doctor agrees to the time set for that appointment. In the case of the external business role, the *Accompaniment* (e.g., client, supplier, patient) usually provides input to some IO

attributes; that is, in the same example, the *Patient* will provide different data values such as his/her *ID* and *preferred time* of appointment, for the *Appointment* IOi. The collaboration between a creator, alterer or experiencer and an internal accompaniment can derive new FSRs, called complementary FSRs which are normally initiated by the internal accompaniment. For example, 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 they are defined through the complementary FSR "*Counselor, Medical System provide feedback to Doctor in Create Prescription"*. To identify the business role of an accompaniment, we should ask questions related to its FSR class (Create, Alter, etc.) and the functional roles *creator*, *alterer* and *experiencer*. For example, to identify the accompaniment in a Create FSR, 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>?
- Pattern: What data/attributes of the <IO> does the <Accompaniment> need to check/confirm and in what way?
- Pattern: How does the <Accompaniment> help the <Creator> during the creation of an <IO>?

Figures 3.3(b) and 3.3(c) are screenshots from the NALASS tool[15] regarding the HIS case, which show some indicative questions and answers to identify the business roles and show how business roles are written. For example, the *Creation* FSR of the *Prescription* IO involves $S=Cr_1,Ac_1 \rightarrow$*Doctor, Patient.*

(note: Counselor doctor may be added as an internal accompaniment)

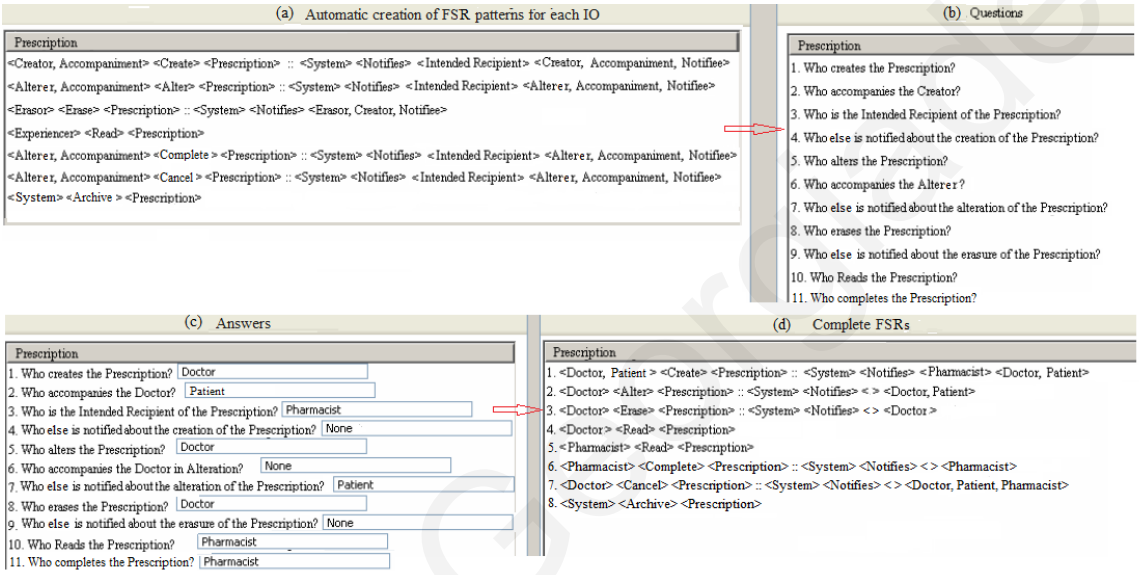


FIGURE 3.3 A NUMBER OF PREDEFINED QUESTIONS (B) CREATED AUTOMATICALLY BY THE FSR PATTERNS (A), AND THE RESULTING FSRS (D) CREATED AUTOMATICALLY BY THE ANSWERS TO THE QUESTIONS (C), FOR THE PRESCRIPTION IO. SCREENSHOTS ARE TAKEN FROM OUR SOFTWARE TOOL THAT IMPLEMEN

*Definition 4. IO* is the Information Object, identified in a previous step, which is involved in the four FSR patterns *Creation, Alteration, Reading* and *Erasure.* It contains specific attributes according to its IO category. Some attributes are compulsory and

[15] The FSR syntax as provided by the tool needs some final refinements

others are optional. Categories of IOs and attributes will be discussed in step 4. As an example, the pattern for an IO categorized as *physical object* is:

$$IO \ni (Ph_i \cup Per_j \cup Docum_p \cup Pre_r),$$

$$1 \le i \le n, 1 \le j \le m, 0 \le p \le u, 0 \le r \le w$$

where:

*n* is the (positive integer) number of *Physical* attributes (compulsory), *m* is the (positive integer) number of *Peripheral* attributes (compulsory – peripheral attributes contain information about other entities related to the IO, such as the business roles involved in the four different FSR patterns of the IO), *u* is the number (non-negative integer) of *Documentation* attributes, *w* is the number (non-negative integer) of *Presentation* attributes (if $p$=0 → there are no documentation attributes; if $r$=0 → there are no presentation attributes).

The following is an example of a number of physical and peripheral attributes contained in the *Book* IO, for a Library IS:

$$BookIO \ni \{Material, Content, Numberofpages, Bookkeeper, Supplier, Student\}$$

Specific questions to derive particular attributes and their values are defined during the fourth step of the methodology; hence a related discussion will follow in the next step.

*Definition 5.* (*R)eceiver* represents one or more animate or business entities who receive a notification as a result of the creation, alteration, reading or erasure of an IO. *R* may include two distinct subsets, the ***I**ntended **R**ecipient* (IR) subset and the ***No**tifiee* (No) subset; the former includes the business roles who will take action within the IS after they

are notified about the creation, alteration (including erasure) of an instance of the IO. The action to be taken needs to fulfill the purpose of the IO[16] within the IS, and the fulfillment is achieved by creating or altering instances of other related IOs. For example, in the simplified

$$FSR_{Cre}^{Pat} = \langle Assistant \rangle \langle Create \rangle \langle Patient \rangle$$

*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 FSR *Create Prescription*, *Pharmacist* is an IR of the IO *Prescription*, 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 IOs (e.g., IO *Drug*) and new FSRs (e.g., *Create Drug*) in which the IR this time plays the role of creator or alterer (e.g., Pharmacist who was an IR in the FSR *Create Prescription* of the IO *Prescription*, is an alterer in the FSR *Alter Drug* of the IO *Drug*). As we will show in later sections, the use of IR helps in linking use cases in a use case model, classes in a Class diagram and processes in a DFD. To identify the IR business roles, we can ask the users the following relevant questions, based on the

[16]The *IR* is drawn from the genitive case of purpose.

purpose of the IO, which is fulfilled by the business role 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 in the FSR *Create Patient*, since s/he fulfills the purpose of the patient (to receive examinations, etc.) through executing new FSRs (e.g. FSR *Create Examination*).

In contrast to the IR subset, the *notifiee* subset 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 *other Stakeholders* (St). Notifiees also include the business roles of *Creator* (denoted by Cr), *Alterer* (Al), or *Experiencer* (Ex), with their *accompaniments*, if any, as defined within the *Subject* set of each relevant FSR pattern, who need to receive notification about the creation, alteration or erasure of an instance of the IO they interact with. If any of the Subject roles is also an IR, then this role will appear only as an IR. Additionally, the notification (its content or layout) sent to each notifiee may be different, based on the preferences of each notifiee. Furthermore, for the *Alteration* FSR, when the *Alterer* is a

different business role from that of the *Creator*, for the same instance of the IO, then the *Creator* and the *Alterer* must be both notified about the alteration of that IOi. For the *Reading* FSR, when an instance of the IO is read, usually the access/retrieval time must be recorded and also the *Creator* must be notified. As an indicative example, we present the pattern of *Receivers* in the *Creation* FSR:

$$R = (IR_k \cup Cr_i \cup Ac_{i,j} \cup St_p),$$

$$0 \le k \le n, Cr_i \in S, Ac_{i,j} \in S, 0 \le p \le z$$

where:

n and z are non-negative integer numbers denoting the numbers of intended recipients and stakeholders, respectively (If n=0 → there are no intended recipients to be notified; if z=0 → there are no stakeholders to be notified).

And an example of the *Receiver* pattern's realization for the

$$FSR_{Cre}^{Pre} = \langle Doctor, Patient \rangle \langle Create \rangle \langle Prescription \rangle < FC > ::$$

$$SystemNotifies \langle R \rangle \langle FC_R \rangle :$$

$$R = \left(IR_k \cup Cr_i \cup Ac_{i,j} \cup St_p\right) \rightarrow \{Pharmacist, Doctor, Patient, Relative\}$$

Based on the functional roles *Intended Recipient* and *Notifiee*, we can derive a number of questions (in addition to the ones mentioned above), for each FSR class, to identify intended recipients and notifiies of the IS. Below we present some sample questions (each question has its own pattern, based on which it is instantiated; patterns are not presented here for simplification) for the $FSR_{Cre}^{\mathrm{Pre}}$ example illustrated above:

–Who will receive 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 of the *Prescription* IO from *Pending* to *Completed*, therefore s/he is an IR)

*Definition 6. Functional Condition FC* is a notion derived from the linguistic notion of the adverbial adjunct. In linguistics, an adjunct is an optional, or *structurally dispensable*, part of a sentence that, when removed, will not affect the remainder of the sentence [Lyons, 1968]. Similarly to the linguistic adverbial adjunct, a functional condition usually establishes the circumstances in which each CAREN function takes place. It can be denoted in the FSR with a single word (e.g., stylus), a phrase (e.g., with stylus, doctor's office), or a clause (e.g., after patient is examined).

Based on the semantic roles of the linguistic adverbial adjuncts, we utilize a number of categories of functional conditions that establish the major circumstances in which each CAREN function takes place. For each *functional condition*, the analyst needs to derive specific questions, the answers to which will give information about the users' requirements in regards to the circumstances they will operate the IS. The questions mainly involve the *Subject* group of the FSR (business roles of creator, accompaniment, etc.), who initiate or experience the action or experience, respectively, denoted by the

relevant functions. Therefore, the users with these particular business roles should be among the recipients of the questions. New questions are also derived when the planned circumstances are not satisfied (example a(*f*) below**)**. We present below the most common FC categories, together with sample questions for the simple form of the FSR <Doctor><Create><Prescription>. Additionally we need to mention that the analyst should try to receive concrete and not vague answers (e.g., 'afternoon' is vague, while '14:00-19:00' is specific[17]). The FCs of each FSR are written as

$$FC_k \in \{tp, d, f, p, i\} \text{ and } 0 \leq k \leq n$$

where n is the number of functional conditions, and:

(a) Temporal conditions establish when (time point; e.g., 8:00 – 14:00), for how long (duration; e.g., 1 month) or how often (frequency; e.g., every day) an action (denoted by the functions *Create*, *Alter* or *Erase*) or experience (denoted by the function *Read*) occurs. Below we present indicative questions and answers:

tp: When can the doctor create the prescription? *Answer: 8:00 – 14:00*

d**:** How long does the Doctor need to create the Prescription (the whole procedure of filling data, not the saving procedure)? *Answer: 5 minutes*

f: How often does a Doctor need to create prescriptions? Or, How many prescriptions can a doctor create (per day)? *Answer: Maximum 10 prescriptions / day*; What happens when this limit is reached?

[17] Unless we use aliases (see paragraph about adjectives in step 4).

(b) Locative conditions establish the place (from) where an action (denoted by the functions of *Create, Alter* or *Erase*) or experience (denoted by the function of Read) occurs. An indicative example of question-answer could be:

p: Where should the Doctor create the prescription? *Answer: Computer at his/her office.*

(c) Instrumental conditions establish the instrument of the action (denoted by the functions *Create, Alter* or *Erase*) or experience (denoted by the function of *Read*). An indicative example of question-answer follows:

i: How/what instrument does the doctor (use to) create a prescription? *Answer: keyboard/stylus*

Putting them together, the functional conditions for the Create Prescription FSR are syntactically realized as follows:

$$FC \ni \left(8:00-14:00, 5 minutes, 10\, prescriptions / day, DoctOfficeComputer, keyboard \vee Stylus\right)$$

Another use of the functional conditions is to determine the *priority level of a requirement*. For example, the requirement *"Assistant creates appointment with voice"* can be determined to have a lower priority than the requirement *"Assistant creates appointment with keyboard"* (where *with voice* and *with keyboard* are instrumental functional conditions) because the latter is easier and less costly to be implemented.

To summarize, the complete realization for writing the *Creation* FSR, based on the partial examples given above is:

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

Or

$$FSR_{Cre}^{Pre} \langle Doctor, PatientCounselor \rangle \langle Create \rangle \langle Prescription \rangle \langle 8:00-14:00, 5min, \ldots\ldots \rangle$$

$$\langle Counselor \rangle \langle Give \rangle \langle \Pr escriptionHelp \rangle \langle ByPhone \vee ByEmail \vee ByForm \rangle$$

By the end of this step, for each identified IO, the four patterns of the FSRs, the business roles and functional conditions of each FSR, the questions that derive the business roles and the functional conditions, as well as the answers to the questions, need to be derived. Figure 3.3 presents this procedure for the *Prescription* IO with the use of the simple syntactic form of the FSR pattern since the complete syntax is not yet provided by the tool. We observe that during this step the analyst might answer some of the questions from the material already collected during step 1 of the methodology with the use of the data flow questionnaires and the data flow table. The analyst may confirm the answers with the client's approval.

### 3.3.4 Define the attributes of each Information Object

NLSSRE provides specific attribute categories, each of which is linked to an IO category, and defines each attribute category as compulsory, optional or not applicable. In this way the analyst will know which categories of attributes to utilize for each identified IO taking into account the category of the IO, and then search to find attributes for that particular attribute category linked to the particular IO. Table 3.4 indicates how some of

the attribute categories, provided by NLSSRE, are linked to the IO categories *physical object* (divided to *business role* and *inanimate*), *procedure* and *document*.

TABLE 3.4 THE ATTRIBUTE CATEGORIES ARE LINKED TO EACH CATEGORY OF IO.

| IO category / Attribute Category | | **Physical Object** | | **Procedure** (e.g., examination, translation) | **Document** (e.g., essay, book, e-book, appointment) |
|---|---|---|---|---|---|
| | | **Animate Business Role** (e.g., patient, student) | **Inanimate** (e.g., car) | | |
| **Physical** | **Animate** | C | n/a | n/a | n/a |
| | **Inanimate** | n/a | C | n/a | O |
| **Purpose** | | O | O | O | O |
| **Temporal** | | n/a | n/a | C | n/a |
| **Possessional** | **Inalienable** | O | O | O | O |
| | **Alienable** | O | O | O | O |
| **Aliases** | | O | O | O | O |
| **Compositional** | | O | O | O | O |
| **Locative** | | O | O | O | O |
| **FSR** | | C | C | C | C |
| **Part-Whole** | | O | O | O | O |
| **Procedural** | | n/a | n/a | C | n/a |
| **Document** | | O | O | O | C |
| **Documentation** | | O | O | O | O |
| **Presentation** | | O | O | O | O |
| **Comparative** | | O | O | O | O |
| O: Optional C: Compulsory n/a: Not Applicable | | | | | |

To construct the various attribute categories[18] we take into account the nature of each category of IO, the FSRs, and also linguistic notions that define relationships between objects, as does the notion of genitive case (examines relationships between nouns), or that provide additional information about an object, as does the adjective. We elaborate on these two notions in the following paragraphs of this section.

For each attribute category, the analyst should try to identify the attributes of each IO, based on specific features of both the attribute category and the IO category. The analyst should use different methods. One method is to utilize the business documents collected

[18] NLSSRE tries to cover a considerable number of attribute categories; however, the list provided could be expanded by the analyst.

during the first step of the methodology or the data flow table created during the same step, as well as existing field knowledge, in order to create a list of possible attributes for each IO. A subsequent task involves asking the users questions in order to cross-check the attributes list and keep only the attributes that are related to the business context. A second method involves asking additional questions clearly related to the attribute category, in order to derive new attributes.

For example, to derive the physical attributes of the *Patient* IO, the analyst can create a long list of animate physical attributes derived from the collected business documents, the data flow table and scientific sources such as the *Dictionary of the Physical Sciences* [e.g., Emiliani, 1987; Meyers, 2001]. Indicatively, the provided list could have the attributes of *tension*, *temperature*, and *mass*. Not all attributes necessarily apply, therefore the analyst should discuss with the user, in order to select the attributes relevant to the business role *patient* in the Hospital IS. Indicatively, the final list could have the attributes of *temperature* and *mass* only, which are relevant—based also on user's preference—to *patient*. Of course, NLSSRE, with the use of other categories of attributes, can cross check some of the physical attributes and also derive some new ones. For example, *temperature* is also a *situational*[19] attribute further categorized as *possessional inalienable*[20], and from this *temperature* attribute and its corresponding categories (e.g., possessional, physical) and related IOs (e.g., *Patient* IO), the analyst should ask the user questions such as "*Describe: what conditions can affect the Patient's mind?"* or "*What*

[19] *Situational* attributes comprise a broader attribute category, and they are related to characteristics of the business role.

[20] Derived from the genitive case of *temporary or abstract inalienable possession*, in which the noun of possession indicates an abstract possession which affects the body or mind (e.g., disease, fever, anger, cold)

*affects the situation of the patient's body?*" and derive the proper attributes (e.g., *temperature*, from the received answer *fever*, and *pressure*). When we do not suspect the answer in advance, or when we want to derive new information, we provide open questions to the user such as the ones above.

Another situational attribute category is the one which indicates the *purpose* of the IO entity; that is the purpose for which the entity is used. For example, for the *Doctor* IO, the analyst could make questions such as "*What responsibilities/duties/tasks does the doctor have in the hospital/clinic?"* Possible answers could be *to examine*, *treat*, *prescribe*, and *diagnose.* Therefore information related to these responsibilities could be attributes of the *Doctor* IO. For example, from the types and numbers of examinations a doctor provides, the analyst can derive or verify the doctor's specialties and the number of examinations the doctor can perform daily. *Specialty* and *Number of examinations daily* should be attributes of the *Doctor* IO. Additionally, searching for attributes can also lead to new IOs. For example the aforementioned responsibilities can give rise to the procedural IOs *Examination*, *Treatment*, *Prescription*, and *Diagnosis*. For these new IOs the *Doctor* IO will be their *Creator*, and therefore some specific attributes of the *Doctor* IO will constitute some of their peripheral attributes (e.g. *Doctor ID* and *Doctor Name*).

*FSR attributes* are a compulsory part of an IO, since they provide information about other entities related to the IO under study; these entities are mainly business roles, but they can also be other stakeholders or functional conditions, all of them constituent parts of the FSR of the IO under study (e.g., information about the *Doctor* business role who plays the *Creator* of the *Prescription* IO in the *Create Prescription* FSR, will be

attributes of the *Prescription* IO). Since all FSRs must have been specified, as per step 3, the analyst should include attributes of all the related FSR entities in each IO.

Another example can be given with the use of the *locative attributes* category drawn from the genitive case of *spatial location*. For each IO, the analyst should check for its related locative attributes, such as the attributes *Hospital* and *Clinic* for the business role IO *Doctor*. Hence, relevant questions should be provided to the relevant users (e.g., doctors and clinic manager), such as *"Where should/could a doctor examine patients?"* (possible answer: *Hospital*), *"Can a doctor examine patients outside the Hospital?"*, *"Can a doctor from an external clinic examine a patient?*"

*Aliases* are shorter names of attributes already defined, and they aim to make attributes easier to be expressed and understood. They are denoted by adjectives. For example, the entire attribute (including its value) *time of work: 08:00-14:00*, defined by the genitive case of *time*, can be replaced by the attribute alias *morning* which is an adjective that denotes time. In this way we can achieve easily understood expressions. For example, when defining a business rule, instead of writing (in its simplified form) "*Doctor of Time of Work 08:00-14:00 Creates Prescription"*, we could write "*Morning Doctor Creates Prescription"*. Therefore we could provide simple questions to the IS user, such as *"How would you like to call the doctor who works from 08:00 to 14:00? Morning?"*

Another category is that of *comparative* attributes used when we need to compare entities of the same capacity (e.g., the same business role). They usually draw on relative adjectives (e.g., previous, next, first, last), which is a category of unambiguous

adjectives[21]. For example, the adjectives *first* and *second* could derive the *Rank* attribute for the IOs of *Doctor* and *Nurse*. For the *Doctor* IO, *Rank* could take the values *Consultant* (from the adjective *First*) and *Specialty Registrar* (*Second*), while for the *Nurse* IO, *Rank* could take the values *Advanced Practice* (*First*) and *Registered (Second)*. Therefore we could provide specific questions to the user, such as *"What is the Rank of each doctor?"*

Beyond the identification of specific attributes, the adjective types can also be used to derive subordinate or parent IOs. For example, the adjective type *color* can determine the sub-IOs *Blue Prescription* and *Red Prescription* for the *Prescription* IO. Or the adjective type *rank* can determine the sub-IOs *Professor* (First), *Reader* (Second), *Senior Lecturer* and *Lecturer* for the *Academic* IO.

### 3.3.5 **Define business rules**

After the identification and definitions of IOs, their FSRs and their attributes, the analyst should proceed to the identification and definitions of business rules. According to Leffingwell [2011], a business rule defines or constrains an aspect of the business that is intended to assert business structure or influence the behavior of the business. In NLSSRE, business rules focus on business policies, an example of which could be a university policy to expel any student who fails more than two courses in the same semester, which results in the alteration of the *Student* IOi state or its complete erasure. Another example is a Cypriot hospital policy that foreign patients (those who do not

[21] Most of the adjective types are inherently or potentially ambiguous (ref) and should be avoided (e.g., big, small, happy, etc.) in formalization.

speak Greek) must be (or *are preferred to be*) examined by English-speaking doctors; this is a policy that affects the organization and structure of the hospital. The business rules can be divided into two types:

(1) *Inter-related business rules*. These rules are created from combinations of two or more attributes between different interrelated IOs of any category, but with more emphasis given on business roles. In particular, the values of one or more attributes of one or more IOs determine the values of one or more attributes of one or more related IOs. The interrelated IOs can be identified easily through their co-involvement in the same complete FSRs. For example, for the simplified form of the FSR below:

$$FSR_{Cre}^{Pat} = \langle Receptionisht, Patient\rangle\langle Create\rangle\langle Patient\rangle\langle Doctor, Receptionist\rangle$$

where:

$$PatientIO \ni \{Name, Surname, ID, Temperature, Language, \dots\},$$

$$DoctorIO \ni \{Name, Surname, ID, Specialty, Language, Gender, Rank, Schedule, \dots\}$$

*Patient* is an accompaniment in the creation of a *Patient* IOi, and *doctor* is the intended recipient of *patient*, since the doctor will examine, diagnose and prescribe for the patient; therefore *Patient* IO and *Doctor* IO are interrelated. The procedure of deriving the business rules is facilitated with the use of relevant questions, such as:

*Question: Does a patient's language determine the language of a doctor assigned to the patient? If yes, how?*

Based on the users' answers, we construct each interrelated business rule, in its *general form*, *general initialization form* and *personalized initialization form*. For the

second form, each business rule can be applied to all instances of the participant[22] which determines the rule (Patient, in our example – see General rule 1 below). For the last type of rule, the rule can be personalized for each participant separately. In this case, each participant must be asked to choose between the options he or she is provided. For the same example, the user agrees to the following business rule in the three forms described above:

(i) General rule 1: A patient's language determines the language of a doctor assigned to the patient.

(ii) General initialization rule 1.1: If patient's language is not Greek, then doctor's language is preferred to be English.

(iii) Personalized initialization rule 1.1: Each patient is asked to choose his/her doctor's language, from the available list.

And the rule is expanded according to the participant's answer, such as in the example below, for the patient's choice of a *French-speaking* doctor:

(iii expanded) If there is a doctor whose language is French, then doctor's language for the doctor assigned to this patient is preferred to be French.

Furthermore, the flexibility of a business rule is determined with the use of grades denoted by the expressions *must*, *preferred to*, and *may.* In particular, the three terms are used as follows:

(a) Must: the rule is applied only when its 'then' statement is fulfilled (e.g., for the business rule "*If patient's language is not Greek, then doctor's language must be*

[22] Participant refers to each constituent part of the FSR, such as a business role, the IO and a stakeholder.

*English",* if no *doctor* speaks English, then the *foreign patient* will not be assigned a doctor).

(b) Preferred to: if the 'then' statement of the rule cannot be fulfilled, then the participant of the 'if' statement should be asked if s/he accepts another option (e.g., if the *doctor* does not speak English, and there is one or more doctors who can speak other languages, then the *foreign patient* should be asked if s/he prefers a doctor who talks another language, e.g., French).

(c) May: if the 'then' statement of the rule cannot be fulfilled, then the user (without asking the participant of the 'if' statement, e.g. patient) can choose another option of his/her own preference.

The realizations (initializations) of interrelated business rules can be written in a formalized form, following an extended version of the FSR syntax, called *detailed FSR*. Writing the business rules as detailed FSRs makes easier the transformation process which is described in step six. The detailed FSR syntax uses adjectival and peripheral attributes. Adjectivals are positioned on the left of each participant and denoted by the Greek letter **α** (alpha), while peripherals, denoted by M, are positioned on the right and distinguished from their participants by the Greek letter **φ** (phi) **(**φ denotes the genitive case and corresponds to the word "of"). The detailed FSR pattern shown below is for single entities (e.g., one *Subject*, one *Receiver*, chosen here for simplification) and can take the form of more complex combinations of entities (participants):

$$\left\langle a_0^n S_1 \varphi \mathrm{M}_0^m \right\rangle \left\langle F \right\rangle \left\langle a_0^p IO \varphi M_0^q \right\rangle \left\langle FC \right\rangle :: SystemNotifies \left\langle a_0^s R_1 \varphi \mathrm{M}_0^t \right\rangle \left\langle FC_R \right\rangle$$

where *n, m* denote the numbers of adjectival and peripheral attributes of $S_1$, respectively, *p, q* denote the numbers of adjectival and peripheral attributes of IO,

respectively, and *s*, *t* denote the numbers of adjectival and peripheral attributes of $R_1$, respectively.

Below we provide an indicative example of the realization of the general business rule 1.1, written as a detailed FSR:

$$FSR_{Cre1}^{Pat} =< Receptionist, Patient > \langle Create \rangle < foreignPatient >$$

$$< Doctor\varphi Lang.English, Receptionist >$$

where *foreign patient = patient who does not know Greek*

(2) *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, for the above FSR, is 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.*

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

### 3.3.6 Create SRS document and semiformal models (DFDs class & use-case diagrams)

This step involves the use of transformation rules on the FSRs and the IO attributes in order to create the software requirements specifications. For the creation of the SRS document, class diagrams and DFDs, we will focus specifically on the rules per se. However, we will explain in greater detail, in the next chapter, use case modeling adaptation, because use case modeling is considered as an approach that attempts to cover apart from specification, which is the main concern of object and structure modeling, the stages of elicitation and analysis, by also focusing on the use of natural language. We will show how the entire NLSSRE methodology can be adjusted for use case model development and how it concludes with the construction of use case diagrams and use case specifications.

#### 3.3.6.1 Adaptation for creating Data Flow Diagrams

Specific rules are used to transform the FSRs and attributes of each IO to DFDs. The FSRs of creation, alteration, reading and erasure of each IO are grouped under one comprehensive function named *Manage <IO>*. The *Manage* functions constitute the functions of the 1st level DFD (see Figure 3.4), while the *Create*, *Alter*, *Read* and *Erase* functions of the FSRs under each *Manage IO* constitute the functions of the 2nd level DFD (see Figure 3.5). For the 3rd level DFD, the second level functions are decomposed to the CAREN sub-functions, according to Figure 3.2. For example, the 2nd level function *Create Prescription* is decomposed to *Enter Data* (incorporates the *Read* and *Compare* sub-functions) and *Save* (see Figure 3.6). The use of the Intended Recipient is used as a

rule to link functions at the same level (e.g., link *Manage Prescription* and *Manage Drug*). Another indicative rule is that the business roles of the FSRs correspond to DFD actors and are represented by a circle. Furthermore, for the functions *Create*, *Alter* and *Erase*, the business role(s) that appear on the left of the name of each function in the corresponding FSR syntax provide data input to the function, hence an arrow from each DFD *Actor* (business role) feeds the relevant function.

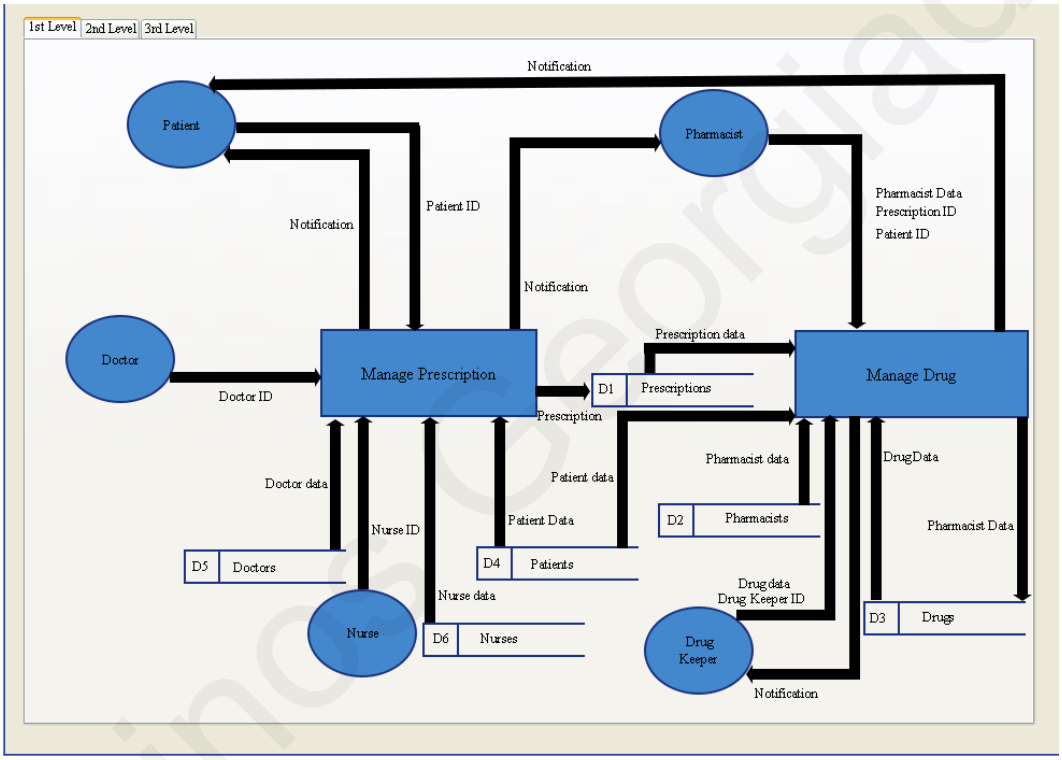


FIGURE 3.4 1ST LEVEL DFD CREATED AUTOMATICALLY BY NALASS.

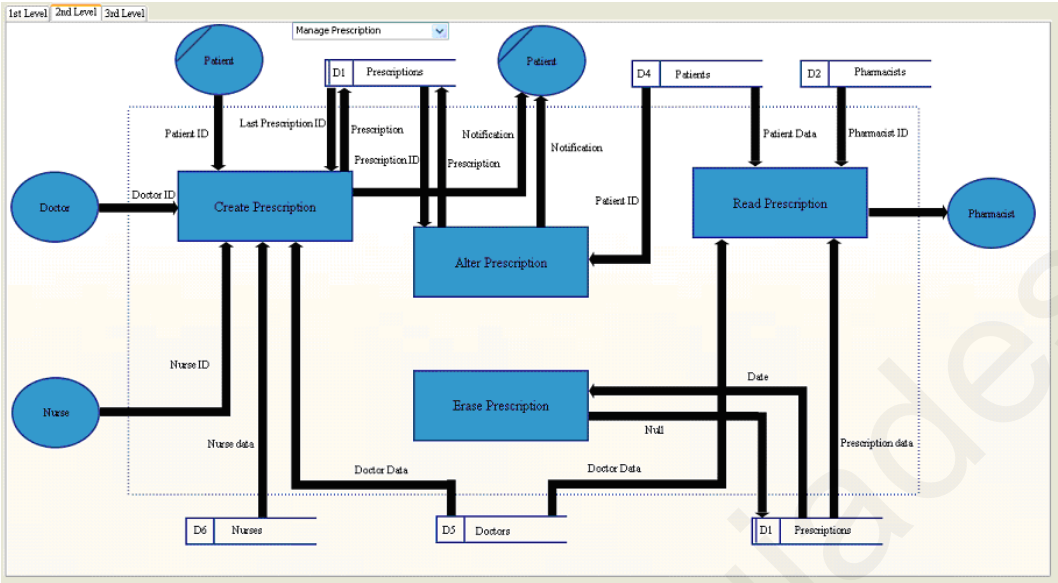


FIGURE 3.5 2ND LEVEL DFD CREATED AUTOMATICALLY BY NALASS.

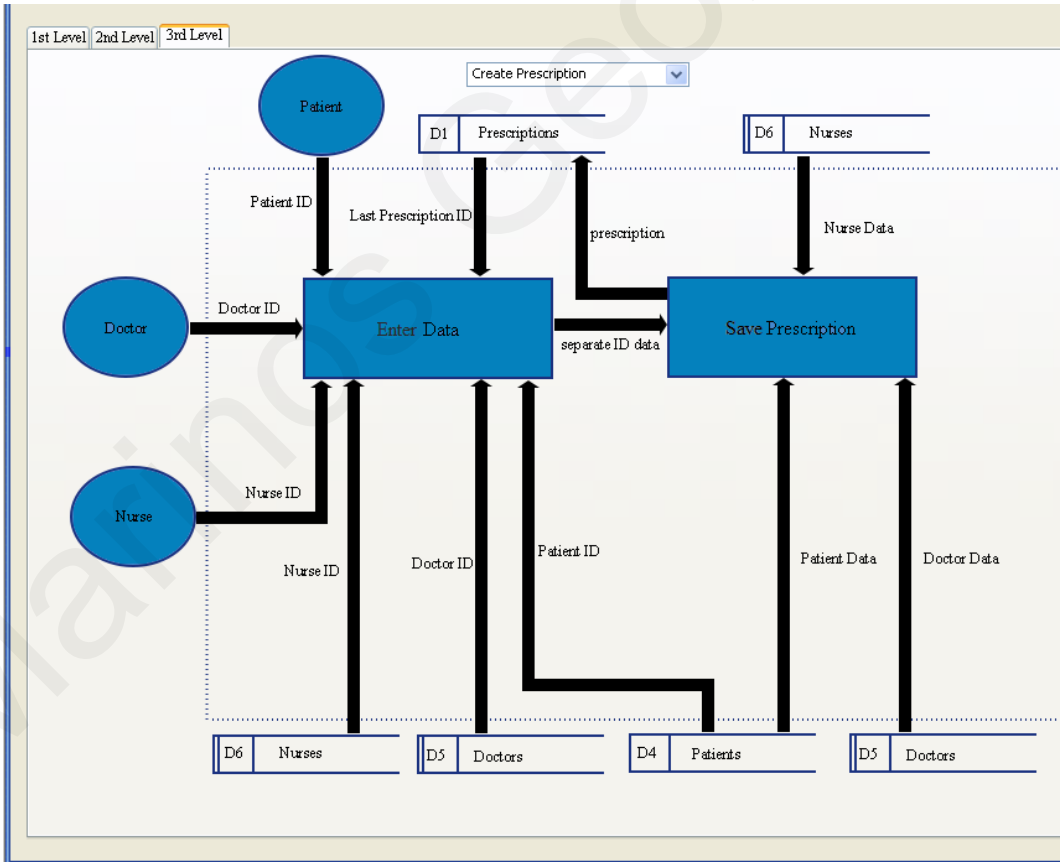


FIGURE 3.6 3RD LEVEL DFD CREATED AUTOMATICALLY BY NALASS.

The DFDs generated by NLSSRE and NALASS are based on the IOs, first, and, secondly, on the CAREN functions. Therefore, we can name them Object-Related Data Flow Diagrams *which are defined as data flow diagrams whose functions are applied on information objects (Information Objects).* Thus, ORDFDs consist of the CAREN functions.

Below we list the most basic rules of this transformation:

- The first level ORDFD will include all the *Manage IO* functions (figure 3.4). Functions are represented by a rectangle.
- The second level ORDFD will include all the 2nd level functions (*Create, Alter, Read, Erase)* of each first level function (*Manage IO)* as shown in Fig. 3.5.
- For the third level DFD, the second level functions are decomposed to the CAREN sub-functions, according to Figure 3.2.
- The functional roles *Creator*, *Accompaniment*, *Alterer*, *Intended Recipient*, *Experiencer* and *Notifiee* correspond to actors (or business actors or business roles) of a traditional DFD and are represented by a circle.
- For the functions *Create*, *Alter* and *Erase*, the business role (s)/ actors (s) that appear on the left of the name of each function, in its syntax, provide data input to the function, hence an arrow from each of these actors goes to the relevant function (e.g. from *Doctor* to *Create Prescription* - Fig. 3.5).
- For the *Read* function, in the 2nd level of decomposition, the business role of *Experiencer* receives the IO in a special format/layout for reading (viewing, listening, etc.). Hence an arrow from the *Read* function goes to the *Experiencer*

actor (business role) in the ORDFD as shown in Fig. 3.5 (*Read Prescription – Pharmacist*).

- The *Create, Alter* and *Erase* functions trigger a data flow from the relevant function to the relevant *datastore,* because the IO is changed and needs to be (re)stored; hence an arrow goes from each function to the datastore (e.g. from *Create Prescription* to *Prescriptions*).
- The *Create, Alter* and *Erase* functions trigger data flows from the relevant *datastore* (which is created because of these functions) to the relevant *function,* because the function needs to check the IO before altering it; hence an arrow goes from the datastore to each function (e.g. from *Prescriptions* to *Create Prescription*).
- The *Read* function triggers a data flow from the relevant datastore to the *Read* function; hence an arrow goes from the datastore to the function (e.g from *Patients* to *Read Prescription*).
- The entities that appear on the right of *SystemNotifies* in the syntax of the *Notification* function receive an arrow (data flow) from the relevant function which appears on the left of the *Notification* function (e.g. from *Create Prescription* to *Doctor, Nurse*, *Pharmacist,* and *Patient*).
- The use of the Intended Recipient is used as a rule to link functions at the same level, 1st or 2nd. The *Intended Recipient* of an IO needs to *Read* that IO. Thus a link from the relevant datastore of that IO to the *Manage IO* function of the new IO in which the *Intended Recipient* is involved as its *Creator* or *Alterer* needs to take place. E.g. the *Pharmacist* is the Intended Recipient of the Prescription as

shown in the syntax of *Create Prescription* (Fig. 3.3.3d), and the *Pharmacist* will *Read* the *Prescription* in order to *Manage Drug*. Hence a link from *Prescriptions* (datastore) to *Manage Drug* is created (Fig. 3.4).

#### 3.3.6.2 Adaptation for creating Class Diagrams

Specific rules are used to transform the FSRs and attributes of each IO to class diagrams. Each IO is transformed to a Class, and its CAREN functions become the methods of the class. The attributes of each IO are those defined during the fourth step of the methodology, and can be refined and codified within this step. As an example, information about *Doctor,* which constitutes one or more attributes of the IO *Prescription*, can be refined and codified to the specific attributes of *Doctor ID, Doctor Signature, Doctor Name,* and *Doctor Surname*. Further rules regarding the relationships between classes and cardinality are realized by NALASS, such as the rule which states that an association relationship exists between the IO and each business role which also constitutes an IO, in the same FSR. Additional rules state that there should be a *one-to-many association* between *Creator (e.g., Doctor* in the *Create Prescription* FSR*)* and *IO (Prescription)* (apart from rare cases where there could be more than one creators for the same IO), a *one-to-many association* between the client business role (*Patient – otherwise called external accompaniment role*) and the IO (*Prescription),* and there could be a many-to-many association between *Creator* (*Doctor*) and *internal Accompaniment* (*Nurse or Counselor*). Figure 3.7 shows the *Prescription* and *Drug* classes, with their attributes (types) and relationship, as generated automatically by NALASS.

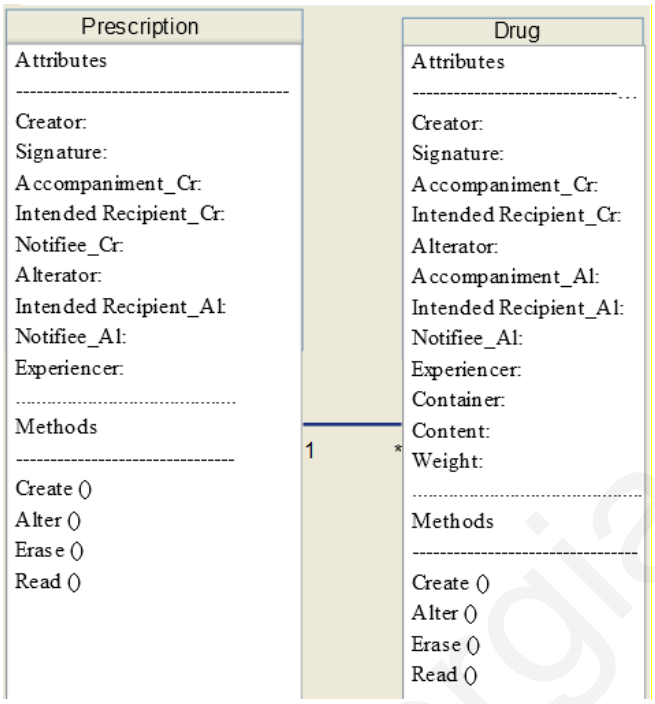


FIGURE 3.7 GENERAL FORM OF A CLASS DIAGRAM CREATED AUTOMATICALLY BY NALASS

#### 3.3.6.3 Adaptation for creating the SRS document

This transformation process (fig. 3.8) receives as inputs the IOs, their attributes, complete FSRs (including detailed ones related to business rules) and constraints, the SRS template that determines the organization and formatting of the SRS document, and the rules to convert the aforementioned inputs into a well-structured SRS document, written in structured, less semi-formal NL, thus effectively hiding from the users the semi-formal organization of requirements. The NALASS tool reads the template and applies: (a) rules for the layout and formatting of the new SRS document, related to additional language refinements of the text, fonts type and size, line spacing, etc.; (b) substitution rules, by replacing the template variables included inside "< >" with the

appropriate values for the identified IOs, the FSRs participants, attributes, business rules and functional conditions.

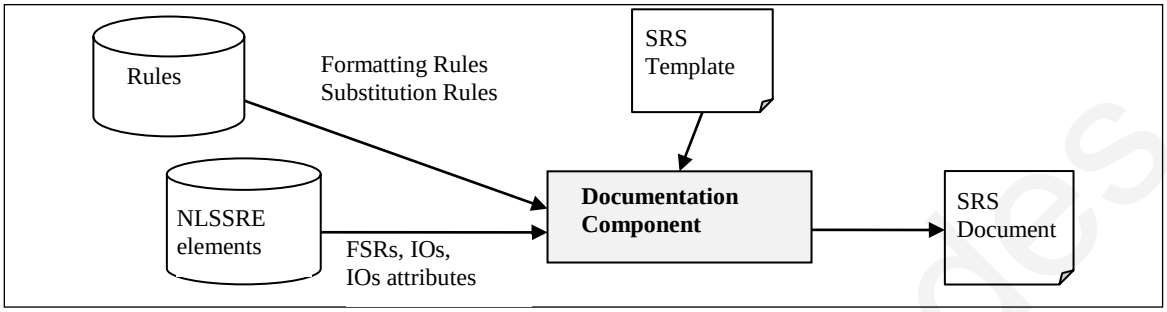


FIGURE 3.8 CONFIGURATION OF NALASS'S DOCUMENTATION COMPONENT

Table 3.5 shows a portion of the main focus of the NLSSRE SRS (some elements such as business rules are not presented for simplification), which corresponds to *Section 3 Specific Requirements* of the IEEE SRS document template (IEEE, 1998), which is itself the most substantial section in the SRS structure, accounting for about 70% of the SRS template definition; the template can take different organizational types, such as functional or object-oriented. All other items, such as the glossary of terms and the initial snapshots of the software system's user interface can be appended by the requirements analyst in the document file generated by NALASS.

The way the template is built and processed does not need any further grammatical and syntactical checks, since, for example, plural is covered by the use of "(s)" at the end of the noun, the third person singular verb form is covered by the use of "(s)" at the end of each verb, and for the genitive case, we use "of" instead of "'s" for simplification and avoidance of mistakes. Future developments include the use of a grammar checker

accompanied by a dictionary that will make such checks, thus substituting the grammatical refinements currently performed on the text as dictated by the template.

NALASS can export the SRS document to either HTML or rich-text format (RTF). This allows the requirements analyst to generate and show the specification outside NALASS's interface either in electronic or printable format.

TABLE 3.5 PORTION OF THE NLSSRE SRS TEMPLATE ON THE LEFT AND ITS CORRESPONDING REALIZATION FOR THE HOSPITAL INFORMATION SYSTEM CASE STUDY ON THE RIGHT.

| **Specific requirements Template (organized by object)** | **Specific requirements (organized by object)** |
|---|---|
| ○ Classes/Objects<br>☐ <IO 1><br>• Attributes (direct or inherited)<br>○ FSR Attributes<br>▪ Cr: <Cr 1> ID, <Cr 2> ID, …, <Cr n> ID<br>▪ Ac: <Ac 1> ID, <Ac 2> ID, …, <Ac n> ID<br>▪ No: <No 1> ID, <No 2> ID, …, <No n> ID<br>▪ <IR 1> ID, <IR 2> ID, …, <IR n> ID<br>▪ …….<br>○ Physical Attributes<br>……<br>• Functions (direct or inherited)<br>○ Create <IO 1><br>▪ Description: <Cr 1> , …., <Cr n> create(s) <IO 1> with the assistance of <Ac 1>, …, <Ac n> .<br>▪ Details:<br>▪ If <At 1>: <InputValue> is True for "<Constraint At 1>", then record <At 1>: <InputValue>. If False, then show message "<At 1>: <InputValue> " is not valid.<br>…<br>○ Alter <IO 1> …<br>○ Read <IO 1> …<br>○ Erase <IO 1> …<br>• Messages (notifications received or sent)<br>○ System notifies <No 1>, …, <No n>, <IR 1>, …, <IR n> that <IO1> is created.<br>○ …<br>☐ <IO 2><br>… | ○ Classes/Objects<br>☐ <Prescription><br>• Attributes (direct or inherited)<br>○ FSR Attributes<br>▪ Cr: Doctor ID<br>▪ Ac: Counselor ID, Patient ID<br>▪ No: Doctor ID, Patient ID<br>▪ IR: Pharmacist ID<br>▪ Doctor Name<br>▪ ……<br>○ Physical Attributes<br>……<br>• Functions (direct or inherited)<br>○ Create Prescription<br>▪ Description: Doctor create(s) Prescription with the assistance of Counselor, Patient.<br>▪ Details:<br>▪ If Doctor Name: <InputValue> is True for "Only alphabetic characters are allowed", then record Doctor Name: <InputValue>. If False, then show message "Doctor Name: <InputValue> " is not valid.<br>…<br>○ Alter Prescription ...<br>○ Read Prescription …<br>○ Erase Prescription …<br>• Messages (notifications received or sent)<br>○ System notifies Doctor, Patient, Pharmacist that Prescription is created.<br>○ …<br>☐ Drug<br>… |

## *3.4 Requirements Change*

As the requirements definition is developed, the analyst normally develops a better understanding of users' needs. This feeds information back to the user, who may then propose a change to the requirements. Furthermore, it may take several years to specify and develop a large system. Over that time, the system's environment and the business objectives change, and the requirements evolve to reflect this. From an evolution perspective, requirements fall into two classes (Sommerville, 2008):

*a. Enduring requirements:* These are relatively stable requirements that derive from the core activity of the organization and which relate directly to the domain of the system. For example, in a hospital, there will always be requirements concerned with patients, doctors, nurses and treatments.

*b. Volatile requirements:* These are requirements that are likely to change during the system development process or after the system has been become operational. Volatile requirements fall into four classes:

- *Mutable requirements*: Requirements that change because of changes to the environment in which the organization is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected. Or new government healthcare policies are introduced.
- *Emergent requirements*: Requirements that emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements. New requirements may emerge from new stakeholders who were not originally consulted.

- *Consequential requirements*: Requirements that result from the introduction of the computer system. Introducing the computer system may change the organizations processes and open up new ways of working which generate new system requirements. - *Compatibility Requirements* that depend on the particular systems or business processes within an organization. As these change, the compatibility requirements on the commissioned or delivered system may also have to evolve.

Therefore even if NLSSRE could settle all that and could get an accurate and stable set of requirements, the latter may change. This is, of course, more likely to happen with large projects where many stakeholders are involved—therefore more requirements need to be elicited—and a longer time framework is required.

To solve the volatility of emergent requirements, which can be caused by stakeholders who are not identified or omitted requirements by the analyst, NLSSRE provides a well-structured elicitation process that attempts to identify the stakeholders and especially the end users of the system, as well as the information objects. The data flow table, the specific questionnaires, and later on the specific questions derived from the predefined types of functions minimize the probabilities of specifying the wrong requirements. However, for large systems, there might be the case that some key stakeholders are missed or some requirements are not derived from the stakeholders, including, for example, new IOs or, mainly, functional conditions and attributes (also non-functional requirements, however NLSSRE does not focus on them). Therefore for large projects, we recommend the application of short several NLSSREE cycles. The stakeholders and requirements identified in each cycle will guide the analyst to identify new stakeholders and new requirements in every next cycle. For example, during the first cycle, the analyst

will identify the major stakeholders and the most significant IOs, s/he will analyze them to define the IOs CAREN functions and relations upto developing Use Case Specifications and diagrams. The first cycle will result to the identification of new stakeholders, especially end-users, and IOs, and the analyst will need to perform another cycle to elicit requirements from the new stakeholders and analyze them accordingly.

To handle the volatility of mutable, consequential and compatibility requirements, that is, to actually manage change after software development, the SRS document must be easily modifiable. To make the requirements document more modifiable, related requirements should be grouped together and a requirement should not appear in more than one place in the document. The requirements document should also have a table of contents and cross-references if necessary. As a rule of thumb, the lower the number of redundant requirements in the SRS document the higher the level of modifiability. We consider that the SRS produced by NLSSRE is well-structured, complete, correct and modifiable, since one of the main advantages of NLSSRE, due to its formalized and specific nature, is that it generates non-redundant requirements—and redundancy is a major obstacle to modifiability (more about how NLSSRE handles modifiability is described in Chapter 6). Of course, some additional traceability policies need to be specified or enhanced. When changes are proposed, we have to trace the impact of these changes on other requirements and the system design. Traceability is the property of a requirements specification that reflects the ease of finding related requirements. There are three types of traceability information that need to be considered:

- *Source traceability* information links the requirements to the stakeholders who proposed the requirements and to the rationale for these requirements. When a

change is proposed, we use this information to find and consult the stakeholders about the change. This information can be derived from NLSSRE through the FSR specifications where the internal business roles (creator, alterer) are actively involved in the requirements. Additional notes may be added to the specification document.

- *Requirements traceability* information links dependent requirements within the requirements document. We use this information to assess how many requirements are likely to be affected by a proposed change and the extent of consequential requirements changes that may be necessary. Again, in NLSSRE, through the use of FSRs, we can identify which business roles are involved in the different FSRs; such FSRs are usually interconnected.
- *Design traceability* information links the requirements to the design modules where these requirements are implemented. We use this information to assess the impact of proposed requirements changes on the system design and implementation. In NLSSRE, class diagrams and data flow diagrams are linked to the FSRs and the IOs, however additional traceability information could be provided, especially with the enhancement of NALASS.

Furthermore, for small systems we could investigate the case of using traceability matrices, and for large systems, as aforementioned, the enhancement of the dedicated CASE tool for traceability.

## *3.5 Chapter Summary*

This chapter presented NLSSRE and showed how it aims to formalize and automate the major activities of RE, including requirements discovery, analysis and specification. NLSSRE is designed so that the analyst is guided in advance, through a step-by-step approach, what specific types of data, functions, business rules and conditions to use and search for, what questions to ask, in what specific way to analyze the answers to the questions, and how to write them in a specific formalized way. We explained the constituent parts of the methodology, which are its architecture (underpinning background), its application steps and techniques, a modeling language for representing requirements as formalized sentences (FSRs) and the NALASS software tool that automates the entire process.

# 4 Adaptation for formalizing use case development

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 (2000), 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 (Dias, 2008).

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 methodology as part of a series of steps for the development of the UC model, which will be described in the following paragraphs. Formalization is mainly achieved with the use of predefined types of use cases—corresponding to the CAREN functions—and actors—corresponding to business roles—, formalized sentential patterns—corresponding to FSRs—, 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 NALASS which is also described through indicative examples.

In particular, the steps of our adapted 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)

## *4.1 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.

Since each IO corresponds to a UC module, then 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 paragraphs of this section. Figure 4.1 shows an example

of a use case diagram (UCD) corresponding to the Appointment UC module of the HIS case example.

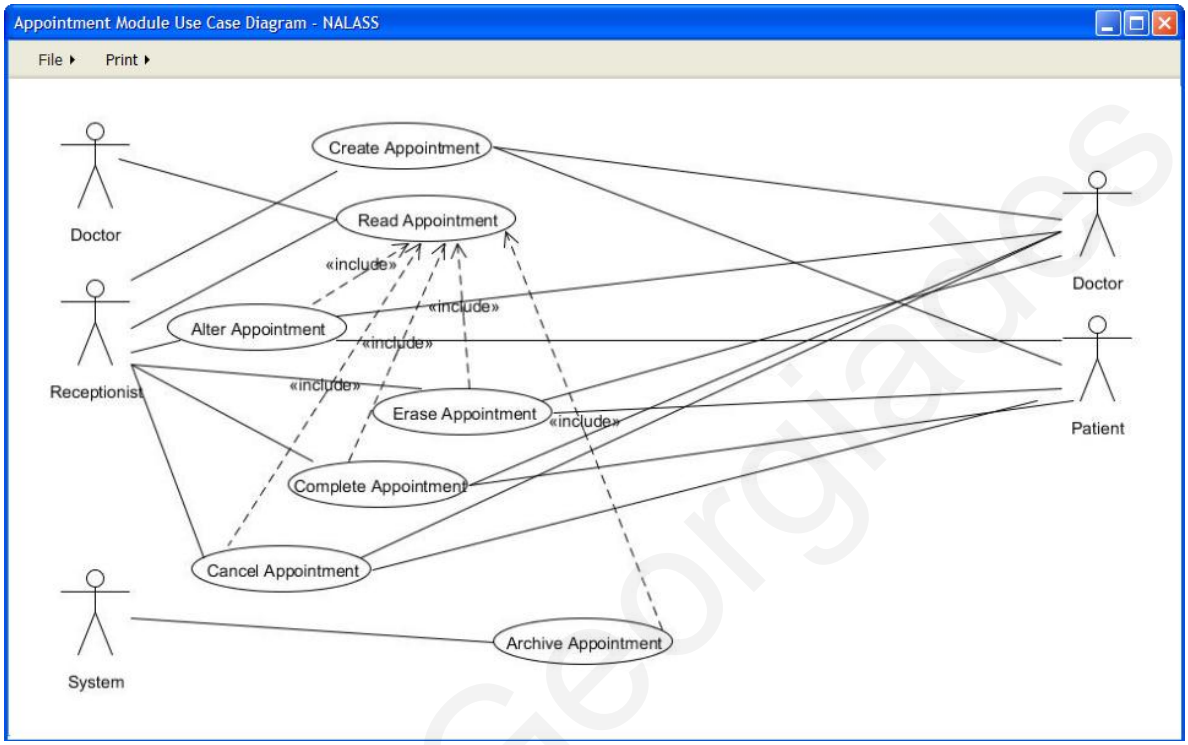


FIGURE 4.1 THE USE CASE DIAGRAM OF THE APPOINTMENT MODULE, AS CREATED BY NALASS.

## *4.2 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. 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. Accordingly

we have the use case types *Create IO*, *Alter IO* (including alter-related use case types explained later), *Read IO*, and *Erase IO*.

As mentioned in step 3 of the NLSSRE methodology (section 3.3), 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) (Podeswa, 2005). We do not focus on programmer's level requirements or abstract-level requirements, the latter of which are generally represented by business use cases. According to Podeswa (2005) and de Cesare (2003), 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".

For the use case approach, similarly to NLSSRE perspective, the formalization concept of the 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 implemented as a system or a business use case, 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 participates in the enrollment. *Student* will be also a different UC module, as it is a different IO.

Below, for each use case type, we describe adaptation and other issues such as relationships between use cases and what actions can be derived for each basic use case.

**UC Create IO**: 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.

**UC Alter IO**: During the execution of this UC, the actor can change the existing values of the attributes of an IOi. Similarly to the *Alter* CAREN function class, a significant attribute that changes during alteration of an IOi is the attribute *State*. We will follow the same examples used for earlier to show how the UC *Alter IO* is adjusted. The change from one state to another (e.g., from *Pending* to *Complete*), for a particular IO, often derives a new 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 use cases *Alter IO* or *Create 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 *Alter* and *Create* 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 the *Create* UC or the basic[23]*Alter* UC, we recommend to represent this self-contained information (pre-conditions, post-conditions, actions) as a new 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* 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

[23] 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.

in generalization relationships[24] between use cases, such as those depicted in the example of figure 4.2 where the student, due to the nature of his/her role, can move to different states during his/her studies.

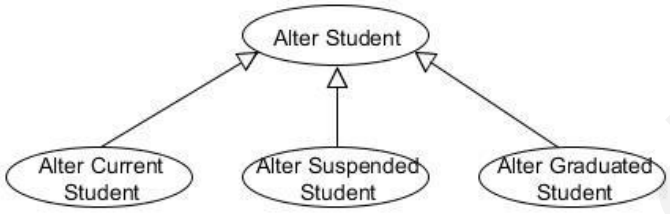


FIGURE 4.2 GENERALIZATION RELATIONSHIPS

**Read**: Forms usually need to be read when an end-user primary actor[25] creates a new IOi or changes the state of an existing IOi. The reading process should be represented as an "include"[26] use case *Read IO* for the use cases *Alter IO*, *Cancel IO*, *Complete IO*, etc., as depicted in figure 4.3 and table 4.2 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 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 4.1 action

[24]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 (Zielczynski, 2007).

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

[26]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 (Coleman, 1998). 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.

2 (in step 7). This issue is discussed further in step 7, on constructing the UC specifications.

*Read IO* may also be initiated directly by a primary actor, when the latter needs to read information provided by an existing IOi form, e.g., by receptionist and doctor, as shown in figure 4.1, or by doctor and pharmacist, as shown in figure 4.3.

For inter- and intra-Reports, as discussed in 3.3 (step 3), 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* (figure 4.3), 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.

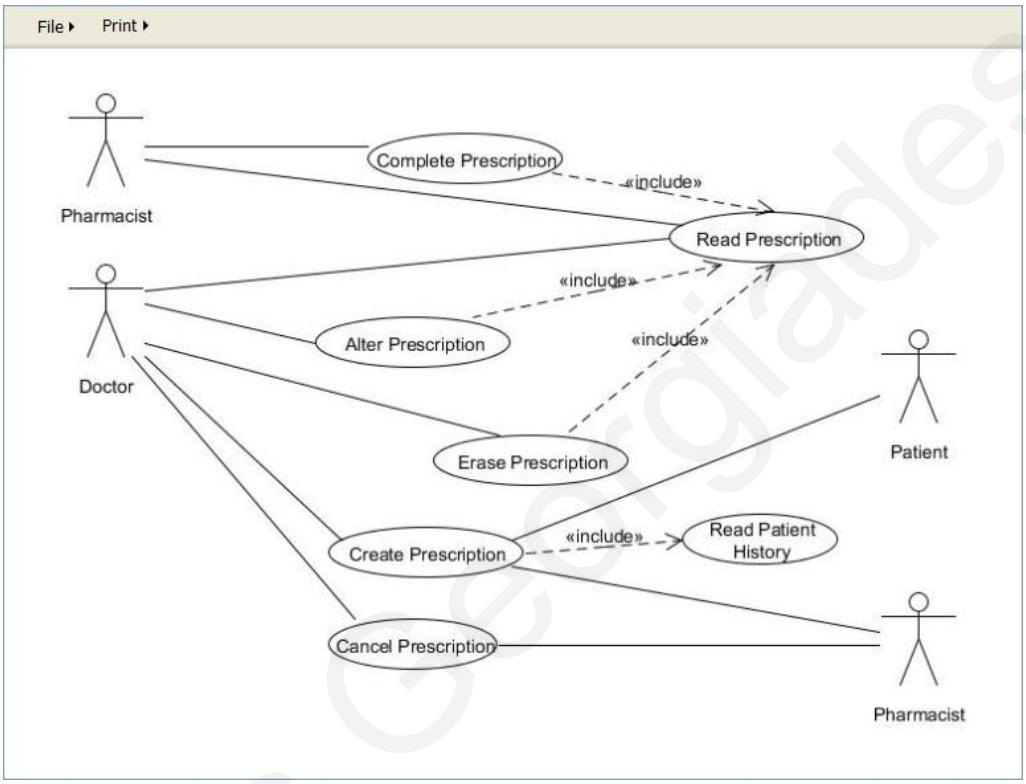


FIGURE 4.3 PART OF THE USE CASE DIAGRAM OF THE PRESCRIPTION MODULE, WHICH IS CREATED AUTOMATICALLY BY NALASS.

**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.

**Notify**: 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. 4.4) included in and invoked by their including UC.

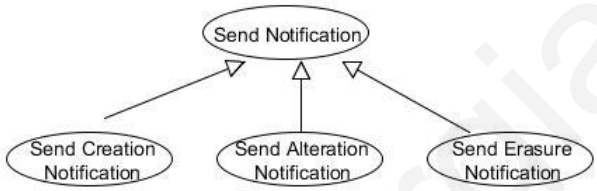


FIGURE 4.4 UC SEND NOTIFICATION MAY BE SPECIALIZED ACCORDING TO THE TYPE OF USE CASE WHICH INVOKES IT (E.G., UC CREATE IO INVOKES UC SEND CREATE NOTIFICATION)

## *4.3 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 (i) the system *end-users* who use the system in an operational sense and interact directly with it; (ii) *customers* who are external users that use the system to buy products and services, or search for information relating to products; or (iii) *trusted external users* who have a particular relationship with the organization and may be given specific privileges in the system (suppliers are examples of such users). In contrast, other stakeholders refer to: (i)

*business users* who are interested in the system's functions and output, as support for achieving their business objectives; (ii) *managers* who are responsible for the strategic use of IT in their business unit and for the overall strategy of the organization and the way information systems can both support and enable the strategy; (iii) *information users* who are external users that use the system not to buy anything but mainly to be informed or provide information about other system users or entities (a patient's relative in a hospital IS is an example of such a user); and (vi) *Shareholders* who are external users that have invested in the organization and have financial interest (Avison and G. Fitzgerald, 2003). 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 (2009) and Sybase (2002), 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.

Similarly to NLSSRE, 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 regarding 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 other stakeholders. 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. The functional roles actors can play and their identification process through questions are similar to what is performed by NLSSRE, so we will not repeat them here. What is worth mentioning here is the identification of new use cases and relationships from involvement of the accompaniment role.

In particular, the collaboration between a primary actor and an accompaniment can derive both include and extend[27] relationships, where extending or included use cases are invoked by their base use cases and triggered by the accompaniments. These use cases

[27]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].

are called complementary. For example, as illustrated in figure 4.5—and earlier in section 3.3 (step 3)—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.

In summary, the entities that play the role of creator, accompaniment, alterer, experiencer or intended recipient are identified as actors (primary or secondary). Every other stakeholder plays the notifiee role. Actors should appear in the UC diagrams and be mentioned in the UC specifications. Other stakeholders should not appear in the UC diagrams but should be mentioned in the UC specifications. We have also indicated that relationships between actors in one use case—such as that between creator and accompaniment—may lead to the identification of new use cases.

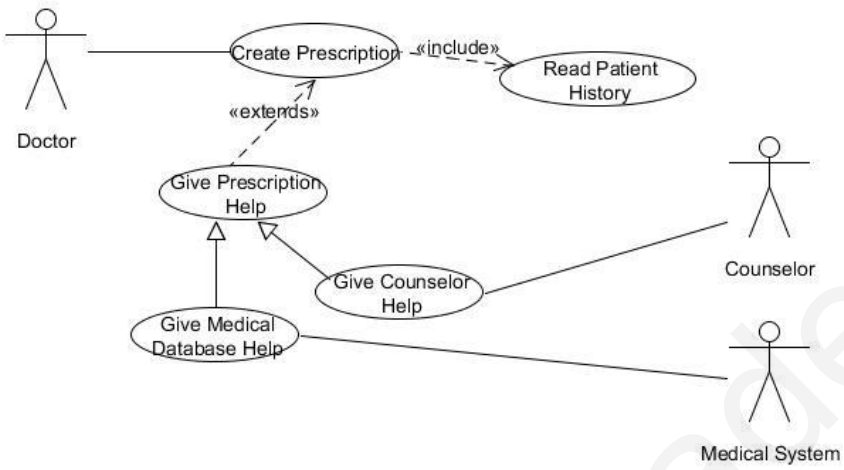


FIGURE 4.5 COMPLEMENTARY USE CASES DERIVED FROM RELATIONSHIPS BETWEEN ACTORS (THIS IS THE OTHER PART OF THE PRESCRIPTION MODULE DEPICTED IN FIGURE 4.3).

## *4.4 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 FSRs. As discussed in earlier sections, 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. We prefer to name the FSRs for use case

modeling as FSUCs, just for terminology purposes, to show that we deal with use case modeling. Therefore, a formalized sentential use case pattern FSUC is a structured, semi-formal way of writing a use case of an IS, which corresponds to the FSR pattern provided by NLSSRE. 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 other stakeholders, and *Functional Condition FC* is a clause that adds further information about the function, commonly by establishing the circumstances 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.

Functional conditions may derive business rules that influence the actions of the UC specifications (e.g., the FC 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 some indicative rules that illustrate how the FSUC can assist in creating the UC diagrams:

1. The complementary sentences may be used as included or extending use cases, accordingly, to their base use cases which are expressed by the main sentences (e.g., the UC *Create Prescription* is extended by the UC *Counselor Give Prescription Help*.)
2. As already mentioned, functional conditions may derive business rules that influence the actions of the UC specifications, or they may also derive "include", "extend" or "generalization" relationships.
3. The verb and the indirect object of the FSUC make the name of the UC, such as *Create Prescription*, and *Give Prescription Help*.
4. In a *Create, Alter* or *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.
5. The actors on the right of the first actor (primary), in the *Actor* group are accompaniments (not notifiees, which should not appear at all) and are therefore secondary actors and should be positioned on the right of the use cases of the UCD.

## *4.5 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. Figure 4.6 below shows a part of the subsystem *Hospital Reception* composed of the UC modules *Patient* and *Appointment*[28].

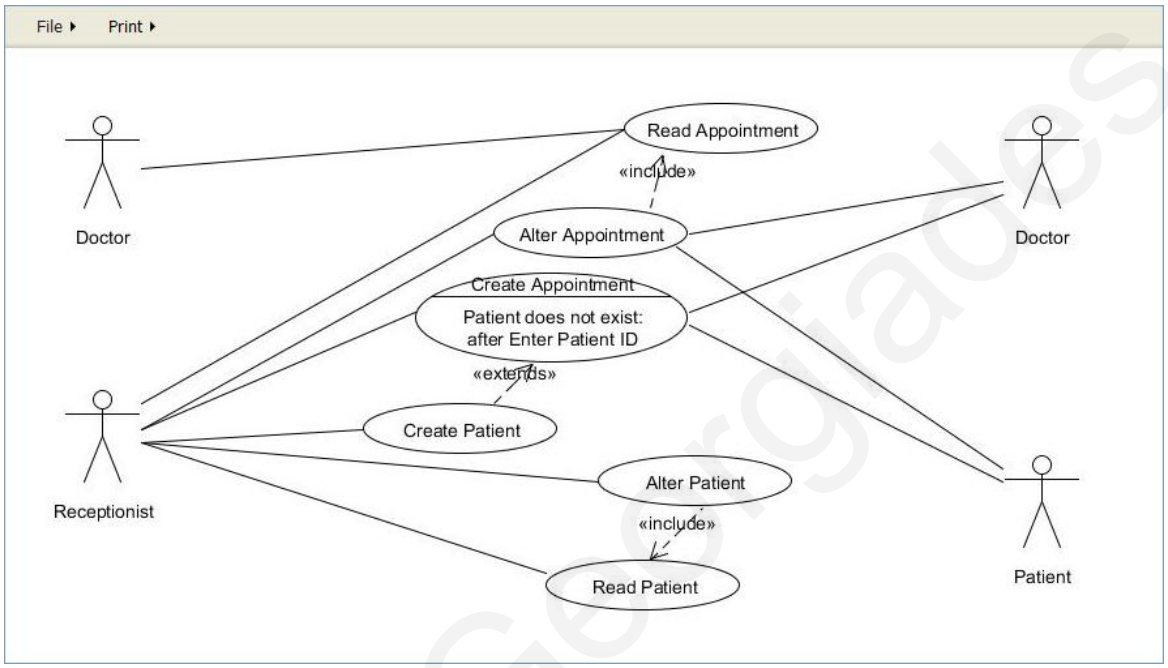


FIGURE 4.6 HOSPITAL RECEPTION SUBSYSTEM UCD DEVELOPED FROM 2 DIFFERENT MODULES: PATIENT AND APPOINTMENT.

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 dissertation. 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

[28] 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.

*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 example, 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[29] 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

[29] 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

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 figure 4.7 below.

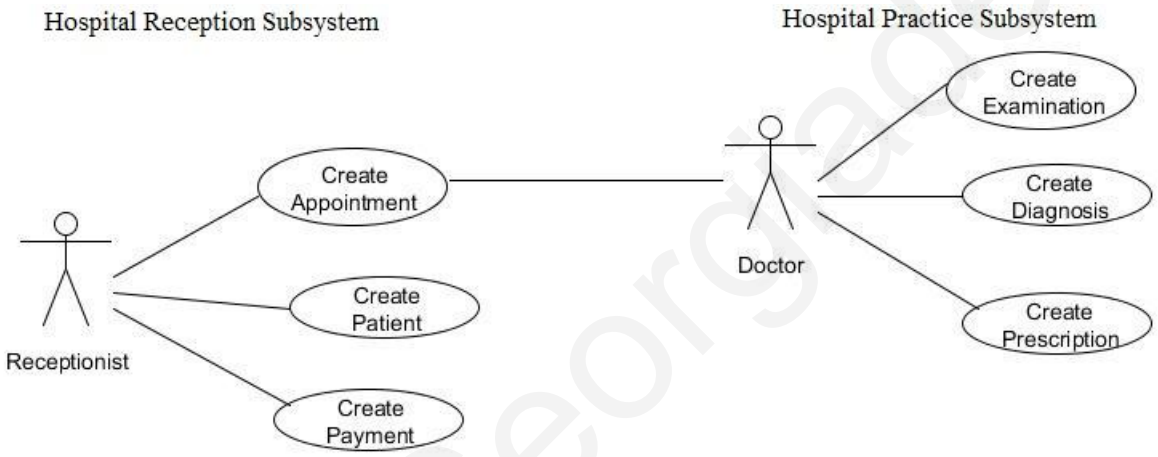


FIGURE 4.7 DIFFERENT SUBSYSTEMS ARE LINKED TOGETHER TO CONSTRUCT THE ENTIRE SYSTEM'S UCD.

## *4.6 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 (Dias, 2008). 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 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 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 (Alhir, 2002).

## *4.7 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*, *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 4.1.

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 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 4.4.

*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 4.4.

*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. (2008), 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[30], 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 4.1 actions 3, 4, and 6; table 4.4 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 4.2 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 4.2 action 2).

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

- *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 4.1 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 4.1 action 7; table 4.2 action 9; table 4.4 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 4.1–4.3 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 4.4 (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.

TABLE 4.1 BASIC FLOW PATTERN FOR UC CREATE IO.

| | |
|---|---|
| 1. | <Actor> selects create <IO>. |
| 2. | System displays new <IO> creation form, including required and optional fields. |
| 3. | <Actor(s)> enter(s) <IO><IO.attribute.value>. |
| 4. | System must check <IO><IO.attribute.value>. |
| 5. | <Actor> 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>*. |

Repeated

TABLE 4.2 BASIC FLOW PATTERN FOR UC ALTER IO.

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

Repeated

TABLE 4.3 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. |

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

*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 figure 8, 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 figure 4.6, 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 4.4 includes two extension points at actions 6.1 and 7.1 of the basic flow.

TABLE 4.4 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 prescription. |
| **Preconditions** | 1. Examination is at Complete state (*from* UC *Create Examination*).<br>2. Doctor is authenticated (*from* UC *Doctor Creates Authentication*). |
| **Actors** | Doctor (Primary), Patient, Pharmacist (Secondary) |
| **Stakeholders** | Patient's Relative |
| **Post-Conditions** | Prescription is in *Pending* state |
| **Flow of Events** | 1. Doctor selects create Prescription by clicking on 'create prescription' button.<br>2. System displays new prescription creation form, including required and optional fields.<br>3. Doctor Patient enter(s) Patient ID.<br>3.1. The System checks Patient ID.<br>4. Doctor enter(s) Drug Name.<br>4.1. The System checks Drug Name. [Extension point: UC 22 Get prescription help]<br>5. Doctor enter(s) Drug Dosage.<br>5.1 The System checks Drug Dosage. [Extension point: UC 22 Get prescription help]<br>12. Doctor clicks on the Submit button.<br>12.1. The Doctor adds Doctor's digital signature to the Prescription (BRU.001)<br>12.2. The System adds a unique identifier to the Prescription. (BRU.002)<br>12.3. The System saves the Prescription in the database.<br>13. The System notifies the Doctor, Pharmacist, and Patient that Prescription is created *via UC 15.*<br>14. Use case ends. |
| **Exception condition** | 3.1. The System displays 'Invalid Patient ID' message, if patient ID is incorrect. Prescription cannot be saved.<br>4.1. The System displays 'Invalid Drug Name' message, if Drug Name is incorrect. Prescription cannot be saved.<br>5.1. The System displays 'Invalid Drug Dosage' message, if Drug Dosage is incorrect. Prescription cannot be saved.<br>……<br>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.<br>BRU.002: The Prescription identification number must comply with the format specified by NEHTA. |

## *4.8 Chapter Summary*

This chapter presented how the NLSSRE methodology can be adapted to formalize and automate the development of the use case model. The major steps of the adaptation process involve formalization of the process of identifying the UC elements and formalization of the use case specification template with the main focus on its transactions flow sections. Formalization is mainly achieved with the use of predefined types of use cases—corresponding to the CAREN functions—and actors—corresponding to business roles—, formalized sentential patterns—corresponding to FSRs—, 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 NALASS which was also described through indicative examples.

# 5 The NALASS Tool

To reduce the time required for the manual application of the NLSSRE methodology, and also to provide a friendly graphical environment for the Information Systems (IS) analyst, a software tool is required. Therefore, we created NALASS (Natural Language Syntax and Semantics)[31], a supporting software tool that automates all the stages of the NLSSRE methodology, including requirements discovery, analysis and specification. For the requirements discovery stage, specific sets of questions are automatically created based on the specific predefined types of data attributes and patterns of formalized sentences that are given in advance; for the requirements analysis stage, the requirements are automatically organised and classified based on the same types of data attributes and patterns; and for the specification stage, the tool can automatically generate Object Related Data Flow Diagrams, Class Diagrams, Use case specifications and diagrams, and the Software Requirements Specification (SRS) Document.

NALASS includes 7 architectural components as depicted in figure 5.1: (i) the FSRs component that uses the predefined and other manually entered FSR patterns as well as the identified IOs to generate, on one hand, the FSR patterns for each IO, and, on the other hand, the complete FSRs fed with the received answers; (ii) the Attributes component that uses predefined and other manually entered types of attributes and the identified IOs to generate the attributes types for each IO, on one hand, and the complete attributes formed by the received answers, on the other hand; (iii) the Questions component, which

[31] The CASE tool and its manual can be obtained upon request to marinos@studyhood.com.

processes the elements of the FSRs patterns and attributes for each IO, to generate the question sets (for each IO) to be submitted to the user; and

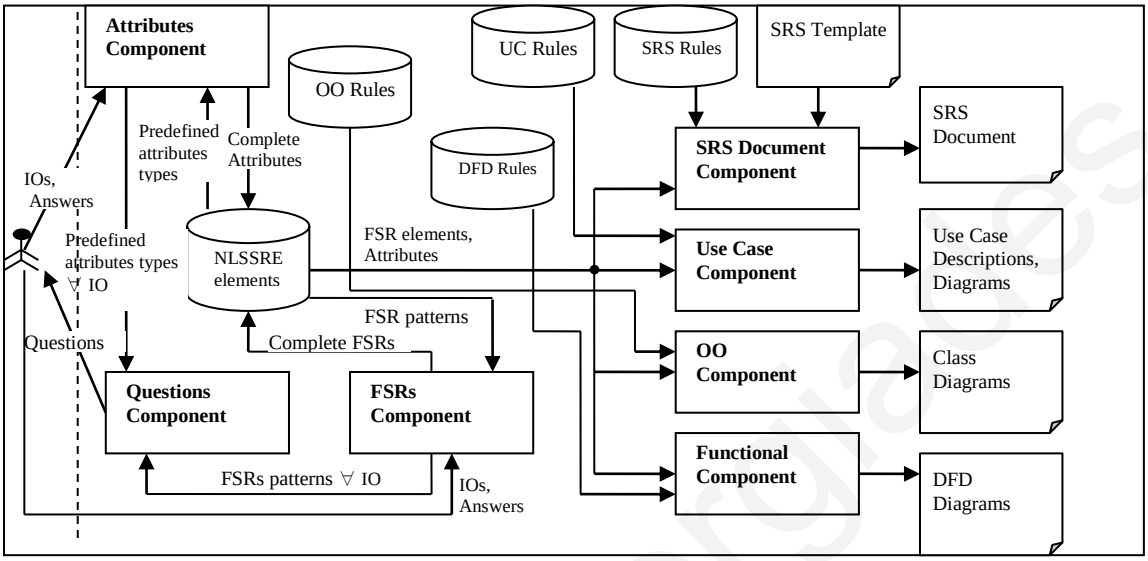


FIGURE 5.1 CONFIGURATION OF NALASS.

(iv), (v), (vi) and (vii) the Documentation, Use Case, Object Oriented and Functional components that process the elements of the completed FSRs, the completed attributes and specific rules to automatically generate the SRS document, Use Case specifications and diagrams, Class diagrams, and DFDs, respectively. NALASS also allows the analyst to enter users and roles of the IS, and it automatically builds a Data flow table, from which the analyst will be able to identify the actual IOs.

During the description of the methodology, in chapters 3 and 4, we illustrated the application of the above components. In particular, the application of the FSR component was mainly illustrated in section 3.3.3, the Atributes component in section 3.3.4, the Questions component in 3.3.3-3.3.5, the Documentation, the OO and Functional components in 3.3.6, and the Use Case component in chapter 4. Therefore, in this chapter

we will provide a summary of the above components through the presentation of the major interface components of the tool, namely Administration, Plan and Execution. In the *Administration* component, the analyst can create/add new types of IS elements, such as FSR patterns and data attributes that may apply to any project, as reusable elements. Figures 5.2 and 5.3 are screenshots of adding a new FSR class and editing an exisitng one, respectively, while figures 5.4 and 5.5 show how we can add and edit participants for each FSR class (e.g., each Create FSR class will have a Creator, and Accompaniment, an Intended Recipient, etc.). The latter two figures show that the analyst can define a number of parameters for the participants of each FSR class. For example, the parameter shape denotes the notation of the participant in a diagram, while the parameter position denotes the position of the participant in the FSR pattern, which will facilitate the correct construction of an FSR and the application of rules to build the diagrammatic notations and the SRS document.

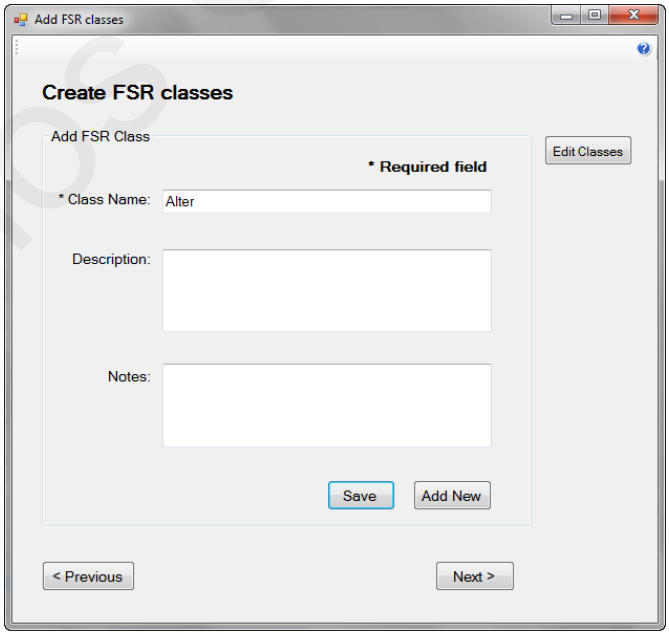


FIGURE 5.2 ADDING A NEW FSR CLASS

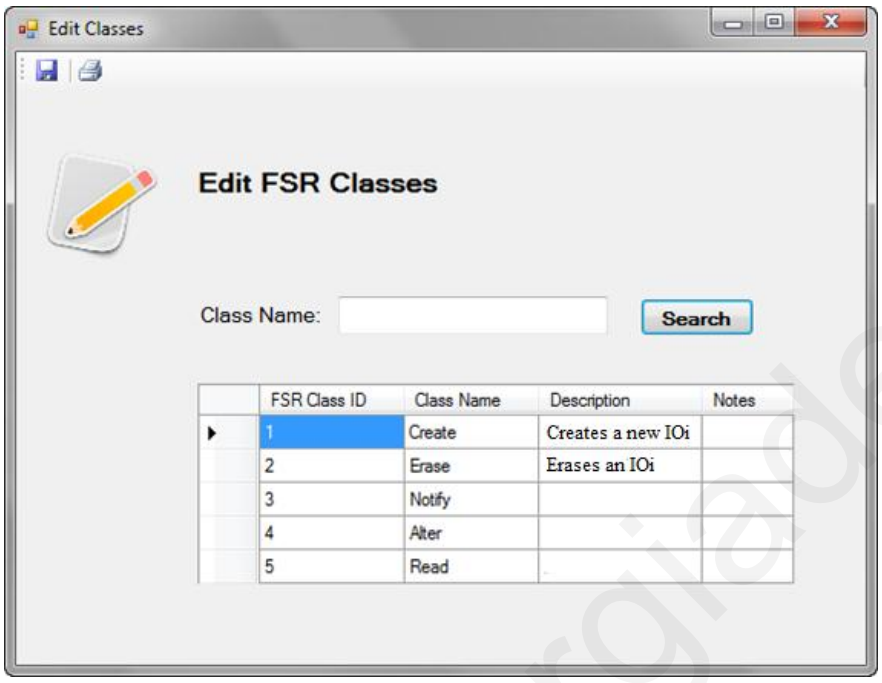


FIGURE 5.3 EDITING AN FSR CLASS

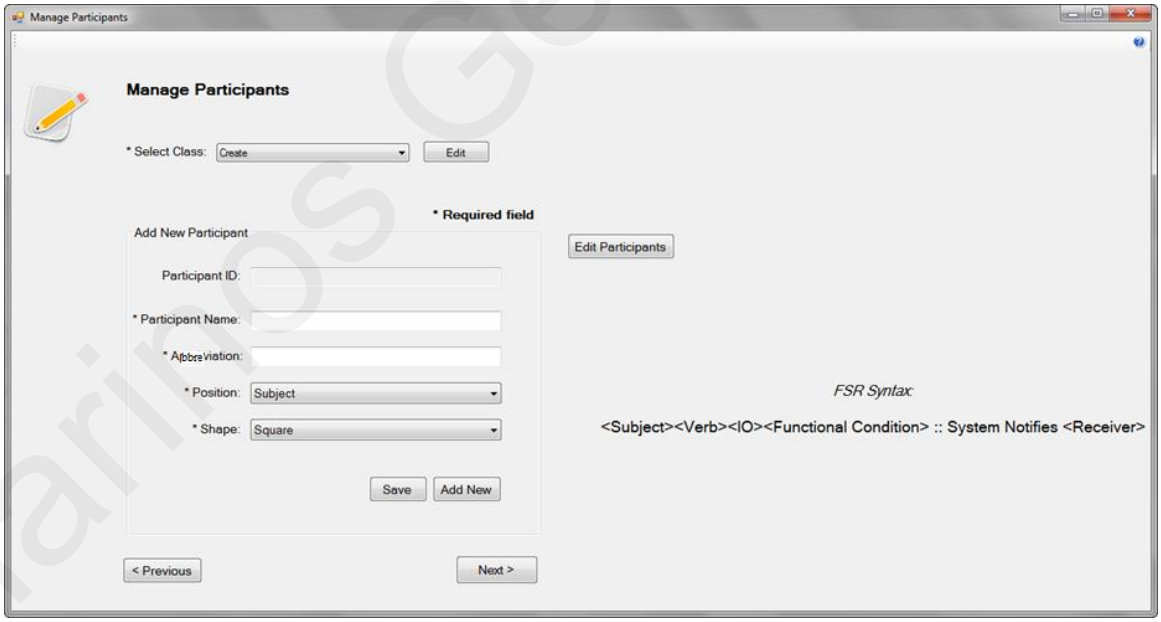


FIGURE 5.4 ADDING PARTICIPANTS FOR AN FSR CLASS.

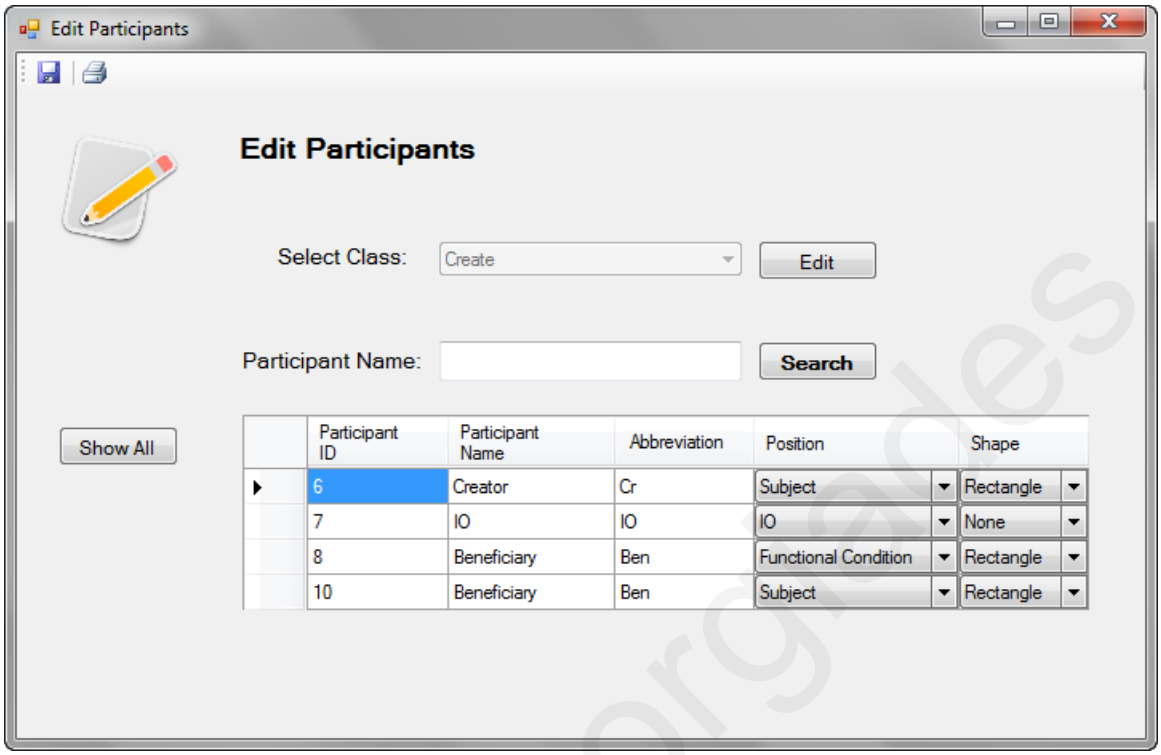


FIGURE 5.5 EDITING PARTICIPANTS FOR AN FSR CLASS.

With the use of the *Plan* interface component, the analyst builds the particular elements of a particular project, including its IOs, FSR classes, attribute types, questions for each IO, etc. For example, for identifying the IOs, the tool guides the analyst first to enter users and roles, and then it generates the Data flow table (fig. 5.6), which the analyst can use to note down the information exchanged between the system users.

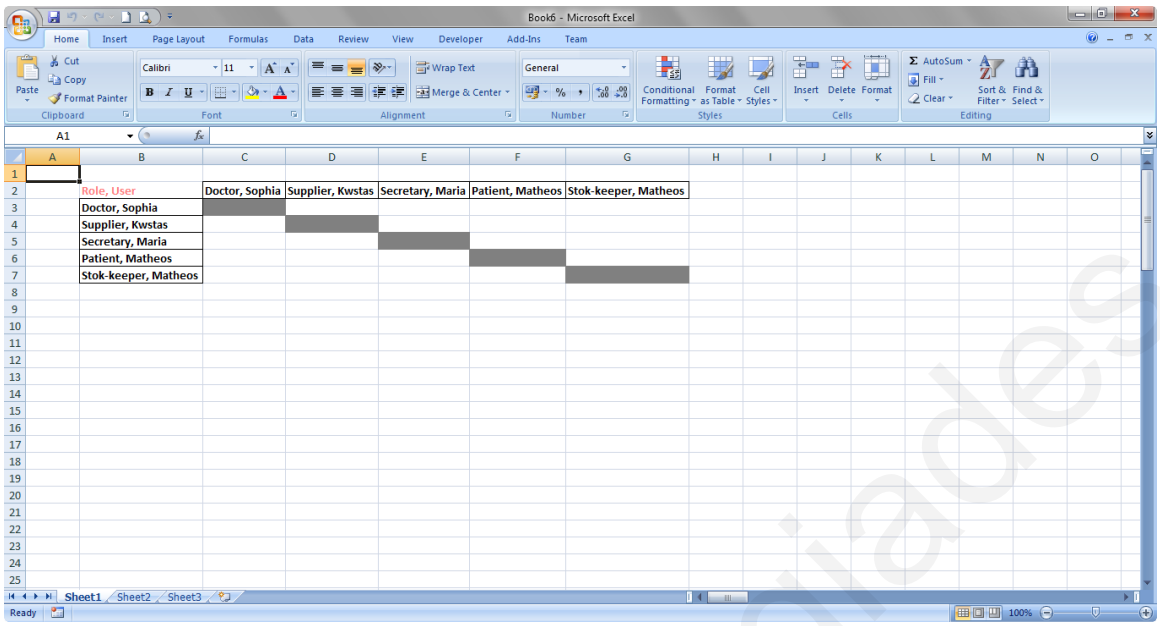


FIGURE 5.6 THE DATA FLOW TABLE.

Subsequently, the analyst determines the actual IOs from a candidate list of IOs derived from the information in the data flow table (among other techniques), and adds them into the new project, as illustrated in figure 5.7.

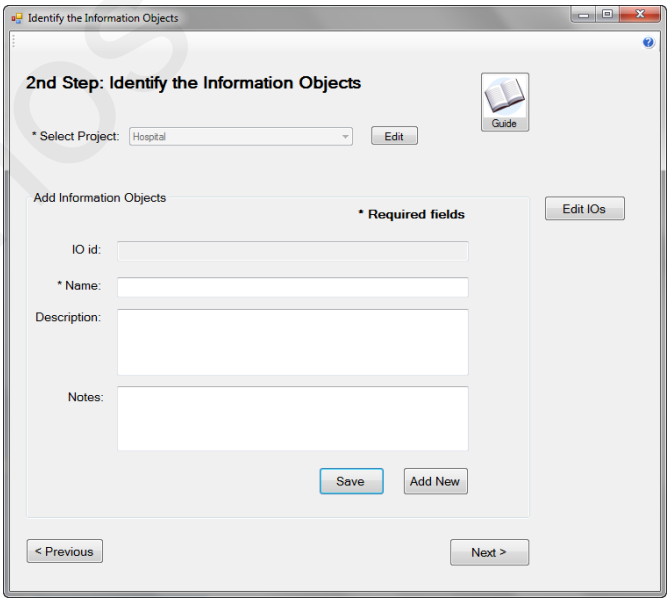


FIGURE 5.7 ADDING INFORMATION OBJECTS.

The activities within the Plan interface component are completed with the addition of a number of attributes for each IO as well as the automatic creation of FSRs for each IO and the questions for the FSRs of each IO, as illustrated in figure 5.8.

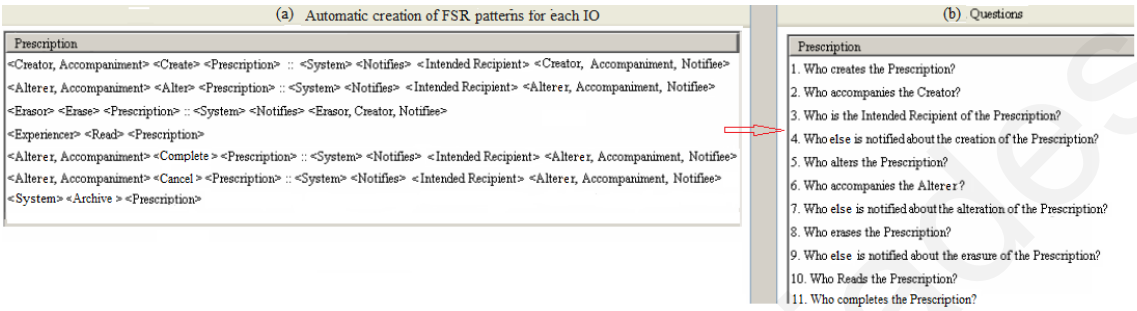


FIGURE 5.8 AUTOMATIC CREATION OF FSRS AND QUESTIONS FOR EACH IO.

In the Execution section, the analyst is usually at the user's environment submitting questions to the users and noting down the answers. The answers to the questions feed the FSR patterns as they are the values of the constituent elements (participants) of the FSR patterns, as shown in figure 5.9 (e.g. Creator takes the value Doctor). The answers also feed the attributes of each IO.

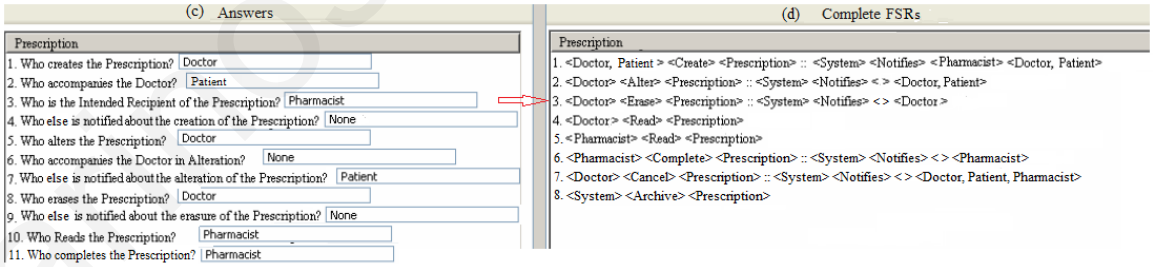


FIGURE 5.9 THE ANSWERS TO THE QUESTIONS FEED THE FSR PATTERNS.

Subsequently the FSRs and their constituent elements, as well as the IO attributes, with the use of specific rules are transformed to DFDs, Class diagrams, Use case specifications and diagrams, and the SRS document, as already described together with

the depiction of screenshots in earlier chapters. Here we provide only a small summary of each component:

*Functional Component*: The functional component utilizes specific rules to transform the FSRs and attributes of each IO to object-related DFDs. For example, the FSRs of creation, alteration, reading and erasure of each IO are grouped under one comprehensive function named *Manage <IO>*. Section 3.3.1 provides a detailed description of such rules, together with examples of resulting diagrams.

*Object Oriented Component*: The OO component utilizes sspecific rules to transform the FSRs and attributes of each IO to class diagrams. For example, each IO is transformed to a Class, and its CAREN functions become the methods of the class. Section 3.3.2 provides a detailed description of such rules, together with an example of a resulting diagram.

*Use Case Component*: The Use Case component utilizes sspecific rules to transform the FSRs (including also detailed FSRs which correspond to business rules) and attributes of each IO to use cases, use case diagrams and specifications. For example, as a rule of thumb, each FSR corresponds to a use case. Chapter 4 includes a detailed description of such rules, together with examples of resulting diagrams.

*The SRS Document Component*: The SRS Document component utilizes sspecific rules to transform the FSRs (including also detailed FSRs which correspond to business rules) and attributes of each IO to construct the SRS document. Section 3.3.3 includes a description of such rules, together with an example of a resulting SRS document.

# 6 Evaluation of NLSSRE

In order to prove the usefulness of the NLSSRE methodology, we performed an experiment through which we compared it to the classical RE OO approach based on use-case driven analysis, by applying both in a real setting[32]. Our evaluation, based on the method proposed by Geisser et al. [2007] for evaluating RE methodologies, tested the following for each approach (including its underlying tools):

• The effectiveness in terms of the achieved quality of the requirements specification produced by the application of each approach. The specification produced by both approaches followed the organization provided by the relevant template of the IEEE Recommended Practice for Software Requirements Specifications (IEEE 1998) (shown also in Table 3.5), and additionally integrated use case specifications and diagrams.

• The efficiency in terms of the output/effort ratio.

## *6.1 Experiment description*

Two novice software engineers (for evaluation purposes we will call them SE1 and SE2) were assigned to test the two methodologies. They were both graduate students of the University of Cyprus who attended several courses in software development over a

[32] Two preliminary evaluations of the methodology preceded the third final evaluation presented in this dissertation. The two previous evaluations assisted in clarifying and establishing several concepts of the methodology, such as the overall application framework, the definition of the IO, the IO guide, the identification of roles and the data flow table, the alter-related functions, and other issues. In particular the two preceded evaluations concerned the application of the methodology in two real-life settings, for the RE task for the development of a Banking IS and for a Dentistry IS. Both case studies were conducted by postgraduate students with extensive knowledge in the field of software engineering. Relevant training was given to them to become familiar with the NLSSRE methodology and the dedicated software tool provided at the time of the evaluation. All case studies are available upon request from the corresponding author.

period of three years prior to the experiment. Additionally, a specific one-week training and lecture were given to SE1 regarding the use of the NLSSRE methodology—including its adaptation to use case modeling—and the NALASS tool, and a similar one-week course was given to SE2 in the form of a knowledge refresher on the classical approach and use case modeling[33]—SE2 was already familiar with the classical methods from corresponding courses in his studies. We assigned SE1 and SE2 the requirements engineering task for the development of a sub-system of the Library Information System (LIS) of the University of Cyprus.

An existing, high-quality object-oriented SRS document, as defined by the IEEE template mentioned above, was used and served as a benchmark for the quality assessment of the specification developed by each student. The benchmark SRS was created by the analysts of the fully functional (currently in daily operation) LIS. This document was further refined during the LIS development, through a number of revisions, and finally reflected the high performance of the existing LIS and the high satisfaction of its users. We performed additional processing on the benchmark SRS to achieve a clearer focus on atomic requirements involving functions, data, functional conditions and business rules, with the aim of ensuring a high degree of comparability. Moreover, we used the use-case diagrams and use-case descriptions/specifications as were created originally by the LIS developers, for comparison with the ones derived from both NLSRRE and the classical approach. SE1 used the NLSSRE methodology, with its use case adjustment, and its corresponding CASE tool, while SE2 employed the classical RE

[33]Methodological guidelines by Cockburn 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.

method with the use of open-ended interviews, the IEEE SRS template, and Cockburn's use case methodological guidelines.

## *6.2 Evaluation criteria and analysis of results*

The evaluation was based on: (a) a number of factors which determine the quality of each produced specification, and, in turn, the effectiveness of each RE approach; and (b) the measurement of the output/effort ratio, which determines the efficiency of each RE approach.

### 6.2.1 Quality of Specification

We have formed a number of quality factors to evaluate the quality of each produced specification document (final version), based on the quality frameworks of Moody (2003), Moody and Shanks (1998), and Sharma (2009), as well as on the IEEE Recommended Practice for SRS (1998). Each quality factor was objectively measured against quality metrics. Table 6.1 shows the summarized results for the quality of each produced specification, followed by discussion of the metrics and results.

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

TABLE 6.1 OBJECTIVE QUALITY METRICS USED TO DETERMINE THE EFFECTIVENESS OF THE METHODOLOGY.

| Quality Factors | Metrics | Specs from NLSSRE | Specs from Classical |
|---|---|---|---|
| **Completeness**[34] | 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% (5/55Pr) |
| | Percentage of missing post-conditions (Pos) | 6% (4/61Pos) | 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 Data Classes | 0% (0/ 12) | 17% (2/12) |
| | Percentage of missing Class Functions | 0% (0/56) | 22% (10/46) |
| | Percentage of missing Class Data attributes | 4% (7/180) | 13% (20/150) |
| | Percentage of superfluous Classes | 8% (1/12) | 10% (1/10) |
| | Percentage of superfluous Class Functions | 5% (3/56) | 15% (7/46) |
| | Percentage of missing business rules (BR) | 14% (5/35 BR) | 41% (12/29 BR) |
| **Correctness**[35] | Number of textual requirements with no identifier | 0 | 3 |
| | Number of spelling errors | 22[36] | 57 |
| | Grammatical errors | 0 | 34 |
| | 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 violations to SRS writing standards | 0 | 6 |
| | Percentage of redundant use cases | 0% | 12% (7/55 UCs) |

[34] This metric is applied to comparable use cases and classes between each approach and the benchmark SRS. The denominator of the fraction in parenthesis refers to the number of elements existed in the benchmark for the comparable use cases or classes, accordinlgy. For example, 68 is the number of primary actors existed in the benchmark use case model for the comparable use cases (57) between the benchmark UC model and the NLSSRE UC model. Accordingly, for the classical approach this number is 58 (for 48 comparable use cases).

[35] For correctness we took into account non-missing plus 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.

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

| | | | | |
|---|---|---|---|---|
| **Consistency** | | 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, UC diagrams, requirements, figures) | 0 | 6 |
| | | Percentage of redundant functions (functions, sub-functions) | 0 | 23/52 |
| | | Number of redundant data (classes, attributes, relationships) | 0 | 26 |
| | | Number of requirements using words from more than one language | 0 | 2 |
| | | Number of missing use cases with regard to the relevant textual IEEE specs document[37] | 0 | 2 |
| | | Number of missing use case associations with regard to relevant textual IEEE specs document[35] | 0 | 14 |
| | | | | |
| **Understandability** | **Unambiguity** | Number of requirements written as optional sentences | 0 | 12 |
| | | Number of requirements written as subjective sentences | 0 | 10 |
| | | Number of requirements written as vague sentences | 0 | 44 |
| | | Number of requirements written as weak sentences | 10 | 63 |
| | | Number of requirements written as implicit sentences | 0 | 25 |
| | **Readability** | Number of non-atomic requirements | 0 | 34 |
| **Complexity** | | Functional levels | 2 | 1 |
| | | Total Functions/ Total Data classes | 4.5 (56/12) | 3.4 (34[38]/10) |
| | | Total Actions/ Total Use Cases | 15 (855/57) | 15 (720/48) |
| **Modifiability** | | Number of redundant requirements | 0 | 13 |
| **Prioritization** | | Number of requirements that were not prioritized | 7 | 15 |

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 of the benchmark SRS, which included 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

[37] This metric concerns the comparison between the use cases defined in the UC model and the functions in the OO textual SRS that should become use cases, of the same approach. Therefore, it indicates the consistency between the produced use case model with its corresponding textual OO SRS.

[38] This is the number of the original functions provided in the classically produced SRS, before they were processed by us to be formed as atomic functions which are 46.

approach or the classical approach that has equivalent functionality with a use case of the benchmark SRS. 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 benchmark 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 NLSSRE and 48 for the classical approach, compared to the 60 use cases proved by the benchmark SRS. Similarly, the number of comparable data classes was 12 for NLSSRE and 10 for the classical approach, compared to the 12 data classes proved by the benchmark SRS.

*a. Completeness.* Completeness refers to the extent to which the requirements document contains all necessary requirements (Moody, 2003). In our evaluation, these requirements include functions, data, use cases, actors, associations and relationships, use case diagrams, use case specifications including actions (in both normal and exception flows), pre-conditions, post-conditions, business rules and functional conditions. To assess the completeness of each SRS document, we check for necessary information which is missing or information which is superfluous. Completeness is mainly focused on the content of the SRS document and not in the way it is written. In our experiment, we observed that the requirements document of the classical approach included several missing and superfluous functions, data classes, attributes, use cases, actors, use case associations and relationships, UC specification actions, pre-conditions, post-conditions, and business rules. One of the major problems that arose from the use of the Cockburn's UC specification template, for the classical approach, 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 UC

specification 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, NLSSRE 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 NLSSRE for identifying the UC elements, with the use of predefined use case types, from—CAREN functions—and actors—from business rules—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 and guidelines for identifying functional conditions, use case pre-conditions and post-conditions, and an understandable way of expressing the content of textual OO and UC specifications. However, although the error rate was lower, our approach was not 100% complete. The very small number of superfluous elements (functions, use cases) resulted from the inclusion of one superfluous IO; in our approach, the identification of IOs (and therefore, UC modules) is an activity performed manually by the analyst, as the first step of NLSSRE, with the help of the provided relevant guide which, although providing specific steps for the identification of IOs, could probably be enriched further. There were also many fewer omitted data attributes and almost no omitted business rules. 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 types of functions, and thus, use case types, are not provided directly by our approach—although we give specific guidance as illustrated in step 3 of NLSSRE and step 3 of the UC adaptation process —and it is up to the analyst to identify them. Furthermore, the use of NALASS helped avoid missing the requirements elements. As the results show, the analyst who applied the classical approach missed considerably more (as a percentage) functions, preconditions and postconditions than the analyst who applied our approach. The use of NALASS helped to minimize missing these elements because it automatically provided the functions of each IO, guides and types to identify the IO attributes, preconditions of each use case and also different options for each use case regarding the new state (postcondition) of the IO, such as *Pending*, *Completed*, etc., so the analyst could decide accordingly. The two missing post conditions occurred, because the identification of postconditions 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 [Moody and Shanks, 1998], that is, in our case, naming rules, definition rules, diagrammatic conventions, etc. for the creation of an IEEE specification organized by object, use-case diagrams and use-case descriptions. Since the NALASS tool automatically provides the CAREN functions (use case types) and also uses specific transformation rules and a template for writing the OO IEEE requirements, as well as specific conversion and authoring rules for writing and drawing the use-case model, in the correct syntax and grammar, and due also to the tool being powerful for automatically drawing good diagrammatic notations, very minor

problems appeared when using NLSSRE, 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 NALASS-produced document, many more errors were found in the classically-produced document, including spelling and grammatical mistakes as well as requirements 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 requirements describing the same element (function, object, etc. – e.g., two or more use cases describing the same functionality) with different terms, or two requirements with the same identifier, or missing elements from a use case diagram or use case specifications while they are defined in the textual OO specification. In NLSSRE, contrary to the classical approach, for each IO identified by the analyst, the NALASS tool provides clearly and automatically the CAREN functions or use cases for each IO identified by the analyst, and it also guides the analyst in identifying and defining additional alter-related functions or use cases (e.g., Cancel IO, Complete IO), attributes, functional conditions and business rules. For instance, the SRS document of the classical approach uses different wording/terminology for the same type of IS elements; for example, for the creation function of each object class, different verbs were used, such as 'create', 'record', 'complete', and others. The NALASS tool also provides the specification template for each identified use case, with specific types of

actions as defined in step 7 of the use case adaptation process. In contrast, the use-case model of the classical approach used different wording/terminology for the same type of IS elements; for example, for the *creation* function of each data class and for the *creation* use case of each UC module.

*d. Unambiguity*. Unambiguity is determined by checking that every requirement stated in the SRS has only one interpretation. The assessment involves performing checks of lexical ambiguity by considering optional clauses such as "possibly" and "if needed" and subjective clauses such as "similar" and "better", checks of syntactic ambiguity by considering the correct use of the syntactic parts of a sentence, and checks of semantic ambiguity. NLSSRE is completely unambiguous since it avoids any words or phrases that fall in these categories. It also follows a strict syntactic order for the sentences written, including the basic syntactic parts of the linguistic sentence. Its only weakness falls in the definitions of business rules where terms such as 'is preferred to' and 'may' are used (e.g., "Every book is *preferred to* exist in four copies"). A possible solution to this would be to use illustrative comments or additional business rules to clarify the clause (e.g., "Three copies are also adequate, and every book must exist in at least two copies").

*e. Readability.* Readability refers to the ease with which the requirements can be read by users, adhering to the basic principle that the combination of good readability and unambiguity in a document leads to good understandability. The presence of long sentences (e.g., with conjunctions or disjunctions) makes the requirements document more difficult to read. However, reducing sentence length does not always improve understandability, since the addition of subordinate clauses often aids comprehension [Entin and Klare, 1985; Davison and Kantor, 1982]. Therefore, to make the requirements

document more readable and understandable, this additional information could be added in the form of comments and examples, wherever required. Additionally, NLSSRE does not use disjunctions or conjunctions (apart from one instance of the clause "if…then"), therefore its sentences are small enough to be easily readable (see example of Table 3.5, second column).

*f. Complexity.* Complexity refers to the size of the different constructs and the way they are decomposed and related. A low-complexity requirements document does not necessarily mean more understandable requirements. The requirements must be structured and decomposed at an achieved level where they will be both well-understood and easily transferred to the design and implementation models. Since we followed the IEEE template organized by object (Table 3.5), each data class consists of attributes, single-level functions and messages. The inherited functions need to have a reference to their super class. Similarly, inherited attributes should make reference to their super class. In NLSSRE, functions follow the CAREN types *Create, Alter, Read,* and *Erase,* and each one triggers a *Notification* function which is described in the *Messages* section; additionally, the lower level functions of *Read, Enter data*, *Compare*, *Save*, *Remove*, and *Present*, which comprise a CAREN function accordingly, are described under each CAREN function of the data class (Table 3.5). Finally, under each function the relevant conditions and business rules are written. However, the classical SRS document often puts many of the sub-functions and business rules together under one function with an overall description. This difference between the two methodologies is the reason that NLSSRE scored higher in complexity. However, that does not mean that its text is less understandable, but that it has a greater level of decomposition of data and functions.

Even its higher score, the NLSSRE specification's complexity is near to that of the specification created through the classical approach due to the fact that the number (size) of functions specified in the SRS created through the classical approach included functions which were redundant; therefore the classically created SRS included more functions that it should.

*g. Modifiability.* The requirements document is modifiable if its structure and style are such that any necessary change to the requirements can be made easily, completely and consistently. To make the requirements document more modifiable, related requirements should be grouped together and a requirement should not appear in more than one place in the document. The requirements document should also have a table of contents and cross-references if necessary. As a rule of thumb, the lower the number of redundant requirements in the SRS document the higher the level of modifiability. In regard to the application of NLSSRE, no serious redundant or ambiguous requirements were found, due to the architecture of the methodology, which assists the analyst in distinctly identifying and defining every requirement element. Therefore it creates an SRS document with a high level of modifiability, in sharp contrast to the classical approach that produced many redundant requirements and, therefore, a low level of modifiability.

*h. Prioritization.* An implementation priority should be assigned to each requirement, to indicate that the most essential features should be implemented first and the lower priority features should be implemented only if sufficient time and resources are available. Prioritization may be specified between the different elements of the IS as defined by NLSSRE or between the same elements. For example, the functions of every IO are of higher implementation priority than functional conditions. Or, there is prioritization

between the types of functions of an IO, viz. from higher to lower priority, the order is as follows: create, alter or erase, read (e.g. reports), notify. Or there may be prioritization between the functional conditions of the same FSR, as in the example of the simplified-form FSR "Librarian creates/records new book with stylus" which is determined to have a lower implementation priority than the FSR "Librarian creates/records new book with keyboard", because the latter is easier and less costly to implement. The analyst can also apply a weighting scheme to define numerical criteria for the parameters that determine prioritization, such as value, cost, effort, risk (e.g., for the parameter *value* that represents the importance of a requirement, we could have '1=high importance', '2=medium importance', '3=low importance'). The prioritization feature is still not provided by our CASE tool and is considered part of future work. In contrast, the classical approach scored lower in prioritization, since it did not provide any specific criteria to prioritize requirements.

### 6.2.2 Effort

To assess the efficiency of the methodology, we measured the time spent for the requirements discovery, analysis and specifications phases.

In our experiment, for the requirements discovery stage, SE1 asked each user (six users in total) in one round of interviews, 34 closed-ended questions, each one including on average fewer than four sub-questions per question—Table 3.3 shows a part of the questionnaire. Therefore, on average each user was asked approximately 120 questions, and the average time required for each question-answer pair was 50 seconds, for a total of roughly 100 minutes per user. These questions aimed to discover the information about

items exchanged between the different roles of the system in order to fill the data flow table. In addition, the users wrote a paragraph about their everyday work. As described, during this step the questions of the questionnaire were predefined and specific; they were also created automatically by the tool, after SE1 entered the business roles as input to the questionnaire. In contrast, SE2 made three visits to the users (due to a lack of preparation in identifying all the roles and users of the system, SE2 initially identified five users and only during the third visit identified the sixth user). The questions provided to the users were more general than those used with NLSSRE and did not guide the user to the answer since the majority of the questions were open-ended and not linked specifically to the elements of the analysis stage. In particular, SE2 asked each user on average 12 different open questions and 20 closed questions, most of them supplementary to the open ones. On average, SE2 asked 32 questions per user and required 2.5 minutes for each question-answer pair, for an average total of 80 minutes per user. SE2's open-ended interview approach sometimes was the reason for the users not understanding the topic, therefore SE2 asked questions repetitively or received incorrect answers. The repeating of questions was one of the reasons each question-answer pair took more time compared to the NLSSRE approach. In another case, SE2 did not receive an answer to a particular question and instead moved on to the next question.

In summary, in one visit with each of the six users interviewed through the NLSSRE approach, the analyst SE1 received specific answers to 120 questions per user in 100 minutes per user, whereas with the classical approach, the analyst SE2 received answers (many of them were vague) from six users, for about 32 questions per user and 80 minutes per user spread over three visits. The total time spent with the use of the classical

approach was 8 hours, while the time spent with the use of NLSSRE was 10 hours. NLSSRE required more time, but the results taken were specific and accurate, contrary to the vague, redundant and incomplete answers received during the application of the classical approach.

We believe that RD with NLSSRE will become faster and more reliable if conducted differently such that the analyst first receives the descriptions of each user's work, and based on the description identifies the roles and users of the IS in order to finalize/refine the questions on the questionnaires. The refined questionnaires could then be sent to the users, who could respond at their own pace without feeling pressure from the analyst's presence. In conducting the interviews at the user's workplace, we noticed that both the users and the analyst felt pressure to quickly ask and answer.

For the analysis and specification stages, the time spent with use of the classical approach was 105 hours broken down into 45 hours for the analysis of requirements and development of the SRS document, 50 hours for the development of the use case diagrams and specifications, and 10 hours for final checks including prioritization of requirements. The time spent on the analysis and specification stages using NLSSRE was 33 hours composed of three hours for IO identification, nine hours for definition of attributes, 16 hours for definition of business rules, conditions and constraints, and five hours for final checks including prioritization of requirements). This significant difference occurs not just because of the formalization provided by NLSSRE but also because of the use of the CASE tool that automates the entire procedure. Once the FSR patterns and attributes were fed with the corresponding answers, the SRS document and diagrammatic notations were created automatically. During this stage, SE1 visited the

users twice, once to confirm the answers to the questions and the second time to present the entire SRS document to the customer for approval. SE2 visited the users twice, principally to clarify with them different issues for the analysis and organization of requirements, and a final time for the approval of the SRS document.

SE1 and SE2 completed SRS documents of 47 and 43 pages, respectively.

In summary, the results of the application of the NLSSRE have proven to be faster and more accurate while exhibiting fewer ambiguities. Moreover, the structured English text resulting from NLSSRE was easier to comprehend and agree upon at the client site. Therefore, this small-scale evaluation indicated that our methodology is efficient and effective and intrinsically provides a very strong element of validation because the NLSSRE-produced requirements are delivered in a Natural Language form which is understandable to the client who gives the final approval to the requirements.

## *6.3 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?

Regarding the first issue, recall that the students had to apply two methodologies for the development of an SRS document enriched with use case models. Such a task is usually performed by requirements engineers during the requirements discovery, analysis and specification phases of a typical software development lifecycle. Given the state of practice in the software industry, whether for students or professional requirements

engineers, RE is likely to require training. The students of our experiment are graduate students with extensive knowledge of software and computer engineering mostly in theory and less in practice, since they were involved only with a few real projects in the past. One of the students received training in using the NLSSRE methodology, and the other student was given a refresher course in using the classical approach[39] with which he was already familiar. 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 methodologies 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 methodologies to larger scale systems seems likely to demonstrate at least a proportional increase in the differences between the two methodologies. The involvement of more users and business roles, information objects, use cases, relationships, use-case specification actions, pre-conditions, post-conditions, business roles and conditions would be more easily handled by a structured, formal and understandable methodology, such as NLSSRE, than from the classical approach.

[39]Here only knowledge refreshing was performed, since the approach was taught in corresponding classes during his first degree.

## *6.4 Other limitations and implications*

Sections 6.2.2 and 6.3 indicated a number of threats and limitations when applying the methodology, regarding effort required, training, level of requirement engineer, and applicability to larger contexts. Additionally, the application of the methodology indicated implications in the order of applying two steps. As previously mentioned requirements elicitation will become faster and more reliable if conducted differently such that the analyst first receives the descriptions of each user's work, and based on the description identifies the roles and users of the IS in order to finalize/refine the questions on the questionnaires. The refined questionnaires could then be sent to the users, who could respond at their own pace without feeling pressure from the analyst's presence. In conducting the interviews at the user's workplace, we noticed that both the users and the analyst felt pressure to quickly ask and answer. Another possible limitation of NLSSRE—and NALASS—is that it did not provide rich alternative expressions in asking questions. Using synonyms or stating the questions in other words, for example, would help the respondent think better of the answer. Use of alternative expressions would be even more useful if provided automatically by NALASS. Another limitation comes from the application domain of the methodology. We have mentioned that NLSSRE is mostly appropriate for engineering the requirements for the development of the operational part of an IS. That is why the methodology was empirically evaluated through its application to relevant contexts, such as an LIS, a Dentistry IS and a Banking IS. Therefore, by having applied the methodology in the aforementioned contexts, we did not demonstrate that the use of NLSSRE may be feasible in other application domains, such as decision support systems or executive information systems.

Furthermore, NLSSRE has been evaluated through its application to real life projects related mainly to the public sector. Given that NLSSRE is a newly proposed methodology that could be fully evaluated only by applying it to the development of a wide range of software systems, it would be interesting to conduct a short scale application to other real-life projects, again under a controlled environment but maybe this time with the contribution of industrial collaborators. In industrial projects, techniques for requirement engineering are often seriously affected by restrictions in time and budget, which in turn they affect employees' attitude toward a new methodology. People in the industry, comparing to public sector, are often more reluctant to new methods. To apply efficiently the methodology in the industrial context, we may need to pay additional attention to human and psychological factors. Therefore, we need ways to 'sell' the methodology and also to pass it easily to the users during its application. A good presentation of the methodology's capabilities could be given to the system's stakeholders to increase the acceptance levels of the methodology. On the other hand, a good user training manual should be provided to the analysts to assist them in understanding, passing to the client, and applying easily the methodology. The NLSSRE activities which require significant users' involvement are, mainly, requirements elicitation, and, secondly, requirements prioritization and approval. For large projects, where time and budget are likely to affect these activities, a more systematic gradation or prioritization of requirements is necessary. Also, the elicitation of requirements could follow a refinement process and be organized incrementally with respect to time and budget such that customers' expectations are satisfiable within any refinement step. Therefore, strategies and methods are needed that highly adapt to almost arbitrary project

situations. In such a context the methodology needs to be more flexible and also easily and quickly understandable to the users. Additionally, because industrial settings are usually more dynamic and volatile, requirements change more easily. Especially, for large projects, NLSSRE could follow a more agile-like development, that is for example, delivering requirements incrementally, starting from the most significant ones, and once they are implemented, moving to the next cycle of requirements.

Another factor that could establish confidence, in the industrial setting, that the software system is 'fit for purpose' the verification and validation process. Software validation or, more generally, verification and validation (V & V) is intended to show that a system conforms to its specification and that the system meets the expectations of the customer buying the system. This means that the system must be good enough for its intended use. Inspections and testing are the most common techniques used for verification and validation, with software testing to be the most reliable as well as the one favored to strengthen the customer's confidence about the capabilities of the new system. The customer will feel more confident when a prototype or an executable version of the program is available. An advantage of incremental development, where NLSSRE can be applied in cycles, is that a testable version of the system is available at a fairly early stage in the development process. Functionality can be tested as it is added to the system so we don't have to have a complete implementation before testing begins.

Overall, therefore, the goal of software testing is to convince system developers and customers that the software is good enough for operational use. Testing is a process intended to build confidence in the software. Further to software testing, the formalized nature of NLSSRE makes it self-verified, since requirements are specifically identified,

interrelated and written, without ambiguities and redundancies. Chapter 9, on future work, expands on requirements testing.

# 7 Discussion

The aim of the proposed methodology is the formalization and automation of major parts of Requirements Discovery, Analysis and Specification. In particular the methodology worked towards its aim by implementing the following elements:

- Specific questions and guidance for discovering, analyzing and specifying requirements.
- Predefined types of functions, specific categories of data, and methods to identify and define business rules and functional conditions (the circumstances within which each function is performed)
- Specific patterns for writing requirements as structured, semi-formal NL sentences.
- Specific rules to transform the abovementioned identified requirements into diagrammatic notations, including class diagrams, data flow diagrams and use-case diagrams, as well as use-case and textual specifications, the latter following a certain IEEE SRS template (IEEE, 1998).
- A CASE tool that automates the entire procedure
- Adaptation for formalizing use case model development

In sections 2.4 to 2.11 we gave a detailed description of related work in the above as well as in more general areas and compared it with our methodology. In this chapter, we pay special focus only on other research related closely to the concepts of our methodology and we discuss how our methodology is different and achieves its aim and objectives.

## *7.1 Formalization in NLSSRE comparing to other related approaches, in general*

Current approaches in RE, both those based on NL and those not based on NL, fail to provide a specific, easily understood formalization of the major parts of requirements discovery, analysis and specification. The problem originates from the weakness of existing approaches to formalize the requirements discovery (RD) activity. On the one hand, some approaches use NL parsing techniques to retrieve requirements from pre-existing requirements documents, but this method is not reliable, because of ambiguities, redundancies and inconsistencies present in such documents. On the other hand, other approaches avoid formalization and use open-ended questions that lack specificity and formality (Gervasi and Zowghi, 2005). In either case, often the result is a requirements document with ambiguities, redundancies and inconsistencies. During analysis and specification of requirements, current approaches are also weak in providing a specific and easily understood formalization of the elements of an information system, so that the analyst will know specifically what data, functions, business rules and functional conditions to use and search for, as well as how to define them. Instead, they use general guides and templates, such as the traditional IEEE SRS document template or templates related to the more contemporary Use Case specification. The use of such templates also results in requirements documents written in a free, informal version of NL which promotes ambiguity and redundancy. The informality in such documents hinders also the use of automated tools for system modeling, since informal NL is inherently complex, vague and ambiguous.

## *7.2 In NLSSRE, Analysis and Specification guide Discovery, specifically*

Additionally, the second main reason for frequent inadequacy of existing RE approaches and models is that they mainly focus on how RE process activities are interrelated and organized during their application/execution. The significant difference between the existing approaches and the proposed methodology does not lie on the way they are applied, but on the way they are built. Specifically, in the existing approaches, we cannot find a direct, specific link at an architectural level between analysis and elicitation or between specification and elicitation. The lack of a direct connection between elicitation and the later stages of RE means that the way elicitation is built is not actually related with the way analysis is built. The main concept of NLSSRE's architecture starts with the principle that if the analysts know, in advance, specifically what types of functions, data, business rules and conditions (RA) they should search for and write down, then they will be able to ask specific questions (RD) regarding that particular information. Thus, for example, NLSSRE provides a specific number of system functions, including *create*, *alter*, *read*, *erase*, *notify*, and *alter-related functions*, specific types of business roles, including *creator*, *accompaniment*, *intended recipient*, and specific types of functional conditions, altogether, on the one hand, written as a semi-formalized NL sentential requirement, and, on the other hand, providing guidance to develop specific questions, the answers of which will feed the FSRs and the attribute values.

## *7.3 Goal-oriented approaches, Use Case Driven Analysis and NLSSRE*

The most widely used structured approach for RE, which involves the use of NL, is the Use Case-Driven Analysis (UCDA), which is mainly used for object oriented analysis and design (Dias et al., 2008). UCDA was also used for the comparative evaluation of the proposed methodology. Goal-driven requirements engineering is another approach that conceives requirements as goals, and it can use NL and also questions to refine goals and sub-goals. However, goals are defined at a more abstract level and existing methods do not provide a formalization of the main functions of an information system, such as create, alter, read, erase, notify, and other alter related functions, such as complete, archive, reserve, etc. Although goal oriented approaches, such as KAOS (Dardenne et al., 1993; Van Lamsweerde et al., 1991; Van Lamsweerde, 2001) separate RE concerns into goals, agents, actions, objects and scenarios, they do not relate all these together under a semi-formal NL sentence, in order for both the client and other analysts or programmer to obtain a better understanding. Techniques provided by van Lamsweerde (2000b) for finding out sub-goals and requirements refer to keep asking HOW questions about the goals already identified. Formal goal refinement patterns may also prove effective when goal specifications are formalized; typically, they help finding out subgoals that were overlooked but are needed to establish the parent goal. Keep asking WHY questions about operational descriptions already available is another technique for finding out more abstract, parent goals is to. Analyzing requirements in terms of goal decomposition and refinement can be seen as teasing out many levels of requirements statements, each level addressing the demands of the next level. This approach to the clarification of requirements is especially appropriate in the case of non-functional requirements (such as

flexibility, robustness, reusability, maintainability), where initial requirements can be difficult to make precise. A goal-oriented approach would allow the requirements to be refined and clarified through an incremental process. Chung's NFR framework (1998) is a goal- and process-oriented approach for dealing with non-functional requirements.

As already mentioned, UCDA has gained a wide acceptance among the many methods in requirements engineering (Dias et al., 2008), 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 (Pooley and Stevens, 1999). 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 (Kim et al., 2004) 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.

Although UCDA offers a more compact framework for analyzing requirements in contrast to the classical generic approach, which is basically performed with the use of a generic SRS template (e.g., IEEE SRS template (IEEE, 1998)), building the UC model and especially writing use case textual specifications is still a difficult and time-consuming activity. The major difficulties in producing high quality use case models originate from the elicitation process; as Kim et al. (2004) 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 (Fliedl et al., 2002; Dias et al., 2008) 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 (Liu et al., 2004), 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 (Gervasi and Zowghi, 2005) 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[40]. 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. In particular, these approaches do not provide:

[40] 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.

(i) 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.

(ii) 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

(iii) 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.

In using UCDA with NL for requirements specification, although NL facilitates communication between the analyst and the domain expert, its free, informal style, increases the risks of ambiguity, inconsistency and incompleteness of the use case description/specification. El-Attar and Miller (2006) state that these problems produce low quality information systems (ISs). In order to avoid these typical problems with natural language, it is important to use a more structured or formal technique for such a description; here is where a more analytical conceptual model is involved. In the relevant literature, some structured techniques for the description of use cases have been proposed. . In the relevant literature, some structured techniques for the description of use cases have been proposed. In Eriksson et al.'s work (2004), a tabular representation is used, and in Leite et al.'s (1997), 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 (2007) 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 (Hsia et al., 1994) or statecharts (Glinz, 1995; Seybold et al, 2006)) 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.

In contrast, our approach differs from the aforementioned ones, since it guides the analyst how to define specific sets of questions from predefined patterns of functions—CAREN for Create, Alter, Read, Erase, Notify—and specific types of data, business rules and functional conditions. The answers to these questions feed and complete the analysis and specification stages. Therefore, the way we discover the requirements is clearly connected to the analysis and specification of requirements. As previously mentioned, in the current literature—to the best of our knowledge—this link does not exist, and therefore the resulting requirements documents produced from current approaches need to be re-organized, re-validated and re-adjusted. While other approaches elicit user requirements, which are abstract and vague in nature, and use techniques to transfer them

into system requirements, our approach differs in that the elicitation activity is targeted directly on the system requirements, in terms, mainly of the system functionality. Therefore, during the analysis activity, there is no need to analyze any user requirements and derive the system requirements. What the analyst needs to do is to organize requirements under IOs. These requirements concern the system functions, which are already predefined from the provided CAREN functions. It more detailed terms the semi-formalized nature of the proposed methodology, comparing to the aforementioned approaches, the formalization of the NLSSRE elements is facilitated with a thorough mapping of a considerable NL elements. In particular, for requirements analysis, data analysis is facilitated by the use of semantic types of the genitive case, other grammatical cases, nouns, adjectives and adverbials; the types of functions are determined with the use of semantic types of the verb of the linguistic sentence; business roles are mainly linked to the use of semantic roles of subject and indirect object of the sentence; analysis of functional conditions is facilitated by the use of adverbial adjuncts; and finally, business rules, which define or constrain some aspects of the business by describing the behavior/reaction of people and data through their relationships, are derived from relations (combinations) between attributes of data entities (business roles and data objects). For requirements specification, functions, data, functional conditions and business rules are written as formalized sentences - FSRs), according to predefined patterns which are derived from the syntax of the linguistic sentence. FSR patterns, business rules and types of attributes are used to derive questions (RD process), the answers to which produce the complete FSRs (including also detailed FSRs which incorporate the business rules). Finally, specific transformation rules are utilized to

process the complete FSRs and attributes to derive diagrammatic notations such as DFDs, UML class and use case diagrams, as well as use case specifications and the SRS document.

Finally, in adapting NLSSRE to formalizing and automating the development of the use case model, our methodology (i) formalizes the elicitation process of UCDA with the use of predefined types of use cases and actors, FSRs, as well as guidelines to derive their associations, relationships and business rules; (ii) formalizes the system's functionality and specification 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; (iii) provides CASE-tool support and achieves time-saving and error-free UCDA, with the use of NALASS that covers all the stages of the UC model development and results to the construction of UC diagrams and specifications.

# 8 Conclusions

Requirements engineering (RE) is the first and most critical activity of the software process as errors at this stage inevitably lead to later problems in the system design and implementation (Sommerville, 2010). Several studies have shown that a substantial percentage of software projects continue to fail, often because requirements are ill-defined, ambiguous, or incomplete. It is also evident that the least understood parts of requirements engineering are the activities of requirements discovery, analysis and specification. Research also shows that the majority of people involved in software requirements elicitation, analysis and specification prefer to use free, common natural language (NL), as the means to discover and document requirements (Mich et al. 2004; Gervasi and Zowghi 2005). Natural language is more understandable to both users and analysts, on the one hand, and on the other, it is easier to move from one type of natural language (informal – during elicitation) to another (formal – during analysis and specification).

This dissertation presented NLSSRE, a compact and clear-cut methodology that is intended to formalize and automate a large part of the Requirements Engineering (RE) process, including discovery, analysis and specification of user requirements for the development of information systems. NLSSRE is designed so that the analyst is guided in advance, through a step-by-step approach, what specific types of data, functions, business rules and functional conditions to use and search for, what questions to ask, in what specific way to analyze the answers to the questions, and how to write them using formalized sentential requirement patterns. The formalized requirements are then easily transformed, with the use of specific rules, into diagrammatic notations, including class

diagrams, data flow diagrams and use-case diagrams, as well as use-case and textual specifications, the latter following a certain SRS template. The formalization of NLSSRE is achieved with the aid of NL elements such as verbs, nouns, genitive case, adjectives and adverbials, while its automation is realized with the use of a dedicated CASE tool. The proposed methodology also aims at being adjusted for formalizing the entire process of use case model development, since existing use case driven analysis approaches often result in poorly defined use case models.

Specifically, the first step of NLSSRE guides the analyst to identify specific discrete data entities, called Information Objects (IO) which are *defined as digital representations of tangible or intangible entities, described by particular attributes, and which the users need to Create, Alter, Read, Erase and be Notified (CAREN) by the messages they can trigger.*

The next step involves the application of specific functions on every IO, as well as the written specification, in the form of formalized sentences (Formalized Sentential Requirements – FSRs), of the IO, its functions, the involved business roles, and the functional conditions. NLSSRE provides specific FSR patterns, based on which it guides the user to derive specific questions to identify the business roles and possible values of the functional conditions; the answers to these questions assist in forming the complete FSRs. Writing the requirements as formalized sentences does not only help to make expression of requirements more disciplined, understandable and organized, but also leads to the identification of entities (business roles and functional conditions) that are involved during the application of a (CAREN) function on an IO. Furthermore, such

formalization makes easier the transformation of requirements into diagrammatic notations and specifications.

The subsequent step involves the identification of the IO attributes. NLSSRE provides specific categories of attributes, each of which is linked to a category of IO, as compulsory, optional or not applicable. In this way the analyst will know which category of attributes to search for so as to link it with each identified IO taking into account its category. For each category of attributes, the analyst works towards identifying the attributes of each IO by using different methods, such as questions and use of other data collected from the system users or domain experts.

After the identification and definitions of IOs, their FSRs and attributes, the analyst proceeds to the identification and definition of business rules, as the next step. Business rules are created from combinations of two or more attributes between different IOs of any category, but with more emphasis given on the category of business roles. The complete FSRs may lead the analyst to identify more easily the interrelated IOs, due to their involvement in the execution of the same (CAREN) function. Then, through a number of questions that actually constitute the business rules in their general form, the analyst aims at investigating whether business rules may be derived by means of combinations between attributes of the involved IOs.

The final step concerns the application of specific rules to transform the FSRs and attributes of each IO to DFDs, Class Diagrams, Use Case Specifications and Diagrams, and an IEEE SRS-like document.

To evaluate the effectiveness and efficiency of the methodology, we performed a short-scale experimental study through which we compared the NLSSRE methodology to

the classical OO RE approach based on Use Case Driven Analysis (UCDA), by applying both of them in a real-life setting. The results showed that the proposed methodology performed much better than the classical approach in various objective quality metrics, such as completeness, correctness, understandability, modifiability and prioritization. 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, where our approach performed much faster than the classical one. Of course, there is always room for improvement, such as the use of a dictionary to check user's input for spelling or the possibility of adding illustrative comments to requirements to make them more readable.

# 9 Future Work

It is our belief that this novel work has achieved significant steps toward providing straightforward and automated support for eliciting, analyzing, and documenting NL requirements. Future steps will involve:

(i) Enhancing the process of identifying user roles, as well as key stakeholders, during the first step of the methodology. Our methodology proposed a new technique to facilitate this process, that is, the use of the data flow table through which new roles can be derived by studying the interactions between users (exchanging information, products, or instructions), and a number of existing techniques, such as a brief description of each user's everyday work, and the study of existing documentation. Additional techniques and methods in the literature could be used, such as StakeNet (Ling, 2010) which identifies and prioritizes requirements using social networks, and it is more appropriate for large-scale projects, with a big number of users and roles.

(ii) Enriching the guide for information objects identification, during the second step. The current guide provided by NLSSRE utilizes specific rules that could assist the analyst in identifying the actual information objects. Through the application of the existing guide in real-life projects within different contexts, this guide could be enriched with additional rules some of which could be standardized according to the application context each time.

(iii) Extending the methodology and its CASE tool in order to support the requirements design phase, with the creation of sequence, collaboration and state diagrams. The construction of such diagrams may be facilitated with the application of

specific rules on the NLSSRE elements including, mainly, the information objects and their functions and sub-functions, as well as the use case actions.

(iv) Investigating the potential of automatically producing Z specifications from FSRs. Formal specifications, such as Z are based on mathematics. Even though formal specifications are very precise and accurate and they have been considered to be more effective in representing software specifications, they are not widely used in software development (Shukur et l., 2002) due to the additional technical knowledge needed (Mehandjiska and Palac, 2002). The major obstacle of the conversion of natural language into formal specification is from the inborn characteristic of ambiguity of natural language and the different level of the formalism between the two domains of natural language and the formal specification. The use of an intermediate controlled language as provided by NLSSRE, through the use of FSRs, limits ambiguity and can facilitate the transformation process. To achieve this, specific transformation rules need to be applied on the NLSSRE elements, with special focus on the FSRs.

(v) Generating the SRS document with additional requirement organizations, and incorporating automatically into the SRS document, with the use of NALASS, elements such as use case descriptions, scenarios and diagrams. Currently, the SRS document automatically produced by NALASS organizes requirements similar to the IEEE SRS template, following an OO organization. Future work may include producing a functional-oriented structure by applying transformation rules on the NLSSRE elements and especially on the FSRs. Additionally, new sections can be created in the SRS document for placing use case descriptions and diagrams with the corresponding

references to the relevant IOs and FSRs. Similarly, DFDs and class diagrams can be placed in the appropriate sections.

(vi) Testing NLSSRE on large scale projects. As mentioned in chapter 6, the involvement of more users and business roles, information objects, use cases, relationships, use-case specification actions, pre-conditions, post-conditions, business roles and conditions would be more easily handled by a structured, formal and understandable methodology, such as NLSSRE, than by a classical approach. One issue that needs to be taken into consideration for large-scale projects is the involvement of more than one analysts—as a way to achieve better project management—and how work should be shared between the analysts in order to achieve good collaboration and reliable results. Our recommendation is that they should work together to establish an agreed group of stakeholders and system users, and, subsequently, the user roles of the system. Based on the identified user roles or on some basic identified user roles, they should work separately, each one for a different group of user roles, to identify the candidate information objects. Then they should determine together the actual IOs, and they should subsequently undertake, each analyst separately, the analysis activity for a different group of IOs. During this task, they should try to include in each IO group the most interrelated IOs; therefore, some rules should be defined to achieve good grouping and separation of both IOs and stakeholders.

(vii) Enriching the guidelines for facilitating more precise and straightforward identification of alter-related use cases including Cancel IO, Complete IO, etc. Currently, specific guidelines are provided to identify alter-related use cases. These guidelines concern the use of the attribute state, pre-conditions and post-conditions, as well as the IO

category. Future work may involve the investigation of using other attributes that could lead to new alter-related use cases, or even if the attribute state may lead to new alter-related use cases depending on new IO categories or sub-categories or on any standard types of pre-conditions and post-conditions.

(viii) Additional strengthening of the NALASS tool could include the establishment of a dictionary to check user's input for spelling. Another future task could be the application of specific rules to convert the semi-formal form of the SRS document into a completely free NL form, thus effectively hiding from the users the semi-formal organization of requirements. Furthermore, the development of a web version of the tool could be a future task, since now the tool is only available in a desktop version.

(ix) Further concepts from natural language such as hyponymy and meronymy may be used to identify IO attributes and relationships between IOs. Hyponymy, is one of the most important structuring relations in the vocabulary of a language (Aronoff and Miller, 2002). This is the relation between apple and fruit, car and vehicle, slap and hit, and so on. We say that apple is hyponym of fruit, and conversely, that fruit is a hyperonym of apple. Hyponymy is used for the identification and construction of object classes and their interrelations, as well as for the development of a lexicon to keep track of data. Meronymy describes the part-whole relation. It attempts to take into account the degree of differentiation of the parts with respect to the whole and also the role that these parts play with respect to their whole. Meronymies determine the structure of an object (object properties) and its relation to other objects. Meronimity assists the process of deriving new objects. Different kinds of meronymies mean different structure of the objects and their relations.

(x) In addition, the use of NLSSRE in other development approaches, like component-based and agile process models could be investigated emphasizing on how the methodology itself should be changed to accommodate the specific characteristics within each of the aforementioned models and furthermore how these approaches may be benefitted from the disciplined formalized concepts of NLSSRE. Some ideas are given in this paragraph regarding the use of NLSSRE with the three abovementioned approaches.

While phased development does requirements engineering in an early phase that precedes the majority of coding, agile development does requirements engineering continuously throughout the project. Agile development, contrary to phased development, usually does not embody the requirements in a written document (Kovitz, 2003). Typically (though not necessarily), developers in an agile project promise very small deliverables which they release to the customer once per "iteration"—one to three weeks, on most projects (Kovitz, 2003). Beyond the current iteration, they don't offer the customer much certainty about precisely what functionality will be delivered on a given date. NLSSRE, contrary to classical phased development, can create quickly, with the use of the NALASS tool, the SRS document, and therefore the development process can start quickly; this stands mostly for small projects. However, for large projects that include a very big number of IOs (e.g. 200), users and roles, the NLSSRE can be used initially to identify the most significant IOs, their relationships and attributes, and based on them to construct the corresponding FSRs and business rules and a portion of the SRS document. The developers then can develop the code based on this part of the SRS document, and subsequently proceed with the identification of another portion of IOs, roles, etc. In this case, NLSSRE provides agile development with more certainty about the functionality,

and every partial implementation developed by agile methods can facilitate NLSSRE for the identification and establishment of the remaining IS elements.

Requirements engineering in the context of off-the-shelf component-based system development is a difficult issue. Most actual approaches are not requirements-driven, which does not allow to gain a great customer acceptance. Otherwise, they have difficulties getting a natural matching between customer requirements and component features, which does not facilitate the user involvement (Le and Rolland, 2001). The description of requirements through NLSSRE may be processed and expanded to combine customer's vision and product supplier's vision under a common representation. Additionally, the NLSSRE artifacts, such as IO and CAREN functions of each IO, as well as some specific attributes for each IO, may create combinations for the development of general components or more specific components for particular contexts (e.g. for an HIS).

(xi) Finally, variations or specific steps of NLSSRE could be used in other Computer Science domains, like verification & validation of specifications, testing (test case production based on NL), etc. Especially, as per the latter suggestion, it is the task of the requirements based test to demonstrate that the system does what it should do according to the written agreement between the user organization and the developing organization. Since NLSSRE provides a semi-formal way of writing requirements without ambiguities and redundancies and with completeness in terms at a great extend in terms of objects and functions, testing should focus more on functional conditions, business rules and data attributes, which altogether may extend the already identified functionality. NLSSRE can facilitate building test cases based on the provided types of the aforementioned elements but also with the formalized types and structure of the use case transaction flows for

building use case scenarios, as a type of testing. In particular, both the procedures on how to create scenarios and how to use scenarios in testing could be investigated. There could also be additional support for modeling dependencies between scenarios (not only with includes and extends relations). To derive tests from scenarios the dependencies between scenarios have to be known, else crucial parts of the system will/may not be tested for. Additionally, to enhance requirements verification, the semi-formal requirements document produced by NLSSRE could be cross-checked with the use case model; that is the SRS document could have links between the use cases and the functional requirements. Each requirement (FSR) will have some kind of identifier. This identifier will then be referred to by the use case. One use case may fulfill one or more functional requirements. One attribute of the use case will be a list of the identifiers of the requirements it fulfills. Since NLSSRE use case specification flows are described with specific types of actions in a semi-formal NL, they can be easily related to the functional requirements of the SRS document. This is useful for traceability purposes, both relating use cases with requirements and the generated test cases with requirements. When requirements change, it is possible to know which use cases might be impacted and, if it is the case, update them. Test cases related to these use cases can also be updated or regenerated (assuming an automatic approach).

# References


Ahli, S. 2002. *Guide to Applying the UML*. Springer.

Ambriola. V, and Gervasi, V. 1997. *Processing natural language requirements.* In Proceedings of the 12th international conference on Automated Software Engineering (Lake Tahoe, Ca, November). IEEE Computer Society, 36-45.

Ambriola. V, and Gervasi, V. 2006. *On the systematic analysis of natural language requirements with Circe*. Automated Software Engineering 13, 1, 107-167.

Aronoff, M. and Miller, J. (eds) 2002. *The handbook of linguistics*. Oxford: Blackwell.

Avison, D, and Fitzgerald, G. 2003. *Information Systems Development: Methodologies, Techniques and Tools.* 3rd Ed. UK, McGraw-Hill Education.

Bailin, S. 2002. *Object-Oriented Analysis.* In Encyclopedia of Software Engineering, J. Marciniak, Eds, 2nd ed. New York: John Wiley and Sons.

Ben Achour, C. 1998. *Guiding Scenario Authoring*. In Proceedings of the 8th European-Japanese.

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

Borland Software Corporation, "CaliberRM 2006 User Tutorial," Nov 2006, http://info.borland.com/techpubs/caliber_rm/2006/EN/CaliberRM%20Tutorial.pdf.

Burg, M. 1997. *Linguistic Instruments in Requirements Engineering*. Amsterdam, IOS Press.

Chalmeta, R. and Grangel, R. 2008, *Methodology for the implementation of knowledge management systems*. Journal of the American Society for Information Science and Technology 59, 5, 742–755.

Chung, L., Nixon, A., Yu, E. and Mylopoulos, J. 1999. *Non-Functional Requirements in Software Engineering*, Kluwer Publishing.

Coad, P. and Yourdon, and E. 1990. *Object-Oriented Analysis*. Englewood Cliffs, NJ, Prentice-Hall.

Cockburn, A. 2000. *Writing Effective Use Cases*. Reading, Massachusetts: AddisonWesley.

Conger, S. 1994. *The New Software Engineering*. Belmont, CA, Wadsworth Publishing Company.

Coleman, D. 1998. "A Use Case Template: Draft for discussion", Fusion Newsletter, April 1998. http://www.hpl.hp.com/fusion/md_newsletters.html

CREWS, http://sunsite.informatik.rwth-aachen.de/CREWS/, Accessed 21 January 2011.

Da Silva, L. and Leite, P. 2006. G*enerating Requirements Views: A Transformation-Driven Approach*. ECEASST 3.

Dardenne, A., van Lamsweerde and S. Fickas. 1993. *Goal directed Requirements Acquisition, Science of Computer Programming*, Vol. 20, 1 993, 3-50.

Dascalu, S., Fritzinger, E., Cooper, K., and Debnath, N. 2007. A *Software Tool for Requirements Specification: on Using the STORM Environment to Create SRS Documents*. In Proc. of the Second International Conference on Software and Data Technologies (ICSOFT-2007), July 2007, pp. 319-326.

Davison, A. and Kantor, R. 1982. *On the failure of readability formulas to define readable texts: A case study from Adaptations*. Reading Research Quarterly 17, 2, 187-209.

De Cesare, Lycett, S. and Paul, R. 2003. *Actor Perception in Business Use Case Modeling*, Communications of the Ais, Vol 12, pp. 223-241.

Denger, C. 2002. *High Quality Requirements Specifications for Embedded Systems through Authoring Rules and Language Patterns*, M.Sc. Thesis, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany 2002.

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

DOORS. Telelogic's DOORS. "Requirements management traceability solutions". Available online as of March 31, 2007 at http://www.telelogic.com/products/doorsers/index.cfm.

El-Attar, M. and Miller, J. 2006. *Matching Antipatterns to Improve the Quality of Use Case Models*, in Proceeding of the 14th IEEE International Requirements Engineering Conference (RE'06)", p.99-108.

Ellison, J. and Moore, P. 2002. *Trustworthy Refinement Through Intrusion-Aware Design (CMU/SEI-2003-TR-002)*. Technical Report. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr002.html>.

Emiliani, C. 1987. *Dictionary of the Physical Sciences: Terms, Formulas, Data*. New York, Oxford: Oxford University Press.

Entin, E., and Klare, R. 1985. *Relationships of measures of interest, prior knowledge, and readability to comprehension of expository passages,* In Advances in reading/language research, B. Hutson, Eds. Greenwich, Conn, Jai Press Inc, 9-38.

Eriksson, M., Börstler, K., and Borg, K. 2004. *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, pp.73-82.

Fabbrini, F., Fusani, M., Gnesi, S., and Lami, G. 2001. *An Automatic Quality Evaluation for Natural Language Requirements*. In Proceedings of the Seventh International Workshop on Requirements Engineering: Foundation for Software Quality (Interlaken, Switzerland, June). Essener Informatik Beiträge.

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

Fuentes, R., Gómez-Sanz, J. and Pavón, J. 2005. *Requirements Elicitation for Agent-Based Applications*. In Proceedings of AOSE'2005. pp.40~53

Geisser, M., Hildenbrand, T., Rothlauf, F., and Atkinson, A. 2007. *An Evaluation Method for Requirements Engineering Approaches in Distributed Software Development Projects*. In *Proceedings of the* Second International Conference on Software Engineering Advances (Cap Esterel, French Riviera, France, August). IEEE Computer Society Press, 39-39.

Georgiades, M., Andreou A., and Pattichis, C. 2005. *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, 73-74.

Georgiades, M. and Andreou A., 2010. *Automatic generation of a Software Requirements Specification (SRS) document*. In Proceedings of the Intelligent Systems Design and Applications Conference (Cairo, Egypt, November). IEEE, 1095-1100.

Georgiades, M. and Andreou, A. 2010. *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, October). IADIS Digital Library, 11-18.

Georgiades, M., and Andreou, A. 2010. *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, October). IEEE Press, 256-263.

Georgiades, M. and Andreou, A. 2011. *Formalizing and Automating Use Case Model Development*, The Open Software Engineering Journal. Accepted.

Georgiades, M. and Andreou, A. 2011. *A Methodology to Formalize and Automate the Requirements Engineering Process with the Use of Natural Language*, Requirements Engineering. Submitted.

Georgiades, M. and Andreou, A. 2011. *A Novel Software Tool for Supporting and Automating the Requirements Engineering Process With the Use of Natural Language*, International Journal of Computer Science and Technology. Accepted.

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

Glinz, M. 1995. *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), pp. 254-271, September.

Goldin, L. and Berry, D. 1997. *Abstfinder: A prototype natural language text abstraction finder for use in requirement elicitation*. Automated Software Engineering 4, 4, 375–412.

Hall,T., Beecham, S., and Reainer,A. 2002. *Requirements problems in twelve software companies: An empirical analysis*. IEEE Proceedings - Software 149, 5, 153–60.

Hofmann, H. F. and Lehner, F. 2001. *Requirements Engineering as a Success Factor in Software Projects*, IEEE Software, Vol. 18, No. 4, pp.58-66.

Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y. and Chen, C. 1994. *Formal approach to Scenario Analysis*, *IEEE Software*, vol. 11, number 2, March 1994.

IBM Rational Rose. Available online as of July 12, 2010 at http://www-306.ibm.com/software/rational/

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

Iivari J. 1991. *Object-oriented information systems analysis. A framework for object identification.* In Proceedings of the Twenty-Fourth Annual Hawaii International Conference on Systems Sciences, B. Shriver Eds. IEEE Computer Society Press.

Jacobson, I. 2004. *Use cases - Yesterday, today, and tomorrow*, Software and System Modeling, vol 3, number 3, pp. 210-220.

Kassel, N. and Malloy, B. 2003. *An Approach to Automate Requirements Elicitation and Specification*. In Proceedings of the Seventh IASTED Int. Conf. on Software Engineering and Applications (Marina Del Ray, CA, November). IASTED/ACTA Press, 544-549.

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

Kotonya, G. and Sommerville, I. 1998. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons.

Kovitz B. 2003. *Hidden skills that support phased and agile requirements engineering*. Requirements Engineering 8(2):135–141.

Lami, G., Ferguson, R. Goldenson, D. Fusani, M. Fabbrini, F., and Gnesi, S. 2005. *QuARS: Automated Natural Language Analysis of Requirements and Specifications.* In Proceedings of Seventeenth Annual System & Software Technology Conference (Salt Lake City, Ut, April).

Larman, C. 2002, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Second Edition, Prentice-Hall.

Le, T. and Rolland, C. 2001. *Functional matching in COTS-based development context*, INFORSID 2001, Genève, 29 may, 2001, pp. 87-110.

Leffingwell D. 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise.* Addison-Wesley Professional.

Leite, J., Rossi, G., Balaguer, M., Kaplan, G., Hadad, G. and Oliveros, A. 1997. *Enhancing a Requirements Baseline with Scenarios*, in Proceedings of Requirements Engineering, Annapolis, USA.

Leite, P. and Gilvaz, A. 1996. *Requirements Elicitation Driven by Interviews: The Use of Viewpoints,* Proceedings of 18th International Workshop on Software Specification and Design (IWSSD-8), pp. 85-94.

Li, Y., Huang, D., Tsang, E, and Zhang, L. 2005. *Weighted fuzzy interpolative reasoning method.* In Proceedings of the fourth international conference on machine learning and cybernetics (Guangzhou, China, August). Lecture Notes in Computer Science, vol. 3930, Springer, 3104-3108.

Ling, S., Quercia, D. and Finkelstein, A. (2010). *StakeNet: using social networks to analyse the stakeholders of large-scale software projects.* In Proceedings of the 32nd IEEE International Conference on Software Engineering. ICSE (1), pp. 295-304.

Liu, D., Kalaivani, S., Armin, E. and Behrouz, F. 2004. *Natural Language Requirements Analysis and Class Model Generation Using UCDA*, Lecture Notes in Computer Science Volume 3029/2004, Innovations in Applied Artificial Intelligence: 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE, Springer, pp. 295-304.

Loucopoulos, P. and Karakostas, V. 1995. *System Requirements Engineering*, McGraw Hill, London.

Loucopoulos P. and Kavakli E. 1995. *Enterprise Modeling and the Teleological Approach to Requirements Engineering.* International Journal of Intelligent and Cooperative Information Systems, Vol. 4, No. 1 pp. 45-79.

Lyons, J. 1968. *Introduction to Theoretical Linguistics*. Cambridge U.P., London, Print.

Macaulay, L. 1996. *Requirements for Requirements Engineering Techniques*. IEEE Proceedings of ICREMEYERS, R. 2001. Encyclopedia of Physical Science and Technology, 3rd ed., Academic Press.

MagicDraw. Available online as of July 12, 2010 at http://www.magicdraw.com

Maiden, N. 2004 *Discovering Requirements with Scenarios. The ART-SCENE Solution,* in *ERCIM News*, vol. 58, July 2004.

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

Mehandjiska D., and Palac, J., 2002. *Towards Bridging Component Specification Technologies*, International Conference on Software Engineering, the 20th IASTED International Multi-conference Applied Informatics (AI2002), Austria.

Meyer, B., J. Nawrocki, Walter, B. 2008. *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

Mich, L., Franch, M., and Inverardi,P. 2004. *Market research for requirements analysis using linguistic tools*. Requirements Eng. J. 9,1, 40–56.

Moody D. 2003. *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).

Moody, D. and Shanks, G. 1998. *What Makes a Good Data Model? Evaluating the Quality of Data Models.* Australian Computer Journal, 97-110.

Moreno, A. and Reind P. van de Riet. 2001. A*pplications of Natural Language to Information Systems*. In Proceedings of the 6th International Workshop NLDB'01 (Madrid, Spain, June). Lecture Notes in Informatics, vol. 3, GI.

Neill, C., and Laplante, P. 2003. *Requirements Engineering: The State of the Practice*, IEEE Software, vol. 20, no. 6, pp. 40-45, Nov./Dec. 2003, doi:10.1109/MS.2003.1241365

Nguyen, L. and Swatmann, P. A. 2000. *Managing the Requirements Engineering Process*, School of Management Information Systems, Deakin University, Geelong, Australia.

Nuseibeh, B. and Easterbrook, S. 2000. *Requirements Engineering: A Roadmap.* In Proceedings of International Conference on Software Engineering (ICSE) - Future of SE Track (Limerick, Ireland, June). ACM Press, 35-46.

Ochodek, M. and Nawrocki, J. 2007. "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.

Pfleeger, S. 2001. Software Engineering: Theory and Practice, Second ed: Prentice-Hall, 2001, Chap. 4.Pohl, K. 2010. Requirements Engineering - Fundamentals, Principles, and Techniques. Springer

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

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

Rational Software Corporation, "Rational RequisitePro User's Guide," June 2003, http://www.se.fhheilbronn.de/usefulstuff/Rational%20Rose%202003%20Documentation/reqpro_user.pdf.

Rayson, P., Emmet, L., Garside, R., and Sawyer, P. 2000. The REVERE Project: Experiments with the Application of Probabilistic NLP to Systems Engineering. In Proceedings of the 5th International Conference on Applications of Natural Language to Information Systems (Versailles, France, June). Springer-Verlag, London, 288-300.

Respect_IT, *"A KAOS Tutorial"*, v1.0, Oct 2007, Available as of November 2009 at http://www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf

Robertson S. 2001. Requirements trawling: techniques for discovering requirements. Int. J. Hum.-Comput. Stud. 55(4): 405-421 (2001)

Rolland, C. and Proix. C. 1992. A Natural Language Approach for Requirements Engineering. In Advanced Information Systems Engineering, P. Loucopoulos Eds. Springer-Verlag, 257-277.

Rolland, C., Souveyet, C., Achour, B. 1998. "Guiding GoalModeling Using Scenarios", *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1055-1071.

Saeki, M. 2010. Semantic Requirements Engineering. *In Intentional Perspectives on Information Systems Engineering*, Springer-Verlag Berlin Heidelberg.

Sawyer, P., Rayson, P., and Garside, R. 2002. REVERE: Support for Requirements Synthesis from Documents. Information Systems Frontiers 4, 3, 343-353.

Scenario Plus, http://www.scenarioplus.org.uk/, Accessed 20 January 2011.

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

Sharma, A. 2009. Requirements quality assessment for outsourcing. Master's thesis. Eindhoven University of Technology, Eindhoven, Netherlands.

Shlaer, S. and Mellor, S. 1992. Object lifecycles - modeling the world in states. Yourdon Press: I-XIII, 1-251.

Shukur, Z., Zin, A., and Ban, A., 2002. M2Z: A Tool for Translating a Natural Language Software Specification into Z. International Conference on Formal Methods and Software Engineering, ICFEM 2002: 406-410.

Sommerville I., Sawyer P., Viller S. 1997. Viewpoints for requirements elicitation: A practical approach. In: Proceedings of 3rd IEEE International Conference on Requirements Engineering. CO, USA, pp.74-81

Sommerville, I. and Ransom, J. 2005. An empirical study of industrial requirements engineering process assessment and improvement. ACM Trans. on Software Eng. and Methodology (TOSEM) 14, 1, 1-33.

Sommerville, I. 2005. Integrated Requirements Engineering: A Tutorial. IEEE Software 22, 1, 16-23.

Sommerville, I. 2010. *Software Engineering*, 8th ed, Addison Wesley

Song, I., Yano, K., Trujillo, J., and Luján-Mora, S. 2005. *A Taxonomic Class Modeling Methodology for Object-Oriented Analysis.* Information Modeling Methods and Methodologies, 216-24.

Sybase, 2002. *PowerDesigner*. *Object Oriented User's Guide*. Sybase.

Thayer, R. and Dorfman, M. 1997. *Software Requirements Engineering*, 2nd ed. Los Alamitos, CA: IEEE Computer Society Press.

The Standish Group. 2003. "What are your requirements?" Technical report, The Standish Group International, Inc.

The Standish Group. 2009. "Chaos summary 2009". Available as of December 2010 at http://www.standishgroup.com.

Tjong, S. and Berry, M. 2008. *Can Rules of Inferences Resolve Coordination Ambiguity in Natural Language Requirements Specification?* In Proceedings of the Eleventh Workshop em Engenharia de Requisitos (Barcelona, Spain, September).

Tjong, S, Hallam, N., and Hartley, M. 2006. *Improving the Quality of Natural Language Requirements Specifications through Natural Language Requirements Patterns.* In

Proceedings of the Sixth IEEE International Conference on Computer and Information Technology (Seoul, Korea, September). IEEE Computer Society, 199-199.

Van Lamsweerde, A. 2000. *Formal Specification: a Roadmap*, In: The Future of Software Engineering, ACM Press, **2000**, pp.147-160, 112875.

Van Lamsweerde, A. 2000. *Requirements Engineering in the Year 00: A Research Perspective*. Invited Keynote Paper, In Proc. ICSE'2000:22nd International Conference on Software Engineering, ACM Press, 2000, pp. 5-19.

Van Lamsweerde, A. 2001. *Goal-Oriented Requirements Engineering: A Guide Tour,* In Proceedings RE'01, 5th IEEE International Symposium on Requirements *Engineering, Toronto, August 2001, 249-263.*

Van Lamsweerde, Dardenne, A., Delcourt, B. and Dubisy, F., *The KAOS Project: Knowledge Acquisition in Automated Specification of Software,* In Proc. AAAl Spring Symp. Series, Track: "Design of Composite Systems", Stanford University, March 1 99 1 , 59-62.

Van Vliet, H. 2008. *Software Engineering: Principles and Practice*, 3rd edition, John Wiley & Sons

Videira, C. and da Silva, A. 2005. *Patterns and metamodel for a natural-language-based requirements specification language.* In Proceedings of the Seventeenth Conference on Advanced Information Systems Engineering, CAiSE Forum (Porto, Portugal, June). 189-194.

Videira, C., Ferreira, D., and da Silva, A. 2006. *Using linguistic patterns for improving requirements specification.* In Proceedings of the First International Conference on Software and Data Technologies Setubal, Portugal). INSTICC Press, 145-150.

Westfall, L. 2006. "Software Requirements Engineering: What, Why, Who, When, and How". The Westfall Team.

Wiegers, K. E. 2004. "In search of excellent requirements". Process Impact Web site. Available online as of January 22, 2010 at http://www.processimpact.com.

Wiegers, K. 2006. *More About Software Requirements*. Microsoft Press, 2006

Wikipedia, 2010, "Information Systems," Available online as of July 15, 2010 at http://en.wikipedia.org/wiki/Information_systems.

Zave, P. and Jackson, M. 1997. *Four Dark Corners of Requirements Engineering*. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 1, pp. 1-30.

Zhang, Z. 2007. *Effective Requirements Development - A Comparison of Requirements Elicitation Techniques.* Software Quality Management XV: Software Quality in the Knowledge Society, E. Berki, J. Nummenmaa, I. Sunley, M. Ross and G. Staples (Ed.) British Computer Society, pp. 225-240

Zielczynski, P. 2007. *Requirements Management Using IBM. Rational RequisitePro*. IBM Press.

Zowghi, D and Coulin, 2005. *Requirements Elicitation: A Survey of Techniques,Approaches, and Tools.* Book Chapter in "Engineering and Managing Software Requirements" Edited by Aybuke Aurum and Claes Wohlin, Published by Springer.

# Appendix A Requirements Discovery Questionnaire

Part I. Documents

1. What documents, in electronic or paper form (e.g. forms, receipts, reports), which you create from scratch or change/complete after the recipient or someone else created them, do you send/ give/ show to $User_{1..n}$ with $Role_{1..m}$?
    a. in person
    b. through another person or service
    c. electronically (e.g. e-mail, internet)

    Auxiliary questions:

    - Do you give/ send/ show any documents to $User_{1..n}$ with $Role_{1..m}$?
    - Do you write any documents for $User_{1..n}$ with $Role_{1..m}$?
    - Do you sign any documents for $User_{1..n}$ with $Role_{1..m}$?

2. What documents, in electronic or paper form (e.g. forms, receipts, reports), which you modify/ complete after the recipient or someone else created them, and/or asked you to modify/ complete them, do you send/ give/ show to $User_{1..n}$ with $Role_{1..m}$?
    a. in person
    b. through another person or service
    c. electronically (e.g. e-mail, internet)

    Auxiliary questions:

    - Do you modify any documents for $User_{1..n}$ with $Role_{1..m}$?

3. What data are or should be included about the document's
    a. Creator?
    b. Author?

c. Purpose?
d. Recipient(s)?
e. Communication channel?
f. Form?
g. Other people (e.g. users) that should be notified about the creation or modification of the document?
h. Procedure (s) mentioned in the document or related to the ones mentioned in the document?

4. What feedback (vocal or written) do you receive from the recipient, after you send the document to him/ her?
    a. Does s/he makes any change and sends it back to you?

5. What initiates the sending procedure?
    a. A request (written or vocal) from the recipient?
    b. A request (written or vocal) from another user?
        i. What is the role of this user?

6. Can you please provide a copy of each aforementioned document?

Note for the Requirements Engineer: The same set of questions should be provided to the same user but in the role of the receiver.

Part II. Physical Items

1. What physical items do you send/ give to $User_{1..n}$ with $Role_{1..m}$?
    a. in person
    b. through another person or service
    c. by post
2. What data are or should be recorded in the system or provided to the recipient about this item's
    a. Purpose?
    b. Recipient(s)?
    c. Communication channel?
    d. Other people (e.g. users) that should be notified about giving/ selling this item?

3. What documents (e.g. forms, orders, receipts, reports) are related to this transaction (sending) and to this item in general?
    a. Do you send any of these documents to any user or stakeholder?

4. What feedback (vocal or written) do you receive from the recipient, after you send/ give the item to him/ her?

5. What initiates the sending procedure?
    a. A request (written or vocal) from the recipient?
    b. A request (written or vocal) from another user?
        i. What is the role of this user?

6. Can you please provide a copy of each aforementioned document?

Note for the Requirements Engineer: The same set of questions should be provided to the same user but in the role of the receiver.

Part III. Requests

1. What other requests or orders do you give to $User_{1..n}$ with $Role_{1..m}$?
    i. In hand
    ii. Through another person
    iii. By voice (e.g. via telephone, skype)
    iv. Written
    v. Electronically (e.g. e-mail, internet)
    vi. Optically

2. Describe in detail the content of each request.

3. What response do you receive by the recipient?
    i. In hand
    ii. Through another person
    iii. By voice (e.g. via telephone, skype,)
    iv. Written
    v. Electronically (e.g. e-mail, internet)
    vi. Optically

4. What does trigger this request?

5. What decisions do you or the recipient take (if any), after your communication?

Note for the Requirements Engineer: The same set of questions should be provided to the same user but in the role of the receiver.

# Appendix B Experimental Evaluation Results

Below we provide an indicative example of the application of NLSSRE to the development of a part of a Library Information System.

**Step 1. Collect the Candidate Information Objects**

a. Identify roles and users

To identify the business roles, we first received a description of each user's work, from a representative number of users (1 user for each role, due to the small size of the project). To define this representative sample, we first discussed with a coordinative role, such as the main librarian. Below is the list of the roles for the LIS we handled:

*Librarian, Member, Client, Supplier, Stock Keeper, Accountant*

b. Collect information about items exchanged between users of each role

A data flow table was created to identify the information and items exchanged between the roles involved in the system. Here is a portion of the table:

TABLE B.1 PORTION OF THE DATA FLOW TABLE FOR THE LIS CASE STUDY

| Sender \ Receiver | Librarian | Member | Client | Supplier |
|---|---|---|---|---|
| Librarian | | Book, journal, article, notification for subscription application/renewal, reservation, book borrowed report, nook delivery, lending form, payment receipt | Payment receipt, lending form | Order form |
| Member | Book return, journal return, article return, ID, subscription, subscription payment application/renewal, book request, article request, journal request, reservation | | | |
| Client | Payment per item, lending form completed | | | |
| Supplier | Book, journal, article, payment receipt | | | |

From the above information, we derived the list of the candidate IOs. The list includes the following:

*Librarian*, *Book*, *Member*, *Client*, *Supplier*, *Journal*, *Article*, *Application*, Book Borrowed Report, *Payment*, *Order form*, *Payment Receipt*, *Lending form*.

**Step 2. Identify the Information Objects**

a. Apply rules to identify the actual IOs from the candidate ones

Three sample rules are as follows:

- Lending of a *book* from the *Librarian* (Library) to a member or client, causes the creation of instances of other IOs, such as a *Payment* IOi, a *Delivery* IOi, and a *Book Order* IOi. It also causes the alteration of the attribute *State/Status* of a *Book* IOi, from *available* to *lent*). Thus *Book* is an IO.
- The *Supplier*, in the role of the *Sender*, sells books to the *Librarian* who is the *Receiver*. This transaction causes instances of other IOs to be created such as a Supplier's *Payment* IOi and one or more *Book* IOi corresponding to new books acquired by the library. Therefore *Supplier* and *Librarian* are IOs.
- A *Book Borrowed Report*, which is a collection of attributes of other IOs (*Book* and *Member*) and can be created automatically by the system, is not an IO.

The result of this step was the list of the actual Information Objects: *Librarian, Book, Member, Client, Supplier, Journal, Article, Payment, Delivery, Order, Subscription, Lending, Reservation,* and *Renewal.*

b. Categorize IOs

Categorizing IOs helps us applying more effectively the subsequent steps, e.g., identifying attributes based on the IO category.

*Librarian, Assistant, Accountant*: Internal Business Roles

*Book, Article, Receipt*: Physical Object, Document

*Member, Client, Supplier*: External Business Roles

*Reservation, Lending, Subscription,Renewal,Payment, Delivery, Order*: Procedure

**Step 3. Develop FSRs for each IO**

a. Apply CAREN functions and their sub-functions on each IO

(for simplicity, sub-functions are omitted from the example)

The alteration CAREN function can lead to the identification of new functions or IOs, as illustrated below. Such an IO is the *LIS* which should perform automatically, in the functional role of *Creator* in the FSRs, the *Lend* and *Reserve* functions which are alteration-like functions. These concepts are illustrated clearly in the use case diagram of figure B1.

i. *Librarian IO*

Create, Alter, Alter::Replace, Read, Erase, Notify

Normal state: Works; Alteration States: Worked, Ill

ii. *Book IO*

Create, Alter, Alter::Archive, Alter::Reserve, Alter::Lend, Read, Erase, Notify

Normal state: Available; Alteration states: Lent, Not Available

*Lend* and *Reserve* states lead to the creation of new IOs → IO *Lending*, IO *Reservation*

iii. *Member* IO

Create, Alter, Read, Erase, Notify

Normal state: Subscribed; Alteration states: Unsubscribed, Suspended

*Subscribed* state leads to the creation of a new IO → IO *Subscription*

b. Specify the IO, its CAREN functions, the involved functional roles and functional condition types, in the form of formalized sentential requirement (FSR) patterns

i. *Book* IO FSRs

<Cr, Acc> <**Create**> <Book> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Alter**> <Book> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Archive**> <Book> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Reserve**> <Book> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Lend**> <Book> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Exp> <**Read**> <Book> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Erase**> <Book> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

ii. *Member* IO FSRs

<Cr, Acc> <**Create**> <Member> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Alter**> <Member> <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Suspend**> < Member > <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Unsubscribe**> < Member > <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Exp> <**Read**> < Member > <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

<Al, Acc> <**Erase**> < Member > <T,I,P,A>:: SystemNotifies <IR> <R> <T,I,P,A>

c. <u>Make questions, derived from FSRs, to find the business roles and values for functional conditions</u>

The answers should be given by the system users or by the information received during the first step of the methodology. Using synonyms also helps the respondent think better of the answer. Here we provide indicative questions for the *Create* FSRs, regarding the IOs *Book* and *Member*.

i. *Book IO Questions*

- Who is responsible to record a new book in the system? (*Librarian*)

- Does/Should any human or computer system help the stock to record a new book, e.g., by providing some relevant data, about the book, its

category, etc.? (*Classification System*)

- What instruments does/should the librarian use to record a new book? (*Keyboard*, *ISBN bar code scanner*)

- Who is/should be notified about the creation/cataloguing of a new book? (*Librarian*, as s/he is the creator; *Member*, as the intended recipient who e.g., requested or reserved this book, and so s/he is expected to borrow it).

- How the *Librarian* and *Member* should be notified? (e-mail)

ii. *Member IO Questions*

- Who is responsible to record a new member in the system? (*Librarian*)

- Does/Should any human or computer system help the librarian record a new member, e.g., by providing some relevant data, about the member? (*Member*)

- What instruments does/should the librarian use to record a new member? (*Keyboard*, *ID Scanner*)

- Who is/should be notified about the creation of a new member? (*Librarian*, as s/he is the creator; *Member*, as the client).

d. <u>Specify complete FSRs, based on the answers received for each FSR pattern element</u>

i. *Book IO complete FSRs (only the Create FSR is presented)*

<Librarian, Classification System> <**Create**> <Book> <Keyboard, ISBN bar code scanner>:: SystemNotifies <Member> <Librarian> <e-mail>

ii. *Member IO complete FSRs (only the Create FSR is presented)*

<Librarian, Member> <**Create**> <Member> <Keyboard, ID scanner>:: SystemNotifies < > <Member, Librarian> <e-mail>

**Step 4. Define attributes for each IO**

- This step involves identifying attributes for each IO, based on their category and the category of the IO, and making specific questions to the user to confirm the attributes and derive new ones. Here we provide a number of indicative attributes.

i. Attributes derived from the use of the *FSR* attribute category for:

   i. *Book IO* (from the Create Book FSR only)

      *Librarian* related attributes

      *Classification System* related attributes

      *ISBN Bar Code related attributes*

   ii. *Member IO* (from the Create Member FSR only)

      *Librarian* related attributes

      *Member ID*

ii. Attributes derived from the use of the *Comparative* attribute category for:

   i. *Book IO*

      *Rank by lending time: 1-day; 2-days; 3-days; 1-week; 1-month*

   ii. *Librarian IO*

      *Rank by years of work: 5-year; 15-year;*

iii. Attributes derived from the use of the *Animate* attribute category for Librarian IO:

      *Name, Surname, Age, Address, Telephone No*

iv. Attributes derived from the use of the *Document* attribute category for Book IO:

      *Number of pages, Title, Number of words*

**Step 5. Define Business Rules**

a. Inter-related business rules (between 2 or more IOs)

i. *Between Book and Member*

Since *Book* and *Member* can be found in the same FSR, there is a big possibility for an interrelated business rule to exist.

$FSR_{Book}^{Cre} =<$ Librarian, Classification System $><$ **Create** $><$ Book $>$
$<$ Keyboard, ISBN bar code scanner $>::$ SystemNotifies $<$ Member $><$ Librarian $><$ e − mail $>$

*Rule1: If Member.ID causes Book.Condition=Bad, THEN Member.Balance is charged.*

*where Bad is defined accordingly, by the Library Organization.*

*Rule2: If Book.State!=Reserved AND Book.Availability!=0, THEN Member.ID can borrow book.*

*Rule3: If Book.State=Reserved AND Book.Availability=0, THEN Member.ID can apply to reserve book.*(there are more cases for reservation-this is only one of them)

*b.* Intra-related business rules (between attributes of the same IO)

*i. Between Book attributes*

*Rule1: If Book.Condition=Bad, THEN Book.Archived=Yes.*

**Step 6. Develop SRS and Diagrams**

*a.* SRS Document. The SRS document follows an organization by object. Here we provide a portion of the specification of the IO *Book*, based on the provided predetermined template.

| **Specific requirements Template (organized by object)** | **Specific requirements (organized by object)** |
|---|---|
| ○ Classes/Objects<br>☐ <IO 1><br>• Attributes (direct or inherited)<br>○ FSR Attributes<br>▪ Cr: <Cr 1> ID, <Cr 2> ID, …, <Cr n> ID<br>▪ Ac: <Ac 1> ID, <Ac 2> ID, …, <Ac n> ID<br>▪ No: <No 1> ID, <No 2> ID, …, <No n> ID<br>▪ <IR 1> ID, <IR 2> ID, …, <IR n> ID<br>○ Physical Attributes<br>……<br>• Functions (direct or inherited)<br>○ Create <IO 1><br>▪ Description: <Cr 1> , …., <Cr n> create(s) <IO 1> with the assistance of <Ac 1>, …, <Ac n> .<br>▪ Details:<br>▪ If <At 1>: <InputValue> is True for "<Constraint At 1>", then record <At 1>: <InputValue>. If False, then show message "<At 1>: <InputValue> " is not valid.<br>…<br>○ Alter <IO 1> …<br>○ Read <IO 1> …<br>○ Erase <IO 1> …<br>• Messages (notifications received or sent)<br>○ System notifies <No 1>, …, <No n>, <IR 1>, …, <IR n> that <IO1> is created.<br>○ …<br>☐ <IO 2><br>… | ○ Classes/Objects<br>☐ <Book><br>• Attributes (direct or inherited)<br>○ FSR Attributes<br>▪ Cr: Librarian ID<br>▪ Ac: Classification System<br>▪ No: Librarian ID<br>▪ IR: Member ID<br>○ Physical Attributes<br>▪ Condition<br>▪ Material<br>▪ Size<br>○ Document Attributes<br>▪ ISBN<br>▪ Title<br>▪ Number of pages<br>▪ Number of words<br>• Functions (direct or inherited)<br>○ Create Book<br>▪ Description: Librarian create(s) Book with the assistance of Classification System.<br>▪ Details:<br>▪ If Book.Title: <InputValue> is True for "Only alphabetic characters are allowed", then record Book.Title: <InputValue>. If False, then show message "Book Title: <InputValue> " is not valid.<br>…<br>• Messages (notifications received or sent)<br>○ System notifies Librarian, Member, that Book is created. |

b. Use Case Model. Here we present the Use Case diagram for the *Book* IO. The FSRs of the *Book* IO were used to facilitate the transformation.

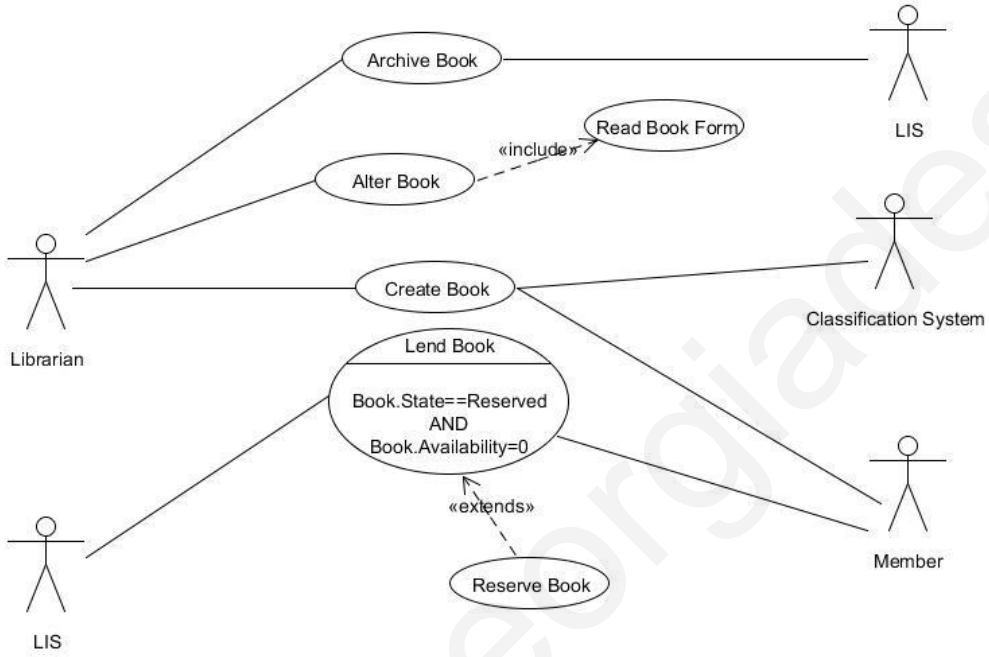


FIGURE B.1. PORTION OF THE ENTIRE USE CASE DIAGRAM FOR THE LIS.