NOTE TO USERS

uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Nayanamana Samarasinghe
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Supporting Instances in Use Cases

TITRE DE LA THÈSE / TITLE OF THESIS

Stéphane Somé
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Jean-Pierre Corriveau

Liam Peyton

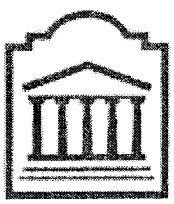
Alan Williams

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

SUPPORTING INSTANCES IN USE CASES

Nayanamana Samarasinghe

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of
Masters of Computer Science



School of Information Technology & Engineering
University of Ottawa

Your file Votre référence
ISBN: 0-494-14946-9
Our file Notre référence
ISBN: 0-494-14946-9

ABSTRACT

Requirements Engineering includes elicitation and analysis of system requirements. There are many techniques used for requirement elicitation. The use case approach is one such technique. This technique uses textual use cases written in a natural language for gathering requirements. Textual use cases are simple and easy to understand because they are written using free form text. But in order to automate requirements analysis, use cases need to be expressed using a restricted form of language according to a set of guidelines. An advantage of a restricted form of natural language is that it can be used to mechanically process use cases while still being understandable to end users.

A use case specifies interactions between a system and its environment. Sometimes interactions need to be expressed between instances. In the existing restricted natural language syntaxes there is usually no distinction between entities and instances of entities. Another issue is that use cases are expressed in term of domain elements. In most of the cases, a domain model is created and updated manually in an ad hoc manner.

One of the contributions of this thesis is a restricted natural language syntax for use cases, where instances are explicitly expressed. Another contribution is an approach to automate domain element extraction from use cases. Finally, we implemented our results as part of UCEd (Use Case Editor): a tool that supports use cases based requirements engineering.

ACKNOWLEDGEMENTS

I am thankful to God for giving me the opportunity of doing a Masters degree in Computer Science. It was a one of my aspirations and a timely decision in life.

I was very fortunate in finding an amicable, encouraging and supportive supervisor: Dr. Stéphane Somé for my degree program. I'm very greatful for the financial assistance that was funded by him towards my studies. I also appreciate the proactive interactions initiated by him, which ultimately resulted in the effectiveness and efficiency of my research work.

I'm also greatful to Dr. Stan Szpakowicz in guiding and mentoring me in the field of Natural Language Processing. His teaching was undoubtedly an initial guidance for my research that involved writing of use cases using Natural Language.

It would be incomplete if I do not mention my professors and colleagues during my undergraduate studies at University of Colombo, Sri Lanka. I should thank specifically Dr. Nihal Kodikara and Dr. Ruwan Weerasinghe for their tremendous support, guidance and in moulding me for graduate studies.

My family back home has permanently been in my mind during my work. My mother had been the first teacher for me in life. If not for her guidance I would not have come this far and I am very much greatful to her for all the support that was extended by her. In addition, I should also specifically mention my late father, sister and brother for their love and contributions towards my success.

Last but not least, I should not forget all my friends whom had been a constant encouragement for me during good and bad times during the period of my study.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

Requirements Engineering is merely the high level specification of software systems. Requirements engineering is not a smooth and a clear cut process. This is because there are not much well established guidelines for requirements engineering. It is also impossible and impractical to have a set of universal guidelines or a framework for supporting requirements engineering. One main reason for this fact is that in addition to dissimilar systems, the nature of a future system varies drastically with other proposed similar systems.

According to Frederick Brooks, "The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements.... No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later." [1]

The above quotation highlights the fact that if precise requirements cannot be gathered during requirement elicitation, the after effects could be irreversible. Requirements gathering become complicated as the volume of requirements increases. Most software development processes including the Rational Unified Process (RUP) [2], require that extensive guidelines be followed for requirements gathering.

There are various techniques used for requirement elicitation and many are under research. Use cases approach is one of the major elicitation techniques. Since their introduction of use cases by Jacobson [3, 4] different guidelines have been proposed for writing use cases in text form. One of these guidelines was proposed by Alistair Cockburn [5, 6]. Cockburn suggested that use cases be written using a restricted form of natural language in order to ensure consistency and facilitate the mechanical processing of use cases. Although Cockburn did not explicitly define a restricted natural language grammar for use cases, several such grammars have since been proposed [7-10]. Most grammars do not provide explicit support for writing instances based use cases. This hinders their expressive power for requirements elicitation.

Another problem with requirements elicitation is a gap between domain knowledge and requirements captured. Use cases are a good mechanism for capturing system requirements in a way that minimizes this gap. However, extracting domain knowledge from a use case model is essentially a manual process.

Instances play an important role in the expression of requirements. Sometimes it is not possible to clearly specify requirements without using instances. Consider as an example, a use case related to a telephone call between two parties. Suppose that these parties need to be represented by a single domain entity: *caller*. In such a context, without instances it is difficult to clearly write the appropriate use case descriptions. For instance, the following statement of interaction *"Ann calls John"* where both *"Ann"* and *"John"* are instances of domain entity: *caller* would not be possible.

Use cases need to be parsed in order to identify domain elements. But parsing involves knowledge of language elements. Such knowledge is often contained in a domain model. In addition, domain modeling supposes being able to categorize entities and find relationships between them. As an example, we would like to be able to determine which entity is a concept from the environment, which entity is a system concept, which entity is an attribute, and so forth. Manual extraction of missing elements of a domain is a tedious and a time consuming task. Therefore, an automatic process for extracting missing domain elements appears to be effective and efficient.

We implemented a natural language grammar that supports writing instances based use cases. This grammar is integrated to a parser, which supports extraction of missing domain elements. However, this parser exhibits certain limitations. One of these limitations is that our natural language parser does not allow distinguishing between different categories of entities. A solution to that limitation is offered by the integration of our approach to the UCEd (Use Case Editor). UCEd is a framework for use cases modeling. UCEd use cases are written in a semi-formal language. The framework involves a domain model that is updated manually as and when domain elements are identified in use case descriptions. UCEd has the advantage of distinguishing between different categories of domain entities. In addition,

UCEd has the capability of generating a state model for each use case model. UCEd generates a state model by using domain operations and postconditions. A limitation of UCEd is that it doesn't support instances. The integration of our grammar with UCEd provides a solution to the limitation of not being able to distinguish categories of entities on one hand. On the other hand, this integration provides a solution to UCEd limitation of not being able to support instances and the manual extraction of domain elements.

When instances are integrated into UCEd use cases, the corresponding state models generated become complicated. This is because the state conditions generated should now reflect the impact of instances. We propose a solution for this limitation by defining parameter based operation postconditions.

After integrating instances into use cases and implementing the automatic domain extraction utility in the UCEd framework, we applied our approach to a case study in order to verify the accuracy of our work. The case study consisted of a set of instances based use cases. The corresponding instances based state conditions of the state model generated was then reviewed and it proved to be valid. Further, a domain model was generated from scratch using the domain extraction utility. The resulting domain generated also appeared to be in consistent with the corresponding use case model of the case study.

1.1 Thesis Contributions

This thesis offers three major contributions:

- A use case grammar that explicitly supports expression of interactions involving instances.
- An automated approach to extract domain elements from use case descriptions.
- Integration of instances and domain extraction to the UCEd framework providing enhancements to both our use case parsing approach and UCEd.

1.2 Thesis Outline

This thesis is structured as follows:

- Chapter 2 describes the context of our work, including the level of abstraction of use cases that is relevant to our work.
- Chapter 3 details different formats for elaborating use cases. It also illustrates different grammars and guidelines for writing use case descriptions. Subsequently, a critique on all of the grammars reviewed is stated.
- Chapter 4 presents a natural language parser that is used to demonstrate the contributions of the thesis. It also discusses a case study which is used for evaluating the functionality of the natural language parser.
- Chapter 5 presents the UCEd framework that is used to implement some of the proposed enhancements for the shortcomings observed in the natural language parser.
- Chapter 6 explains how instances are implemented in UCEd use case grammar and the impact of instances based use cases on the corresponding domain and state models that are been generated.
- Chapter 7 presents a semi-automatic technique for extracting a domain model which is integrated to UCEd. The benefit of this domain extraction approach is that it facilitates identification of domain sub-entity types.
- Finally, Chapter 8 recalls the main contributions of the thesis, related work, limitations and some directions for future research.

Chapter 2 - Context of work

Requirements Engineering is one of the most important and complicated tasks in Software Engineering. It has been always a challenge to reduce the communication gap that exists between domain specialists and requirement analysts during requirements analysis. One of the reasons is that informal languages used to capture requirements from domain specialists are inherently ambiguous. This chapter starts by introducing requirement engineering. Then we will focus our attention to problems that hinders the requirements engineering process. Afterwards we will brief on formal and informal languages used for requirements specification. We will thereafter introduce scenarios and use cases that are commonly used for requirements engineering. In the latter part of this chapter, we will introduce an iterative software development process, which employs use cases at different levels of abstraction. Finally, we will state and justify the abstraction level of use cases in our approach.

2.1 What is Requirements Engineering?

The primary measure of success of a system is the degree to which it meets its intended purpose. Requirements Engineering is the process of discovering this purpose, by identifying stakeholders and their needs, documenting them in a form that is easy to analyze, communicate and subsequently to implement them. There are various issues related to the Requirements Engineering process. Stakeholders of a system may have different perspectives of the environment, which leads to conflicting goals for a future system. The goals of a system may even not be explicit or difficult to articulate [11].

Pamela Zave defines Requirements Engineering as: "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" [12]. This definition tends to give an idea of the role of Requirements Engineering. It highlights the importance of real world goals that motivate the development a software system. The goals

of a system can be ascertained by the 'why' and 'what' of the system [11]. Another interesting aspect that is highlighted by this definition is the evolving nature of requirements. Requirements may evolve during all phases of the software development process, and may not be limited to the requirements analysis phase. This is exemplified by the Rational Unified Process (RUP) [2]. RUP is an iterative development process. It categorizes the software development cycle into four phases: inception, elaboration, construction and transition. During each of these phases, requirements may evolve and needs to be taken into account.

Requirements elicitation is one of the core activities of Requirements Engineering. It is the systematic extraction and inventory of requirements of a system from a combination of stakeholders, system's environment, feasibility studies, market analysis, business plans, analysis of competing products and domain knowledge.

Requirements Engineering is a complex task. Isolation of the correct set of requirements for a future system is a tedious task. A computer system is not just limited to a physical system, but also includes the human protocol that is involved in order to completely define the problem domain. This means when writing requirements specification, we should not only consider the system, but also the interactions that exist between users and stakeholders of the system. There are many conflicting methods used for requirements engineering. Most of these conflicts arise because the methods used makes different assumptions about what is useful. Such conflicts can be resolved by asking of anything one proposes to do: "What is the use of doing that" [13]. Regardless of the method of requirements specification, requirements analysts should make sure that real users are involved during requirements capture. In any context, requirements specification phase should not be treated as the design phase of a system.

Requirements Engineering generally requires various skills: technical skills, ability to acquire an understanding of a system and interpersonal skills.

In order to build a high quality system, we should be able to capture good requirements. High quality requirements can only be defined by understanding the clear separation between

requirements specification and design. In addition, usage of precise specification notation plays an important role in this regard [13].

2.2 What are the problems of Requirements Engineering?

Requirements are easily understood when they are elaborated using natural language sentences. However the inherent ambiguities of the natural language result in severe problems during requirements elicitation. This problem becomes worst when the system to be modeled is of intricate nature. Many natural language words and phrases have dual meanings which can be altered by the context in which they are used. Weak sentence structure can also produce ambiguous statements [14]. The software industry has produced a significant number of aids or tools to correctly document quality requirements.

There are quality attributes that requirements specifications are expected to exhibit. These attributes are also described as the quality indicators of the requirements gathered. However in practice most requirements specifications do not meet these qualities. This is not a problem of applying the right methods and processes until we yield the appropriate level of requirements quality. A closer look would reveal that it is not a simple task. This is because the qualities themselves are part of the problem [15].

There are eleven quality indicators for requirements specification as described in [14]: Complete, Consistent, Correct, Modifiable, Ranked, Testable, Traceable, Unambiguous, Adequate, Validatable and Verifiable. These indicators are not independent of each other. It has been suggested in [6] that a requirements quality model should focus on adequacy as it is the most important quality.

A discussion on some of the important quality indicators is specified below [14, 15].

Adequate: Requirements specification is understandable if the meaning of its statements can be easily grasped by all its readers. Sometimes representation of requirements contradicts

with the way that the stakeholders view the system. Stakeholders are unable to assess the adequacy of requirements in such a situation.

Unambiguous: A requirements specification needs to be as formal as possible. But in most cases requirements are described using a natural language which is inherently ambiguous.

Consistent: A consistent specification is one where there are no conflicts between individual requirement statements. However, most of the times, requirements are given in chunks. Therefore the constituent requirements may have several consistency issues.

Correct: For a requirements specification to be correct, it must accurately and precisely identify the individual conditions and limitations of all situations that the desired capability will encounter and it must also define the capability's proper response to those situations.

Validatable: In order to validate requirements specification, stakeholders of the system must be able to substantiate that the requirements are true as stated. As such, requirements specification must be able to understand, analyze and accept.

Verifiable: In order to be verifiable, requirements at one level of abstraction must be consistent with those at another level of abstraction.

Modifiable: In order for requirements specifications to be modifiable, related concerns must be grouped together and unrelated concerns must be separated. This characteristic is exhibited by a logical structuring of the requirements document.

Ranked: Specification statements can be ranked according to stability and importance. This is established in requirements document's organization and structure. In addition, more complex the problem addressed by the specification, more difficult the task to design a document. Therefore, ranking specification according to stability and importance can conflict with structuring the document to be modifiable.

Testable: In order for a specification to be testable it must be stated in such a manner that pass/ fail.

Complete: A complete requirements specification must precisely define all real world situations that will be encountered. During the life cycle of a system it is not practical at some point to produce and freeze a complete requirements specification. The reason is that requirements and systems evolve with time. In addition, stakeholders do not always understand fully what they really need during the initial stage of the development cycle.

Traceable: Each statement of requirement must be uniquely identified to achieve traceability. This can be facilitated by using a consistent and logical scheme for assigning identification to each specification statement within the requirements document. The structure of a requirements document can be assessed by relatively simple algorithms. This can be achieved if each specification is uniquely identified and expressed as a simple statement.

2.3 Formal and Informal Languages

Requirements of a system can be elaborated using a formal specification language or a natural language (i.e. an informal language).

Formal languages are used to define requirements using a mathematical model. They provide the means to build system models by specifying their structure and behaviour. When formal methods are used in conjunction with tool support, automated verification of complex requirement models is possible [16].

A natural language is a convenient medium for writing requirements specification. It allows non-technical users and domain specialists to easily understand produced requirements. However due to the inherent ambiguity of the language there may be limitations in requirements understanding. Some of the common problems using a natural language includes under and over specification of requirements [16].

The language gap that exists between domain specialists and requirement analysts when formal languages are used for requirements specification is discussed in [17]. One of the main issues in Requirements Engineering is intra-language communication. The task of a domain specialists and a requirement analyst can be viewed as one of translation between two specialized worlds: application and computing domain. It is important that a requirements specification exhibits fluency of the application domain to some extent. This facilitates the domain specialist's task of requirements verification and validation. At the same time mere understanding of the syntax and semantics of a specialized language cannot bridge the communication gap. One reason for this limitation is described in [17] as: "of far greater significance are the unstated assumptions that reflect the shared ("common sense") knowledge familiar with social, business and technical contexts within which the proposed system will operate". Modeling the common sense knowledge in a language is very complicated.

Moreover, there are several requirements specification languages. Therefore, clients usually depend on other professionals to interpret their wishes and to translate as necessary into the specialized jargon. It is unrealistic to expect the clients to learn the "language of computing" [17].

Formal Languages are widely used for critical systems. But it may be impossible to represent certain requirements using formal languages, because they cannot be formalized easily. In such situations a natural language can be used in conjunction with a formal language to prevent refining out of such requirements [17]. For instance, descriptive textual and graphical material greatly assists human understanding when used with formal languages.

To minimize the limitations of both the formal and natural languages, a restricted form of natural language can be used. Attempto [18] is a system that translates restricted form of English sentences into discourse representation structures, and optionally into Prolog. The advantage of a restricted language is that it reduces the ambiguity of free form text to some extent, and can be accurately and efficiently processed by a computer. An approach that is

based on restricted form of natural language can reap the benefits of both the formal and natural languages.

2.4 Scenarios and Use Cases

In requirements engineering we frequently hear the terms: "Scenarios" and "Use Cases". These terminologies are related to each other to some extent. However, use cases and scenarios are different. We need to identify both terms clearly and be able to distinguish them. In a real world problem domain there are interactions between the system being modeled and its external actors. A scenario is a sequence of interactions [6]. Such a sequence has no branching or alternatives, but is subjected to specific conditions [6]. A use case is a collection of scenarios. All interactions in a use case relate to the same goal. Interactions start with a triggering event and end when the goal is delivered or abandoned, and the system completes its responsibilities with respect to the interaction [6].

A use case can also be viewed as a pattern or a class of interactions, and a scenario as an instance of a use case.

2.4.1 Scenarios

A scenario is a sequence of interactions that happens under certain conditions, to achieve a primary actor's goal. A scenario has a particular result with respect to the goal. The interactions starts from the triggering action and continues until the goal is delivered or abandoned, and the system completes whatever responsibilities it has with respect to the interaction [6].

Listed below are some advantages of scenarios in requirements engineering [15],

- Scenarios view a system from the viewpoint of external users. This style of writing scenarios gives the users a feel of what the ultimate system would provide. In addition, scenarios assist in validating systems for requirement adequacy.

- Scenarios can be used to write partial specifications of a system. That is they can be used to decompose a system function from a user's perspective.
- Scenarios are written in the form of user-system interactions and in most cases using natural language sentences. Thus, they provide a high degree of understandability of requirements.
- Scenarios can be written in a user oriented manner. Therefore, they may be used to represent user functions. This facilitates short feedback cycles between users and requirement engineers.
- Interaction sequences in scenarios can be used for deriving test cases. This feature can ultimately be used to verify and validate system requirements.

A good example of the last benefit is the SCENT approach [19] where scenarios are used to systematically derive test cases for system tests. This is done by formalizing natural language scenarios into statecharts, annotating statecharts with helpful information for test case generation and by using path traversal in the statecharts to determine concrete test cases. Since the testing cost of a software is significant, such an approach minimizes cost and reduces time to market while increasing the ultimate software quality.

There are some drawbacks with scenarios. Scenarios can be viewed as separate entities and this could lead into severe inconsistency problems. Another problem is that scenarios cannot describe requirements such as data persistence or state dependent requirements. These requirements can be expressed using other models such as object models or state automata. Scenarios must be used in conjunction with these models further yielding the possibility of inconsistencies.

2.4.1.1 Inter-Scenario Relationships

Scenarios are partial descriptions. Several scenarios are interrelated. Relationships between scenarios can be elaborated at a very detailed level. Four types of inter-scenario relationships are identified in [20],

- ***Containment Dependency:*** If S1 and S2 are two scenarios, a containment dependency exists, if S2 is used in the description of S1. Include relationships of use cases is an example for this kind of dependency.
- ***Alternative Dependency:*** If S1 and S2 are two scenarios, an alternative dependency exists, if S2 is an alternative to S1. An example for this category of dependency is main and alternative scenarios in use cases.
- ***Temporal Dependency:*** This dependency captures the different types of temporal relationships that may exist between scenarios.
- ***Functional (or Logical) Dependency:*** This dependency captures the coexistence of two or more scenarios inside the same logical structure. A use case that contains two or more scenarios is a good example in this regard.

2.4.2 Use Cases

A use case is a collection of possible scenarios between a system under construction and external actors, characterized by a goal a primary actor has towards the system's declared responsibilities, showing how the primary actor's goal might be delivered or might fail [6].

Use Case scenarios can be categorized as follow [6],

- ***Main Scenario:*** This is the simplest scenario. It is achieved when everything goes right and the use case goal is delivered without difficulty.
- ***Alternate Scenario:*** These scenarios can be further classified into recovery and failure scenarios.
    - ***Recovery Scenario:*** A Scenario that goes into a failure situation, but ultimately recovers and succeeds by delivering the use case goal.
    - ***Failure Scenario:*** A Scenario that ultimately ends in a failure.

Use Cases are a powerful technique for functional requirements capture. They allow structuring of requirements documents with user goals. In addition, they provide a means to specify the interactions between a software system and its environment.

Use Cases describe the outward visible requirements of a system. Not only can they be used for requirements analysis, they can also be used to create user guides, test plans, etc. In addition, use cases contribute to the architecture and scheduling of software projects [4].

Most object-oriented methodologies assume the prior existence of a requirements specification. Elicitation of requirements is someone else's problem. This separation of requirements generation from requirements analysis into disjoint customer and developer activities is not only ineffective but also inconsistent [21]. These issues can be eliminated by adopting a use case driven method.

Use cases are integrated in the Unified Modeling Language (UML) [22]. UML is a popular graphical modeling language. However, the UML merely serve as a "table of contents" for use cases [23] showing only the relationships between actors and the use cases.

The UML does not specify how use cases should be written. Various approaches have been proposed for writing use cases. A detail review of some of these approaches can be found in section 3.1.

2.4.2.1 UML Use Case Relationships

There are three kinds of use case relationships according to the UML. They are described as follows in [3].

- ***Generalization:*** A child use case can inherit the behaviour and meaning of the parent use case. The child may add to or override the behaviour of its parent.
- ***Include:*** An include relationship between use cases means that a base use case explicitly incorporates the behaviour of another use case at a location specified in the base use case. Include relationships can be used to avoid describing the same flow of events several times by putting the common behaviour in a separate use case.
- ***Extend:*** An extend relationship between use cases means that a base use case implicitly incorporates the behaviour of another use case at a location (i.e. extension point)

specified indirectly by the extending use case. Extend relationships are used to model optional parts of use cases.

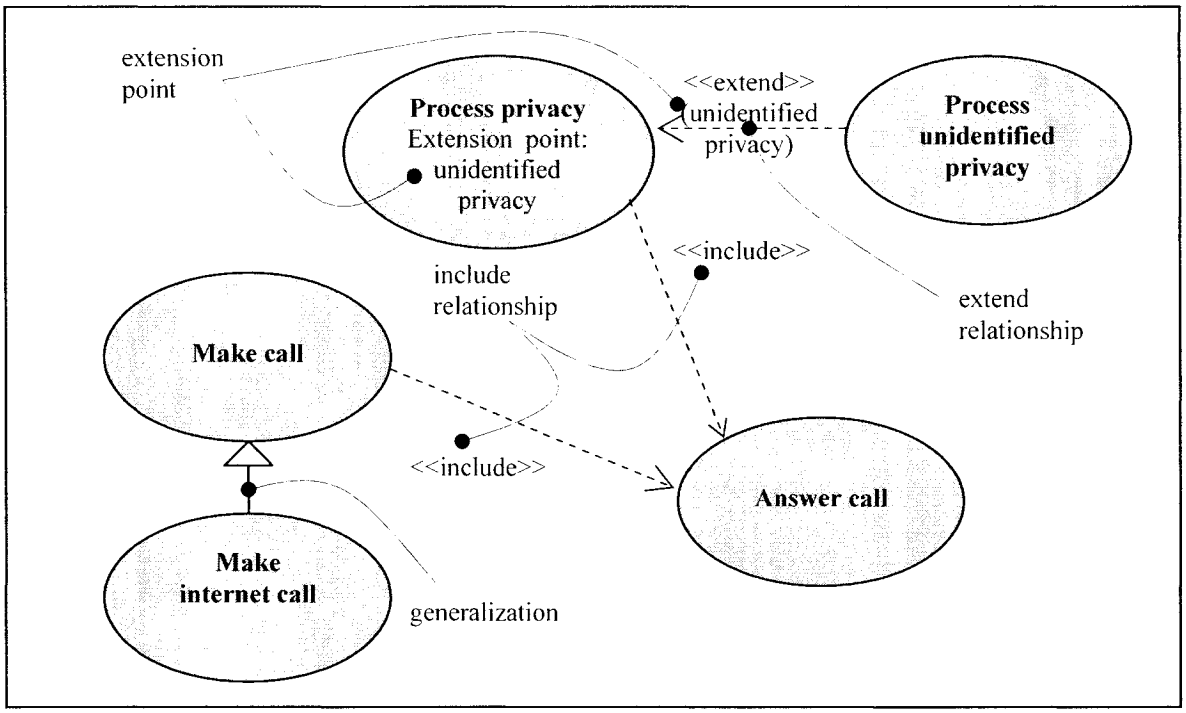


Figure 2-1: Generalization, Include and Extend Use Case Relationships

Figure 2-1 describes use case relationships for a Telephone PABX system. As represented, *"Make internet call"* use case is a specialization of *"Make call"* use case. *"Answer Call"* use case is included in base use cases: *"Process privacy"* and *"Make call"*. In this case the base use cases pull the behaviour from the included use case. Since common behaviour is allotted to include use cases, use case behaviour redundancy can be avoided. Lastly, *"Process unidentified privacy"* is an extension use case. It is inserted in the base use case *"Process privacy"* at the extension point *"unidentified privacy"* when the extension condition *"number type is unidentified"* is satisfied. An extension use case pushes behaviour to the base use case.

2.5 Abstraction levels in use case descriptions

Use cases can be written at different abstraction levels. During requirements gathering, the flow of events depicting interactions between the system and the environment are captured

using use case sentences. As the system requirements are refined, we may need to detail the requirements that were initially captured.

Following are possible levels of abstraction for a use case description.

Abstract use cases: Initially, use cases need to be identified from a set of informal requirements. Names of use cases are derived from verb phrases that describe an actor's functionality. [9]. These use cases are abstract and do not contain details. UML use cases falls into the category of abstract use cases.

Detailed black box use cases: These use cases are described using structured text in detail. However, all use case details are described at a black box level, without internal details of the system. Essential use cases [24] are a lightweight, technology free method of writing black box use cases.

Detailed grey box use cases (scenarios): This type of use cases are described with internal system details. Use cases at the grey box level may be represented using message sequence charts (MSC) [25], use case maps (UCM) [26], sequence diagrams [3], etc.

2.6 The role of use cases in the software development process

Use Cases are used in many software development processes such as the RUP. It is a process that is iterative and risk driven. As we mentioned before, RUP distinguishes four primary phases: *inception*, *elaboration*, *construction* and *transition*.

The general tasks in each of these phases are described as follows [4],

Inception phase: The scope of the project is determined and business cases are created. At the end of this phase it should be possible to determine whether it makes good business sense to proceed with the project.

Elaboration phase: Requirements analysis and risk analysis is done. In addition, a baseline architecture and a plan for the construction phase are created.

Construction phase: Progresses through a series of iterations. Each of the iterations includes analysis, design, implementation and testing.

Transition phase: Complete things of the developed product. This may include beta testing, performance tuning, creation of documentation such as training, user guides, sales kits, etc.

So how and where do use cases fit into all these? Figure 2-2 is extracted from the ideas taken from [4]. It attempts to describe which use case level of abstraction is used in the different phases of RUP.

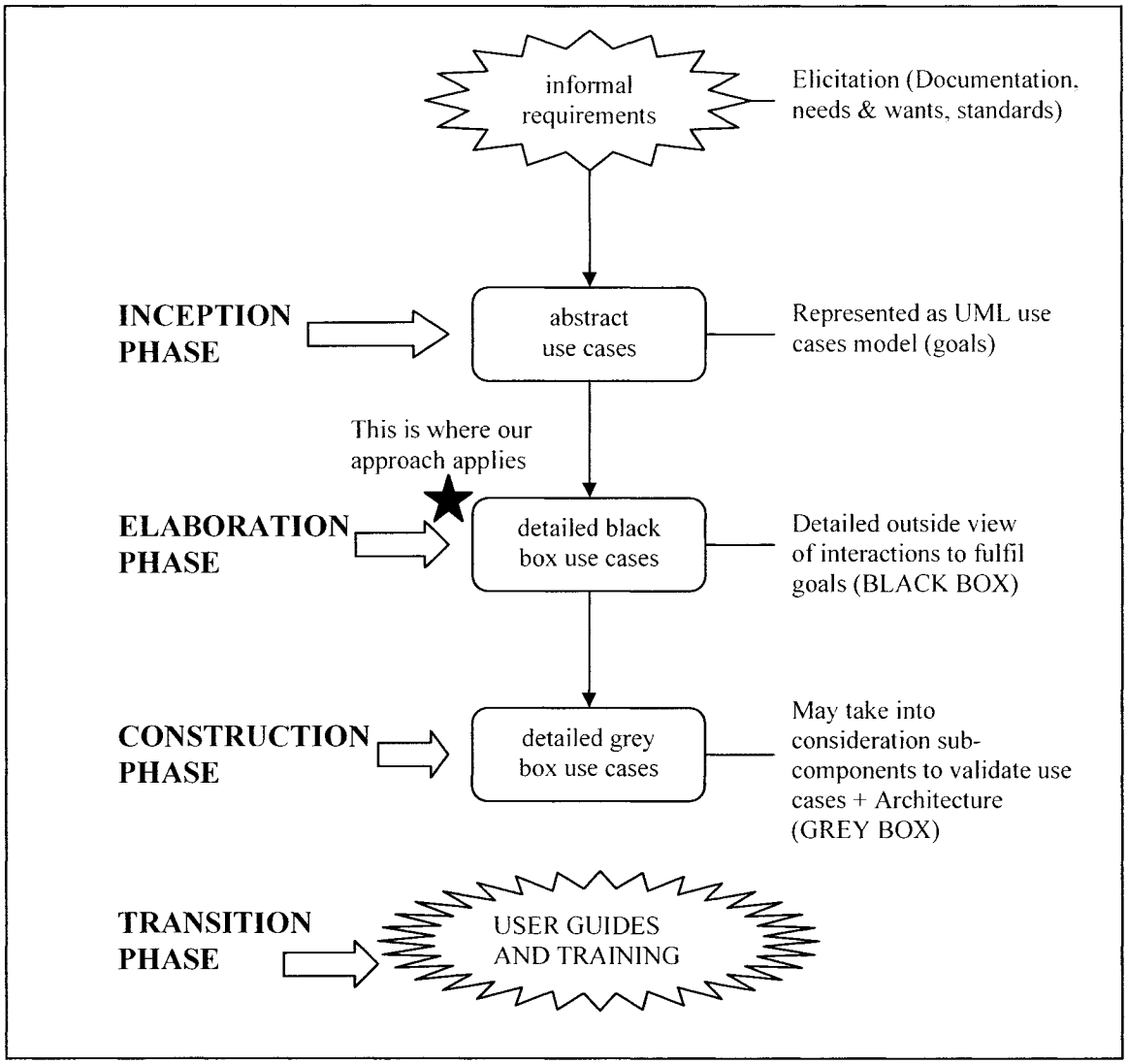


Figure 2-2: Iterative Software Development Process

A software project starts with a set of informal requirements. These informal requirements can be in the form of needs and wants of stakeholders, project documentation, standards, etc. Subsequently, these informal requirements are refined into abstract use cases.

It is only at the inception phase that high-level, abstract use cases are developed to assist scoping of the project. These abstract use cases can be goal-oriented and developed using UML. As we mentioned before a UML use case model is merely a table of contents. It is in this phase that the following tasks are determined: what to include in the project, what belongs to another project, what can be realistically accomplished given the project schedule and budget.

Abstract use cases defined in the inception phase are further refined into detailed use cases in the elaboration phase. They contribute to risk analysis and baseline architecture. They are also used to plan for the construction phase. Detailed use cases specify the detailed outside view of interactions to fulfil user goals. It is the *"black box"* view of the system.

In the construction phase, use cases are the starting point for design and development of test plans. More detailed use cases are developed as part of the analysis in each iteration. In this phase, more focus is made to the software architecture. Scenarios involving sub-components are essential for use case validation [19]. Detailed use cases constructed during this phase represents the *"grey box"* view of the system.

In the transition phase detailed use cases from the construction phase are used to develop user guides and training. Use cases from the elaboration and construction phases are used to derive test cases for verification and validation.

2.7 Where does our approach lies?

In the preceding sections we discussed how use cases can be refined at each phase of an iterative software development process. We noticed that use cases can be described at different levels of granularity according to the development phase involved.

In the Requirements Engineering phase, use cases are used to capture an external (black box) view of the system. When we model an external view of a system we should be careful in understanding the context of the system that we are dealing with. That includes drawing a line at the system boundary, so that actor operations and system reactions are clearly defined. Requirements Engineering is not concerned in how the system works in the inside. A black box view of the system is important to provide a way for requirement analysts to come to a common understanding with domain experts before committing to a particular architecture. In such a context, domain experts are in a position to easily validate the requirements modeled by the system.

Therefore, our work is based on the black box view given by use cases. In relation to the RUP, the approach applies to the elaboration phase of the software development process, as specified in Figure 2-2.

2.8 Chapter Summary

This chapter first introduced Requirements Engineering; subsequently it described various problems of the Requirements Engineering process. We also focussed our attention on formal and informal languages that can be used for requirements elicitation. Afterward we introduced use cases and scenarios. The most important part of this chapter is the introduction of an iterative software development process: RUP. This process applies use cases at different levels of abstraction according to the development phase. Requirements Engineering that is concerned with defining "what" the system is about, is carried out using abstract and black box detailed use cases. Our work focuses on black box detailed use cases. Use cases at this level can be formalized and still be understood by domain specialists. The communication gap between the domain specialists and requirement analysts can therefore be minimized.

Chapter 3 - Description of Use cases

Use cases are a popular mechanism used for requirements elicitation. They are part of the UML specification. Use cases are also integrated in software development approaches such as the Rational Unified Process (RUP). There is no standard guideline for writing use cases. This has caused introduction of several approaches for description of use cases. We will start this chapter by a review of different use case formalisms. Then we will focus on different guidelines and restricted languages for writing use cases. Thereafter we will present a critique of the discussed restricted languages. Subsequently, we will describe the importance of having instances in use cases. Finally, we elaborate on why domain knowledge is needed for use case analysis.

3.1 Review of use cases description formats

Ivar Jacobson, who is considered the inventor of use cases have contributed a lot of ideas to use cases, which have been integrated into the UML specification. Jacobson has stated that use cases should be formalized as far as possible. He provides descriptions for several use case formalisms [27]. First is the "basic use case" which is described using structured text that includes alternative and exceptional behaviour. Second is a "class association" that includes use case <<uses>> and <<extends>> stereotypes and actor inherent hierarchies. Third is an "interaction diagram" which shows different paths that a conversation will follow. These include iteration, repetition, branching and parallelism. Fourth type is a "contract" which specifies object's interface in detail. A fifth formalism is a state transition diagram which describes how use case operations can form state transitions. In addition, several other authors have proposed more ideas in formalising use cases. As discussed in Chapter 1, use cases can be refined and formalized at different levels of abstraction.

In this section, we will describe some of the techniques that have been introduced for writing use cases.

3.1.1 Wirfs-Brock use cases formats

Wirfs-Brock introduces three formats for describing use cases in [28].

- ***Narrative Form***: This form is merely writing text in a paragraph format. It describes the intent of the user in performing a use case, and the high level actions of the user in the course of the use case. Figure 3-1 presents a use case for making a payment [28] in a narrative form.

> The user can make online payments to vendors and companies known to the bank. Users can apply payments to specific vendor accounts they have. There are two typical ways to make payments: the user can specify a one-time payment for a specific amount, or establish regular payments to made on a specific interval such as monthly, bi-weekly, or semi-annually.

Figure 3-1: Wirfs-Brock's narrative form of use case: "Make Payment"

- ***Scenario Form:*** This form describes a particular path through a use case written from the actor's point of view. It is represented as a sequence of events or list of steps to accomplish, and each step is a simple declarative statement with no branching. A step may describe actors and their intentions, or system responsibilities and actions. A scenario of a use case for making a payment [28] is presented in Figure 3-2.

Scenario: Make Payment
1. user requests a payment template 2. system displays the requested payment template 3. system displays the recent payment history to payee 4. user enters the payee notes, amount and account ***Required information:*** payee notes amount and account ***Optional:*** comments 5. user submits the payment information 6. system applies the payment to payee 7. system adds a new payment to recent payment list 8. system redisplays the payment list 9. if user requests to setup payment then Goto Edit Payment Template information

Figure 3-2: Wirfs-Brock's scenario form of use case: "Make Payment"

- ***Conversational Form:*** An example for a use case in conversational form [28] is shown in Figure 3-3. It describes a use case for making a payment.

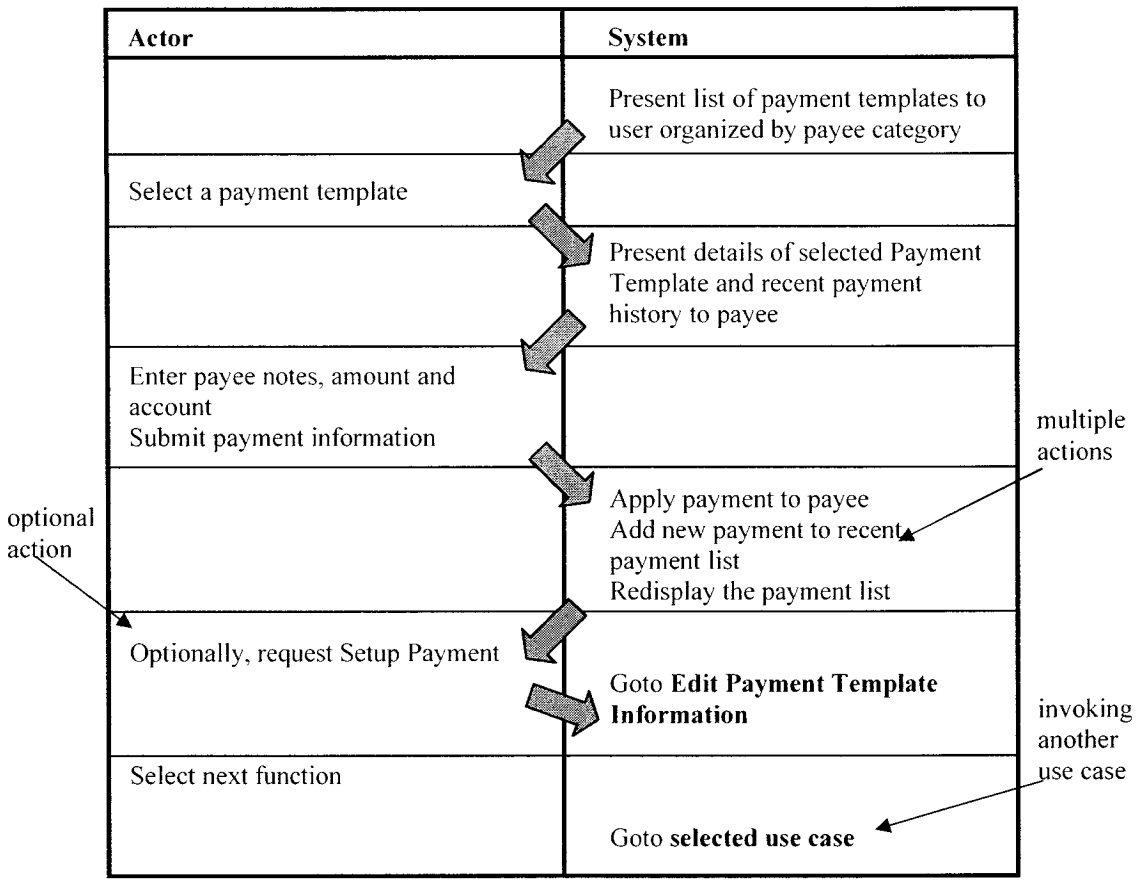


Figure 3-3: Wirfs-Brock's conversational form of use case: "Make Payment".

A conversational form can represent one path through a use case that emphasizes interactions between an actor and a system. It can show optional and repeated actions. Also each action can be represented by one or more sub-steps. A conversational form can represent actor actions, and system responsibilities and actions.

3.1.2 Essential use cases

Essential use cases are part of Usage-Centred Design, as developed by Larry Constantine and Lucy Lockwood [29]. They state that conventional use cases contain too many hidden assumptions about the form of the user interface that is yet to be designed. These assumptions are problematic since they force the UI design details to be embedded in requirements, making them difficult to be modified later [30]. Essential use cases are

designed to overcome this problem. They are abstract, technology free and implementation independent descriptions of the requirements. Essential use cases can be a viewed as detailed black box view of use cases.

Essential use cases are written in a format representing a dialogue between users and the system [30]. The format closely resembles the one proposed by Wirfs-Brock in writing use cases in a conversational form (refer to section 3.1.1).

3.1.3 Cockburn use cases

According to Cockburn, use cases should be written in a textual, goal-oriented form. He recommends a set of guidelines to be used in writing use cases. In addition, he emphasizes the usage of a restricted grammar that consists of simple grammatical rules [5]. Use cases written by different teams of a project appear to be consistent when there are written using a standard restricted grammar. Reading and understanding well written, standard use cases is easy. But learning to write a use case is not that easy. The writer has to master three concepts: what is the scope of the system under discussion?, who has the goal (i.e. the primary actor)?, and the level of the goal. The scope of the system could be functional, design, etc. The level of a user goal may exist at a summary, user or sub-function level.

Cockburn also defines a use case template which could be used in writing use cases [5]. The template that he recommends, which is presented in Figure 3-7 does not have to be followed strictly. An alternate or modified template can be used according to the context of use.

3.1.4 Use Case Maps (UCMs)

A Use Case Map (UCM) can be used to describe *causal relationships* between *responsibilities*, which are bound to underlying *organizational structures of abstract components*. These can be used to represent scenarios that intend to bridge the gap between use cases (i.e. requirements) and detailed design [26]. UCMs are usually described using a visual notation.

Scenarios that are contained in a use case can be described using a path that causally links responsibilities, which may be bound to components. A Responsibility is a system behaviour such as an action, operation or a task. Responsibilities can be sequential, in alternative or in parallel. Components are entities composing the system including software entities. When paths become too complex to fit in to one UCM, they can be refined using stubs [26]. A stub may contain separate sub-maps (i.e. plug-ins). Therefore, a UCM can be hierarchical.

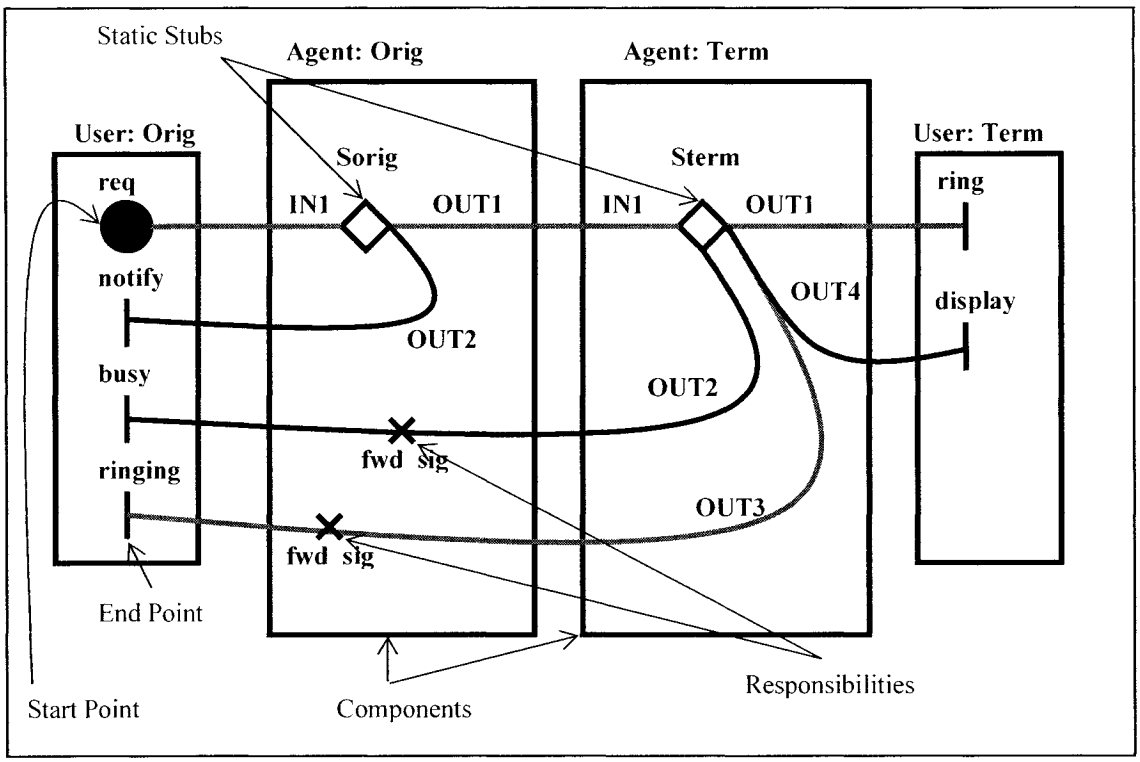


Figure 3-4: An example UCM for a telephony system

A simple example for a UCM, which is extracted from [31] is presented in Figure 3-4. It is a UCM that describes the connection phase of a simple agent-based telephony system. In this example originating and terminating users both have agents that handle their part of the call. It shows graphically the paths that run through responsibilities. This UCM is a top-level UCM (i.e. a root map). The stubs in the root map of the UCM provide sub-map level requirements, so that the system can be modeled at a lower level of detail.

3.1.5 Message Sequence Charts (MSCs)

Andersson and Bergstrand have presented a method using MSCs to formalize use cases at three levels: system level, structure level and basic level [27]. The interactions between a system and its environment are called messages. The system can be either seen as a "black box" or can be described in more detail. Sequence diagrams are similar to MSCs in the UML [22].

Primary uses of MSCs includes: Requirements definition of a system, viewing the execution trace that can be used to match requirements and present a stepwise execution "event-by-event" to monitor a system behaviour [25].

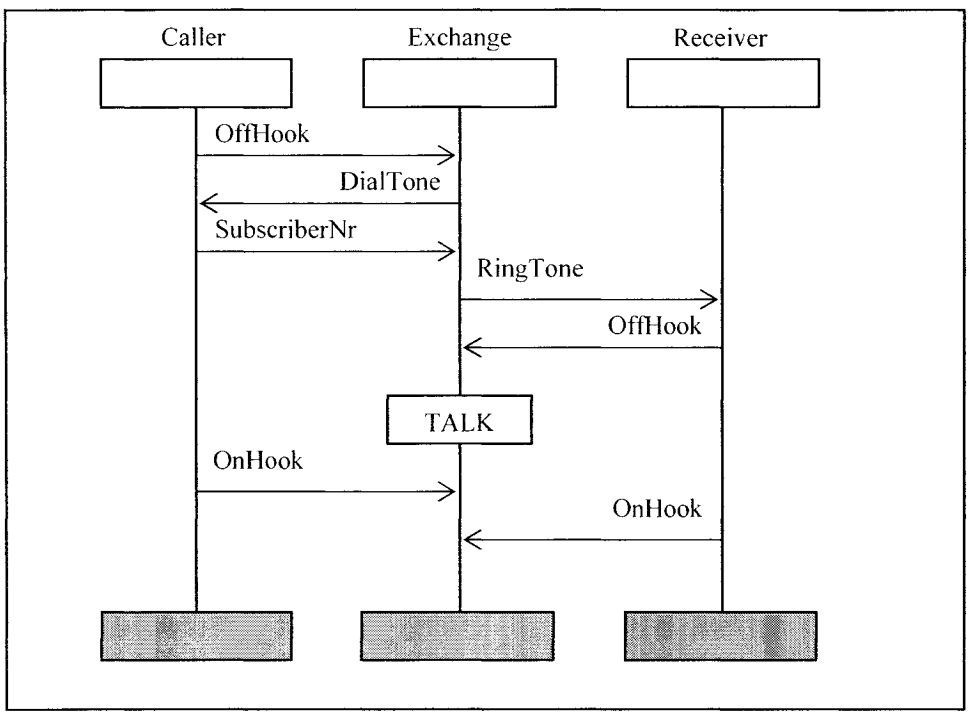


Figure 3-5: An example MSC for a telephony system

A simple example MSC for a telephony system, which is extracted from [25] is presented in Figure 3-5. Interactions in a MSC can be described by a stimuli and response messages that are exchanged between environment and system. Interactions could also occur between internal components of a system [25]. These messages are represented as arrows in the MSC.

The vertical lines are time axes and the boxes on top represent components in the system or the environment.

3.2 Review of guidelines for use case writing

We stated in Chapter 1 that requirements expressed in the form of natural language sentences are easier to understand by stakeholders. However, free form natural language suffers from an inherent ambiguity. A compromise between formality and understandability is to use a restricted form of natural language. Several guidelines have been introduced for writing use cases based on restricted languages. These guidelines specify how the language should be controlled in writing use cases. There is a trade-off between the degree of restrictions and expressive power. Too many prescriptive guidelines restrict what can be expressed in use cases, making the language inadequate for requirements acquisition and validation. On the other hand, too few guidelines give too much freedom of expression to the author, decreasing use cases quality, and making automated processing difficult [10]. In this section we review some of the guidelines that are used for writing use cases.

3.2.1 Cooperative Requirements Engineering With Scenarios (CREWS) guidelines for use cases

CREWS guidelines in writing use cases are summarized in [10]. CREWS general guidelines in use case authoring divide a use case into three sections: normal course of events scenario, variations to the normal course scenario and alternative courses to the scenario. CREWS provides guidelines, which describes how to write a use case scenario and what to put in it. In addition, CREWS introduces style guidelines as part of defining best practices for writing use cases.

- ***SG1 :*** write the UC normal course as a list of discrete actions in the form: <action #> <action description>. Each action description should start on a new line. Since each action is atomic, avoid sentences with more than two clauses.
- ***SG2 :*** use the sequential ordering of action descriptions (and hence their unique number identifiers) to indicate strict sequence between actions. CREWS imposes a precise meaning on the ordering of actions in this list. Variations should be written in a separate section.
- ***SG3 :*** iterations and concurrent actions can be expressed in the same section of the UC, whereas alternative actions should be written in a different section.
- ***SG4 :*** use consistent agent, object and action names in all action descriptions in a UC. Avoid use of synonyms and homonyms, and anaphoric references such as he, she, them and it. Be consistent in your use of terminology.
- ***SG5 :*** use present tense and active voice when describing actions.
- ***SG6 :*** avoid use of negations, adverbs, and modal verbs in the description of an action.

Figure 3-6: CREWS style guidelines for writing use cases

CREWS style guidelines are presented in Figure 3-6. They highlight a restricted language for writing use cases. They also provide recommendations such as avoidance of synonyms, homonyms and anaphoric references to improve consistency.

3.2.2 Cockburn use case guidelines

Cockburn defines a series of guidelines in defining use case steps. We state below some of these guidelines that are extracted from [5].

- Use simple grammar: The sentence structure should follow a simple grammatical form (e.g. subject....verb...direct object....prepositional phrase).
- Clearly specify who initiates the action.
- Write use case steps from a "bird's eye view".
- Show that the process is moving forward. When use cases are written at a very low-level of detail, it appears that the process which is described by the use case steps moves very slowly. In such a situation, a use case may contain a larger number of use case steps. This is because when use case steps are written in a very low level of detail, more steps may

need to be written in order to fully explain the use case. Cockburn suggests that we should not have more than nine steps in a well-written use case.

- Show actors intentions, not their movements: User's movements in operating a user interface should be avoided.
- Include a reasonable set of action: Thought should be given to the appropriate level of use case step abstraction.

Cockburn recommends a template for writing textual use cases [5], which is presented in Figure 3-7. It is a one-column, numbered-step version. The discussion that follows explains each part of this template.

<Name of use case: should be the goal as a short active verb phrase>
Context of use: <a longer sentence of the goal and normal occurrence conditions (optional)>
Scope: <design scope, what system is being considered black box under design>
Level: <one of: Summary, User, Sub-function goal>
Primary Actor: <a role name for the primary actor, or description>
Stakeholders & Interests: <list of stakeholders and key interests in the use case>
Precondition: <what we expect is already the state of the world>
Minimal Guarantees: <how the interests are protected under all exits>
Success Guarantees: <the state of the world if goal succeeds>
Trigger: <what starts the use case, may be even a time event>
Main success Scenario: <put here the steps of the scenario from trigger to goal delivery, and any cleanup after>
<step #><action description>
Extensions: <put here the extensions, one at a time, each referring to the step of the main scenario>
<step altered> <condition> : <action or subordinate use case>
Technology and Data Variations List: <put here the variations that will cause eventual bifurcation in the scenario>
<step or variation #><list of variations>
Related Information: <whatever your project needs for additional information>

Figure 3-7: Cockburn suggested format for use case writing

The template starts with a description of the context of the use case, the goal of the use case and the condition under which it can be executed.

The scope of a use case can be defined in various ways. One perspective is where the scope can be further sub-categorized into system and strategic scope [5]. The system scope is everything within the boundary of a system and is associated with a primary actor having a goal. The strategic scope includes use cases and their goals. Strategic scope covers a sphere that contains people and computer systems, which helps stakeholders to monitor the progress of their goals. The scope can also be sub-categorized as functional and design scope [5]. The functional scope describes the services that the system offers that will be captured by use cases. On the other hand the design scope specifies the extent of the system, which may be expressed in terms of hardware and software [5].

Goal of a use case can be sub-categorized into 3 levels: summary level, user level and sub-function level [5]. User level goals are performed at a level where the primary actor is trying to get the work done. A collection of user goals forms summary goal. User goals can be further sub-divided into sub-goals. These sub-goals are also known as sub-functions. Figure 3-8, which is extracted from [5], describes the dimensions of goal levels. It depicts the structure of a sailing ship. As we go from the sky level towards the seabed, use cases gets more detailed and specific.

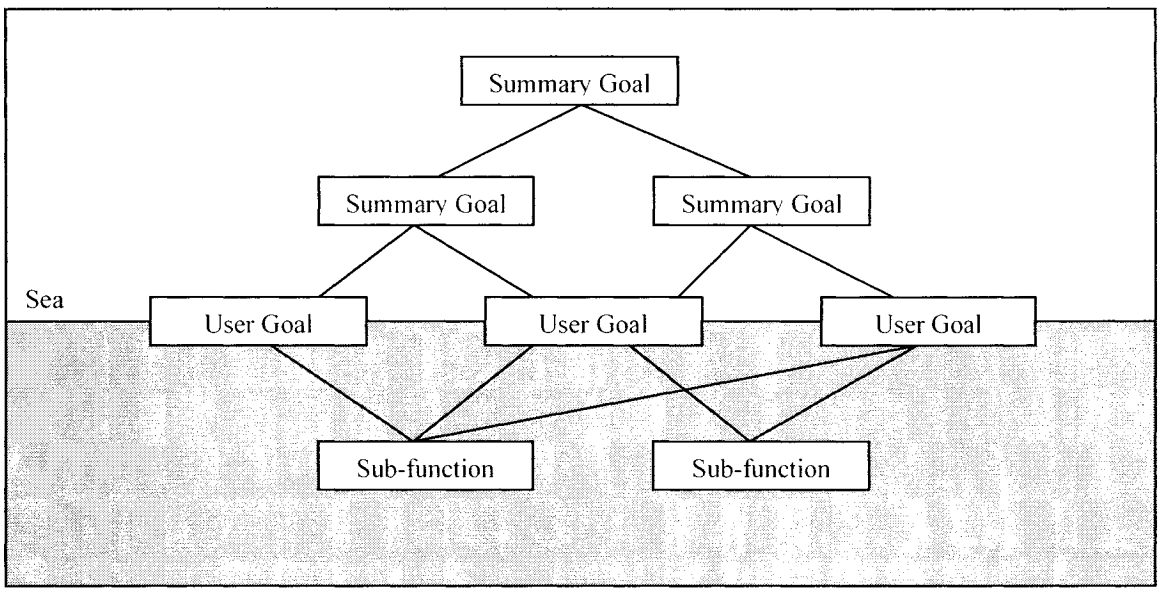


Figure 3-8: Goals in use cases

A stakeholder is someone who has some vested interest in the behaviour of the system. A primary actor is a stakeholder who initiates an interaction with the system to achieve a goal. A precondition of a use case specifies what the system will ensure is true before letting the use case start. Minimal guarantees are the fewest promises that the system makes to the stakeholders, particularly when the primary actor's goal cannot be delivered. On the other hand a success guarantees states what interests of the stakeholders are satisfied after a successful conclusion of the use case. A trigger specifies the event that gets the use case started. A main success scenario is a case in which nothing goes wrong during it execution. On the other hand extensions describe what can happen differently during the execution of the main success scenario.

Technology variations or differences in the data that could be captured from use case steps and extensions are written into the "Technology and Data Variation List" section of the Cockburn use case template. Use case steps and extensions that contain requirements text can be expressed in different ways: "What is happening is the same, but how it is done might vary" [5]. This is because there are some technology variations or some differences in the data that must be captured. Let's consider the use case step: *"caller pays for the call"*. In this case, the *caller* can pay by check, credit card or using an electronic fund transfer. Therefore, the same use case step can be expressed in different ways. These variations of data are written into the "Technology and Data Variation List" section of the use case template.

There are other use case guidelines and templates similar to Cockburn's. Examples are Process Impact [32] which introduces a template that follows a one-column table form for writing use cases and the RUP uses a template. Templates for which are presented in Appendix A.

3.3 Review of Restricted Natural Languages for use cases

The difficulty in processing use case descriptions is attributed to the vagueness of natural language and its free form style of writing use case descriptions. A free form text is hard to process mechanically by a machine. Natural Languages are far from the rigorous syntactic

requirements of computer languages. To assist a computer program in extracting information from use case descriptions, we have to constrain the free form text into a more restricted form. A restricted form of natural language uses a set of syntactic rules.

Various syntactic guidelines have been suggested by different authors for writing use cases in restricted form of language. In this section we will discuss some of these guidelines.

3.3.1 CREWS Content Guidelines

CREWS recommends specific content guidelines for writing restricted form of natural language sentences. A complete description of these guidelines is presented in Figure 3-9.

- ***CG1 :*** <agent> <'move' action><object> from <source> to <destination>.
- ***CG2 :*** <source agent> <'put' action> <object> to <destination agent>.
- ***CG3 :*** <destination agent> <'takes' action> <object> from <source agent>.
- ***CG4 :*** <agent> <action> <agent>.
- ***CG5 :*** <agent> <action> <object>.
- ***CG6 :*** 'If' <alternative assumption> 'then' <action>.
- ***CG7 :*** 'Loop' <repetition condition> 'do' <action>.
- ***CG8 :*** <action 1> 'meanwhile' <action 2>.

Figure 3-9: CREWS content guidelines

An interesting feature in this grammar is that the action verb type of a use case step, determines the location of agents and object in a use case step description. For example, if the action verb type is "put", the source agent appears before the action verb, and the destination agent and object appears after the action verb. Similarly, if the action verb type is "takes", the destination agent appears before the action verb, and the source agent and object appears after the action verb. In addition, as described in CREWS content guideline: CG8, CREWS allows writing use case steps with simultaneous actions.

3.3.2 Use-Case Driven Development Assistant (UCDA) grammar

UCDA is a CASE tool used for natural language requirements analysis and class model generation based on the RUP [9]. Using this tool stakeholder's requests documented in natural language are analyzed and processed by a parser. The grammar that supports the structural elements of sentences as recognized by the UCDA parser is represented in Table 3-1. The typical structural elements of the sentences that are recognized by the parser are the following [9].

N – Noun
V - Verb
ART – Article
ADJ – Adjective
Q – Quantifier
P – Preposition
DET – Determiner
ADV – Adverb
CONJ – Conjunction
auxV – Auxiliary Verb

NP is noun phrase and **VP** is verb phrase.

Complex Structure	**Simplified Structure**
NP1 + Verb1 + NP2 + Verb2 + NP3	(i) NP1 + Verb1 + NP2
	(ii) NP2 + Verb2 + NP3
{Q, ADJ, V-ing, DET} + N	N
auxV + V	V
Sentence1 { AND/OR} Sentence 2	(i) Sentence1
	(ii) Sentence2
NP + VP + NP1 and NP2	(i) NP + VP + NP1
	(ii) NP + VP + NP2

Table 3-1: Rules for sentence construction

UCDA can identify complex sentences and break them down to simple sentences, using the rules for sentence reconstruction, which is presented in Table 3-1. The parser converts the input requests into sentences, and each sentence into an abstract syntactic structure. For example, use case step: *"caller calls the recipient"* can be tagged *as ['caller'/'N', 'calls'/'V', 'the'/'ART', 'recipient'/'N'].*

3.3.3 Liwu Li's grammar for use cases description

A grammar for writing and normalizing use cases, which is introduced by Liwu Li is presented below [8]. It is a set of syntactic rules, which implies a common structure of sentences in use case descriptions.

- Use simple sentences, each of which may permit only one subject and one predicate (i.e. action). Sentences that contain multiple actions should be normalized into simple sentences. A computer program can easily understand simple sentences.
- Use active rather than passive voice.
- Use the same verb for the same action in different sentences
- Use the name of the specific entity to replace a pronoun of that entity (i.e. avoid using pronouns in use case descriptions).
- Use If-clause for sentences having conditions.
- Use *Do...until* and *while* constructs to mark that some steps can be repeated.

A use case step can have a complex structure with multiple verbs. Identifying the appropriate action verb, to isolate the correct use case operation from such a use case step would be a challenge. Li's grammar facilitates identifying the correct use case operation from a complex use case step. The use case sentence grammar proposed by Li is described in Table 3-2.

Syntactic Structure	Sender	Receiver	Operation
subject verb		subject	verb
subject verb object	subject	object	verb
subject verb past_participle		subject	verb of the past_participle
subject verb predicative	subject	class of object	verb + predicative
subject create_verb object	subject	class of object	create_verb
subject destroy_verb object	subject	object	destroy_verb
subject verb object (to) verb1	subject	object	verb1
subject verb object present_participle	subject	object	verb of the present_participle
subject verb object adjective	subject	object	set + adjective
subject verb object preposition object1	subject	object	verb
subject verb object to verb1 object1	subject	object	verb1

Table 3-2: Li's syntactic structures of use case sentences

3.3.4 Use Case Editor (UCEd) grammar

UCEd is a tool that is used for elicitation, composition and simulation of use cases based requirements. It uses a restricted form of natural language for writing use case descriptions. UCEd provides a concrete syntax [7, 33, 34] for writing use case conditions and operations. Conditions are predicates specified on domain entities. A domain entity in UCEd can be a concept, system concept, instance, aggregate or an attribute. Having different sub-entity types for domain entities makes parsing complicated. On the other hand, it improves the expressive power of use case descriptions. It also assists in defining a detailed domain model with concepts, aggregates, attributes, etc. A use case operation is an instance of a concept operation, a branching statement or a use case inclusion directive. Figure 3-10 and Figure 3-11 presents UCEd grammars for conditions and operations respectively.

```
<condition> -> <acondition> "and" <condition>
               | <acondition> "or" <condition>
               | "("<condition>")"
               | <negation> <condition>
<acondition> -> [<article>] <entity> [<verb>] <value>
<article>    -> "a" | "an" | "the"
<negation>   -> "not" | "no"
<entity>     -> <concept> | <concept> <attribute> |
                <concept> <aggregate>
<concept>    -> <word>+ {member of the model concepts}
<aggregate>  -> <word>+ {member of the model concepts}
<attribute>  -> <word>+ {attribute of concept}
<verb>       -> "is" | "is" "not" | "isn't" | "are"
                | "aren't" | "become"  | "becomes"
<value>      -> (<word>)+ {value of the entity}
              | <comparison> {entity is non-discrete ?}
<comparison> -> <comparator> <word>
<comparator> -> ">" | "<" | "=" | "<=" | ">=" | "<>" | "greater
             than" | "less than" | "equal to" | "different to" |
             "greater or equal to" | "less or equal to"
```

Figure 3-10: UCEd grammar for conditions.

```
<operation_spec> -> <concept_operation> | <branching_statement>|
                    <useCase_inclusion>
<concept_operation> -> [<before_delay>] [ after_delay]
                    [<condition_spec] <operation_reference>
<condition_spec> -> "IF" <condition> "THEN"
<operation_reference> -> (<word>)+ {derived from a operation of
the current entity}
<after_delay> -> "AFTER" <duration_spec>
<before_delay> -> "BEFORE" <duration_spec>
<duration_spec> -> <duration_value> <duration_unit>
<branching_statement> -> "GO" "TO" "Step" <word>
                    {corresponding to a step label}
<useCase_inclusion> -> [<condition_spec>] "INCLUDE"
                    <use-case-name>
```

Figure 3-11: UCEd grammar for use case operations

3.4 Critique on restricted languages

As explained in the preceding section, restricted languages enhance the consistency of requirements text. We discussed four such restricted languages that have been introduced by different authors. In this section we will make a comparison of each of the discussed restricted languages.

CREWS guidelines show clearly the relationship between agents and objects for different types of action verbs. They also facilitate writing use case steps with simultaneous actions. UCDA grammar is based on the grammar of English Language. The UCDA grammatical elements include nouns, verbs, etc. Nouns can be used to isolate use case actors and verbs can be used to extract use case operations. Therefore UCDA grammar has a high degree of flexibility, as it is capable of extracting domain elements from free form language elements. UCDA grammar also has the capability of breaking complex sentences into simple sentences for processing. Li's grammar provides emphasis on isolating the correct use case operation from a complex use case sentence, which contains multiple verbs. UCEd grammar provides support for having different types of sub-entities for writing use case descriptions.

The importance of instances for writing use cases is discussed in section 3.5. All grammars that we discussed do not provide ***explicit support*** for instances. In CREWS content guidelines, the subject agent that appears before the action verb of a use case description can be a proper noun. An example for CREWS content guideline *CG5: <agent> <action> <object>* is *"Ann calls John"*, where both *<agent>* and *<object>* syntactical categories belong to the proper noun category. Similarly, the same example follows the grammar rule: *NP1 + Verb1 + NP2* provided by UCDA grammar. This is because both noun phrases *NP1* and *NP2* can also be treated as proper nouns. The above example would also adhere to Li's grammar rule: *subject verb object*, where *subject* and *object* are of proper noun types. However, there is no explicit mechanism to link objects (i.e. instances) to their abstractions (i.e. entity classes). This is left as a manual interpretation performed by the requirements analyst. For instance, in sentence *"Ann calls John"*, manual interpretation need to be performed to recognize *"Ann"* and *"John"* as instances of a same entity type. Our objective is to be able to make such deduction explicit in the grammar.

In addition, all grammars except for UCEd do not provide support for sub-entity types (i.e. concepts, aggregates, attributes, etc.). The possibility to distinguish entity sub-types right from the grammar is a problem similar to that of being able to distinguish instances. UCEd supports sub-entity types by linking use cases to a domain model defined as a stereotyped UML class diagram.

One limitation of all the grammars that we discussed is none of them provide support for pronoun resolution. Pronoun resolution is an important task in discourse analysis, which is useful for Requirements Engineering.

3.5 Why use instances in use cases?

There is a clear separation between an abstraction and an instance. An abstraction describes an ideal essence of a thing [3]. In use case descriptions, abstractions are essentially domain entities. On the other hand an instance denotes a concrete manifestation [3]. An instance and an object are largely synonymous. However, instances do not stand alone. They are tied to an abstraction [3]. The essential fact here is that domain entities are not sufficient in documenting requirements in the form of use cases. That is domain entities themselves do not provide the required expressive power in writing use cases. In order to have a model with better expressive power, we need to have instances for writing use cases. As an example let us consider the use case sentence: *"caller calls recipient"*. This sentence is related to placing a call. The use case operation: *"calls"* is associated with domain entities: *"caller"* and *"recipient"*. These entities illustrate the use case operation at a higher level of abstraction. But in reality we need to have more expressive power than that for requirements elicitation. In order to enhance the expressive power of requirements elicitation, we may have to work at a lower level of abstraction by using instances. The example mentioned above can be elicited at lower level of abstraction as *"John calls Jane"*, where *"John"* and *"Jane"* are instances of *"caller"* and *"recipient"* respectively. Low level, instance-based abstractions is important to specify interactions between specific instances. It may be the case that only specific callers are allowed to call another party. In such a context we need to have instances for modeling requirements, with the required expressive power.

Instances are explicitly supported by other use case formalisms, such as MSCs and Sequence Diagrams. But no clear mechanism exists for providing support for instances in textual use cases. As previously discussed, textual use cases are important for early requirements capture, since they are easy for requirements understanding and validation by the stakeholders of a system.

The following sub-section describes how instances are represented in sequence diagrams.

3.5.1 Support for instances using sequence diagrams

In this sub-section, we describe how sequence diagrams can be used to write use case scenarios that involves instances. Sequence Diagrams are used to model the abstract interaction between classes, as well as the concrete interaction between instances.

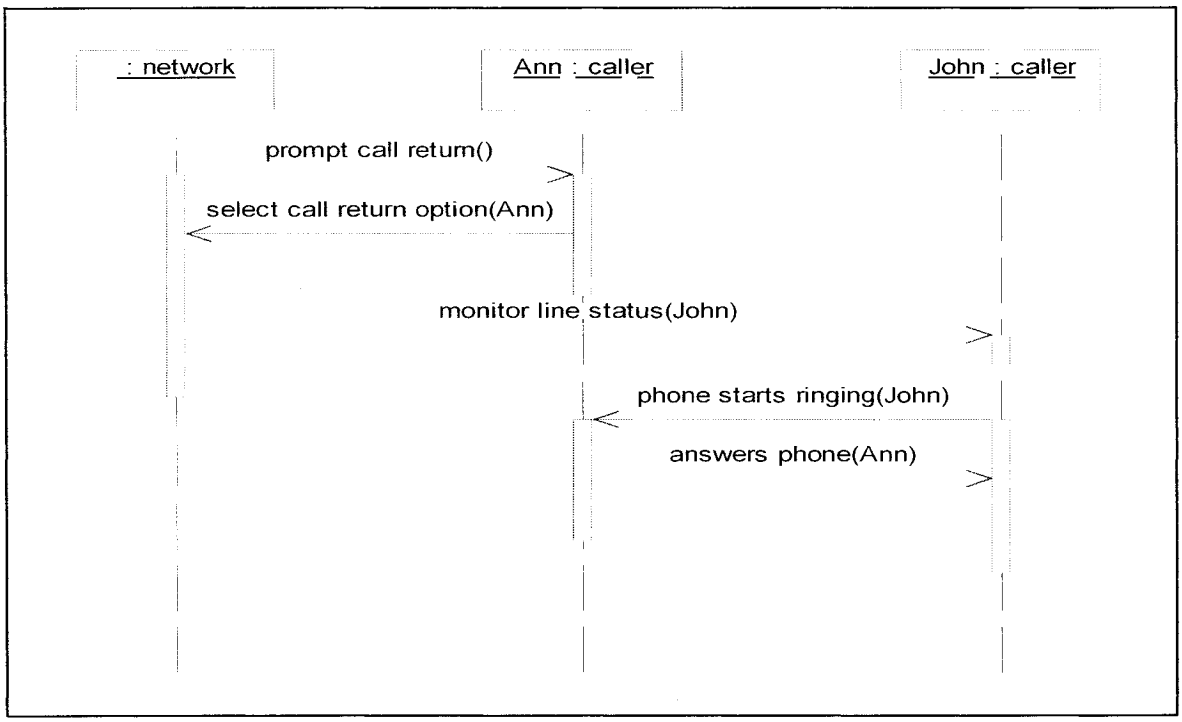


Figure 3-12: A Sequence Diagram that describes the main scenario of use case: "Call Return"

Figure 3-12 represents the main scenario of the *"Call Return"* use case of the Telephone PABX system that is presented in Appendix E. The main difference between this sequence diagram and the instances based use case: *"Call Return"* is that in the sequence diagram instances: *"Ann"* and *"John"* belong to the same entity: *"caller"*, whereas in the use case, the instances: *"Ann"* and *"John"* needed to belong to the entities: *"caller"* and *"recipient"* respectively. This shows some of the difficulties in eliciting requirements without instances. Without instances, representing an interaction between two instances of the same entity is ambiguous. In the above example the roles of the *"caller"* and *"recipient"* entities are

associated with the names of their respective instances. This is because that both the *"caller"* and *"recipient"* entities are referred by the same entity name: *"caller"*.

The boxes across the top of the sequence diagram specify classifiers or their instances typically objects, classes or actors. Because messages can be sent to both instances and classes, instances respond to messages through the invocation of an operation and classes may do so through the invocation of static operations. It makes sense to include both instances and classes in sequence diagrams. Labels that are used to represent instances and classes in sequence diagrams are written in different formats. Labels used to represent instances are written in the standard UML format *"name:ClassName"*, where *"name"* is an optional instance name. *"ClassName"* could be treated as the name of entity of the corresponding instance. Instances that have not been given a name on the diagram are called *"anonymous instances"*. They are represented using a label of the form *":ClassName"*. In Figure 3-12, *"Ann"* and *"John"* are instances of entity: *"caller"*, whereas *"network"* is an anonymous instance.

In the above diagram, instances are passed as parameters to the messages that are initiated by instances of entities. Messages represented by sequence diagrams are operations. This implies that parameters are a powerful mechanism in representing and manipulating operations that are initiated by instances of entities.

3.5.2 Case Studies of use case models that support instances

In [35], the author have introduced several case studies. Each of these case studies consists of a use case model and a domain model. The use cases of each use case model contain use case descriptions, which can be expressed more clearly using instances. In this section, we discuss the importance of having instances for some of these case studies.

Elevator System Case Study: A detailed description of this case study is presented in section 4.7. In addition, instances based use cases of this case study is presented in Appendix I. As specified in Figure 4-18, *"elevator"* domain entity has three instances: *"ElevatorA"*,

"ElevatorB" and *"ElevatorC"*. When a user at a particular floor requests for an elevator, the system selects a particular elevator instance out of the three available elevators. This example shows how instances are important to distinguish a particular instance of an elevator from the other instances of elevators.

Banking System Case Study: This case study describes a use case model relating to a banking system. A bank has several automated teller machines (ATM's), which are geographically distributed, and connected via a wide area network to a central server. This is a client/ server type case study. A customer can withdraw money, query balances or transfer funds between accounts using any of the ATM machines. This is a more complicated case study than the Elevator System case study. ATM machines of the bank can be identified by their respective instances. In addition, there can also be several instances of customers, each of whom is using one or more instances of an ATM machine. Therefore, without having instances it is tedious to express interactions in the use cases of the Banking System case study.

Distributed Factory Automation System Case Study: This is a case study of an assembly plant. Manufacturing workstations are laid out in an assembly line. Each manufacturing work station has an assembly robot for assembling a product and a pick-and-place robot for picking parts off and placing parts on the conveyor. Each robot is equipped with sensors and actuators. As such, instances of a *"workstation"* entity can be used to distinguish the functionality and the characteristics of different workstations. Parts are moved between workstations on a conveyor belt. Typically, a number of parts of the same type are produced, followed by a number of parts of a different type. Therefore, we may also have to have instances of an entity: *"part"* to distinguish the characteristics between different parts.

3.6 Why domain knowledge is needed for use case modeling?

In the preceding section we stated that use case descriptions are best described using a restricted form of natural language. One other advantage of such a restricted form of language is that it could be parsed and processed by a machine. In order to parse a use case

description, we need to be able to identify the syntactic elements (say nouns and verbs) of the language. For that we must have a dictionary that contains domain knowledge. It is from these nouns and verbs of use case descriptions that entities and operations of a domain model are derived. A domain model would also facilitate validation of use case descriptions by assisting in identifying synonyms. Therefore, it is important that we have a domain model for use case modeling. However, it is not realistic to have a complete domain before starting the use case modeling activity. Most often it is a partial domain that is available for use case modeling. As elements are detected in the domain while performing the use case modeling activity, the domain model could be updated as and when required, until the domain model is complete.

The motivation of having a domain extraction from a use case model is driven by the fact that a domain model is a unifying glossary. It helps having domain elements for different stakeholders who "talk the same language". This feature minimizes the ambiguity of the language to a greater extent. One of the reasons to this fact is that a domain model contains non-redundant entities, which are common to all the stakeholders of a system.

A domain model also provides a smoother transition from analysis to design [35]. Further a static domain model could be further be refined into a dynamic object model, which represent the interactions of instances.

3.7 Chapter Summary

Use cases have been formalized and expressed in various ways. Use cases so formalized exist at different levels of abstraction. In section 3.1, we reviewed different use case formats. It is important to note that use cases can be illustrated in various other forms. A use case is used to specify interactions between domain entities. Therefore a use case can also be expressed as collaboration diagrams, state machines, etc. Wirfs-Brock use cases, Essential use cases and Cockburn use cases are detailed black box use cases, whereas UCMs, MSCs, sequence diagrams and state machines can be used to write use cases at a grey box level. In our thesis we lay emphasis on use cases in textual form. We use Cockburn guidelines. The

importance of having use case guidelines is to reduce the vagueness and improve consistency in writing them.

In this chapter, we also reviewed restricted forms of natural language grammars for writing use cases. It is important to have a restricted grammar for writing use cases, so that use case descriptions could be easily processed by a machine. At the same time, it is important that textual use cases have a high degree of expressive power. Support for instances in textual use cases enhances their expressive power. None of the grammars reviewed in section 3.3 allows expression of entities as instances. Other shortcomings include flexibility of the grammar and support for pronoun resolution.

Another observation that we made in this chapter is the necessity of domain knowledge in order to parse use case descriptions. This implies that we at least require a partial domain before starting the use case modeling activity.

Chapter 4 - Proposal for restricted grammar for use cases

We emphasized the importance of having a restricted form of grammar for writing use cases. A grammar improves consistency of use cases. In this section, we introduce a restricted form of grammar for processing use cases. The main features of this grammar are support for instances and automatic extraction of missing domain elements from use case descriptions. As discussed in section 3.6, we need to have domain knowledge in order to parse use cases. But in practice we do not have a complete domain before use cases are captured. Therefore, we need a mechanism to add new domain knowledge to the domain while parsing use cases. The first few sections of this chapter will elaborate on our grammatical approach. Then we will present the design and functionalities of a parser for that grammar. Thereafter, we present two case studies for the purpose of evaluating our use case grammar. Subsequently, we determine why it is not possible to use a corpus as the domain model. Finally, we wrap up this chapter with a discussion on the limitations of the approach and provide suggestions for improvement.

4.1 Traditional method of extracting domain elements from use case models

Requirements gathered from interviews and requirements documents are used for composing use cases. After the formulation of use cases the next step should be the extraction of domain elements from use case sentences. A domain model is merely a UML class diagram [34]. A domain model contains entities (i.e. classes), attributes, operations and associations. In order to parse and identify domain elements from use case sentences we need to have a domain model. But the problem is that initially we do not have a domain for use case modeling. So how can we identify domain elements that are part of use case sentences? Traditional methods suggest that domain elements can be identified using the building blocks of the language: nouns and verbs [8]. Domain entities and their attributes can be identified using nouns. Most domain entities of use case sentences can be extracted using common nouns of a corpus. Domain entities can also have instances. Instances of domain entities can be extracted using proper nouns. However, all proper nouns cannot be located in a corpus or a

dictionary. This is because proper nouns include names of people, locations or things. Names of proper nouns depend on the geographic location, language used and the culture. Therefore, it is impossible to have all proper nouns in the world in one corpus. Domain operations and associations between entities are identified using verbs which are contained in a corpus. Identification of domain elements of this manner is used to drill down from high-level requirements models to detailed design.

4.2 Proposed syntax for writing use cases

Natural Language parts of use cases consists of conditions and operations [7, 34]. A use case condition is a predicative sentence that involves an associated domain entity. A use case operation describes an action initiated by a subject entity in a use case sentence. A use case sentence can be represented using a natural language grammar such as English [8, 9]. In this section we will present a restricted natural language form of grammar for writing use case conditions and operations. Our restricted form of grammar is based on the ideas of [7-9, 34]. We use a notational extension of SWI Prolog called Definite Clause grammar (DCG). A brief introduction on how to write DCG grammar is presented in the following sub-section. A grammar written using DCG is directly executable by Prolog as a syntax analyzer.

4.2.1 Writing DCG grammar

DCG are notational extensions provided by an implementation of Prolog. A grammar includes a set of production rules. Each rule declares things that are always unconditionally true. A set of simple grammar rules for a subset of the English language can be written as follows,

```
sent (Num, sentence(NP, VP)) --> noun_ph(Num, NP), verb_ph(Num, VP).
verb_ph(Num, verb_phrase(Verb, NP)) --> verb(Num, Verb), noun_ph(Num1, NP).
noun_ph(Num, noun_phrase(Det, Noun)) --> determiner(Det), noun(Num, Noun).
determiner(determiner(Word)) --> [Word], {is_determiner(Word)}.
noun(Num, noun(Word)) --> [Word], {is_noun(Num, Word)}.
```

The left hand side and the right hand side of a DCG rule is separated by the "--> " symbol. The left hand side of a grammar rule can contain only non terminals, whereas the right hand side may contain both terminals and non terminals. Non-terminals of a grammar rule starts with a capital letter. Terminals may be represented in square brackets. All of these grammar rules are written using predicates. A predicate is a technique used to represent the meaning of a logical notion. For example, the first rule: *"sent (Num, sentence (NP, VP))"* mentioned above is a predicate. It says that a sentence consists of a noun phrase (*NP*) and a verb phrase (*VP*). This predicate uses the argument *"Num"* in order to ensure number agreement between a noun phrase and a verb phrase. In the above predicate *"sentence"* is a functor. Functors are used to explicitly show the constituent structure of components in a parse tree. Predicates of a Prolog grammar rule may be separated by commas. This illustrates the predicates participates in a conjunction. Contrarily, if the semi-colon is used as the separator between predicates of a grammar rule it participates in a disjunction. In addition to grammar rules, Prolog also defines facts. They are used to declare things that are always unconditionally true. A set of Prolog facts for the English language grammar which is specified above is listed below,

determiner(the).
noun(singular, network).
noun(plural, networks).

Observe how the number agreement of nouns can be logically represented in the Prolog facts. The above facts clearly define that *"network"* is a singular noun, whereas *"networks"* is a plural noun. The domain model for our proposed grammar is a set of Prolog facts. We describe these Prolog facts in the next section.

4.2.2 Domain model for the proposed grammar

The domain model for our proposed Prolog grammar is a set of Prolog facts.

Figure 4-1 presents an excerpt of the Prolog facts that represents domain knowledge for parsing use cases.

```
is_verb(is).
is_verb(are).
.....

is_entity([caller, phone]).
is_entity([recipient, phone]).
is_entity([network, call, status]).
is_entity([caller]).
.....

is_val(unidentified).
is_val(normal).
is_val(idle).
.....

is_instance(recipient,'John').
is_instance(caller,'Ann').
.....

is_action_verb(translate).
is_action_verb(put).
is_action_verb(connect).
.....
```

Figure 4-1: Domain model for the parsing the restricted natural language grammar

To support the description of domain entities in conditions, Prolog fact: *"is_verb(...)"* is used. Domain entities are represented using the Prolog fact *"is_entity(...)"*. Prolog fact: *"is_val(...)"* is used to describe the entities. As an example, consider condition: *"caller phone is idle"*. In this example *"caller phone"* is an entity, "is" is a verb and "idle" is a value. These elements are represented in the domain using Prolog facts *"is_entity([caller, phone])"*, *"is_verb(is)"* and *"is_val(idle)"* respectively.

In our approach we also represent attributes, aggregates, etc. as single entities by using its fully qualified name preceded by the concept name. For example in Prolog fact: is_*entity([network, call, status])*, *"network"* is a system concept and *"call status"* is an attribute of the system concept. But we consider the fully qualified attribute as an entity.

Prolog fact: *"is_instance(...)"* is used to represent instances. In Figure 4-1, the first Prolog fact for an instance: *"is_instance(recipient,'John')"* expresses that *"John"* is an instance of entity: *"recipient"*.

Prolog fact: *"is_action_verb(...)"* is used to represent domain operations. A use case operation exists as a conjugated form of a verb (e.g. *"calls"*). But we represent domain operations using the infinitive form of the verb (e.g. *"call"*) of the corresponding use case operation.

4.2.3 Grammar for use case conditions

A restricted form of grammar for use case conditions is shown in Figure 4-2.

```
1.  comp_pred([Cond, Sep, RemCond]) --> pred(Cond), comp_sep(Sep), comp_pred(RemCond).
2.  comp_pred(Cond) --> pred(Cond).
3.
4.  pred(predicate(entity(Inst,EntPro),verb(Verb), Comp, value(Value))) -->
5.                  entity_cond(_, [], Inst,EntPro, 1),  verb_modifier( Verb),
6.                  ((comparator(Comp),val(Value)) ; ({Comp = []}, val(Value))).
7.
8.  entity_cond([EntHead, EntTail], X, Inst, Ent, N) -->
9.        (
10.                 (({N=:=1},instance(Concept,EntHead)) ; ent_noun(EntHead)),
11.                 {(var(Concept),add(EntHead, X, Y), ((N=:=1,conc([],[],Inst));write('')) ) ;
12.                 (add(Concept, X, Y),add(EntHead,[],Inst))} ,
13.                 {N1 = N+1}, entity_cond(EntTail,Y,Inst,Ent, N1));
14.                 {reverse(X, Ent)},!,
15.                 ({(is_entity(Ent));
16.                 (write_entity(Ent), write_to_log(entity, Ent, predicate))}
17.       ).
18.
19. ent_noun(Word)  -->  [ Word ],  { is_noun( Word ) }.
20. verb_modifier(Word) --> [Word] , {is_verb(Word) }.
21. val(Word) --> [Word] , ({is_val(Word) ; (write_value(Word),write_to_log(value, Word, pred)) }).
22. comparator(Word) --> [Word] , {is_comparator(Word) }.
23. instance(Concept, instance(Instance))  -->  [ Instance ],  { is_instance(Concept, Instance ) }.
```

Figure 4-2: Restricted grammar for use case conditions

Lines 1 and 2 define a composite predicate as being made out of several simple predicates: *"pred"*, separated by appropriate separators. A separator is defined using predicate:

"comp_sep". The separator of a composite condition could be a conjunction (i.e. *"and"*) or a disjunction (i.e. *"or"*). An example of a composite predicate is *"caller is offhook AND recipient is offhook"*. This composite condition is made out of two constituent simple predicates: *"caller is offhook"* and *"recipient is offhook"*.

Lines 4-6 shows that a simple predicate starts with an entity (i.e. as represented by predicate: *"entity_cond"*), followed by a verb and a value. If a comparator (i.e. >, <, >=, <=) is used with a condition value in the predicate, the condition is said to be a non-discrete condition (e.g. *caller ring count is > 5*), otherwise the condition is discrete (e.g. *caller is offhook*).

When parsing a condition entity of a predicate and if it happens to be missing in the domain, steps should be taken to update the missing entity in the domain model. This is shown between lines 8-17 in the grammar. Line 10 identifies whether the condition entity is a normal entity or an instance. If the condition entity is a normal entity, each constituent word in it should belong to the noun category. To ascertain whether a word is a noun we use the predicate: *"ent_noun"*. This predicate will in turn call the predicate: *"is_noun"* in order to find whether it is defined as a noun in the WordNet dictionary. Line 11 is executed only if the condition entity is a normal entity. In this case each constituent word of the entity that is processed is added to the head of the list *X* and thereafter it is assigned to list *Y*. This process is carried out by predicate *"add(EntHead, X, Y)"*, where *"EntHead"* is the currently processing constituent word of the condition entity, which is of noun form. Line 12 is executed if the condition entity refers to an instance. In this case *"EntHead"* is an instance and its corresponding entity is *"Concept"*. The corresponding entity of the instance is ascertained from the domain. Lines 10-13 are performed iteratively until the appropriate condition entity is created by concatenating each of its constituent words (i.e. by iterating until a verb or an instance is found). A verb in a condition can be identified using the predicate: *"verb_modifier"*, which in turn will call the predicate: *"is_verb"*. An instance in the domain is defined using the predicate: *"is_instance"*. Since the list X is updated with constituent words of an entity that is being processed in the reverse order, line 14 formats the condition entity. If line 15 is not satisfied, condition entity that is evaluated will not exist in the domain. In such a situation, predicate *"write_entity"* adds the missing condition entity

into the domain and logs the details of the transaction using the predicate: *"write_to_log"*. A condition value is identified from the domain by using the predicate: *"is_val"*. Line 21 depicts that if a condition value is not in the domain, it is added to the domain by using the predicate: *"write_value"*.

Let's take the example condition: *"caller line status is busy"*. In this case *"caller line status"* is identified as the condition entity, only after iterating the steps in lines 10-12 of the grammar. The iteration stops when it finds the verb *"is"*. Thereafter, *"busy"* is identified as a discrete value of the condition. If the respective condition entity or value is found missing in the domain, the parser will update the domain with the new domain knowledge. In this example *"line status"* is an attribute of concept: *"caller"*. But the parser is not capable of distinguishing the sub-entity type of the condition entity. This is because isolating sub-entity types from a natural language corpus is far from possible.

4.2.4 Grammar for use case operations

The restricted grammar for a use case operation is depicted in Figure 4-3.

```
1.  usecaseop([entity(Inst, Ent), operation(Oper), Phr])-->
2.                          entity_op(Inst, Ent, Oper, [],1),
3.                          operation(Oper, Stem, Phr),
4.                          (chk_instance(Ent); (({is_entity(Ent), write('')});
5.                          ({write_entity(Ent), write_to_log(entity, Ent, operation)}))),
6.                          (({is_action_verb(Stem), write('')});
7.                          ({write_verb(Stem), write_to_log(verb, Stem, operation)})).
8.
9.  chk_instance(instance(Ent)) --> {is_instance( _, Ent )}.
10.
11. operation(Oper, Stem, Phr) --> conjugated_verb(Oper),{stem_verb(Oper,Stem) }, phr(Phr).
12.
14. entity_op(Inst, Ent, Oper, X, N) -->
15.      (
16.              ((conjugated_verb(EntHead),{assert(isVerb(1))});({write('')})),
17.              (({isVerb(1), is_conjugated_verb(EntHead)});
18.              ({not(isVerb(1))},(ent_noun(EntHead);instance(_,EntHead)))),
19.              {((N=:=1, instance(Con,EntHead,_,_),add(EntHead,[],Inst),add(Con, X, Y));
20.              (add(EntHead, X, Y), ((N=:=1,conc([],[],Inst));write('')))},
21.              {N1 = N+1}, entity_op(Inst, Ent,Oper, Y, N1)) ;
22.              ({del(Oper, X, Q), reverse(Q, Ent), retractall(isVerb(_))}
23.      ).
24.
25. conjugated_verb(Word) --> [ Word ], { is_conjugated_verb( Word ) }.
26.
27. phr([NP,PP]) --> nphr(NP),pps(PP).
28. phr(Phr) --> nphr(Phr).
29. phr(Phr) --> pps(Phr).
```

Figure 4-3: Restricted grammar for use case operations

Lines 1-7 define a use case operation. A use case operation consists of an operation entity (i.e. as represented by predicate *"entity_op"*) and an operation. An operation entity could be a normal entity or an instance. To check whether a particular word is an instance when parsing, we use the predicate *"chk_instance"*. Similarly, an entity is identified by using the predicate *"is_entity"*. These parts of a use case operation are identified at line 4. As shown in line 5 of the grammar, if the operation entity is found missing in the domain, it is added to the domain by using the predicate: *"write_entity"*. Operations are represented in the domain by using the Prolog fact: *"is_action_verb(...)"*. Line 7 shows that, if an operation is not in the domain, it is added to the domain by using the predicate: *"write_verb"*. Identification of an operation entity or an instance and its corresponding operation is specified in lines 16-22.

Lines 14-23 illustrate how the operation entity and the conjugated form of the use case operation are identified. Finding an operation entity is not easy. Usually an operation entity

may consist of a combination of words that are of noun form. However, a word which is of noun form may also have a verb sense. This is one problem of the inherent ambiguity of the natural language and we have to deal with it when identifying an operation entity.

According to this grammar each word in a use case description is scanned from the left. If it is an instance the parser identifies its corresponding entity from the domain. If the word is a noun then it is part of an operation entity. The parser will then add the noun to the head of a temporary list. This process iterates until the parser finds a verb. When the parser finds the first occurrence of a verb, flag: *"isVerb"* is updated to value 1. But remember that this verb may have a noun sense and also be a part of the operation entity. To overcome this problem, we employ a heuristic. The heuristic is that we select the last continuous verb from the first occurrence of the verb of a use case description as the use case operation. This iterative process in finding the entity operation is terminated when the token after the last continuous verb is processed. This is because the predicates between lines 16-20 are not satisfied, when it processes the word immediately after the entity operation and when the value of the flag: *"isVerb"* is 1. Line 22 formats the operation entity which is processed.

A use case operation consists of an operation and a phrase. A use case operation is in conjugated form. Conjugated form of an operation is represented in the grammar by using the predicate: *"conjugated_verb"*. But the domain contains operations in its infinitive form. Therefore, we use the predicate: *"stem_verb"* to convert a conjugated form of a verb to its infinitive form. The phrase (i.e. *"phr"*) is the part of the use case description that appears immediately after the use case operation. As shown in lines 27-29 a phrase may be a noun phrase, prepositional phrase or a noun phrase followed by a prepositional phrase. In addition, a phrase may also be composed of other grammatical structures.

Consider the example use case sentence: *"caller display shows the number"*. The parser scans this sentence from its left end and the first verb that we detect is: *"display"*. But in this case this verb should be part of the use case entity: *"caller display"*. This is why we need to employ a heuristic to determine the operation of a use case description. When we find the first verb, we further scan through the sentence until we reach the last continuous verb of the

use case sentence. This last continuous verb: *"shows"* is treated as the use case operation. The portion that appears before the use case operation is identified as the operation entity.

We tag the word forms of the phrase, so that a requirement analyst has the option of further identifying domain sub-entities manually from words that falls into the noun category. The parse tree for the example mentioned above is,

[entity([caller, display]), operation(shows), [[the, []], [noun(number), []]], []]

In this case the analyst has the option of identifying the noun *"number"* as an attribute of the entity: *"caller display"*.

4.2.5 Grammar for writing use cases

The grammar that we use to write use cases is derived from the template used by UCEd. In our grammar we distinguish normal use cases and extension use cases. The next sub-section introduces the abstract syntax used to define use cases in UCEd. The subsequent sub-sections describe the grammar that we use to define normal and extension use cases.

4.2.5.1 Abstract UCEd syntax for writing use cases

UCEd adheres to an abstract syntax that is based on UML for defining use cases. This is shown in Figure 4-4.

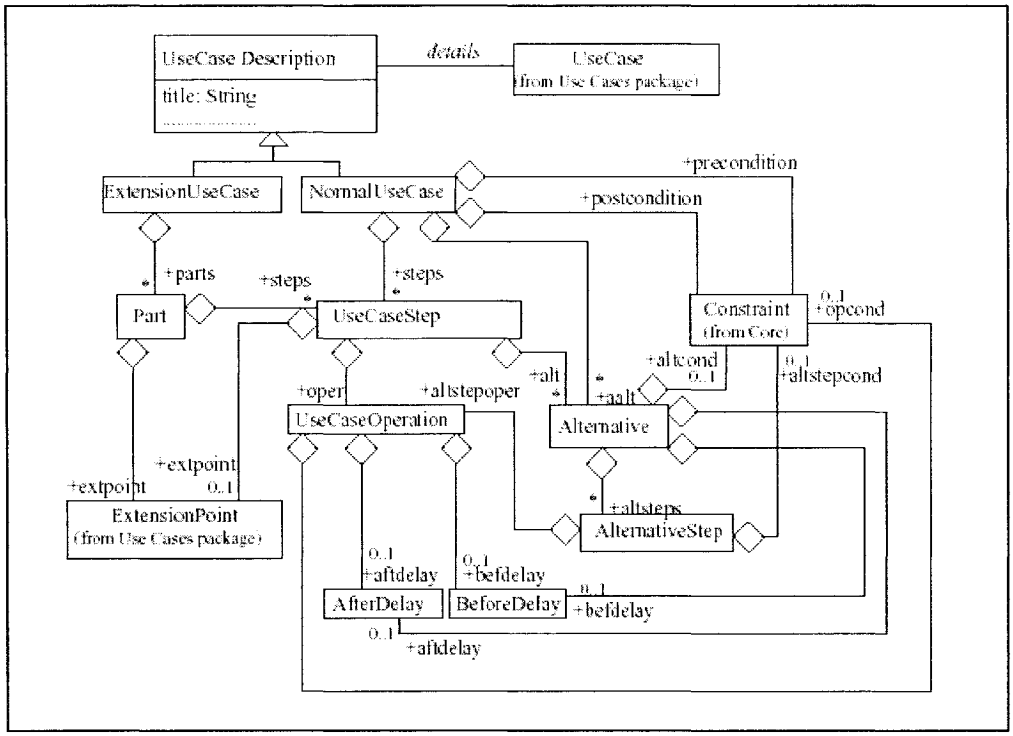


Figure 4-4: UML representation of abstract syntax for use case descriptions

According to Figure 4-4, a use case can be divided into two categories: *Normal Use Case* and *Extension Use Case*.

A normal use case can be considered as a tuple: *[Title, Precondition, Steps, Use Case Alternatives, Postcondition]*. Title is a unique name for the use case, *Precondition* is a condition (or a constraint) that must be satisfied before the use case can be executed, *Steps* are a sequence of steps, Use Case *Alternatives* are a set of alternatives that may apply to each step in the use case and *Postcondition* is a condition that must be satisfied after the execution of the use case. Each step of a use case is a tuple: *[Oper, ExtPoint, Alt]*. *Oper* is a use case operation, *ExtPoint* is an optional extension point and *Alt* is a set of possible alternatives after a step. A use case operation can be a concept operation, branching statement or a use case inclusion directive. A use case operation may include a guard (*OpCond*) that must hold for the operation to be possible, and a timing condition. A timing condition may be in the form of an *"After Delay"* or *"Before Delay"*. Delays are measured from the completion moment of the previous step of the use case. An *"After Delay"* specifies the minimum time period that must pass for the operation to be possible, while a *"Before Delay"* specifies the

maximum time period after which an operation is no more possible. An extension point is some label, which is a reference made after a specific step of a use case, at which point interactions defined in an extension use case may be inserted. An alternative describes an exceptional course of action in the use case and specifies a possible continuation after a specific use case step. An alternative is a tuple: *[AltCond, AftDelay, BefDelay, AltSteps]* where *AltCond* is a guard that must be true for the alternative to hold. The existing framework does not support alternatives of alternatives, instead it employs *"include use cases"* to overcome this barrier and to reduce complexity.

An extension use case can contain one or more parts. These parts are inserted at specific *extension points* in a base use case in the presence of an *extend relationship*. An extension use case is a tuple: *[Title, Parts]*. Each part in *"Parts"* in turn is a tuple: *[ExtPoint, Steps]*. *ExtPoint* is a reference to an extension point and *Steps* are a sequence of steps. The extension use case may participate with its base use case with an *extend relationship* under a specific *extension condition.*

4.2.5.2 Grammar for writing normal use cases

In this section, we present the proposed restricted form of grammar that is used to define normal use cases.

```
1.  use_case([PreCond, UCSteps]) -->
2.                  pre_cond(PreCond), process_step_sep(_),  overall_use_case_steps(UCSteps).
3.
4.  pre_cond(pre_cond([If, PreCond])) --> if_cond_clause(If), comp_pred(PreCond).
5.  pre_cond([]) --> [].
6.
7.  overall_use_case_steps([UCStep, UCSteps]) --> overall_use_case_step(UCStep), process_step_sep(_),
8.                  overall_use_case_steps(UCSteps).
9.  overall_use_case_steps(UCStep) --> overall_use_case_step(UCStep).
10.
11. overall_use_case_step([Step, Exts]) --> use_case_step(Step), process_ext_sep(_),  overall_ext(Exts).
12. overall_use_case_step(Step) --> use_case_step(Step).
13.
14. use_case_step([step(TimingCond, Cond, Step, ExtPoints)]) -->
15. use_case_step_time_cond(TimingCond),
16.                 use_case_step_cond(Cond), usecaseop(Step), ext_points(ExtPoints).
17. use_case_step([include(Cond, Include)]) --> use_case_step_cond(Cond), include_use_case(Include).
18. use_case_step(Goto) --> goto(Goto).
19.
20. use_case_step_cond(Cond) --> step_cond(Cond), { length(Cond,N), N =\= 0 } , process_step_sep(_).
21. use_case_step_cond(Cond) --> step_cond(Cond), { length(Cond,N), N =:= 0 }.
22. use_case_step_cond([]) --> !.
23.
24. use_case_step_time_cond(TimeCond) -->  timingcond(TimeCond), {not(is_list(TimeCond))},
25.                 process_step_sep(_).
26. use_case_step_time_cond(TimeCond) -->  timingcond(TimeCond), {is_list(TimeCond)}.
27  use_case_step_time_cond([]) -->!.
28.
29. ext_points([ExtPoint, ExtPoints]) --> ext_point(ExtPoint) , ext_points(ExtPoints).
30. ext_points(ExtPoint) --> ext_point(ExtPoint).
31.
32. ext_point(ext_point(Ext, ExtPntCond, ext_point_name(ExtPntName))) --> process_step_sep(_) ,
33.                 ext_keyword(Ext) , process_step_sep(_) ,
34.                 use_case_step_cond(ExtPntCond) , process_step_sep(_) ,
35.                 ext_point_name(ExtPntName).
36.
37. include use case([Inc, UseCaseName]) --> inc(Inc), uc  name(UseCaseName).
```

Figure 4-5: Restricted grammar for a normal use case

As specified in lines 1 and 2, a normal use case consists of a precondition and a set of overall use case steps. A precondition is a use case condition, which must be satisfied prior to the execution of the use case. Lines 11-12 depicts that an overall use case step can have a use case step and an optional number of use case step extensions.

Lines 14-16 specifies that a use case step may consist of an operation. Such a use case step may also have optional use case step conditions, timing conditions and extension points. Extension points are used to include extension use cases to a normal use case. As specified by line 17 a use case step may also be defined to have a subordinate use case (i.e. an include

use case). A subordinate use case is executed under an optional condition. Line 37 specifies that an include use case consists of an *"Inc"* keyword and the name of the included use case. Line 18 illustrates that a use case step may also consists of a simple branching statement (i.e. Goto clause).

Lines 20-22 define a use case step condition. These grammar rules calls the predicate *"step_cond"* which ultimately calls the predicate: *"comp_pred"* that is defined in section 4.2.3. Line 22 specifies that a use case step condition is optional. Similarly, lines 24-27 define an optional timing condition. Lines 29-35 describes that a use case may contain a series of use case extension points. Each of these extension points consists of an *"Ext"* keyword which is followed by an extension condition and the name of the extension point.

4.2.5.3 Grammar for writing extension use cases

In this section, we explain the grammar relating to extension use cases. Steps of an extension use case are similar to that of a normal use case. They are executed under specific extension conditions. The restricted form of grammar for an extension use case is represented in Figure 4-6.

```
1. ext_parts([ExtPart, ExtParts]) --> ext_part(ExtPart), ext_parts(ExtParts).
2. ext_parts(ExtPart) --> ext_part(ExtPart).
3.
4. ext_part(part([Part, ext_point(ExtPntName), UCSteps])) --> part_keyword(Part) , process_step_sep(_),
5.                                                           ext_point_name(ExtPntName), process_step_sep(_),
6.                                                           overall_use_case_steps(UCSteps).
```

Figure 4-6: Restricted grammar for an extension use case

As specified in lines 1 and 2, an extension use case may consists of one or more extension use case parts. Each of these use case part is represented by predicate: *"ext_part"*. Lines 4-6 illustrates that each use case part consists of a reference to an extension point of a base use case and a sequence of steps. A reference to an extension point is defined by using the predicate: *"ext_point_name"*, which is a name of a particular extension point. These parts are to be inserted at specific extension points in a base use case.

4.3 High level design of the use case parsing system

In this section we will discuss the high level design of the use case parsing system. Figure 4-7 illustrates the architecture of the use case parsing system. We look at the functionalities of each of its components in the following discussion.

A use case can be treated as a representation where both a domain specialist and a requirements analyst can understand and validate system requirements. In addition, use cases can also be used to refine the black box view of the system to low level system design. Subsequently a domain model needs to be created in order to isolate domain elements and operations. The domain model could be represented as a class diagram for the system. In reality, a use case model that is initially created does not have a complete domain model. Therefore we need some mechanism to update the missing entities in the domain. One technique is to use a natural language corpus to identify missing domain entities, so that a partial domain can be updated or a new domain model can be created if a domain is not in existence. Another technique is to have user intervention in order to identify missing entities in the domain. The latter technique seems to be awkward. It also has the weakness of not being able to fully automate. The former technique appears to be the ideal choice. The natural language parser for processing use cases described in this section is based on the former technique.

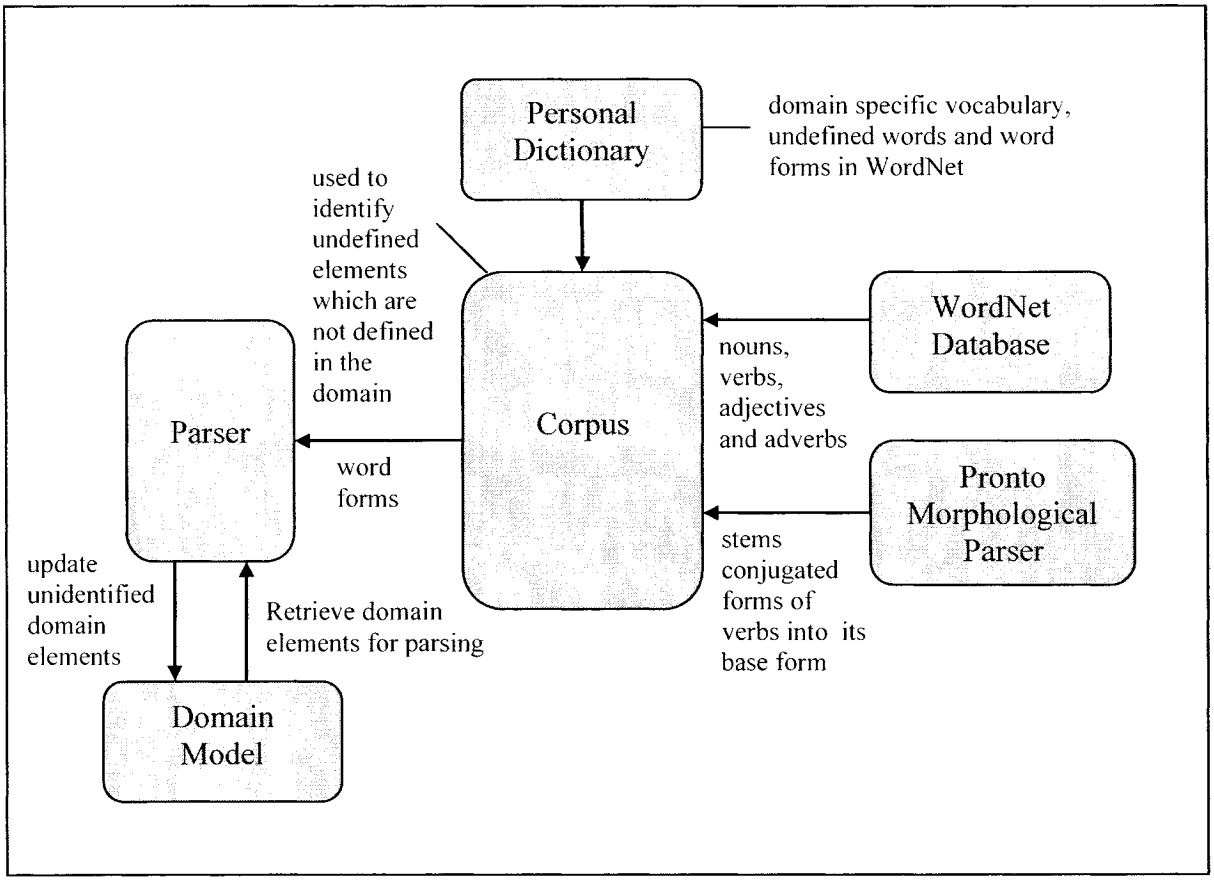


Figure 4-7: High level design of the use case parser

4.3.1 Domain Model

As mentioned in section 4.1, domain knowledge is required to parse use cases. Domain knowledge includes domain entities and operations. A domain model can be represented as a UML class diagram. Having a domain model for parsing use cases help reduce the ambiguity of the language. A domain model can help avoid synonymous words that refer to the same entity. Elements of a domain model are defined such that there are no elements with similar meanings for the same entity. Let's consider the example use case description: *"caller calls the client"*. In another use case of the use case model, the same use case description can be interpreted as *"receptionist calls the client"*. In this case *"caller"* and the *"receptionist"* refer to the same entity which has the meaning of a calling party. Having entities with similar meanings that refers to the same entity is inconsistent and ambiguous. We can overcome this problem by having a domain model, which contain concrete domain entities that are

unambiguous. In a rich domain model (UML), receptionist could be defined as a sub class of caller.

As described in section 4.2.2 the domain model used in our approach is a set of Prolog facts. It is extensible as new domain knowledge can be dynamically added using Prolog assertions. Section 4.2.2 also presents an excerpt of the Prolog facts that represents domain knowledge for parsing use cases.

4.3.2 WordNet Database

We mentioned in section 4.1 that domain elements can be extracted by identifying the nouns and verbs that are contained in use case descriptions. But we need a corpus or a dictionary to identify these different word forms. There are a number of dictionaries that exists, which are used for linguistic analysis. WordNet is one such popular dictionary which is developed by Princeton University [36]. It is a widely used database for English and contains separate databases for nouns, verbs, adjectives and adverbs. Unidentified domain elements of use case descriptions can easily be extracted using the word forms that exist in the WordNet database. WordNet provides a database that is written using Prolog. We use this database for identifying different types of word forms for parsing use case descriptions.

To look up a word form from the WordNet dictionary, we consult the predicate,
s(+Synset_ID,+W_Num,+Word,+SS_Type,+Sense_Number,+Tag_Count).

The important parameters of the above predicate for our work are *"Word"*, *"SS_Type"*, *"Sense_Number"* and *"Tag_Count"*. *"Word"* is the word whose word form we need to know. The type of this word is represented by *"SS_Type"* and could take the values: *"n"* (i.e. noun), *"a"* (i.e. adjective) and *"v"* (i.e. verb). *"Sense_Number"* gives information on how common a word is. Higher its value less common is the word. *"Tag_Count"* indicates how common a word is in relation to a text. Higher its value more common is the word. Since a particular word can exist in more than one word form, we use a heuristic, so that we can really identify whether a particular word belongs to the intended word form. The heuristic that we use is, to select a particular word form for a word, if it is the most common

word for that word category. For this purpose we set *"Sense_Number=1"* and *"Tag_Count>0"*. For example, the above predicate can be written to identify whether a word is the most common word for any of the word categories: nouns, verbs and adjectives. It can be represented as follows,

For nouns, s(_, _, <noun> , n , 1 , *Tag_Count), Tag_Count>0*
For verbs, s(_, _, <verb> , v , 1 , *Tag_Count), Tag_Count>0*
For adjectives, s(_, _, <adjective> , a , 1 , *Tag_Count), Tag_Count>0*

It is not a necessity to choose WordNet as the dictionary for parsing use cases. The analyst has the flexibility of selecting any other word dictionary.

4.3.3 Personal Dictionary

A personal dictionary is useful to define the undefined words that are missing in a dictionary. Although word dictionaries such as WordNet exist, there may be words which are not defined in such a dictionary that belongs to the relevant word form categories. It may also be the case that particular words are new to the English Language.

In the preceding section, we mentioned that WordNet dictionary can be used to identify unknown domain elements that are missing in the domain model. Unfortunately, all of these unknown words do not exist in the WordNet dictionary. One reason is that all word form categories are not supported by WordNet. It is only the nouns, verbs, adjectives and adverbs that exist in the WordNet database. But there are other forms of words such as articles, prepositions, proper nouns, etc. in the English language. These word form categories should be defined in a personal dictionary. Therefore, a personal dictionary is used to supplement WordNet with the definition of missing words. For instance WordNet doesn't define articles, prepositions, etc.

4.3.4 ProNTo Morphological Analyser

ProNTo Morphological Analyser [37] is used to stem a conjugated form of a verb to its infinitive form. Ascertaining a morpheme (i.e. infinitive form) from its conjugated form is not easy. This is because there are several spelling rules that exists to break a conjugated form of a word to its infinitive form [37]. Given a conjugated form of a verb, ProNTo Morphological Analyser outputs the corresponding infinitive form of the verb.

A use case operation consists of an action verb which is in conjugated form. But when we extract a use case operation to the domain model, we need to process the action verb such that it is in infinitive form.

4.3.5 Corpus

This module integrates the data of the WordNet dictionary and the personal dictionary. Therefore, we can ensure that we have sufficient amount of word forms required for use case analysis. In addition, the corpus uses the ProNTo Morphological Analyser to stem the action verbs of use case descriptions into its infinitive form.

A separate corpus also gives the advantage that the parser module is independent from data dictionaries. This gives the added flexibility of replacing the WordNet dictionary with any other dictionary without changing the implementation of the parser module.

4.3.6 Parser

The parser is used to process use case descriptions. The parser module implements the grammar for use cases. Given a use case as the input, the parser can parse a use case and process its constituent elements. The parser is also capable of adding missing elements in a domain while parsing the use cases. Supported use cases may be in the form of normal use cases and extension use cases.

4.4 Functionalities supported by the parser

Our natural language parser carries out the following three tasks: creation of a parse tree for an input use case, extraction of missing domain elements from use case descriptions, update of a log file with the details of new domain knowledge which is added to the domain model. In this section we discuss the functionalities which are supported by the parser. Figure 4-8 illustrates these functionalities. A use case is provided as an input to the parser.

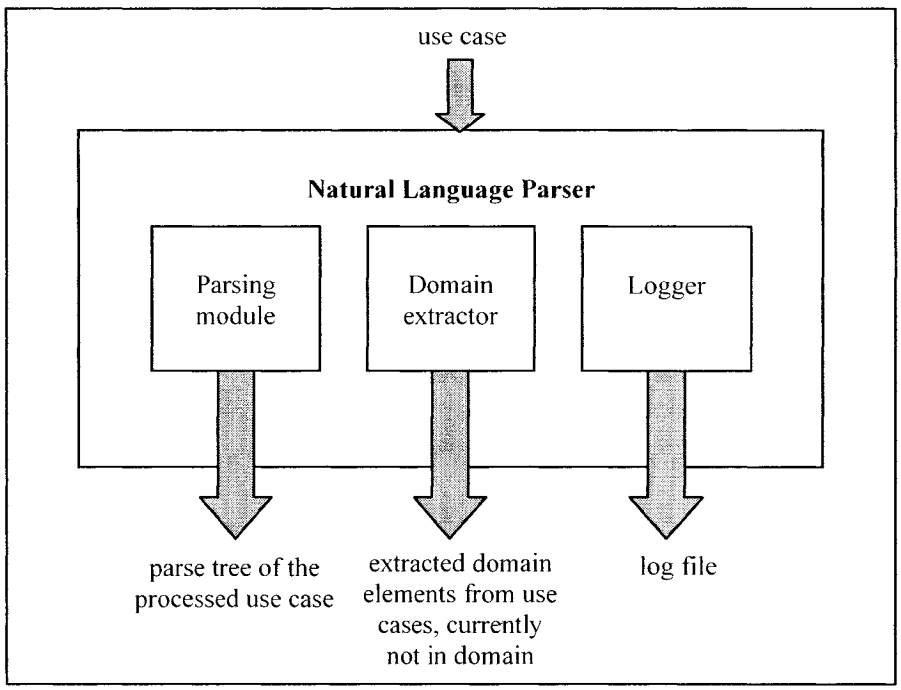


Figure 4-8: Functionalities supported by the Natural Language Parser

4.4.1 Parsing module

The parser converts use case descriptions into an abstract syntactic structure. This structure is useful for identifying domain elements. It is also useful for identifying word forms of elements, so that they can be used to refine domain entities at a later stage. In order to parse use case descriptions we need to have a minimal domain or a corpus to identify domain elements that are contained within it.

A Case Study of a Telephone PABX system is presented in section 4.5. It describes the setting up of a simple phone conversation and features of a typical telephony system. We

applied all use cases that were derived from the case study to the parser. An interesting feature of our natural language restricted grammar is that it supports instances.

In the following, we will illustrate the functionality of our parser using the use case: *"Answer Call"*. It relates to answering an unattended telephone call by a *"message recorder"*. Figure 4-9 presents this use case which is in a form that is required by the parser. The main scenario steps are separated by a comma and the extension steps are separated by a semi-colon. Conditions starts with an "IF" clause and timing conditions starts with a "AFTER" or "BEFORE" clause.

```
1. IF Ann phone is offhook AND John phone is onhook AND John phone indication is ringing AND
2. John phone ring count is > 5 AND network call status is negotiating,
3.
4. message recorder outputs a welcome message followed by a beep,
5. Ann records a message;
6.        IF Ann is silent; AFTER 10 sec;
7.        network terminates the call;
8.        Ann hangs the call,
9. IF Ann is responsive, Ann hangs the call.
```

Figure 4-9: Input representation of "Answer Call" use case for the parser

In this example *"Ann"* and *"John"* are instances. *"Ann"* is the caller and *"John"* is the recipient of the telephone conversation. The example illustrates that if the recipient *"John"* does not pick up the phone after five rings, then the call is forwarded to a message recorder, so that caller *"Ann"* has the option of recording a message for recipient *"John"*. If *"Ann"* do not records a message within 10 seconds of hearing the beep of the message recorder, the network terminates the call. The parse tree for use case *"Answer Call"* is presented in Figure 4-10.

Lines 1 and 2 of Figure 4-9 is a use case precondition. It is processed by the predicate: *"pre_cond"* which is defined at lines 4-5 of Figure 4-5. Lines 1-5 of Figure 4-10 represent the part of the parse tree that is generated for the use case precondition. This precondition is a composite condition. Line 4 of Figure 4-9 is a use case step. It is applied to predicate: "use_case_step" which is located at lines 14-16 of Figure 4-5. Lines 7-8 of Figure 4-10 represent the relevant part of the parse tree for the use case step. Similarly, line 6 of Figure

4-9 is a use case extension condition while lines 7 and 8 represent the use case extension steps. After being processed by the parser, they are represented at line 10 and lines 11-13 of Figure 4-10 respectively.

```
1.  [pre_cond([if, [predicate(entity([instance(Ann)], [caller, phone]), verb(is), [], value(offhook)), AND,
2.  [predicate(entity([instance(John)], [recipient, phone]), verb(is), [], value(onhook)), AND,
3.  [predicate(entity([instance(John)], [recipient, phone, indication]), verb(is), [], value(ringing)), AND,
4.  [predicate(entity([instance(John)], [recipient, phone, ring, count]), verb(is), >, value(5)), AND,
5.  predicate(entity([], [network, call, status]), verb(is), [], value(negotiating))]]]]]),
6.
7.  [[step([], [], [entity([], [message, recorder]), operation(outputs), [[a, [adjective(welcome)], [noun(message),
8.  []]], []]], []),
9.  [[[step([], [], [entity([instance(Ann)], [caller]), operation(records), [[a, [], [noun(message), []]], []]], []),
10.       [exCond([[], IF, predicate(entity([instance(Ann)], [caller]), verb(is), [], value(silent))]),
11.       [[extStep(timing_cond(AFTER, duration(10, sec)),
12.                 [entity([], [network]), operation(terminates), [[the, [], [noun(call), []]], []]])],
13.       [extStep([], [entity([instance(Ann)], [caller]), operation(hangs), [[the, [], [noun(call), []]], []])]]]],
14. [[step([], [IF, predicate(entity([instance(Ann)], [caller]), verb(is), [], value(responsive))],
15        [entity([], [caller]), operation(hangs), [[the, [], [noun(call), []]], []]], []), []]]]]
```

Figure 4-10: Parse tree for use case "Answer Call"

The parse tree is a tagged representation of use case descriptions. Entities and operations can be identified clearly from the parse tree. Entities and operations are tagged using the functors *"entity"* and *"operation"* respectively. The use of functors for tagging the relevant parts of a parse tree, make the parse tree readable. It also provides easy identification of different use case parts. Word forms of words in a use case operation which are located after its action verb is tagged appropriately. Words that cannot be resolved using the corpus are ignored. For example, let's consider the use case step: *"message recorder outputs a welcome message followed by a beep"*. In this case, the parser tags the words: *"welcome"* and *"message"*, which are located after its action verb: "outputs", as nouns. Thus, it could be the case that *"welcome message"* is an attribute of the concept: *"message recorder"*. But this is a decision for a requirements analyst.

4.4.2 Domain extractor

Having a complete domain prior to use case modeling is far from possible. Therefore, the reality is that we need to make updates for new domain knowledge, as and when they are found. For this purpose we use a corpus. For simplicity, we assume that all domain sub-entities (i.e. concepts, attributes, aggregates, etc.) can be represented as a generic entity type. Thus, we explore the possibility of having a fully automated domain extraction process

without user intervention. While parsing use cases, the domain model is updated dynamically with new domain elements found. This avoids the tedious process of manually updating a domain model for new domain knowledge found. Not only failures could be triggered by a manual technique of extracting a domain, but also it could happen inherently, because users do not know how to say what they want. The automatic extraction of domain elements was described in Figure 4-2 and Figure 4-3. In Figure 4-2, when the parser identifies that a corresponding entity is not in the domain it will add it to the domain. Similarly, as described in Figure 4-3, when the parser identifies that an operation is not defined in the domain, it will add it to the domain. To illustrate the automatic domain extraction process, we apply the use case: *"Answer Call"* (refer to Figure 4-9) to our parser. Before running the parser we have a partial domain model. The newly added domain knowledge is represented in Figure 4-11.

After running the parser we notice that the domain model is updated with new domain knowledge. As mentioned before, *"is_entity(...)"*, *"is_val(...)"* and *"is_action_verb(...)"* represents the relevant Prolog facts for entities, condition values and operations respectively.

The first line of the use case *"Answer Call"* starts with the use case precondition. It is a composite condition and *"Ann phone is offhook"* is a constituent simple condition of the precondition. *"Ann"* is an instance of entity *"caller"*, which is already defined in the domain. Therefore, *[caller, phone]* should be an entity. But it is not defined in the domain. Hence the parser adds it as an entity to the domain. Similarly, as specified in line 2, since the corresponding value of this use case condition is not in the domain, it is also added to the domain. Consider the use case operation, *"Ann records a message"*. *"Ann"* is an instance of entity: *"caller"*. But *"caller"* is not yet defined as an entity in the domain. Therefore, as specified in line 13, it is added as an entity to the domain. Similarly, *"records"* which is an action verb, is also not yet defined as a use case operation in the domain. Therefore, as specified in line 14, the parser adds it to the domain. A domain operation is represented with its associated domain entity. Hence, operation: *"records"* appears to be associated with entity *"caller"*.

```
1. is_entity([caller, phone]).
2. is_val(offhook).
3. is_entity([recipient, phone]).
4. is_val(onhook).
5. is_entity([recipient, phone, indication]).
6. is_val(ringing).
7. is_entity([recipient, phone, ring, count]).
8. is_val(5).
9. is_entity([network, call, status]).
10. is_val(negotiating).
11. is_entity([message, recorder]).
12. is_action_verb([message, recorder], output).
13. is_entity([caller]).
14. is_action_verb([caller], record).
15. is_val(silent).
16. is_entity([network]).
17. is_action_verb([network], terminate).
18. is_action_verb([caller], hang).
19. is_val(reponsive).
```

Figure 4-11: Domain knowledge generated for use case "Call Answer"

4.4.3 Logger

The Logger is used to log the details of the new domain knowledge added to the domain model. It is very useful in identifying the incorrect domain elements that are updated in the domain model. As such, the analyst can rectify them manually by updating the incorrect values of domain elements. An excerpt of the log file generated by the parser is presented in Figure 4-12.

```
Added entity : [caller, phone] for predicate: [caller, phone, is, onhook].
Added entity : [recipient, phone] for predicate: [recipient, phone, is, onhook].
Added entity : [network, call, status] for predicate: [network, call, status, is, idle].
Added entity : [caller] for operation.[caller, takes, the, phone, offhook].
Added verb : take for operation:[caller, takes, the, phone, offhook].
Added entity : [network] for operation: [network, sends, dial, tone].
Added verb : send for operation: [network, sends, dial, tone].
Added verb : dial for operation: [caller, dials, the, number].
Added verb : translate for operation: [network, translates, the, digits].
Added entity : [recipient, line, status] for predicate: [recipient, line, status, is busy].
Added verb : put for operation: [caller, puts, the, phone, offhook].
.....
.....
```

Figure 4-12: Excerpt of the log file generated by the parser

4.5 Case Study: Telephone PABX System

In this section, we present a case study in order to illustrate our approach. The case study concerned is a Telephone PABX system.

For the sake of simplicity, we refer to the switching network which services the telephony system as the *"network"*. The example in this case study is about a telephone conversation that is initiated between a *"caller"* and a *"recipient"*. The *"network"* bills the *"caller"* (i.e. customer) after the telephone conversation. We assume that also a telephone service provided by a *"service provider"* has additional features in addition to the basic telephone service. These additional features include *"answering a call"*, *"returning a call"*, *"forwarding a call"* and *"processing for call privacy"* facilities. The answering machine of the call answer facility that records optional messages for the recipient is called the *"message recorder"*.

In order to initiate a telephone conversation the network should be alive (i.e. up). It is assumed that in the event the network connection is down, there cannot be any form of communication initiated between the caller and the recipient. In the event that there is a network malfunction, necessary steps should be taken to repair the network connection, by informing the respective service provider.

4.5.1 Identification of actors and use cases

Actors of a use case model are derived from nouns of requirements documents, especially from those that are subjects of statements. In addition, use case names are derived from verb phrases of requirements text, acting as actor's predicates. Heuristics are used by analysts to distil and identify actors and uses cases [9]. Typical heuristics used to distil actors include identifying entities that supply information, use the functionality, support the system. In addition, system's external resources and other system's that are needed also serve as heuristics in identifying actors of a system. A common heuristic to distil use cases is identification of tasks for each actor [9].

After reviewing the high level requirements of the Telephone PABX system and applying the heuristics for isolating use cases and actors, we identify *"caller"*, *"recipient"*, *"network"*, *"service provider"* and *"message recorder"* as the actors of the system. They are all nouns of the high level requirements. In addition, we identify *"Make Call"*, *"Answer Call"*, *"Forward Call"*, *"Disconnect Call"*, *"Return Call"* and *"Process Call Privacy"* as the high level abstract use cases from the verb phrases of the requirements text. Thus, we can present a high level UML use case diagram, which illustrates the relationships between abstract use cases and interactions that exists between use cases and actors. We present this diagram in Figure 4-13.

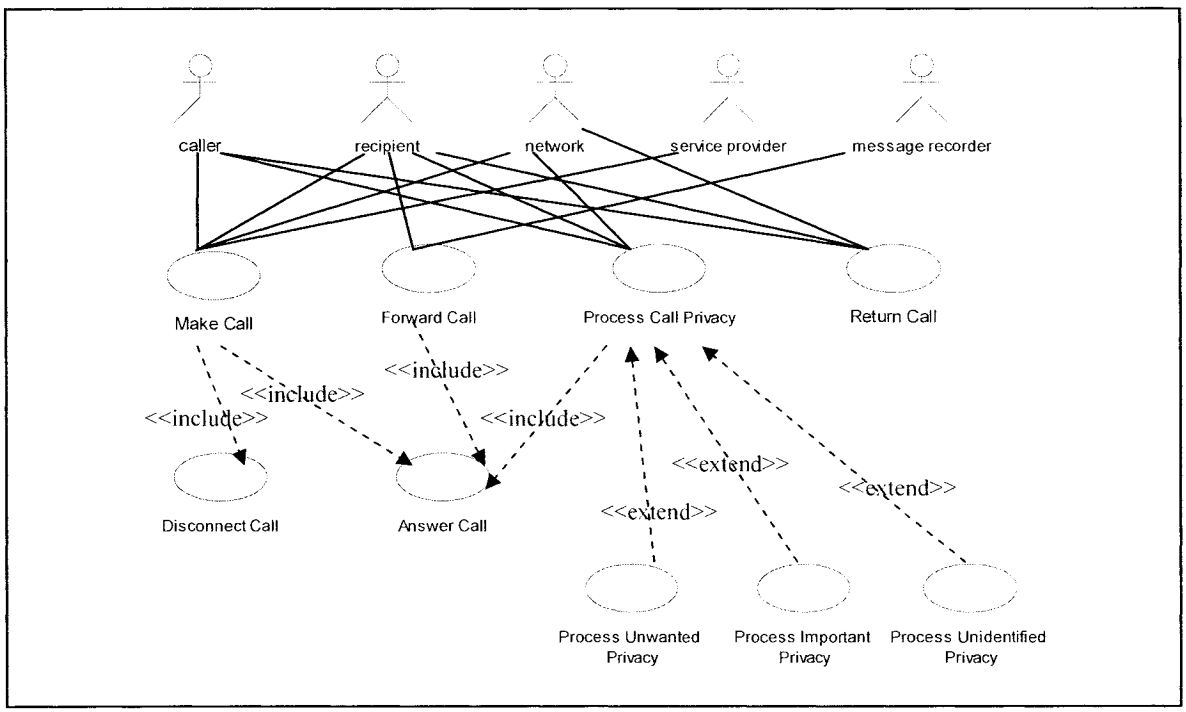


Figure 4-13: Abstract UML use case diagram for the Telephone PABX system

When making a call, if the recipient is not available, the call answering facility can be used by the caller to record a message for the recipient. Similarly, if the recipient has forwarded incoming calls and if the forwarding phone is left unattended, the call answering facility can still be used to record a message for the recipient. It could also be the case that call answering facility can be used and customised for outputting welcome messages and recording messages for different levels of call privacy. Thus, *"Answer Call"* participates with an include relationship with *"Make Call"*, *"Forward Call"* and *"Process Call*

Privacy" use cases. Use case: *"Make Call"* uses *"Disconnect Call"* as an include use case for disconnecting a live call.

Processing of call privacy can be carried out for different levels of privacy. In order to process each level of privacy, we can formulate a separate extension use case. Thus we have three extension use cases: *"Process Unwanted Privacy"*, *"Process Important Privacy"* and *"Process Unidentified Privacy"*. Each of these extension use cases are inserted into the base use case: *"Process Call Privacy"* and participates with extend relationships with the base use case. In addition, extension use cases are executed under specific conditions. For example, *"Process Unwanted Privacy"* use case is executed when condition: *"caller number type is unwanted"* is satisfied.

Figure 4-13 also depicts the interactions that exist between use cases and actors.

After identifying the high level abstract use cases, we focus our attention on refining each use case while preserving its black box view of the system. Before refining each of these abstract use cases, we discuss below its key features.

1. ***Make Call:*** This use case represents setting up a telephone conversation between two parties: caller and recipient. If the recipient is unavailable, call answer facility is used. After the telephone conversation the call is disconnected. Use cases: *"Answer Call"* and *"Disconnect Call"* are included as subordinate use cases (i.e. include use cases) within this use case.

2. ***Answer Call:*** When a caller dials a number to call a recipient, and if the recipient does not answer the phone after a specified number of rings, the *"Answer Call"* feature is initiated. In this case the message recorder will output a welcome message requesting the caller to record a message for the recipient, so that the recipient can check it when he is available.

3. ***Process Call Privacy:*** This feature is provided by the service provider. It determines the action to be taken, which directly depends on the number type of the incoming call. This is considered a privacy feature. Table 4-1 represents the appropriate action to be taken for each caller number type.

Caller number type	*Action taken*
unwanted	Network directs the phone call to the message recorder, which plays a courteous recording to the caller.
friend	Lets the recipient hear the ringing tone in order to proceed with the phone call
family	Lets the recipient hear the ringing tone in order to proceed with the phone call
unidentified	Network interferes to intercept the receiving telephone call
normal	Lets the recipient hear the ringing tone in order to proceed with the phone call

Table 4-1: Actions to be taken for different caller number types

It is from this table that we determine the extension use cases for call privacy. Processing for a normal caller number type is described in *"Process Call Privacy"* use case. We determine extension use cases for processing the other caller number types on the following basis,

- ***Process Unwanted Privacy:*** This is an extension use case that is used for processing of privacy for unwanted caller number types.

- ***Process Important Privacy:*** This is as extension use case that is used for processing of privacy for important caller number types. Important number types are from callers who are family members or friends of the recipient.

- ***Process Unidentified Privacy:*** This is an extension use case that is used for processing of privacy for unidentified caller number types.

4. ***Return Call:*** When a caller dials a telephone number of a recipient and the recipient's line is busy, this option is used to notify the caller whenever the recipient's line becomes free within a specified time interval. If the recipient line becomes free within that specific time interval caller's phone will be rung. When a caller dials a recipient's telephone number, the network attempts to initiate a telephone conversation by switching from the idle state to the negotiating state. If the recipient's line is busy and caller's call return option is activated, the network switches to monitoring state, and starts monitoring the recipient's line for a specific period of time. If the recipient's line becomes free within that time period, the caller will be notified, and the network will switch back to the negotiating state. If the recipient's line is still busy and the specific time interval expires, the system (i.e. network) will cancel the call request of the caller and switches back to the idle state.

5. ***Forward Call:*** When this option is activated the recipient's phone will automatically divert an incoming call from the caller's phone to another phone. The forwarding phone should be configured by the recipient in advance. *"Answer Call"* use case is included as a subordinate use case within this use case.

6. ***Disconnect Call:*** This use case describes the termination of an existing telephone conversation.

After reviewing the features of each of the above use cases of the Telephone PABX system, we need to write them by using a template in a restricted form of language. We used Cockburn's template [5] for writing use cases. In addition, we adhered to Cockburn's guidelines [5] in writing use case descriptions. All the use cases written in textual form are represented in Appendix D. We also present these use cases modified for supporting instances in Appendix E.

We present below how a use case is written textually. For the purpose of our discussion, we consider the use case: *"Make Call"* which is illustrated in Appendix D. Similarly, the rest of the use cases can also be represented textually. An excerpt of the use case: *"Make Call"* of the Telephone PABX system is presented in Figure 4-14.

Title: Make Call
Scope: function
Level: user
Primary Actor: caller
Participants: caller, recipient, network, service provider
Goal: placing a phone call
Precondition: caller phone is onhook AND recipient phone is onhook AND network call status is idle
Postcondition: network connection is up AND recipient phone is onhook and caller phone is onhook and network call status is idle
1. caller takes the phone offhook
2. network sends the dial tone
3. caller dials the call
4. network translates the digits
5. if caller line status is free then network connects the call to recipient phone
6. network sends the ring signal to the recipient phone and caller phone
7. recipient answers the phone
8. if network connection is up then caller talks to the recipient
9. INCLUDE Disconnect Call
4.a. recipient line status is busy
4.a.1. network sends a busy signal to caller
4.a.2. caller puts the phone onhook
4.a.3. Go to Step 1.
6.a. recipient phone ring count is > 5
6.a.1. INCLUDE Answer Call
7.a. network connection is down
7.a.1. caller informs the service provider

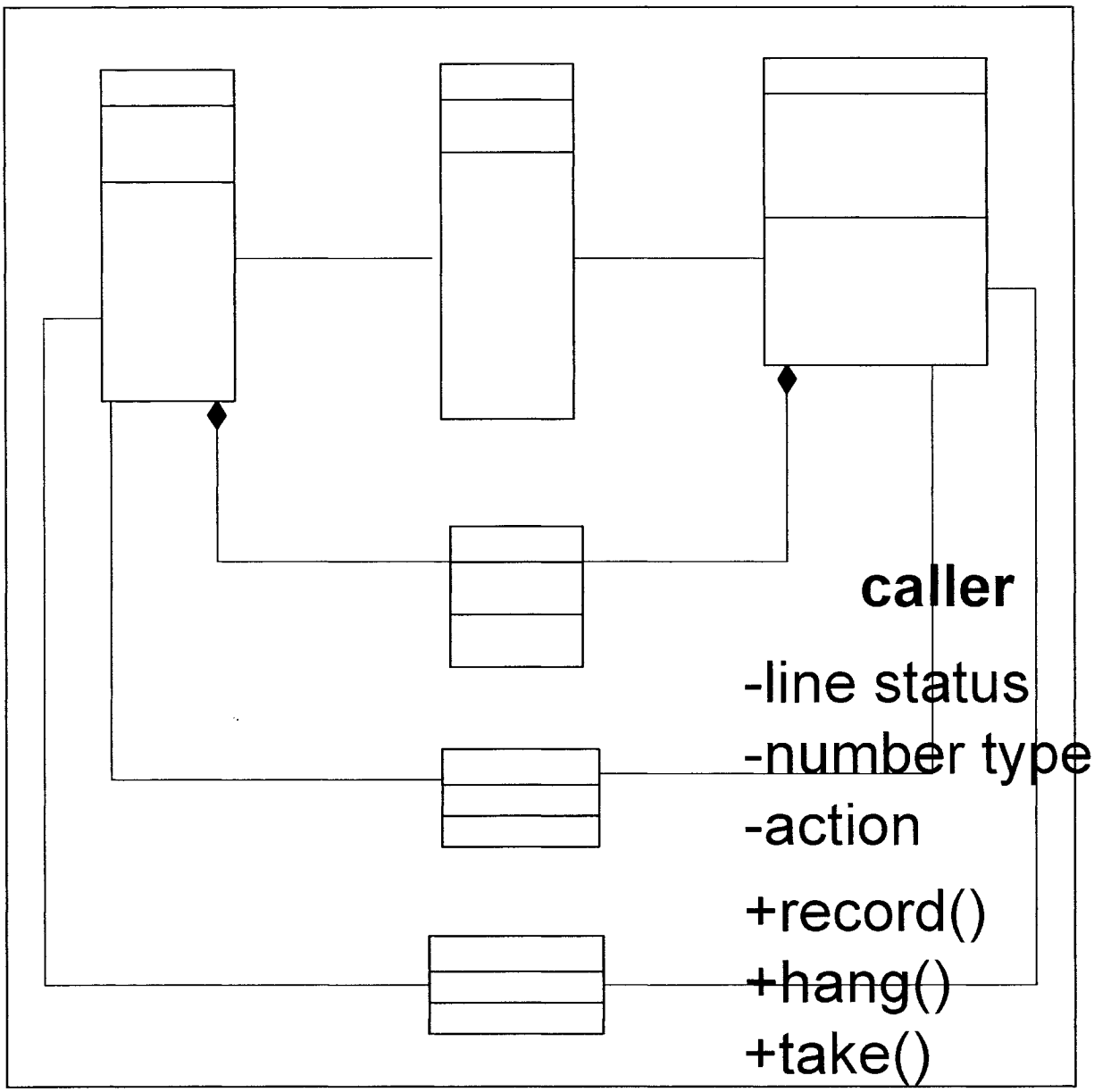
Figure 4-14: Use Case: Make Call of the Telephone PABX system

This use case is written using the Cockburn's use case template [5] specified in Figure 3-7. The title of the use case is the use case name. This use case is written at the user level within the functional scope. The use case is initiated by caller who is the primary actor. The participants of the use case are caller, recipient, network and service provider. The precondition of the use case should be satisfied prior to its execution. The postcondition should be satisfied after the use case is executed. The goal of this use case is placing a telephone call. The main scenario and extension steps of this use case are numbered. Numbering makes it easier to examine the behaviour of a use case. Step numbers clarify the steps and give places to refer to in the extensions section. Numbering use case steps also makes it harder to maintain, but good CASE tools could solve this problem. Main scenario

steps are number sequentially starting from number 1. For numbering use case extension conditions, give the number of the step where the condition would be detected, and put a letter after it (e.g. 4.a.). Extension steps can again be numbered sequentially by prefixing the number of the relevant extension condition (e.g. 4.a.1).

4.5.2 Identifying domain elements

New domain elements are identified using the domain extraction utility of the parser. As mentioned in section 4.4.1, all the use cases specified in Appendix D are applied to the parser. Before applying these use cases to the parser, they need to be formatted in a way that the parser supports. Use cases prepared according to this format is illustrated in Appendix B. When the parser is run and all the above use cases are processed, new domain knowledge identified from the processed use case descriptions are added to the domain model in the form of Prolog facts. These domain elements consist of entities and operations. One limitation of the added domain knowledge is that we have no way of distinguishing between domain sub-entity types. This is because domain sub-entities are represented as a generic entity type in the domain. Therefore, we have to manually identify the domain sub-entity types from the generic domain entities that are extracted. Predicates for operations are specified along with its associated domain entities. A domain model can be constructed and represented in the form of a UML class diagram. The domain model created for the Telephone PABX system is presented in Figure 4-15. Entities: *"caller"* and *"recipient"* form an aggregate relationship with entity: *"phone"*. The cardinality of association relationships are also marked at each end of the appropriate relationship. The parser detects *"caller"*, *"recipient"*, *"network"*, *"service provider"* and *"message recorder"* as the main entities of the system. They are also the actors of the system.



4.6 Performance evaluation of the parser

Database manipulations of a corpus could have a significant impact on the parser which uses such a database. The restricted natural language parser used for parsing use cases makes calls to the WordNet database in order to identify word forms of words in use case descriptions. Making calls to a database is costly as getting a connection from a database itself is expensive. Therefore, we focussed on evaluating the impact on WordNet database operations on the overall performance of the parser. In this section we will explain our performance

evaluation carried out for the restricted natural language parser, which is introduced in section 4.4.1.

First, we calculate the average times consumed for parsing and database operations for each use case of the Telephone PABX system case study. These average values are computed by taking averages for 20 data values for each use case. Using these average values we compute the average percentage time consumed for database operations for each use case. These computed values are presented in Table 4-2. We notice that the average percentage values for database operations are dispersed. Times taken to lookup word forms (such as nouns and verbs) from the WordNet dictionary for different words in use case descriptions are independent of each other. Even the time consumed for querying the WordNet database for getting the word form for different words of the same word category (i.e. nouns, verbs) may vary drastically. In addition, the number of words queried for determining word forms for a particular use case description may vary from another use case description. These are some of the reasons that we noticed for not having a uniform distribution on average percentage time consumed for database operations. However, the important observation that we noticed from this performance evaluation is that having a corpus that is integrated with a database in order to support parsing of use cases is costly. This is because the database operations are expensive. *"Forward Call"* use case takes an approximate average percentage of 52.054% for performing WordNet database operations. This is a significant and a considerable fraction of the overall parsing time.

Use Case	Average time to parse (in seconds)	Average time consumed for database operations (in seconds)	Percentage of average time consumed for database operations
Make Call	9.324	2.53	27.134%
Discount Call	1.803	0.86	47.698%
Answer Call	3.866	1.87	48.370%
Return Call	4.787	1.58	33.066%
Forward Call	2.555	1.33	52.054%
Process Call Privacy	12.438	4.05	32.326%
Process Unwanted Privacy	2.434	0.23	9.449%
Process Important Privacy	3.185	0.30	9.412%
Process Unidentified Privacy	1.233	0.34	27.575%

Table 4-2: Average percentage time taken for database operations by use cases

4.7 Case Study: Elevator System

In section 4.5, we introduced a case study of a Telephone PABX system. It is a case study that is created by us. Hence it exhibits the limitation of not being able to validate its accuracy with an existing source. To overcome this limitation we introduce an "Elevator System" case study which is extracted from [35]. The author also provides the static model of the problem domain for this case study. Thus we have the advantage of applying the use cases of this case study to our parser, in order to generate the corresponding domain, so that we can validate it with the author's domain.

This case study consists of an Elevator Control System that controls one or more elevators. The system has to schedule elevators to respond to requests from users at various floors and control the motions of the elevators between floors.

4.7.1 Problem Description

The high level problem description of the Elevator System case study is mentioned below.

For each elevator, there are,

- ***A set of elevator buttons:*** A user presses a button to select a destination.

- ***A corresponding set of elevator lamps:*** Indicate the floors to be visited by the elevator.
- ***An elevator motor:*** Controlled by commands to move up, move down, and stop.
- ***An elevator door:*** Controlled by commands to open and close the door.

For each floor, there are,

- ***Up and down floor buttons:*** A user presses a button to request an elevator.
- ***A corresponding pair of floor lamps:*** Indicate the directions that have been requested.

At each floor, and for each elevator, there is a pair of direction lamps to indicate whether an arriving elevator is heading in the up or down direction. For the top and bottom floors, there is only one floor button, one floor lamp, and (for each elevator) one direction lamp. There is also an arrival sensor at each floor in each elevator shaft to detect the arrival of an elevator at the floor.

4.7.2 Gomaa's use case model for the Elevator System case study

The Elevator Control System has two actors: one representing the *"elevator user"* who wishes to use the elevator and the second representing the *"arrival sensor"*. The "elevator user" interacts with the system via the elevator buttons and the floor buttons. The "elevator user" actor initiates two use cases, which can be identified by the problem definition.

Select Destination: The user in the elevator presses an up or down elevator button to select a destination floor to which to move.
Request Elevator: The user at floor presses an up or down floor button to request an elevator.

Apart from the above use cases, common behaviour of the above use cases can be factored out into the following use cases. These use cases can form *"include relationships"* with its base use cases (i.e. *"Select Destination"* and *"Request Elevator"*).

Despatch Elevator: The elevator is despatched in response to a user request.

Stop Elevator: The stopping of an elevator.

The UML use case model for the Elevator System case study is represented in Figure 4-16.

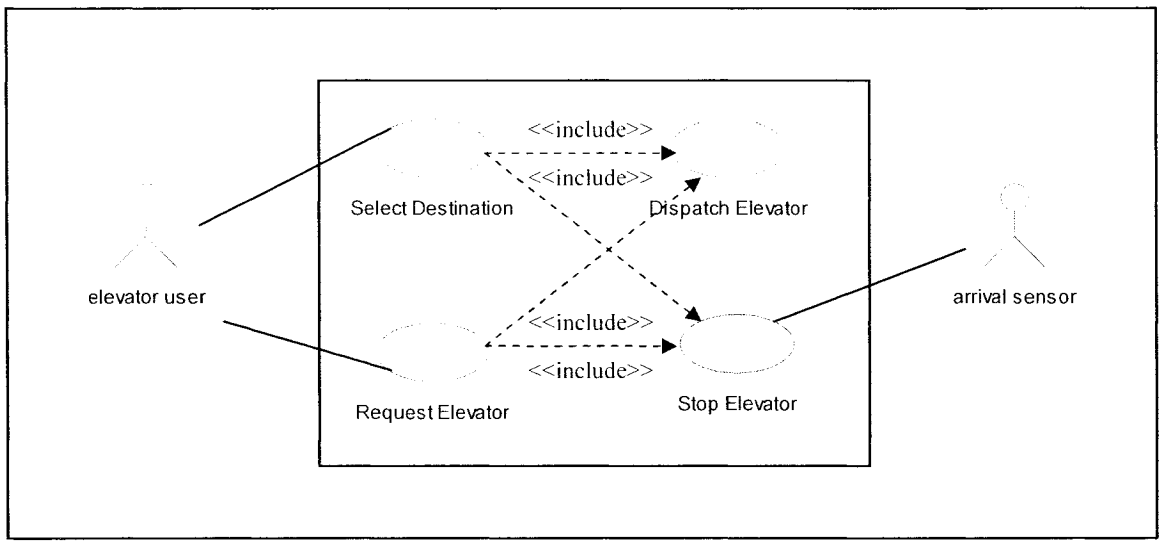


Figure 4-16: Use Case model of the "Elevator System" case study

According the use case model depicted in Figure 4-16, both use cases: *"Select Destination"* and *"Request Elevator"* participates with *"include relationships"* with use cases: *"Dispatch Elevator"* and *"Stop Elevator"*. The system interacts with actors: "elevator user" and "arrival sensor".

The following sub-sections describe Gomaa's use cases of the Elevator System case study in more detail.

4.7.2.1 Use Case: Stop Elevator

Actor: Arrival sensor

Precondition: Elevator is moving

Description:

As the elevator moves between floors, the arrival sensor detects that the elevator is approaching a floor and notifies the system. The system checks whether the elevator should

stop at this floor. If so, the system commands the motor to stop. When the elevator has stopped, the system commands the elevator door to open.

Alternative: The elevator is not required to stop at this floor and so continues past the floor.

Postcondition: Elevator has stopped at floor, with door open.

4.7.2.2 Use Case: Dispatch Elevator

Precondition: Elevator has at least one floor to visit

Description:

The system determines in which direction the system should move in order to service the next request. The system commands the elevator door to close. When the door has closed, the system commands the motor to start moving the elevator, either up or down.

Alternative: If the elevator is at floor and there is no new floor to move to, the elevator stays at the current floor, with the door open.

Postcondition: Elevator is moving in the commanded direction.

4.7.2.3 Use Case: Select Destination

Actor: Elevator user

Precondition: User is in the elevator

Description:

1. User presses an up (or down) elevator button. The elevator button sensor sends the elevator button request to the system, identifying the destination floor the user wishes to visit.
2. The new request is added to the list of floors to visit. If the elevator is stationary, include *"Dispatch Elevator"* use case.
3. Include *"Stop Elevator"* use case.

4. If there are other outstanding requests, the elevator visits these floors on the way to the floor requested by the user, following the above sequence of dispatching and stopping. Eventually, the elevator arrives at the destination floor selected by the user.

Alternative: User presses down elevator button to move down. System response is the same as in the main sequence.

Postcondition: Elevator has arrived at the destination floor selected by the user.

4.7.2.4 Request Elevator

Actor: Elevator user

Precondition: User is at a floor and wants an elevator

Description:

1. User presses an up floor button. The floor button sensor sends the user request to the system, identifying the floor number.
2. The system selects an elevator to visit this floor. The new request is added to the list of floors to visit. If the elevator is stationary, then include *"Dispatch Elevator"* use case.
3. Include *"Stop Elevator"* use case.
4. If there are outstanding requests, the elevator visits these floors on the way to the floor requested by the user, following the above sequence of dispatching and stopping. Eventually, the elevator arrives at the floor in response to the user request.

Alternative: User presses down floor button to move down. System response is the same as for the main sequence.

Postcondition: Elevator has arrived at the floor in response to user request.

4.7.3 Gomaa's domain model for the Elevator System case study

Static relationships in the Elevator Control System are captured using a domain model. The domain model represented in Figure 4-17 shows the real world entities in the problem domain.

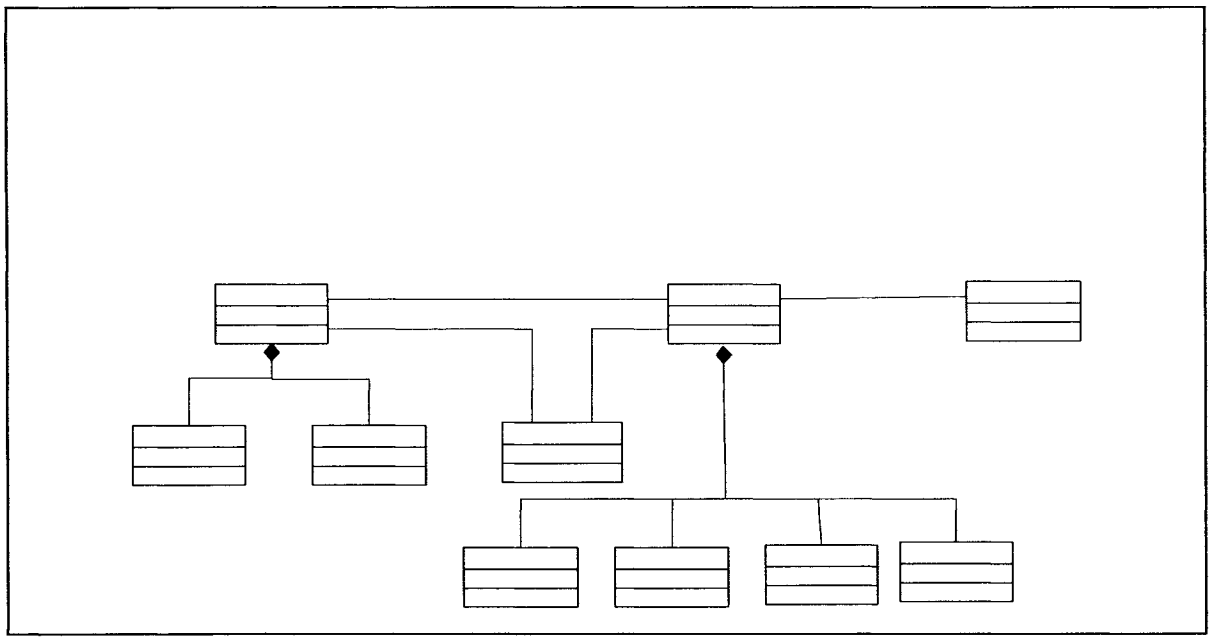


Figure 4-17: Gomma's domain model for the Elevator System case study

The *"elevator"* is a composite entity composed of one *"motor"*, one *"door"*, n *"elevator buttons"* and n *"elevator lamps"*. *"floor"* is also a composite entity, composed of *"floor button"* and *"floor lamp"*. The *"elevator"* entity also has associations with the *"arrival sensor"* entity, which notifies it of imminent arrival at a floor, and the *"direction lamp"* entity, which it switches on and off. The *"direction lamp"* entity has an association with the *"floor"* entity.

4.7.4 Validation of Elevator System case study

The natural language parser that is introduced in section 4.4 accepts input use cases when it is only written according to Cockburn's template. Therefore, Gomaa's use cases specified in section 4.7.2 were rewritten, such that it adheres to the Cockburn's template. These use cases are represented in Appendix F. Thereafter, these use cases are modified to support instances, which are represented in Appendix G.

Subsequently, the Cockburn form of use cases of the Elevator System case study are rewritten, such that it could be provided as input to the Prolog implementation of the parser. These use cases are represented in Appendix H (Similarly, use cases that support instances,

are rewritten to provide as input to the Prolog implementation of the parser, and is presented in Appendix I).

Part of the domain model extracted, after applying the input use cases of the Elevator System to the parser is presented in Figure 4-18.

```
.....
is_entity([system]).
is_entity([elevator]).
is_entity([elevator, door]).
is_entity([elevator, user]).
is_entity([elevator, button]).
is_entity([elevator, floor]).
is_entity([arrival, sensor]).
is_entity([floor]).
is_entity([floor, button, sensor]).
.....
.....
is_instance(elevator, 'ElevatorA').
is_instance(elevator, 'ElevatorB').
is_instance(elevator, 'ElevatorC').
.....
is_action_verb([system], select).
is_action_verb([elevator], arrive).
.....
.....
is_val(stationary).
is_val(floor).
.....
```

Figure 4-18: Part of the generated domain model of the Elevator System case study

The advantage of our extracted domain is that it identifies specific details of domain entities (such as domain operation, possible values, etc.) when compared with the Gomaa's domain model. For example, predicate: *"is_val(stationary)"* represents a domain possible value: *"stationary"*, and predicate: *"is_action_verb([elevator], arrive)"* denotes that *"arrive"* is a domain operation of entity: *"elevator"*. Predicate: *"is_entity"* represents an entity of the system. However, it does not provide the facility to distinguish between sub-entities of an entity. Therefore, sub-entities need to be identified manually. Domain model for our extracted domain is represented in Figure 4-19.

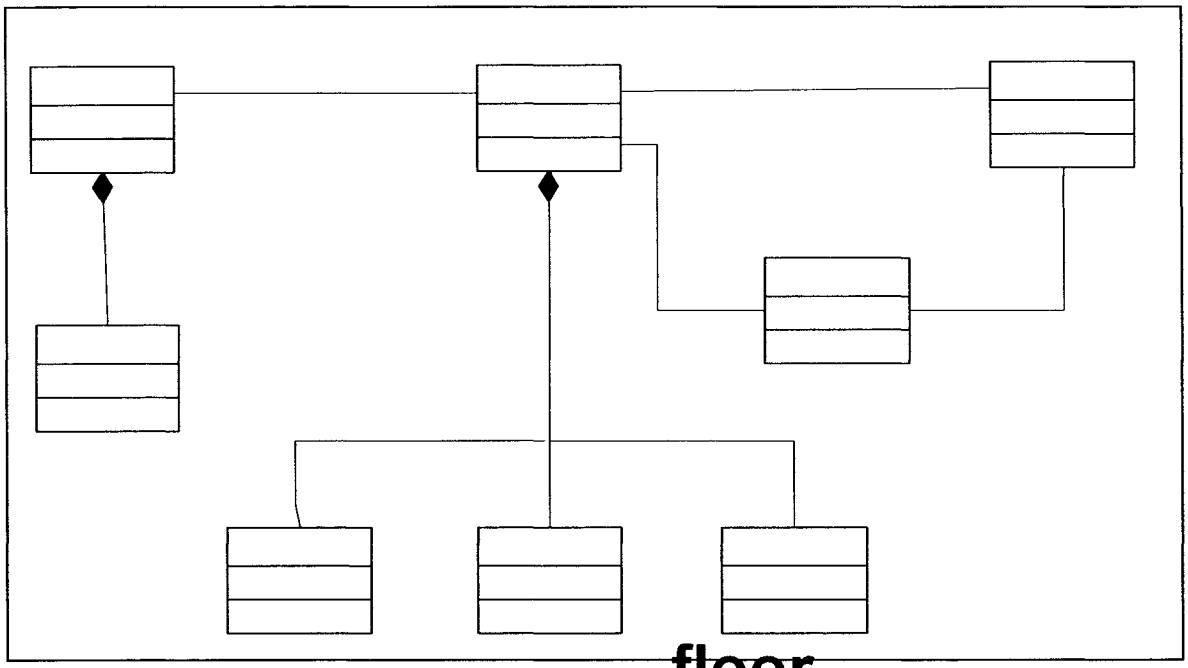


Figure 4-19: Domain model generated by the parser for the Elevator System case study

It can be determined from Figure 4-19 that *"elevator"* is a composite entity composed of n *"elevator users"*, n *"elevator buttons"* and one *"elevator door"*. *"floor"* is a composite entity, composed of *"floor button sensor"*. The *"elevator"* entity also has associations with *"arrival sensor"* entity.

When comparing Figure 4-17 with Figure 4-19, it can be determined that the domain extracted using the parser does not extract *"floor button"*, *"floor lamp"*, *"direction lamp"* and *"motor"* as domain entities. On the other hand, the domain extracted by the parser has identified domain entities: *"system"*, *"user"* and *"floor button sensor"*, which are not determined as domain entities in Gomaa's domain model. These discrepancies are due to the fact that in Gomaa's approach, domain entities are generated intuitively, using the problem definition. Conversely, by using the natural language parser, domain entities are generated by employing an automatic technique, which is based on the syntax that is used to write use case descriptions. However, we do not observe major differences in the domains that are generated using the above two approaches.

4.8 Why can't we use a corpus as the domain model?

Since, we used the WordNet dictionary to extract domain entities and operation, it seems important to ascertain why we can't use a corpus such as WordNet as the domain model.

As discussed in section 4.3.1, if we merely use a corpus as the domain, we may attempt to use different nouns to refer to the same entity in different use cases. This leads to a problem of inconsistency. But by having a well defined domain model with concrete and consistent values for entities we can overcome this problem. Further, if we use a corpus as the domain model we would have redundant elements that are not part of the domain.

Another drawback of using a corpus such as WordNet database as a domain model is the performance reduction caused when executing database operations. This fact is apparent by reviewing the performance evaluation mentioned in section 4.6. If we have a specific and separate domain model, we can evade this performance problem to a greater extent.

Finally, derivation of a domain model is an essential part of requirements engineering.

4.9 Limitations of the Parser and suggestions for improvement

We noticed in section 4.5.2 that entities of our grammar are represented in a generic form. Therefore we do not have any way of distinguishing between sub-entity types of these entities. Having a technique to distinguish between sub-entity types is important when creating a domain model. Otherwise, we need to manually identify the specific sub-entity types of the domain model.

Another limitation is that our grammar does not support processing of requirements descriptions that contain pronouns. Providing support for pronoun resolution in requirements specification is a worthy suggestion for improving our grammar. Pronoun resolution is a separate research area in Discourse Analysis [38].

4.10 *Chapter Summary*

Use cases can be written using restricted form of grammar to improve consistency and reduce ambiguity. In chapter 3, we reviewed several guidelines in writing restricted form of use cases. CREWS guidelines [10] and UCEd guidelines [7, 33, 34] defines a semi-formal types of grammar for writing use case content. Further, [9] and [8] defines natural language friendly techniques in writing use case descriptions. However, these grammars do not provide the functionality for supporting instances. We discussed that instances can be used to improve the expressive power of a natural language used for writing use cases. In addition, none of the grammars referred above do not possess a fully automatic mechanism of adding new domain knowledge to the domain model.

One of the key features of our grammatical approach is the support for instances for use case modeling. This improves the expressive power of requirements elicitation.

Domain knowledge is required for parsing use case descriptions and we need to have some mechanism in order to identify the missing elements in the domain. Our grammatical approach supports parsing use case descriptions and adding new domain knowledge to the domain model. In our approach, nouns and verbs are used to identify missing domain elements and add them into the domain.

However, one of the limitations in our grammar is that it has no mechanism in identifying the sub-entity types of entities. Therefore, we have to identify them using manual inspection.

We use the WordNet dictionary for identifying new domain elements. But we should not use the WordNet dictionary as the domain model. The reason is that if WordNet is used as the domain model it could cause redundancy, ambiguity and performance reduction when parsing use cases.

Chapter 5 - Use Case Editor (UCEd) Framework

One limitation of our grammatical approach mentioned in Chapter 4, is that it does not provide a mechanism to specifically identify domain sub entity types. UCEd is an approach that is rooted in UML. UML [22] is a model driven, extensible language for modeling system requirements. It is standardized by unifying the best features of different modeling languages. Hence, UML has become an industry standard. UML stereotypes have the behaviour of being extensible. Using the extensible property of stereotypes, different types of sub-entities can be defined. Eventually, these sub-entity types extend from a single generic entity type.

UCEd [7, 33, 34] defines a grammar that supports domain sub-entity types. Sub-entity types supported by UCEd are Concepts, System Concepts, Attributes and Aggregates. Support for different types of domain sub-entity types by UCEd is provided using UML stereotypes. Because of this feature of the UCEd grammar, we integrate it with the domain extraction utility of the natural language parser. Selecting UCEd for our work prevents re-inventing the wheel. Integration of the domain extraction utility with UCEd assists us in eliminating one of the limitations of the natural language parser. That is UCEd can be used to distinguish domain sub-entity types when extracting a domain model. This means that UCEd is used to overcome the limitations of the parser which is introduced in Chapter 4. In addition, the enhancements proposed for these limitations will also solve some of the limitations of the UCEd framework. However, our approach could be integrated to frameworks other than UCEd.

At the outset we mentioned that our approach focuses on detailed black box view of use cases. Since we are working only on the black box view of use cases, we are not concerned on the inter-component interactions that exist between system components. Use case modeling and state model generation in UCEd also follows a black box view. UCEd generates a single state machine for a whole system.

We start this chapter my providing a high level overview of the UCEd framework, followed by the activity process supported by UCEd. Subsequently, we discuss each module of UCEd in detail.

5.1 High level overview of UCEd

Requirements are learnt from customers and requirements specification is formalized in a way that is understood by system analysts. Formal requirements documents are difficult for customers to comprehend. Therefore, there is a gap between the customers and analysts view of requirements. This gap is mainly attributed to the fact that requirement engineering being carried out using manual techniques. Thus, an automatic technique to minimize the gap for requirements elicitation is required. UCEd is one such technique.

UCEd is a tool for elicitation, formulation, composition and simulation of use cases. The use cases in UCEd are written using a restricted form of natural language. The tool takes a set of use cases and generates an executable specification, integrating the partial behaviours of all the use cases. A domain model is used for syntactic analysis of use cases and specification generation. UCEd is a hybrid of a model driven (i.e. UML) approach and a system that supports natural language textual requirements. The use cases created using the tool follow the UML specification. An UCEd domain model can be represented as a UML class diagram.

Domain entities in the environment are represented using Concepts. The system elements are denoted by System Concepts. A system may consist of one or more components. However, all interacting components are considered in one black box. Let's consider the example presented in Figure 5-1. Use Case: *"Make Secured Call"* includes three system reactions. Each of these system reactions is executed by a separate system component, and they are triggered after executing the use case step: *"caller takes the phone offhook"*. All the system components: *SystemCA*, *SystemCB* and *SystemCC* can be viewed as contained in a single black box.

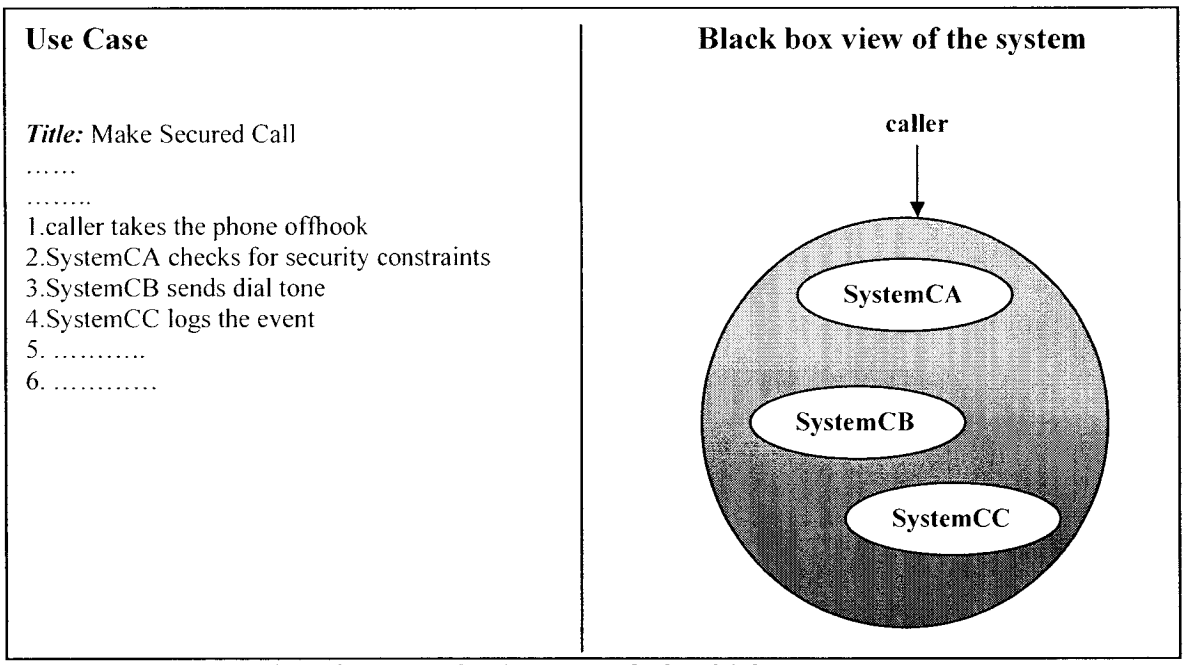


Figure 5-1: Black box view of a system that is composed of multiple system components

5.2 Activity supported by UCEd

Requirements Engineering with UCEd starts with an early view of the requirements that consists of a domain model and a use case model. These requirements may consist of a *"rough"* domain model and a set of use cases. Subsequently, a high level domain model and a use case model together with a state model specification of the system are produced. Refining a rough set of requirements to a high level concrete set of requirements requires manually updating the domain and use case models for new elements that are added. A state model is generated by composing the partial behaviour of all the use cases [7, 34]. Use case composition process is driven by information in the domain model. The state machine created by UCEd can be used as a prototype for use case simulation [7].

The UCEd process is shown in Figure 5-2. The boxes represent activities, while the arrows between them are the data that are exchanged between the activities. For the purpose of explaining the UCEd framework, we use the Case Study defined in section 4.5.

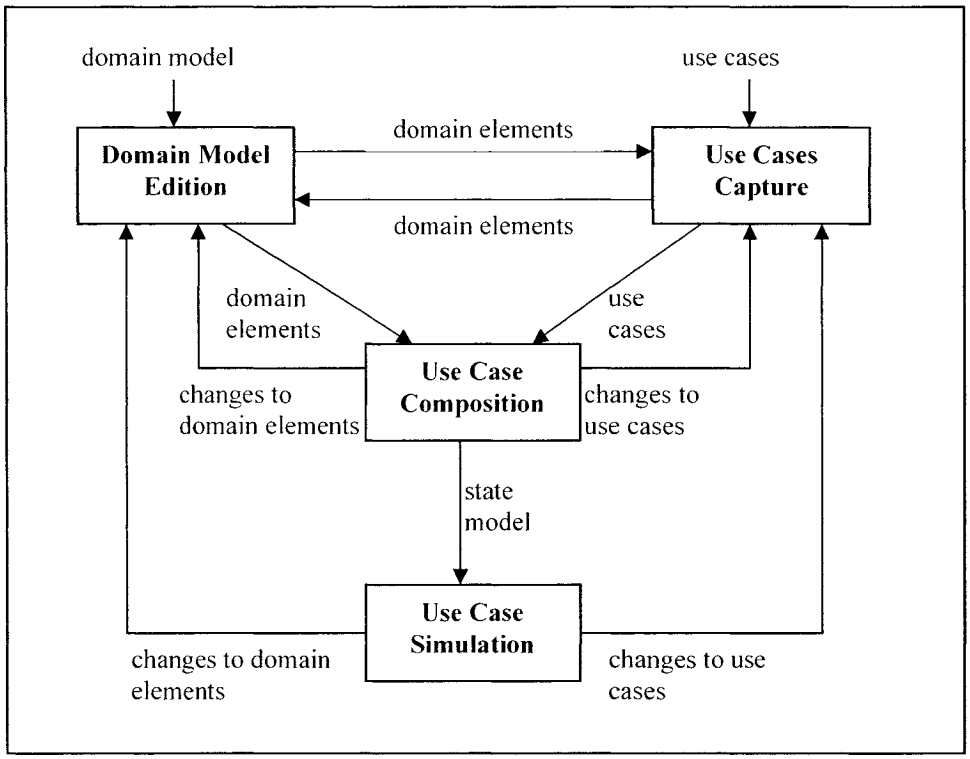


Figure 5-2: UCEd Requirements Engineering process

The next few sections explain the activities supported by the UCEd framework.

5.3 Use case model

Use cases are captured using a Use Case Edition module of UCEd. It is an interface that supports edition of use cases and its descriptions in a field oriented editor, using the template derived by Cockburn [5]. The advantage of such an editor is that the user can easily create use cases without worrying about delimiting the different parts of the use cases. UCEd also checks the use cases and its corresponding domain model for inconsistencies or omissions. The field-oriented editor of UCEd that assists in capturing use cases is represented in Figure 5-3. It shows the *"Make Call"* use case which is illustrated in the Case Study.

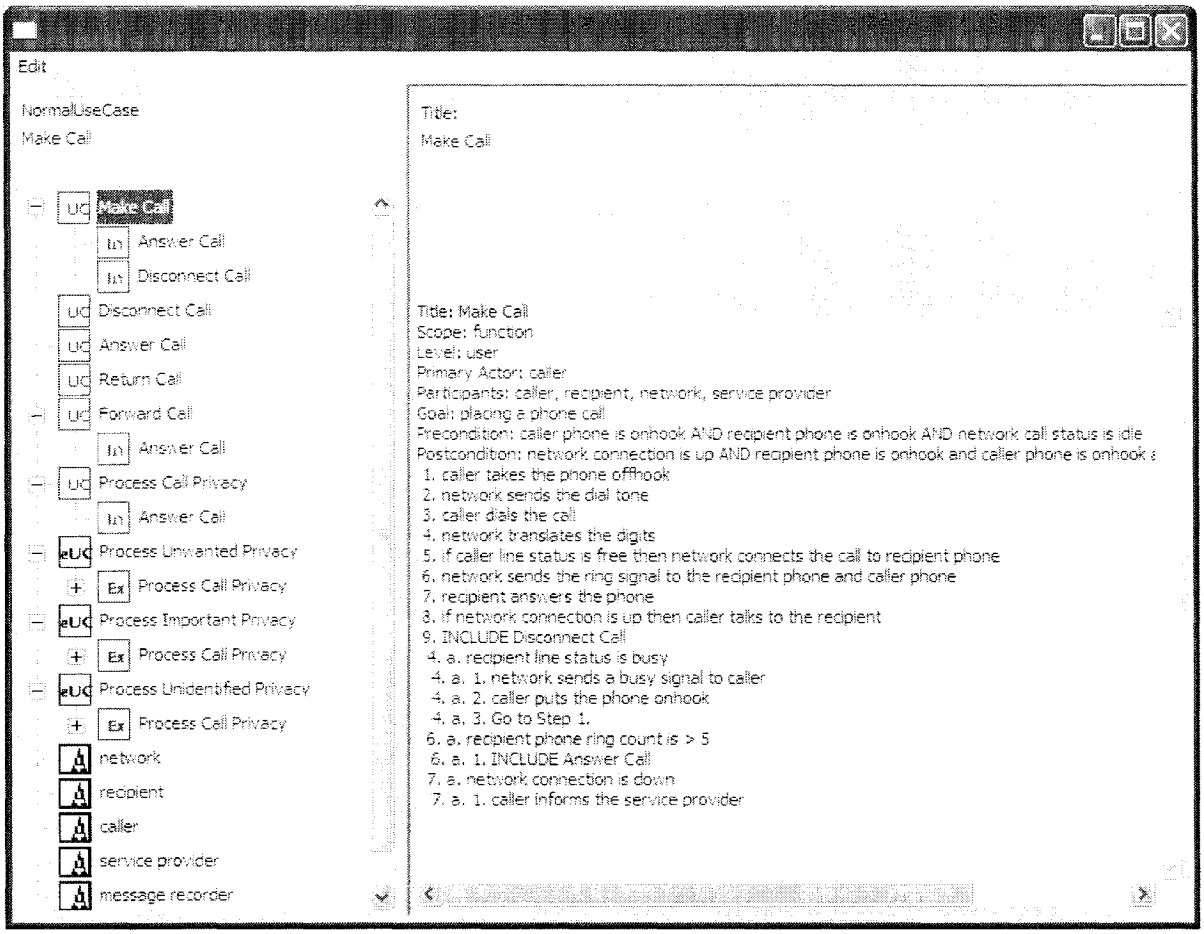


Figure 5-3: Use Case Editor of UCEd

The use case precondition must be satisfied prior to the execution of the use case. The use case represented in Figure 5-3 has 9 use case steps. Step 9 is a use case inclusion step. Use case: *"Make Call"* also has 3 use case alternatives: 4.a., 6.a., 7.a., which are executed under specific extension conditions. Extension use case step 4.a.3 is a *"branching statement"*, which passes the control back to use case step 1. Other use case steps of this use case are *"concept operations"*. Each use case step operation has a precondition and a postcondition. Operation preconditions and postconditions are useful in ascertaining the state conditions that prevails at the end of each step. Postcondition is a condition that must be satisfied after the successful execution of a use case. The state condition evaluated at the last use case step of the success scenario of the use case should be identical to the use case postcondition.

5.4 Domain model

A domain model is a high-level class model that captures domain concepts and their relationships. Domain concepts are entities of a particular system. The domain model includes the system under consideration as a black box and other entities in the environment with which the system will be interacting. The domain model is expressed in the form of a UML class diagram that is extended with stereotypes. An UCEd domain concept is an instance of a stereotype of UML meta-class: *<<concept>>*. There are also constraints to *<<concept>>*. One such case is that an attribute of a concept can be defined as an instance of a stereotype called *<<conceptAttribute>>*. The *<<conceptAttribute>>* stereotype extends UML meta-class *Attribute*. The Domain Editor of UCEd provides an area where domain entities can be captured when defining the domain model. Figure 5-4 shows the view of the Domain Editor.

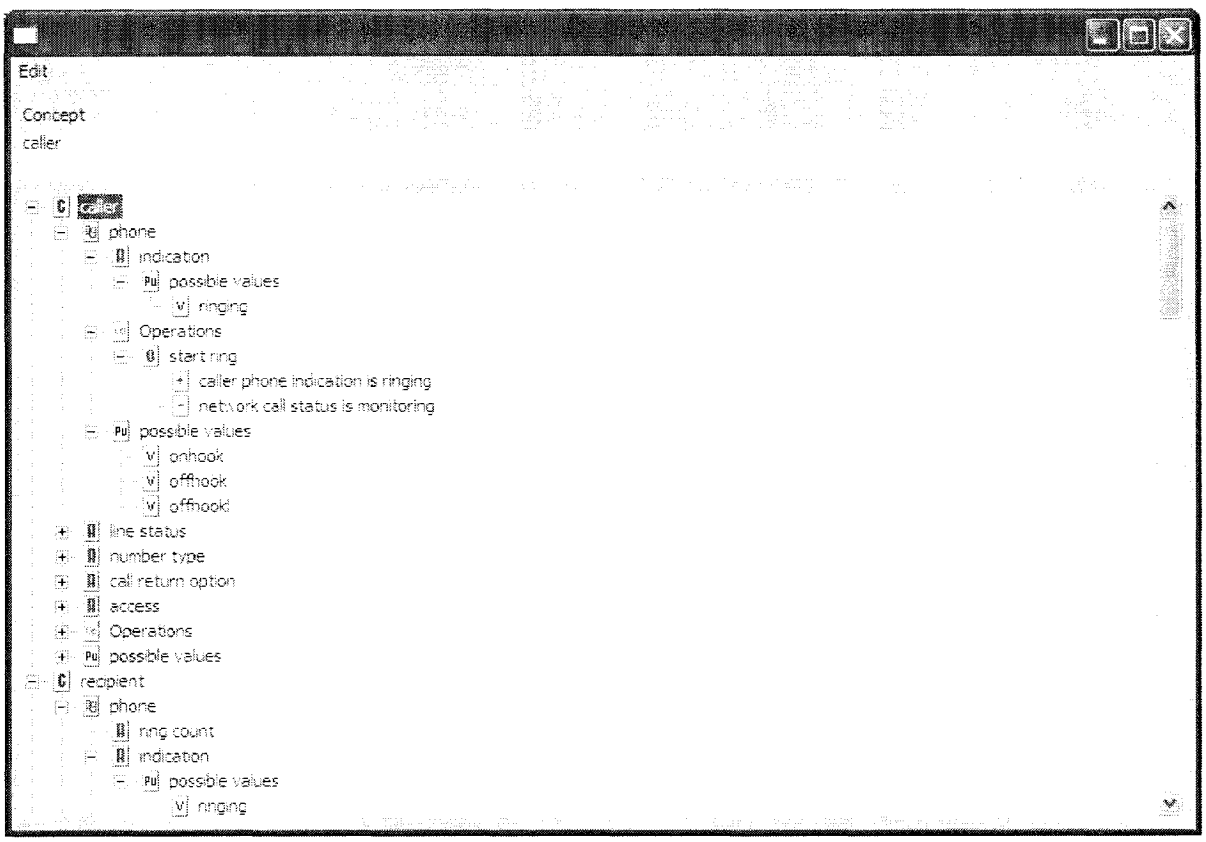


Figure 5-4: Domain Editor of UCEd

Figure 5-4 contains a graphical form of a domain model for the Case Study mentioned in section 4.5. A domain model is used for use case analysis and serves as a lexicon for natural language analysis of use cases. Apart from domain entities, it defines domain operations and postconditions that are used for state model generation.

State changes happen during operation execution. During such an execution, state values for state conditions may be added and removed. An operation withdrawn condition is a set of predicates that are removed after the execution of an operation. An added condition is a set of predicates that becomes true after the execution of an operation. These added and withdrawn conditions are operation postconditions. A special case of an added condition is that of a composite disjunction condition. For example, in our Case Study, the domain model defines: *"recipient line status is free OR recipient line status is busy"* as an added condition of the operation: *"monitor recipient line status"* for the system concept: *"network"*. This means that after the execution of this operation, there will be two possible resulting states that are determined by the predicates: *"recipient line status is free"* and *"recipient line status is busy"*. A withdrawn condition may be specified as an *ANY condition*. When removing conditions it may be useful to refer to all the conditions on an entity, as it is not feasible to separately list all possible individual conditions that must be removed. ANY conditions are useful for this purpose. A *wildcard* "*" may be used to refer to remove all conditions on an entity together with those of its sub-entities. An example of a wildcard ANY condition is *"ANY ON recipient*"*. In this example all conditions of sub-entities associated with *"recipient"* such as *"phone"* will be removed together with those of the entity itself.

5.5 State model

Use case composition is incremental. Each use case is analyzed and its partial behaviour is merged in forming the state model for all use cases that are composed. A state model generation algorithm is used for use case composition [7, 34]. The state model specification derived contains all partial behaviours of use cases. During this process any inconsistency found with the domain and use case models will be detected and reported.

UCEd generates hierarchical type of finite state machines from use cases, based on a predefined algorithm. This algorithm is based on the definition of states by predicate sets and specification of operation effects such as withdrawn and added conditions on predicate sets. Therefore, we state below the basis of functionality of the algorithm,

- As we mentioned earlier operations of domain entities have withdrawn and added conditions, which are expressed in the form of predicates.
- Characteristic predicates are used to describe the states in the state model. Such predicates that are used are formulated using domain model entities. It is said that two states are identical if they have the same characteristic predicates.
- It is said that a state s_b is a sub state of state s_a (i.e. its super state), if its characteristic predicates include those of s_a in a logical sense. In addition, any transition going from a super state s also applies to all sub states of s. Note that a particular state may have more than one super state.

A state transition graph is augmented with states and transitions for each use case, such that the behaviour sequences corresponding to each use case is included as a state transition sequence in the state transition graph. States are determined by withdrawn and added conditions. Let's consider the two sets of predicates: C1 and C2. Suppose "-" is an operator, then C1-C2 is a set obtained by removing all predicates of C2 from C1. Similarly, if "+" is an operator, then C1+C2 is a set obtained by adding all the predicates in C2 to C1. "Given a state s, such that pred(s) are the characteristic predicates of s, the execution of operation op with added condition add_conds(op) and withdrawn condition withdr_conds(op) produces a state s' such that, pred(s') = (pred(s) – withdr_conds(op)) + add_conds(op)" [34].

The state model generation algorithm supports overlapping and connected use cases. Using the said algorithm we can derive an integrated state model such that the resulting state model does not contain any redundant states.

State models generated in UCEd can be described using UML statecharts. Statecharts represent reactive behaviour model descriptions where transitions represent complete

interactions. Each transition consists of a trigger (an operation performed by an actor) and a system reaction resulting from the event. In the state model that we discussed, a transition is a trigger event or a system reaction. Generated state machines include internal states with all outgoing direct transitions triggered by system reactions. These internal states are useful in identifying the common parts of independent use cases. These state machines are transformed into statecharts by merging the states and transitions such that the internal states are removed. It is important to note that a deterministic statechart cannot be generated from a state machine where more than one reaction follows a triggering event or a guard and such a situation is considered to be an inconsistency.

Figure 5-5 shows a state chart that is extracted from use case: *"Return Call"*.

```
**** SCSTATES ****
 SState:1[network call status is negotiating, recipient line status is busy]
 SState:2 ChoicePoint
 SState:3[recipient line status is free, network call status is monitoring, caller call return option is ON]
 SState:4[recipient line status is busy, network call status is monitoring, caller call return option is ON]
 SState:5[recipient line status is busy, caller call return option is ON]
 SState:6[recipient line status is free, caller phone indication is ringing, caller call return option is ON]
 SState:7[network call status is oncall, caller call return option is ON]
**** SCTRANSITIONS ***
  1---select call return option/monitor recipient line status-->2
  2---[recipient line status is busy]/-->4
  2---[recipient line status is free]/-->3
  3---start ring/-->6
  4---TIMEOUT(Timer4:30.0 minute)/cancel call request-->5
  6---answer phone and talks/-->7
```

Figure 5-5: Statechart obtained from the state machine generated from use case: "Return Call"

The statechart represented in Figure 5-5, describes transitions that are specified using the UML format: *[guard] trigger/reaction(s)*. A *Guard* is a condition. *Trigger* includes operations from the environment and timeout events. A *Reaction* is an operation executed by the system. Let's consider the statechart transition: *"1---select call return option/monitor recipient line status-->2*. It specifies that *"select call return option"* is a trigger and *"monitor recipient line status"* is a system reaction. During the transition, the system moves from state 1 to state 2. An interesting feature of a statechart is its representation of transitions at choice points. In the above example, a choice point at state 2 is possible due to the

presence of the disjunction condition: *"recipient line is free OR recipient line is busy"*. During any transition from state 2, only one constituent simple condition of the disjunction condition is satisfied. Hence, state 2 can be treated as a choice point.

5.6 Simulator

State machines generated are used as prototypes to validate the original use cases in order to uncover any possible interactions between them. Use Cases are validated in order to verify whether they are consistent with the corresponding domain model. By uncovering any possible interactions in use cases, we may be able to find possible undiscovered interaction in those use cases. This is useful in finding the missing or undetected requirements of a system. UCEd Use Case Simulator provides a graphical interface that could be used to execute the specification generated by the state machines.

In real software projects a drawback with simulation is the challenge in developing an appropriate prototype. Creating a prototype is not only error prone but is also very costly and a tedious process. Fortunately in UCEd, statecharts have the property of being executable and used as prototypes. Therefore we have the opportunity of applying efficient simulation. The Use Case Simulator component of UCEd facilitates simulation using the generated statecharts as prototypes. The objective of simulation is to reproduce the reactive behaviour described in the use cases and exhibit the global behaviour resulting from their integration.

A view of UCEd Simulator tool is presented in Figure 5-6. Simulation in UCEd is done in an interactive manner where the user will be prompted to select the appropriate actor event required for a simulation transition to proceed with multiple paths of transitions.

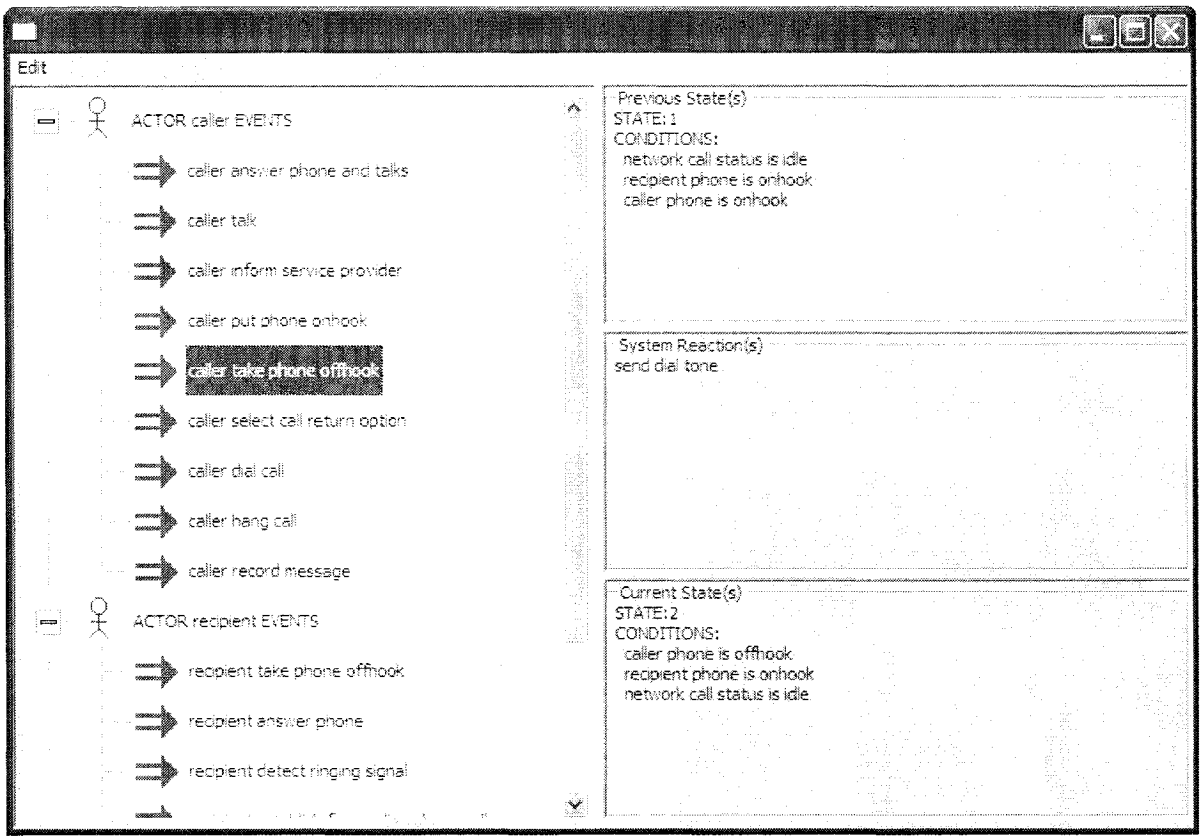


Figure 5-6: Simulating the use case: "Make Call" using the UCEd Simulator

In order to simulate, we apply use case: *"Make Call"* that is presented in Figure 5-3 to the use case simulator. The simulation process of this use case is shown in Figure 5-6. Before starting the simulation process, we set the *"initial state"* of the simulator as *"caller phone is onhook AND recipient phone is onhook AND network call status is idle"*. This is the precondition of the use case: *"Make Call"*. Subsequently, we trigger the first step of the use case: "*caller takes phone offhook*". Then the right side pane of the simulator would show the *"Previous State(s)"*, *"Current State(s)"* and *"System Reaction(s)"*. In this case the previous state is the use case precondition and the current state is *"caller phone is offhook AND recipient phone is onhook AND network call status is idle"*. The system reaction is *"send dial tone"*, which is the second step of the use case. Similarly, rest of the steps of this use case is simulated until it completes its execution. After a successful execution of the use case the state conditions appearing in the *"Current State(s)"* section of the simulator should be equivalent to the use case postcondition.

5.7 Limitations of the UCEd Framework

UCEd framework is developed using a semi-formal type of grammar that is generated using the JavaCC parser generator [39]. Semi formal grammars are not capable of exploiting the full power of a natural language. Requirements expressed using a natural language is easier to understand. Therefore, verification and validation of requirements is easy when they are represented using a natural language grammar. As a result UCEd grammar exhibits a great reduction in expressive power. As discussed in Chapter 4, a natural language grammar can be represented using a Definite Clause Grammar (DCG). Implementation of DCG using a declarative programming language such as SWI Prolog could be very efficient. Natural Language grammars can also be implemented using a Backus-Naur Form (BNF). However, BNF implementations of grammar rely on techniques such as JavaCC lookahead mechanism [39]. The parsing efficiency when using the lookahead mechanism in JavaCC, deteriorates with the increase of the number of tokens that should be looked ahead from a specific choice point. Therefore, DCG implementations of SWI Prolog may be parsed more efficiently than BNF implementations of grammar.

One of the other limitations of UCEd grammar is that it does not provide support for writing instances based use cases. Instances can be used to improve the expressive power of writing use cases. Therefore, this limitation of UCEd hinders the expressive power of requirements elicitation.

Another limitation of UCEd is that a domain model is created manually. The creation of a domain model is done in an incremental way, starting with the basic elements. When use cases are added to the use case model, the domain model needs to be updated accordingly with the new domain knowledge and vice-versa. An automatic extraction of a domain model is more effective and efficient than the manual extraction of domain elements which is cumbersome and time-consuming.

5.8 Chapter Summary

UCEd is composed of a Domain Editor and a Use Case Editor that is used for domain and use case modeling. It is also capable of generating state machines and use case simulation. Simulation is used for building a prototype of the requirements and play with the composed use cases. UCEd use cases can be validated during use case edition, use case composition and use case simulation. Therefore, UCEd has the added advantage of validating and verifying the composed use cases. This facilitates the early detection of problems during the requirement analysis phase. UCEd exploits the extensible nature of UML stereotypes in defining domain sub-entity types. Thus by integrating the features of the natural language grammar defined in Chapter 4 into the UCEd grammar, we can arrive at a better grammatical approach.

The next few chapters elaborate how UCEd grammar is enhanced to overcome the limitations of the grammar that is introduced in Chapter 4. In chapter 6, we implement the notion of instances in UCEd grammar, and observe the impact of instances based use cases on the corresponding domain model and state model generated. Consequently, in chapter 7, we extend the domain model extraction feature that is introduced in chapter 4 and implement it as an enhancement in UCEd, for distinguishing domain sub-entity types.

Chapter 6 - Supporting instances in UCEd

In chapter 4, we presented a grammar which is integrated with instances. Instances improve the expressive power of use cases. This chapter is about integration of instances to UCEd use cases. When instances are integrated with use cases the generation of resulting state model becomes complicated. This is one of the problems when instances are integrated with use cases.

A semi-formal grammar is a restricted form of grammar. A semi-formal grammar lies between a formal grammar and a natural language grammar. UCEd framework employs a semi formal grammar for writing textual use cases. This chapter starts with a review of the semi-formal grammar that is employed by the UCEd framework. Thereafter an enhancement of UCEd grammar is presented for supporting instances in textual use cases. In the latter part of this chapter we explain the complexity of having instances integrated with use cases, which is followed by a solution to reduce this complexity.

6.1 Support for instances in UCEd grammar

In chapter 4, we mentioned that support for instances increase the expressive power of a requirements language. When grammar formalisms become more expressive, it is harder to analyze or parse those using automated tools. That is why most requirements elicitation models use formal grammars for requirements specification. Contrarily, more formal a language is, less expressive it is to represent requirements. UCEd framework is implemented using a semi-formal type grammar. This semi-formal type of grammar is implemented using the JavaCC parser generator. A *"Parser generator"* is a program that reads grammar that describes a language and outputs source code (i.e. parser) that will recognize the language described by the grammar. UCEd grammar is a pure Java implementation. Therefore, we moved the DCG grammar implementation that was introduced in Chapter 4 to JavaCC.

In the following sub-section, we present a discussion on the JavaCC parser generator. It is used by the UCEd framework to generate a semi-formal grammar.

6.1.1 JavaCC parser generator

JavaCC [39] is a 100% Java implementation of a parser generator. The grammar of the existing framework of UCEd is based on JavaCC.

There are other similar parser generators when compared with JavaCC. They can also be used to automate the source code generation of a parser. One such parser generator is YACC [40]. Another similar kind of parser generator is ANTLR [41]. ANTLR is a second generation parser generator that is implemented using Java. It has the option of generating a Java or C++ parser. However, learning curves for both YACC and ANTLR are steep. Relatively speaking the complexity of ANTLR is higher than that of YACC.

JavaCC reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. JavaCC allows defining grammars in EBNF (Extended Backus-Naur) form. Since JavaCC generates top-down parsers, it has the ability to pass values both up and down the parse tree during parsing.

Some grammars use the feature of *"backtracking"* for retracing back to the appropriate choice point when a bad choice is made. The general problem of matching an input with a grammar may result in large amounts of backtracking and making new choices. This may take time and would depend on how the grammar is written. Therefore, JavaCC employs the *"lookahead"* technique [39] instead of *"backtracking"*. It makes decisions at choice points based on some exploration of tokens further ahead in the input stream. Once such a decision is made, it commits to it. Hence no backtracking is performed once such a decision is made. Similarly, ANTLR use the lookahead mechanism for parsing grammars. But the limitation with the lookahead technique is that, greater the number of tokens that are to be looked ahead at a particular choice point, the parsing process will be highly time consuming and inefficient. To overcome this problem to some extent, we could resort to *"local lookahead"* as opposed to *"global lookahead"*. Local lookahead affects only a specific choice point, in contrast to the global scope of a grammar file as defined using global lookahead. UCEd grammar uses the local lookahead technique.

Although a JavaCC grammar is defined using a grammar file, much of the processing of tokens in the grammar have to be handled through Java source code. This highlights the lack of flexibility of JavaCC grammars.

6.2 Why instance support for textual use cases in UCEd is complicated?

A use case step may consist of a use case operation. Object orientation suggests that operations performed on an instance are declared in the instance's abstraction [3]. Consider the example use case description: *"Ann informs the service provider"*, where *Ann* is an instance of the concept *caller*. In this case the operation: *inform* should be associated with the concept abstraction: *caller*. This means an operation of an instance is associated with the relevant entity of the instance in the domain model. Implementation of instances for use cases in the UCEd framework is not simple. This is because even though operations of instances can be associated with the relevant entity, we need to have a mechanism to support for instances based postconditions that are associated with the generic operations of instances (i.e. operations that are associated with the entity of an instance). For instance consider the use case operation: *"caller dials the recipient number"*. Let's consider that the corresponding domain operation of this use case operation is *"dial recipient number"* and the operation postcondition is *"recipient phone is ringing"*. This example appears to be simple. But what happens if the phone system happens to have a multi-dialling facility and a caller instance happens to dial to several recipient instances. Let's consider the use case operation *"Ann dials John Jane numbers"*. This operation is associated with instances. So how can we represent it as a domain operation in the abstraction of the associated instance? We need to find out a way to represent this domain operation in a generic manner. Also in this case the resulting postconditions generated could be *"John number is ringing"* and *"Jane number is ringing"*. Implementing instances based postconditions for generic operations of instances is a problem that we need to solve. To solve this problem we use parameters (or placeholders) in defining generic domain operations and their post conditions. A parameter is represented by the notation: <*p*>. Subsequent sections of this chapter discuss

in detail how parameters can be used to reduce the complexity when instances are introduced for use case modeling.

6.3 Enhancement to UCEd grammar for supporting instances

We presented the grammar which is employed by the UCEd framework in section 3.3.4. However this grammar does not provide support for instances based use case descriptions. Therefore, we needed to modify the grammar to provide support for instances in writing use case descriptions. As discussed in section 6.1, UCEd uses JavaCC to generate a parser from a grammar. But grammars generated by the JavaCC parser generator are not flexible enough for manipulation. This is mainly because manipulation of the JavaCC grammar is done through Java source code. The enhancements made to the UCEd grammar for providing support for instances are presented in Figure 6-1. The addition to this grammar is a rule for defining instances which is represented in line 18. In addition, lines 9-16 represent how instances could be associated with different types of sub-entities when writing different patterns of entities.

```
1.  /* Grammar for conditions */
2.  <condition> -> <acondition> "and" <condition>
3.                | <acondition> "or" <condition>
4.                | "("<condition>")"
5.                | <negation> <condition>
6.  <acondition> -> [<article>] <entity> [<verb>] <value>
7.  <article>   -> "a" | "an" | "the"
8.  <negation>  -> "not" | "no"
9.  <entity>    -> <concept> | <instance> | <concept> <instance> |
10.         <concept> <attribute> | <instance> <attribute> |
11.         < concept> <instance> <attribute> |
12.         <concept> <aggregate> | <instance> <aggregate> |
13.         <concept> <instance> <aggregate> |
14.         <concept> <aggregate> <attribute> |
15.         <instance> < aggregate> <attribute> |
16.         <concept> <instance> <aggregate> <attribute> | .....
17. <concept>   -> <word>+ {member of the model concepts}
18. <instance>  ->  <word>+ {instance of concept}
19. <aggregate> -> <word>+ {member of the model concepts}
20. <attribute> -> <word>+ {attribute of concept}
21. <verb>      -> "is" | "is" "not" | "isn't" | "are"
22.           | "aren't" | "become"  | "becomes"
23. <value>     -> (<word>)+ {value of the entity}
24.         | <comparison> {entity is non-discrete ?}
25. <comparison> -> <comparator> <word>
26. <comparator> -> ">" | "<" | "=" | "<=" | ">=" | "<>" | "greater
27.         than" | "less than" | "equal to" | "different to" |
28.         "greater or equal to" | "less or equal to"
29.
30. /* Grammar for operations */
31. <operation_spec> -> <concept_operation> | <branching_statement>|
32.                <useCase_inclusion>
33. <concept_operation> -> [<before_delay>] [ after_delay]
34.                [<condition_spec] <operation_reference>
35. <condition_spec> -> "IF" <condition> "THEN"
36. <operation_reference> -> (<word>)+ {derived from a operation of
37. the current entity}
38. <after_delay> -> "AFTER" <duration_spec>
39. <before_delay> -> "BEFORE" <duration_spec>
40. <duration_spec> -> <duration_value> <duration_unit>
41. <branching_statement> -> "GO" "TO" "Step" <word>
42. {corresponding to a step label}
43. <useCase_inclusion> -> [<condition_spec>] "INCLUDE"
44.         <use-case-name>
```

Figure 6-1: Enhancement for UCEd grammar for supporting instances

When writing use case descriptions that supports instances, an instance of a concept can be written in different ways (e.g. *<instance>* or *<concept> <instance>*). Similarly, when instances are associated with an attribute of a concept, an aggregate of a concept or an attribute of an aggregate of a concept, they can be written in different ways (e.g. *<instance>*

attribute>, <*concept*> <*instance*> <*attribute*>, <*instance*> <*aggregate*>, <*concept*> <*instance*> <*aggregate*>, <*instance*> <*aggregate*> <*attribute*>, <*concept*> <*instance*> <*aggregate*> <*attribute*>). These various ways of writing instances are handled by the <*entity*> rule of the grammar. If the name of an instance is qualified with a name of a concept, the type of the concept that is associated with the instance could be a normal concept or a system concept.

Let's take an example to show the different types of entity patterns that are associated with instances. Assume that a calling party of a telephony system is represented by the concept: *"caller"*. Instances of this concept are *"Ann"* and *"John"*. In addition, *"line status"* is an attribute and *"phone"* is an aggregate of the concept *"caller"*. Further, *"ringer"* is an attribute of the aggregate *"phone"*. Table 3-1 shows examples for different types patterns of entities. For example, instance: *"Ann"* of concept: *"caller"* can be represented as *"Ann"* or *"caller Ann"* by following the entity patterns <*instance*> or <*concept*> <*instance*> respectively.

Entity pattern	**Example**
<instance>	Ann
<concept> <instance>	caller Ann
<instance> <attribute>	Ann line status
<concept> <instance> <attribute>	caller Ann line status
<instance> <aggregate>	John phone
<concept> <instance> <aggregate>	caller John phone
<instance> <aggregate> <attribute>	John phone ringer
<concept> <instance> <aggregate> <attribute>	caller John phone ringer

Table 6-1: Examples for different entity patterns of the enhanced UCEd grammar

6.4 Support for instances in writing UCEd use case descriptions

UCEd use case descriptions mainly consist of conditions and operations. In this section we discuss the use of instances in the formulation of use case descriptions. We present patterns

for writing use case descriptions involving instances. These patterns are based on the grammar that is represented in Figure 6-1 and are illustrated with examples in Table 6-2.

No.	Description	Pattern	Example
1	Conditions that involves instances of concepts	*<concept> <instance> <verb> <value>*	caller Ann is silent
		<instance> <verb> <value>	Ann is silent
2	Conditions that involves instances of concepts that have attributes	*<concept> <instance> <attribute> <verb> <value>*	recipient John line status is busy
		<instance> <attribute> <verb> <value>	John line status is busy
3	Conditions that involves instances of concepts that have aggregates	*<concept> <instance> <aggregate> <verb> <value>*	caller Ann phone is onhook
		<instance> <aggregate> <verb> <value>	Ann phone is onhook
4	Conditions that involves instances of concepts that have attributes of aggregates	*<concept> <instance> <aggregate> <attribute> <verb> <value>*	recipient John phone ring count is > 5
		<instance> <aggregate> <attribute> <verb> <value>	John phone ring count is > 5
5	Operations initiated by instances of concepts	*<concept> <instance> <operation>*	caller Ann dials the call
		<instance> <operation>	Ann dials the call
6	Operations initiated by instances of concepts that have aggregates	*<concept> <instance> <aggregate> <operation>*	recipient Jane phone starts to ring
		<instance> <aggregate> <operation>	Jane phone starts to ring

Table 6-2: Examples for patterns of instances based use case conditions and operations

Descriptions 1-4 shows different patterns and corresponding examples of instances based use case conditions. Similarly, descriptions 5 and 6 shows different patterns and corresponding examples of instances based use case operations. For example, since the instances based entity of the condition of the first description could be written in two different forms (i.e. *<concept> <instance>* and *<instance>*), there could be two different patterns for writing conditions that involve instances of concepts. In addition, we have also given examples for each of these patterns.

6.5 Enhancements to UCEd domain model for supporting instances

Enhancing the UCEd domain model for supporting instances makes it a hybrid of a UML class diagram and an object diagram. UML object diagrams model the instances of entities contained in a class diagram. An object diagram shows a set of instances and their relationships at a particular point of time [3].

We modified the UCEd domain model in order to introduce instances. These modifications apply to domain operations and their associated postconditions. In section 6.2, we mentioned that in UML, instances are defined in their corresponding abstraction. Instances are introduced to UCEd domain model by defining them as a child of a Concept or a System Concept (we will refer as Concept in the remainder).

In section 6.2, we stated that in order to provide support for instances we need to implement operations and postconditions using parameters. This means that we have to define domain operations and postconditions in a generic sense (or in a universal context). We define operations and postconditions in the context of the corresponding abstraction (or concept) when providing support for instances.

Figure 6-2 shows this fact with an example using an excerpt of the domain model of the Telephone PABX Case Study. The UCEd domain model has specific icons to represent the different types of domain elements. Icons *"C"*, *"I"*, *"O"*, "+" and "-"are used to represent concepts, instances, operations, added post conditions and withdrawn post conditions respectively. As we mentioned earlier, a domain operation is attached with a specific domain entity. In addition, a domain entity can have zero or more instances, each of which can be associated with a domain operation defined for the domain entity. A domain operation may or may not contain parameters. Parameters are used by generic domain operations in order to pass instance values to the corresponding parameters of its generic postconditions. For instance as shown in Figure 6-2, *"caller"* is a concept and *"Ann"* and *"Bob"* are instances of the concept: *"caller"*. One of the operations of concept: *"caller"* is *"dial <p> number"*, which is a generic domain operation. This generic operation consists of a parameter: *"<p>"*.

In addition, this generic operation also contains a generic post condition: *"<p> line status is free"*. Another domain operation of concept: *"caller"* is *"takes phone offhook"*. This is not a generic operation as it does not contain a parameter. It has a generic postcondition: *"<p> phone is offhook"* and a non-generic postcondition: *"line status is busy"*. The system concept of this domain model is *"network"*. It has a generic domain operation: *"monitor <p> line status"*. This domain operation consists of a generic disjunction postcondition: *"<p> line status is free OR <p> line status is busy"*.

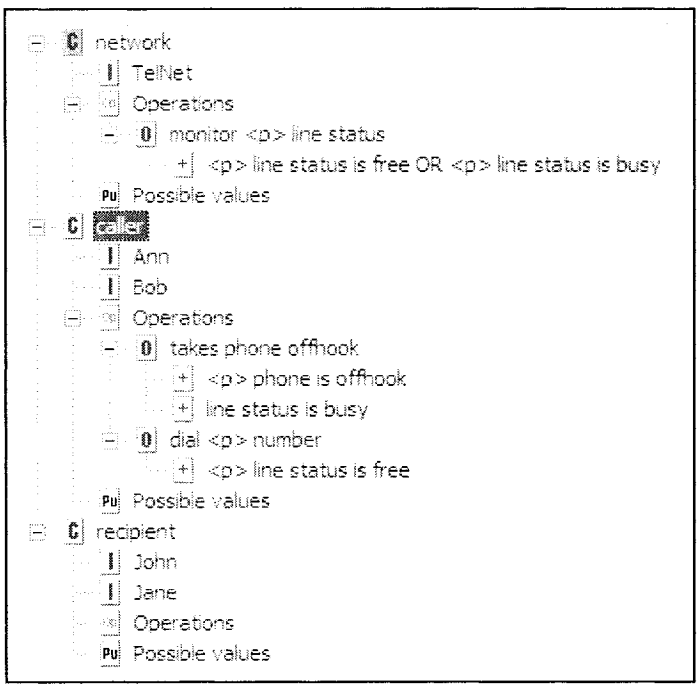


Figure 6-2: Enhanced domain model with parameters

Even if a domain operation does not have parameters, postconditions of that domain operation may have parameters. In such a context, it is the instances of the corresponding entity of the use case operation that may serve as the appropriate values for the parameter contained in the generic operation postcondition. These instance values could be substituted for the placeholders contained in the postconditions that may result in generating a state condition of the state machine. In Figure 6-2, *"takes phone offhook"* is an operation of domain entity: *caller*. This is an operation that does not consist of parameters. In this case, *Ann* is the instance of entity: *caller* of the corresponding use case operation. The domain operation may have postconditions without parameters (e.g. *"line status is busy"*), in which case it does not have any impact of instances on state conditions generated. On the other

hand, the domain operation may also have postconditions defined with parameters (e.g. "*<p> phone is offhook*"). In this case, instance value: *Ann* could be substituted for parameter: *<p>* in forming a state condition (i.e. "*Ann phone is offhook*") in the state machine.

If operations are defined as generic, using parameters, the instance value(s) for the parameter of the domain operation is evaluated from the corresponding use case operation description. The entity name of the instance value(s) for parameter(s) of the use case operation description may or may not be the same as the entity name of the corresponding domain operation. If we consider the domain operation: "*monitor <p> line status*" and the corresponding use case operation step: "*network monitors John line status*", we notice that *John* (of entity *recipient*) is the instance value for parameter: *<p>*. But *John* is not an instance of entity: *network* that is associated with the corresponding domain operation. Let's consider domain operation: "*dial <p> number*" and the corresponding use case operation step: "*Ann dials John number*". In this case we observe that *John* is the instance value for parameter: *<p>*. It is interesting to note that in this case that the entity name of the instance value of the use case operation description is equivalent to the entity name of the corresponding domain operation.

6.6 Impact of instances in UCEd state model generation

It is during the process of use case composition that state machines are generated in UCEd. State machines describe the transition between states that occurs when executing use case operations. The complexity of a state machine increases when instances are introduced to the state model. Each instance associated with an entity of a condition, may result in a separate state in the state model. For example, if there is a use case condition: "*caller phone is offhook*", the existence of two instances: "*Ann*" and "*Bob*" for the concept "*caller*", may now lead to two separate state conditions: "*Ann phone is offhook*" and "*Bob phone is offhook*". Greater the number of instances for a particular concept, larger the number of state conditions that may be generated for the corresponding condition.

The convention of using parameters for defining domain operations and postconditions is done in a compatible way, such that parameter values for instances in use case step operations can be mapped to parameters in the corresponding domain operations. In addition, parameters for domain operations and postconditions are defined in such a way that they provide easy validation of both domain and use case models. State conditions are generated depending on the manner in which placeholders are located in domain operations and postconditions, and how instances are used in use case operations. As mentioned before, instance parameter values in use case operations serves as values for parameters in the corresponding domain operations. These parameter values are substituted for parameters in the corresponding domain operation postconditions when generating state conditions.

The different possibilities of generating state conditions with generic domain operations and postconditions are discussed below.

When both domain operations and postconditions are generic (i.e. they use parameters), instance values for the parameter are evaluated from the instance values which are specified after the conjugated verb of the corresponding use case operation. This is illustrated in Figure 6-3. In this case *"caller"* and *"recipient"* are concepts, and *"network"* is a system concept. *"Ann"* and *"Bob"* are instances of concept: *"caller"*, whereas *"John"* and *"Jane"* are instances of concept: *"recipient"*. Concept: *"caller"* has a generic domain operation *"dial <p> number"* and a corresponding generic domain operation postcondition: *"<p> line status is busy"*. Similarly, system concept: *"network"* has a generic domain operation: *"send <p> ringing signal"* and a corresponding generic domain operation postcondition: *"<p> phone signal is ringing"*. If we consider the use case operation *"Ann dials John number"*, *John* is evaluated as the instance value for parameter *<p>* of domain operation: *"dial <p> number"*. Therefore, after substituting *John* for parameter *<p>*, we arrive at state condition: *"John line status is busy"*.

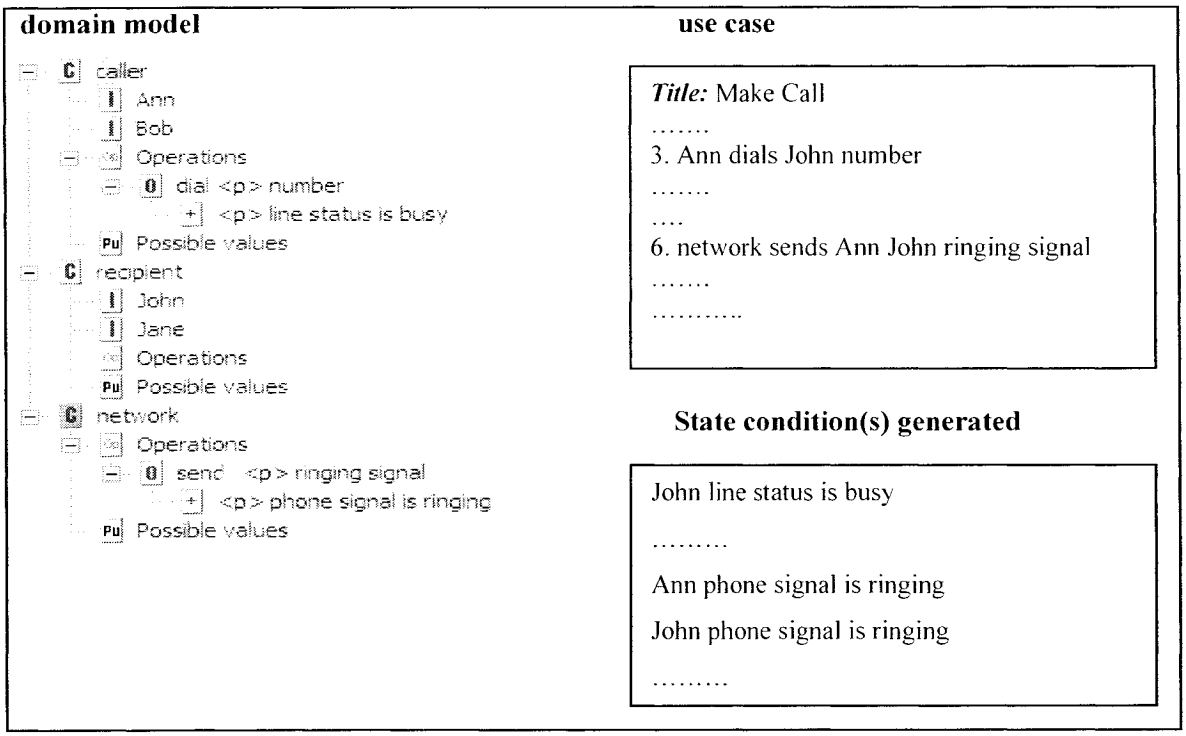


Figure 6-3: State condition(s) generated when both the domain operation and postcondition are generic

A parameter of a generic domain operation could also be assigned with multiple instance parameter values as evaluated from its corresponding use case operation. When considering the use case operation: *"network sends Ann John ringing signal"*, *Ann* and *John* are evaluated as instance parameter values for parameter: *<p>* of domain operation *"sends <p> ringing signal"*. After substituting the instance values to the corresponding domain operation postcondition, the resulting state conditions generated would be: *"Ann phone signal is ringing"* and *"John phone signal is ringing"*. This shows that, number of instances values for a parameter of a generic domain operation determines the number of state conditions generated.

Figure 6-4 illustrates how states conditions are generated when a domain operation is not generic (i.e. it does not contain parameters) and the corresponding postconditions are generic (i.e. they contain parameters). In this case, the value for the parameter of the domain operation postcondition should be the instance value of the entity that initiates the corresponding use case operation. Therefore we have to make sure that the use case

operation is associated with an instance type. Example represented in Figure 6-4 shows that *"Ann"* and *"Bob"* are instances of concept: *"caller"*, whereas *"John"* and *"Jane"* are instances of concept: *"recipient"*. In addition, concept: *"caller"* has a domain operation: *"dial number"* and a corresponding generic domain postcondition: *"<p> line status is busy"*. In this example, instance value: *Ann*, which is associated with the use case operation: *"dials number"*, is substituted with the parameter: *<p>* of the domain operation postcondition: *<p> line status is busy*. This results in generating the state condition: *Ann line status is busy*.

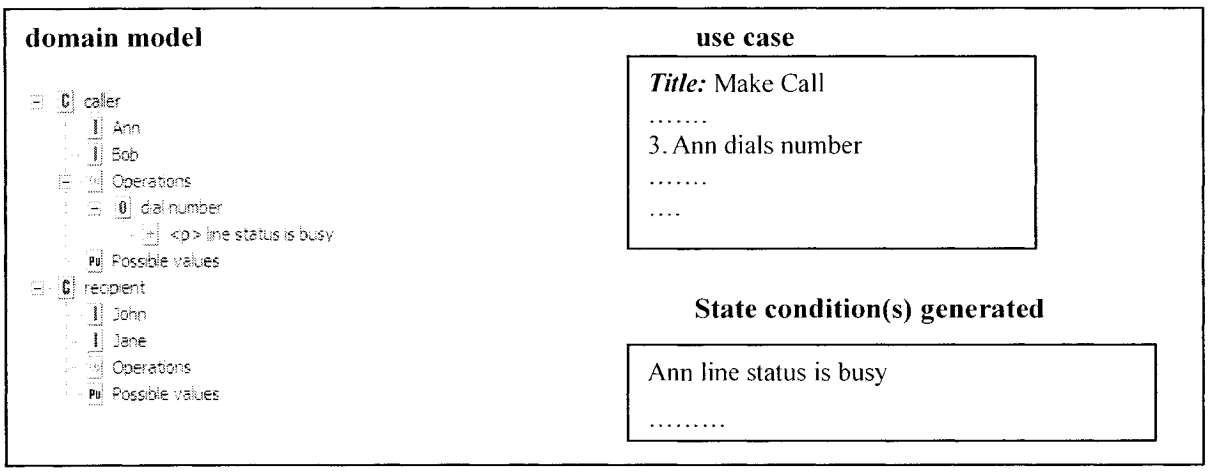


Figure 6-4: State condition(s) generated when post condition is generic.

Figure 6-5 illustrate that if the domain operation is generic and its postconditions are not generic, the resulting state conditions generated are not generic. In this example, *"Ann"* and *"Bob"* are instances of concept: *"caller"*, whereas *"John"* and *"Jane"* are instances of concept: *"recipient"*. Concept: *"caller"* has a generic domain operation *"dial <p> number"* and a corresponding domain operation postcondition: *"recipient line status is busy"*. As a result, state conditions generated in this context describe an entity and not an instance. The state condition generated in this example is *"recipient line status is busy"*. It is a condition that describes about recipient's line status, which is an entity.

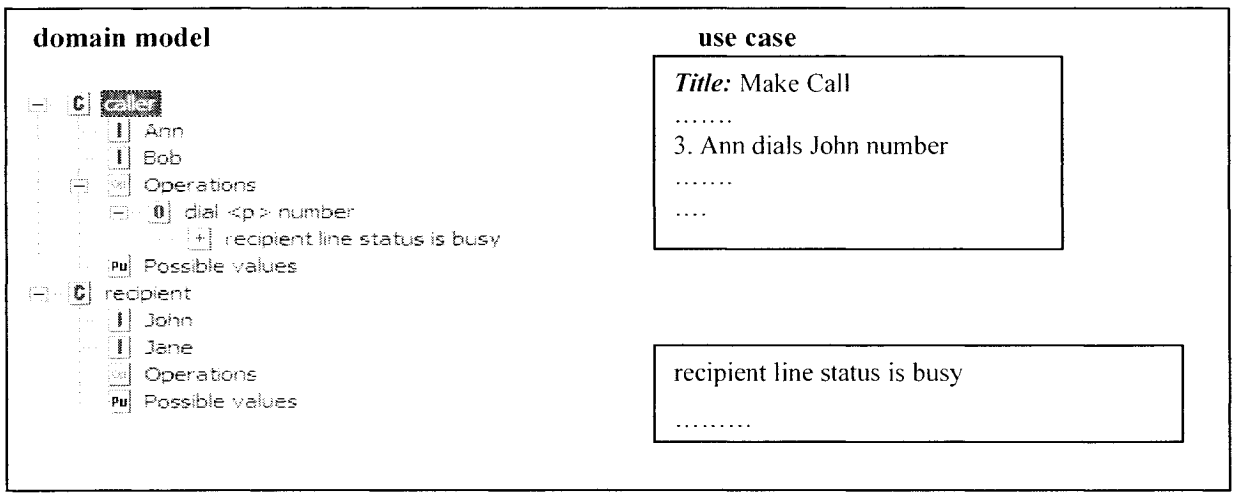


Figure 6-5: State condition(s) generated when domain operation is generic.

Postconditions of domain operations may also be defined as composite conditions. One of the major issues faced with the application of instances is when the generic postconditions of domain operations are composite conditions. The type of the composite condition (e.g. AND condition, OR condition) determines the number of states generated.

If the number of constituent simple conditions in a composite AND generic postcondition of a generic domain operation is *numCond*, and the number of instance values available for a parameter is *numInst*, then the total number of instances based state conditions generated for the postcondition can be evaluated using the formula: *[numInst * numCond]*. Figure 6-6 shows an example of a composite AND generic postcondition and the resulting state conditions generated. In this example *"Ann"* and *"Bob"* are instances of concept: *"caller"*, whereas *"John"* and *"Jane"* are instances of concept: *"recipient"*. Concept: *"caller"* has a generic domain operation: *"dial <p> number"* and a corresponding composite AND generic postcondition: *"<p> line status is busy AND <p> status is oncall"*. The composite AND generic postcondition contains two constituent simple conditions. Use case operation: *"Ann dials John Jane number"* has two instance parameter values: *John*, *Jane*, for parameter: *<p>* of the corresponding domain operation. Therefore, there are *[2 * 2] = 4* resulting state conditions generated for the generic postcondition. It is important to observe that the final resulting state condition generated is a conjunction of all constituent simple state conditions

(i.e. *John line status is busy AND Jane line status is busy AND John status is oncall AND Jane status is oncall*). Therefore there is only a single state that is generated.

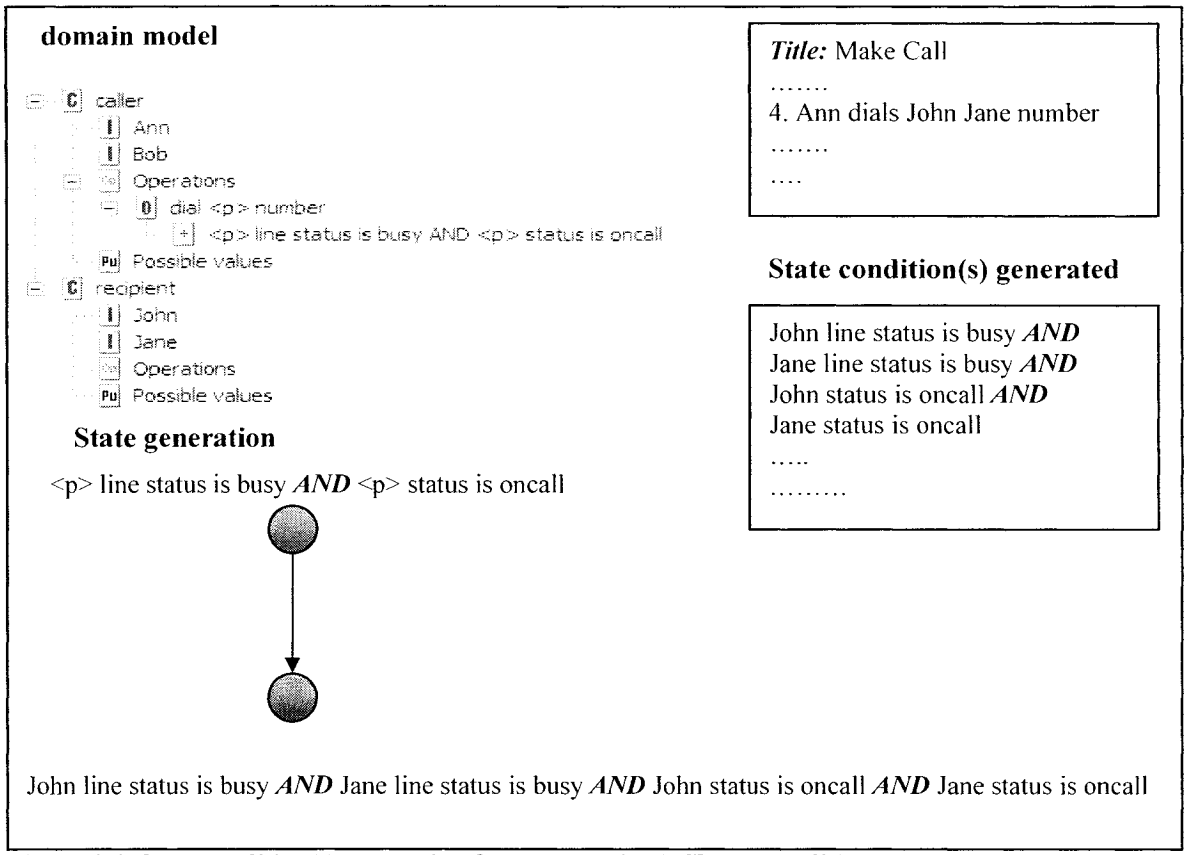


Figure 6-6: State condition(s) generation for a composite AND postcondition

Consequently, in the case of a composite OR generic postcondition, if the number of constituent simple conditions is denoted by *numCond* and the number of instance values available for a parameter of the generic domain operation is *numInst*, the number of state conditions generated would be *[numInst * numCond]*. These states that are generated are independent from each other. This characteristic of state conditions generated for a composite OR generic postcondition may cause a state explosion. Figure 6-7 presents an example of a composite OR generic postcondition. In this example *"Ann"* and *"Bob"* are instances of concept: *"caller"*, whereas *"John"* and *"Jane"* are instances of concept: *"recipient"*. Concept: *"caller"* has a generic domain operation: *"dial <p> number"* and a corresponding composite OR generic postcondition: *"<p> line status is busy OR <p> status is oncall"*. This composite OR generic postcondition contains two constituent simple conditions. In this case, the use case operation: *"Ann dials John Jane number"* has two

instance parameter values: *John*, *Jane*, for parameter: <*p*> of the corresponding domain operation. Therefore, there are *[2 * 2] = 4* resulting ***independent*** state conditions generated for the generic postcondition. Each of these states results in a separate state.

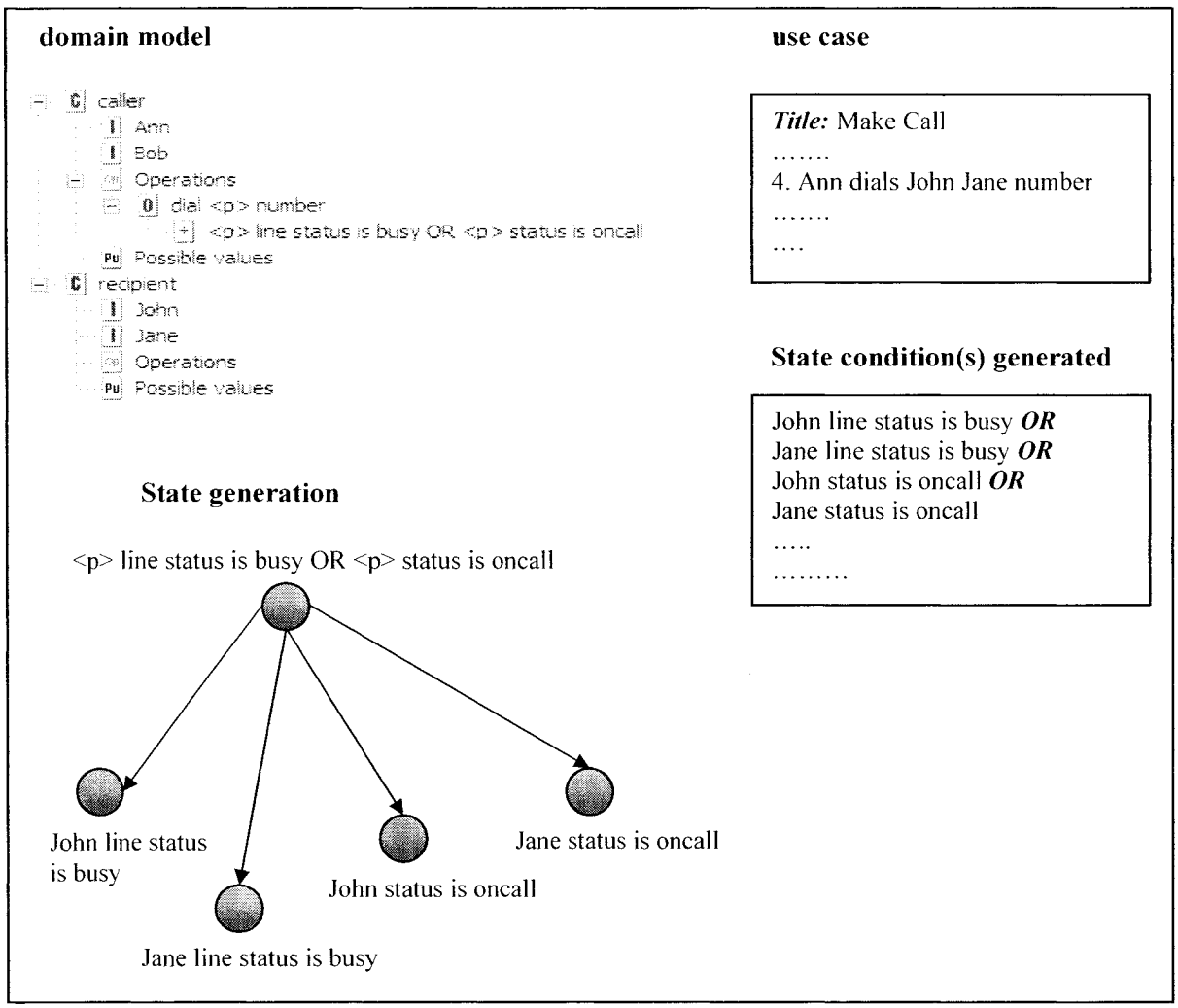


Figure 6-7: State condition(s) generation for a composite OR postcondition

6.7 Implementation issues

UCEd employs a semi formal type grammar implemented using JavaCC. Due to lack of flexibility of JavaCC, much of the work related to the implementation of instances is handled through Java source code. We reviewed alternate grammar formalisms with the intention of ascertaining their flexibility and expressive power for writing instances based use cases and subsequently integrating them into UCEd. Some of these grammar formalisms include Link

Grammar [42] and Phrasys [43]. All of these formalisms exhibited limitations in integrating them to UCEd. Therefore, we had to work out a solution by enhancing the UCEd grammar and finding out ways in manipulating instances through Java source code of the UCEd grammar parser. However this matter is still open to discussion and investigation.

6.8 Chapter Summary

In this chapter we presented integration of instances to UCEd. Operations initiated by instances are defined in the domain model by associating them with the corresponding entities (i.e. abstraction) that they belong to. This means that domain operations for instances are defined in an abstract manner. Further, domain operation postconditions should also be defined in abstract form since they are defined in respect to each abstract operation of the domain model. It is these domain operation post conditions that are used to determine state conditions generated during operation execution. Generation of state conditions become complicated when instances are integrated to use case operation descriptions. To reduce this complexity we introduce the usage of parameters. Parameters can be used to provide the required expressive power for writing use case descriptions and generating the corresponding state model, when instances are integrated into textual use cases. Notice that even sequence diagrams, which are another form of a use case formalism, use parameters to represent instances when message interactions are initiated between instances.

Chapter 7 - Implementation for Domain Extraction in UCEd

Traditionally, a domain model is derived by manually updating domain elements in an ad hoc fashion from an existing use case model. Manual Extraction of a domain model is a tedious and cumbersome task. The natural language parser that is introduced in Chapter 4 provides a solution to overcome this problem by automating the domain model extraction process. However, a limitation of this parser is that it does not allow distinction of domain sub-entity types. A manual inspection is needed to distinguish between domain sub-entity types. A complete domain model cannot be formulated, without separately identifying the different sub-entity types of a domain. UCEd supports sub-entity types because it is based on UML. In this chapter we present how we integrate our results to the UCEd framework for automatic extraction of domain models.

This chapter starts with a brief overview on how automatic domain extraction would be tied to the software development process, followed by the architecture of the domain extraction process that is integrated to the UCEd framework. Thereafter, we present the algorithm of our domain extraction process. Subsequently, an evaluation of the algorithm is presented using the set of use cases of the Telephone PABX system case study. This chapter ends with a discussion of the limitations of the domain extraction tool.

7.1 How would a domain extraction tool be tied into the development process?

Elements of a domain model may be identified using the different word forms of a requirements text. Nouns can be used to extract domain entities and verbs can be used to extract domain operations and associations. Ultimately, this process assists in deriving a complete domain model that corresponds to a requirements model. The domain model so derived can also be represented as a UML class diagram. A UML class diagram can be used in the design phase of the software development life cycle. As such, a domain model can also be viewed as a bridge between the Requirements Engineering and Software Design phases of the development process. Therefore, extraction of a domain model from a use case model

facilitates smooth transition from the requirements analysis phase to the software design phase.

Defining a class model is not an easy task [44]. If the class model is created in a disorganized manner, it means that the domain entities are created in an ad hoc manner, never knowing when the class model will get completed. On the other hand if a rigid model is adopted we have to start by determining the classes, associations, attributes, etc. of the model. But it is hard to complete one step before moving to the next as most of the time we have to come back to the previous steps for updating the model. Thus the best choice would be to adopt an intermediate approach, which is neither disorganized nor rigid but falls in between them. In an intermediate approach a step-by-step approach is followed in identifying the initial set of classes, then adding or deleting from this set as work progresses.

Defining a class model is a very sensitive task. The definition of a domain model may vary from individual to individual and their style. Even the best programmers in the world have considerable difficulty thinking at the level of abstraction needed to create effective models [44]. In addition, educating software developers have focussed much on design and development than modeling of domains. Therefore future software projects may be at risk unless proper training is provided for software professionals [44].

7.2 Architecture of the domain extraction process

During the domain extraction process domain elements of a use case model are extracted. In chapter 4, we presented a natural language parser, which is capable of extracting domain elements from a set of use cases in a fully automated fashion. However, domain entities extracted from that parser were too generic, and a manual technique has to be employed in distinguishing sub-entities of the domain entities extracted. A better choice to overcome this problem is to formulate a method to distinguish sub-entity types of domain entities. UCEd domain model defines sub-entities based on UML stereotypes. Therefore, in order to prevent re-inventing the wheel, we decided to integrate our approach of domain extraction with UCEd. However, it is important to state that our technique of domain model extraction can

be integrated with any other use case analysis tool that supports different types of domain sub-entities.

One of the obstacles in distinguishing between different types of sub-entities while performing a domain extraction is the inherent ambiguity of the language. For example, if we consider the use case condition: *"caller line status is busy"*, we can identify easily that *"caller line status"* is an entity. However, identifying the constituent sub-entity types of an entity is a not easy. It may be that after identifying the concept of the entity: *"caller"*, the type of the next sub-entity could be an attribute or an aggregate. It could also be that the concept of the entity may be one of the word combinations: *"caller"*, *"caller line"* or *"caller line status"*. If the concept of the entity is *"caller"* then *"line"* or *"line status"* could be an attribute or an aggregate of the concept. It appears that the ambiguity of the language is more apparent when a sub-entity type of an entity is composed of more than one word. Because of these ambiguities, our domain model extraction approach is semi-automated.

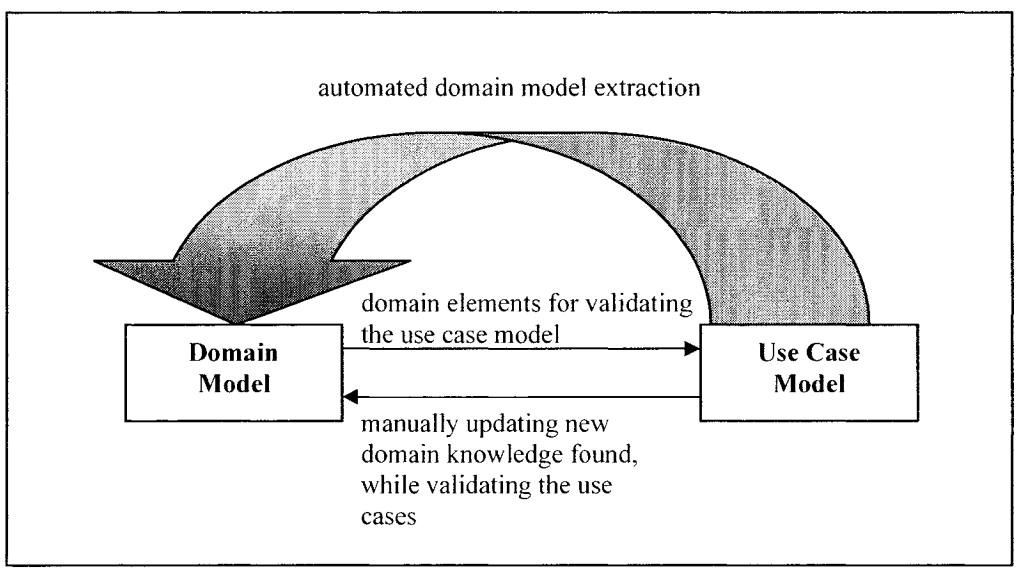


Figure 7-1: Automated domain model extraction process

The usual relationship between a domain model and a use case model is that elements of a domain model are used to validate a use case model, and during the process of validation any new domain knowledge found is updated in the domain model. This is a tedious and a cumbersome approach. An automated approach will reduce the complexity of this process to

a greater extent and eliminate the ad hoc nature of constructing a domain. These two ways of adding new knowledge to the domain is represented in Figure 7-1. An automated domain model extraction process should be able to build up a new domain from scratch or update a partial domain that is already in existence.

7.3 Algorithm for domain extraction

This section discusses our semi-automated algorithm for extracting domain elements from a use case model. For the purpose of demonstration we have integrated the domain extraction tool that we developed to the UCEd framework. However, our algorithm is flexible enough to be ported to other requirements models with minimal changes.

Recall that from Chapter 5 that UCEd domain entities consist of the following;

1. ***Concept:*** Represents an actor in a system's environment. There are also concepts that are defined as sub-concepts of a parent concept. These sub-concepts represent specialization and inheritance behaviour.
2. ***System Concept:*** This represents a component of the system under consideration which is represented as a black box.
3. ***Aggregate:*** Used to represent a "part-of" relationship that participates with a parent concept or system concept.
4. ***Attribute:*** A characteristic of a concept, system concept or an aggregate.
5. ***Object:*** An instance of a concept or a system concept.

Concepts, system concepts, aggregates and attributes have possible values that characterize the state of each of these domain elements at a particular point of time.

The complete algorithm for extracting domain elements from an existing use case model is represented in Figure 7-2.

1. Identify missing domain conditions, operations and actors
   1.1 For each use case,
   For each use case description,
   If (condition: c $\in$ use case description)
   /* A condition is defined as <entity> <verb> <value> */
   If condition entity: e exists in the domain and condition value: v does not exist in the domain,
   Add a possible value: v to the referring entity: e, which is in the domain
   Else if condition entity: e does not exist in the domain
   $C = C \cup \{c\}$
   If (operation reference: o $\in$ use case description)
   /* An operation reference is an operation that is associated with an entity */
   If o is not defined in the domain
   $O = O \cup \{o\}$
   1.2 For each actor: a, who is interacting with the use case model,
   If (a is not defined as a domain entity in the domain)
   $A = A \cup \{a\}$

2. For each element c in C,
   For each sub-entity: se (starting from the leftmost sub-entity) of the condition entity: e
   If sub-entity: se does not exist in the domain
   - Prompt the user to select the type of the sub-entity: se
   - Add the sub-entity: se to the domain
   Insert condition value: v, as a value to the corresponding sub-entity of e

3. For each element a in A,
   If a is not defined in the domain
   - Prompt the user for the type of sub-entity of Actor: a
     /* Sub-entity type of an actor could either be a *"Concept"* or a *"System Concept"* */
   - Add the sub-entity to the domain

Let op be the action specification of the use case operation reference
/* Action specification has the form: conjugated_action_verb [action_object] */

4. For each element o in O,
   For each token: t in o (starting from the leftmost token),
   - If (t $\in$ verb and a starting point of a logical operation)
     op = part of the sentence that starts from t
     Else
     Prompt the user to select the appropriate value for the action specification, op
   - Evaluate the infinitive form of op.

   The part that starts from the first word up to the preceding word of the action verb of o, contains the value of the operation entity: e for op (i.e. part that is towards the left side of o after removing the action specification)
   For each sub-entity: se (starting from the leftmost sub-entity) of operation entity: e
   If sub-entity: se does not exist in the domain
   - Prompt user to select the type of the sub-entity: se
   - Add the sub-entity: se to the domain

   Insert the infinitive form of op as a domain operation to the corresponding sub-entity of e

Figure 7-2: Algorithm for domain model extraction

The inputs and outputs of the extraction algorithm are,

- ***Input:*** A set of use cases from which domain elements needs to be extracted. There could also be an existing partial domain. If not the domain needs to be created from scratch.
- ***Output:*** Complete domain model that consists of all the domain elements of the use case model. Use cases in input can be parsed using the extracted domain.

A use case consists of use case descriptions. Use case descriptions that the algorithm processes are the use case precondition(s), use case postcondition(s), use case steps and use case extension steps. The first step of the algorithm relates to the processing of the use case model in order to identify missing domain conditions, operations and actors. A domain condition is represented in the domain by associating a value to a specific entity. This way the value of a condition describes the state of its entity. Operations are associated with domain entities. Actors are represented as entities (i.e. *concepts* or *system concepts*) in the domain. Step 1.1 relates to processing of use case descriptions of each use case. These use case descriptions are processed in order to identify missing conditions and operations in the domain. A use case description may consist of a condition if it is a use case precondition, a use case postcondition, a use case step or a use case extension step. An UCEd use case condition has the form: *<entity> <verb> <value>*. The value of an UCEd condition describes the entity associated with it. If the entity of a condition is already part of the domain, but the value associated with the entity is not yet defined in the domain, then the algorithm simply adds the value to the corresponding entity of the domain. In the event, the condition entity itself is not part of the domain, the algorithm simply adds the condition into set: *C*. A use case description may also consist of a use case operation reference. A use case operation reference (i.e. *<entity> <operation> ...*) is a phrase that describes an operation initiated by an entity. If the use case description contains an operation reference, the algorithm checks whether the operation reference already exists in the domain. It is important to state that the operation of an UCEd use case operation reference is represented in conjugated form, whereas the corresponding UCEd domain operation exists in the infinitive form. The associated entity of a missing domain operation of an operation reference may or may not be defined in the domain, and will have to be taken into account when processing use case descriptions. An operation reference which is not yet defined in the domain is added to set: *O*. Each use case description is processed in this manner and sets *C*

and *O* are updated with missing domain conditions and operations accordingly. This process is iterated for each use case of the use case model.

Actors are entities that interact with use cases in a use case model. In addition, actors are also entities that initiates use case operations. They can exist only in the form of a concept or a system concept in the domain model. In step 1.2, if an actor described in the use case model does not exist in the form of an entity in the domain, then it is added to set *A*.

The second step of the algorithm relates to the processing of missing domain conditions which is added to set: *C*. An entity of a condition may consist of sub-entities. Part of these sub-entities may have already been defined in the domain, in which case it is the non-existing domain sub-entities that needs to be added to the domain. The algorithm scans a condition for sub-entities from its leftmost word token. It is only the sub-entities not present in the domain that are processed and added to the domain by the domain extraction tool. If a sub-entity is not found in the domain, the user will be prompted to select the type of the sub-entity. After all sub-entities of a condition are added to the domain, the corresponding condition value is added to the appropriate sub-entity of the domain model. If all appropriate sub-entities of a condition are present in the domain, then the associated value of the condition will be inserted automatically without user interaction. This means after learning the domain model by the domain extraction process, the need for user intervention is minimized. The most crucial part of the second step of the algorithm is identification of sub-entities which are not present in the domain. This is because of the inherent ambiguity of the language and has been explained in detail in section 7.2. Given a use case entity that does not exist in the domain, each of its constituent sub-entities has to be distinguished from a set of possible combination of words. This is because a sub-entity may consist of more than one word. Determination of a sub-entity from these choices (i.e. possible set of combination of words) has to be done manually with the help of user interaction. As a result the domain extraction process cannot be fully automated, and have to be carried out in a semi-automated manner. In order to generate a possible combinations of words from which a sub-entity could be identified, the algorithm that is illustrated in Figure 7-3 is used.

```
n = number of words of the entity that consists of sub-entities
m = 1
strArray = an array that consist of the words of the entity.
vec = vector to hold the possible combination of words from which a sub-entity of a entity could be
selected

while (m < n + 1)
{
        for (int i = 0; i < n; i++)
        {
                String tmpString = "";

                for (int j = i; j < i + m && (n - i) >= m; j++)
                {
                        tmpString += strArray[j] + " ";
                }

                if ((n - i) >= m)
                {
                        vec.add(tmpString);

                }
        }
                m++;
}
```

Figure 7-3: Algorithm for deriving the different combination of words for the purpose of determining a sub-entity of an entity

The functionality of the algorithm presented in Figure 7-3 can be explained using an example. Let's consider the example use case condition: *"caller line status is busy"*. The entity of this condition is: *"caller line status"*. As mentioned in section 7.2, the constituent sub-entities of an entity have to be identified using a set of possible combination of words that are generated using the entity string. According to the algorithm, array: *"strArray"* hold the words of the entity string. The output of this algorithm is the set of possible combination of words, which is stored in vector: *"vec"*, from which the user can select the appropriate value for the sub-entity. The algorithm computes the different combination of words of a string as follows,

Given a string $w_1\ w_2\ w_3\ w_4 \ldots w_n$ delimited by spaces where w_1, w_2, w_3, w_4, ..., w_n are the constituent words in the string, we can derive all possible combination of words of the string as follows,

W_1, w_2, W_3, w_4, ..., W_n

W_1 W_2, w_2 w_3, W_3 W_4, ..., w_{n-1} w_n

W_1 W_2 W_3, w_2 w_3 w_4, ..., W_{n-2} W_{n-1} W_n

W_1 W_2 W_3 W_4, ..., w_{n-3} w_{n-2} w_{n-1} w_n

.....

.....

W_1 W_2 W_3 W_4 ..... W_n

These combinations of words will have any pattern that we may need to choose for a particular selection of a sub-entity of an entity string. Therefore, the example entity condition can generate the following combinations of words, out of which the appropriate sub-entities of the condition entity could be determined.

caller, line, status,
caller line, line status,
caller line status

In the above example, *"caller"* is a sub-entity of type concept and *"line status"* is an attribute of the concept. Both of these sub-entities are choices of the possible combination of words that are generated by the algorithm as described in Figure 7-3.

The third step of the domain model extraction algorithm relates to the processing of actors that was added to set: *A*. At this point of execution of the algorithm some of these actors of set: *A*, may already have been added to the domain as domain entities. This is because some of these actors may have been added while processing the second step of the algorithm for entities of conditions. If any of the actors of set: *A* does not exist in the domain, the algorithm prompts the user in order to select their sub-entity type by providing a choice between the alternatives: *concept* or *system concept*. This action requires user interaction.

The fourth step of the algorithm relates to processing of operation references that were added to set: *O*. Each element of an operation reference that was added to set: *O* is scanned from its

left most token until the first word that belongs to the verb category is reached. The part of the operation reference that starts with an action verb is considered as the start of an operation (i.e. action specification). The infinitive form of the operation is then added to the corresponding domain entity of the domain model. It could be the case that the corresponding entity that is associated with the use case operation is still not defined in the domain, in which case the relevant entity have to be added to the domain before adding the associated domain operation. The entity of the operation is the substring that starts from the leftmost token of the operation reference and ends at the word token preceding the action verb. The operation entity needs to be processed for sub-entities before adding it into the domain. This is performed in a similar manner as described in step 2 of the algorithm. The verb form of a word in an operation reference is evaluated using the WordNet lexical database. However this process of treating the first occurrence of the leftmost verb of a sentence as the use case operation is inaccurate. This is because, due to the ambiguity of the natural language, a word which is of verb form may also belong to many other word forms such as nouns, adjectives, etc. In the event of such an ambiguity, the domain extraction tool prompts the user to manually choose the operation of the operation reference.

One of the limitations of the domain extraction utility of the natural language parser which is described in Chapter 4 is that it is not capable of distinguishing domain sub-entity types. The domain extraction algorithm integrated with the UCEd framework provides the flexibility of distinguishing domain sub-entity types and subsequently adding them into the domain. Therefore, this feature is added as an improvement to the former approach in order to resolve a limitation of the former natural language parser. The idea behind the generation of a set of possible combination of words, in order to determine the appropriate name of a particular domain sub-entity is a novel technique in order to overcome the ambiguity of the language to a certain extent. Evaluation of the time complexity of the domain extraction algorithm is hindered due to the interactive nature of the domain extraction application. Computing the interaction time of an interactive application is totally unpredictable and depends drastically on user response time.

7.4 Evaluation of the domain extraction tool

Now that we have formulated the algorithm for extracting a domain model automatically from an existing UCEd use case model, we need to evaluate the correctness and the validity of it. For this purpose we apply all the use cases of the Telephone PABX system which are defined in Appendix E to the domain extraction application.

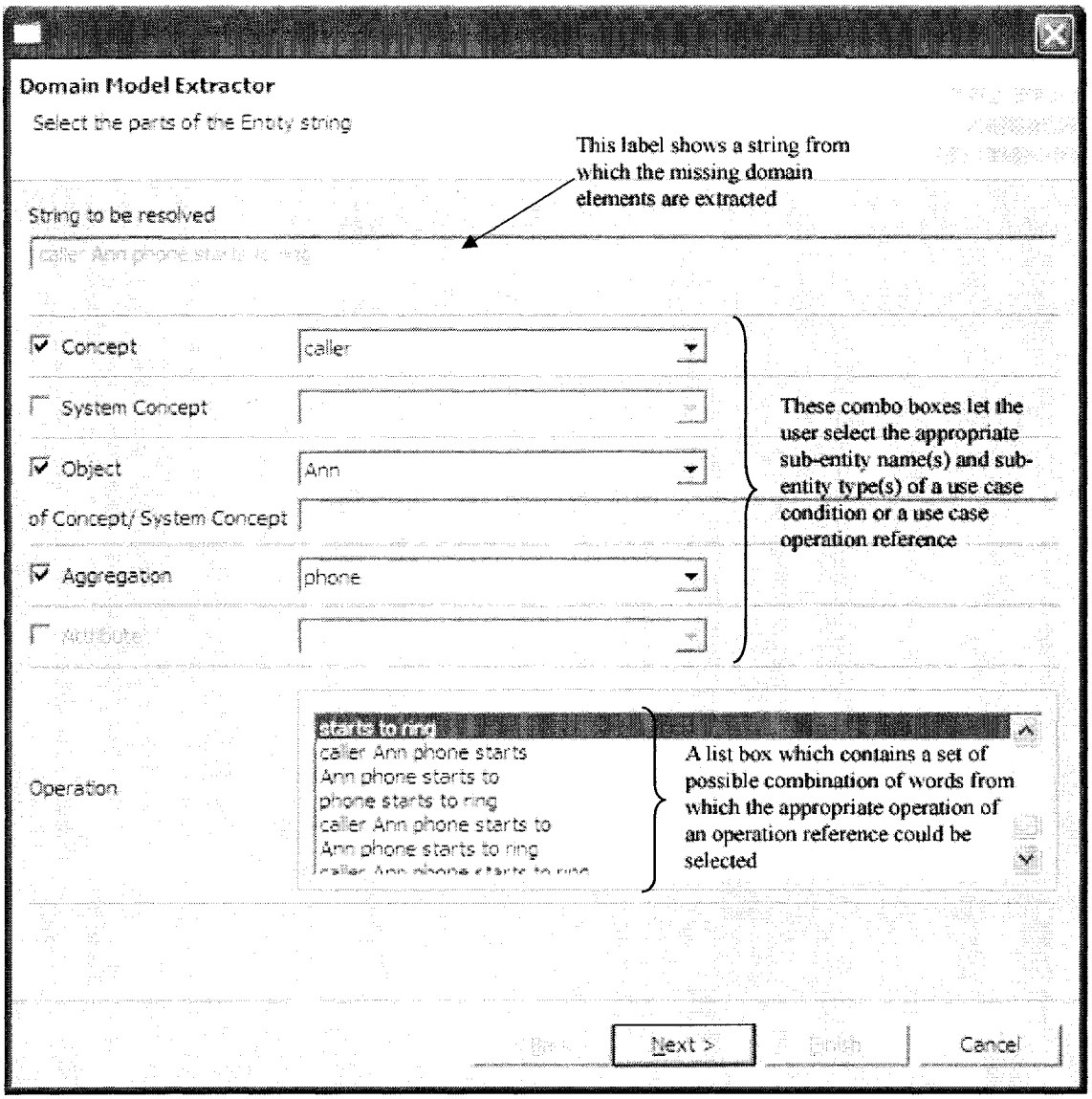


Figure 7-4: User interface of the domain extraction tool

Before we apply the use cases in Appendix E to the domain extraction process, we modify the use case model such that, entities: *"caller"* and *"recipient"* are merged into a single

entity: *"caller"*. This would make the interactions between instances more apparent in the use case model. Initially, we start with an empty domain and build it from scratch by running the domain extraction tool. In order to get a better understanding of the domain extraction tool, we present its user interface in Figure 7-4.

As specified in Figure 7-4, the label at the upper portion of the interface shows the string from which the missing domain elements need to be extracted. This string could be a condition, an operation or an actor. The combo boxes at the mid portion of the interface let the user resolve sub-entity names and sub-entity types of a particular entity. Each combo box identifies a sub-entity type. The name for each sub-entity type is then resolved using a set of combination of words that are populated in the combo boxes. The entity that is made out of these sub-entities may belong to a use case condition, a use case operation or it may even be an actor. The lower portion of the interface shows a set of combination of words from which an operation of a particular operation reference could be identified. If the use case description to be resolved is an operation reference: *"caller Ann phone starts to ring"*, the entity of the operation reference is *"caller Ann phone"*. The user has to manually select each of the constituent sub-entity types and sub-entity names (i.e. concept = *caller*, object = *Ann*, aggregate = *phone*) of the entity concerned, with the help of the possible combination of words that are populated in the appropriate combo box fields. Similarly, the user has to select the appropriate operation of the use case operation reference from a possible combination of words, if the selection chosen by the tool is incorrect. In this case, the correct operation of the operation reference is *"starts to ring"*. Resolving sub-entity types of conditions and actors using the tool is similar to that of operation references, except for the fact that conditions do not have any selections for an operation, and sub-entity type of an actor is selected between the choices: *Concept* and *System Concept*. An entity of an operation reference does not contain a sub-entity of type: *attribute*, whereas a condition may contain a sub-entity of type: *attribute*.

The domain extraction tool will iterate for each use case, processing each of its use case descriptions. If we assume that the first use case that is processed by the tool is *"Make Call"*, the initial use case descriptions that will be processed are the use case precondition and the

postcondition. Both of them are in the form of composite conditions. Therefore they are processed only after segregating them into its constituent simple conditions. Thereafter, conditions and operation references of use case steps and extension steps are processed. There may be use case descriptions that may contain both use case conditions and operation references. Use case steps 5 and 8 of use case: *"Make Call"* have use case descriptions that contain conditions as well as operation references, each of which have to be processed separately by the tool.

When use case: *"Make Call"* is processed, the domain extraction algorithm which is specified in Figure 7-2, will go through all the use case descriptions of the use case, and populate the missing conditions and operation references of the domain, in sets *C* and *O* respectively. This is the functionality of step 1.1 of the algorithm. Domain entities of conditions with missing condition values are processed such that the missing values are added to the corresponding condition entities. Updates for missing condition values for already defined entities of the domain requires no user interaction. When the use case precondition: *"Ann phone is onhook AND John phone is onhook AND network call status is idle"* is processed, all three constituent simple conditions: *"Ann phone is onhook"*, *"John phone is onhook"*, *"network call status is idle"* are added to set *C*, as neither of them exists in the domain. When the postcondition: *"network connection is up AND Ann phone is onhook and John phone is onhook and network call status is idle"* is processed only one of its constituent simple conditions (i.e. *"network connection is up"*) is added to set: *C*, as the other simple conditions are the same as those that were processed by the precondition. Likewise, the conditions of the rest of the use case descriptions that are not defined in the domain are added to set: *C*. Similarly, operation references of use case steps and extension steps that are not in the domain are added to set: *O*. In the case of use case: *"Make Call"* all its operation references are added to set: *O*, since at the outset the domain model is empty.

Thereafter, according step 1.2 of the algorithm, all actors of use case: *"Make Call"* is added to set: *A*, as none of them exists in the domain prior to the application of the domain extraction algorithm.

Step 2 of the domain extraction algorithm relates to the processing of conditions in set: *C*. Consider the condition: *"network connection is up"* that was added to set: *C* in step 1.1 of the algorithm. The entity of this condition is *"network connection"*. However, sub-entity: *"network"* has already been detected as a system concept while processing the use case precondition. Therefore, the user is only prompted to select the sub-entity type for *"connection"* (which is an attribute of system concept *"network"*). After user interaction, the tool will add the new attribute to the domain together with its value: *"up"*. Similarly, the rest of the conditions of set: *C* is processed.

After processing the use case conditions of use case descriptions of use case: *"Make Call"*, step 3 of the domain extraction algorithm will prompt the user for the sub-entity types of the entities that were added to set: *A*, but not yet defined in the domain. Such an instance is prompting for the sub-entity type of entity: *"message recorder"* (which is of type *"concept"*), as it was not added as an entity to the domain while processing the use case conditions.

Step 4 of the domain extraction algorithm relates to the processing of operation references. Step 1 of use case: *"Make Call"* is *"Ann takes the phone offhook"*. It is an operation reference and is added to set: *O*. In this case, the tool will detect automatically that *"Ann"* is an instance (i.e. object) of concept *"caller"*, as it was selected by the user when the use case precondition was processed and subsequently added to the domain. The tool will also detect that *"takes the phone offhook"* as an operation. This is because that WordNet dictionary identifies that *"takes"* as the first occurrence of the word that belongs to the verb category when scanned from the leftmost word of use case step 1. Since the corresponding entity of this operation already resides in the domain, infinitive form of the use case operation: *"take phone offhook"* will be added to the corresponding entity *"caller"* of the domain. Similarly, the rest of the operation references of set: *O* is processed.

However, the automatic selection of a use case operation by the domain extraction tool is not accurate. This is because a particular word that belongs to the verb category may also be a member of other word categories. An example for this fact is the operation of the use case

operation reference shown in Figure 7-4. In this case, the use case operation is initially detected as *"phone starts to ring"*. This is because *"phone"* is the first word that belongs to the verb category, when scanned from the leftmost word of the operation reference. However, it is not the correct operation (i.e. the correct operation should be *"starts to ring"*) of this use case description. Therefore, the correct operation has to be selected manually by the user.

The set of input use cases which is applied to the algorithm is defined using the *use case editor* of the UCEd framework. A partial domain could be defined with the *domain editor* of UCEd. Otherwise, we can start the domain extraction process with an empty domain. After extracting the domain elements using the domain extraction tool which is integrated to the UCEd, domain elements of the complete domain model can be reviewed using the *domain editor* of UCEd. This complete domain model can then be used or refined during system design of the software development process.

After having processed all the use cases of the Telephone PABX system by the domain extraction tool, we notice that the domain model is completely updated. We verified this fact by validating both the domain and use case models of the UCEd framework. The view of the UCEd domain editor, before and after the extraction of the domain model, using the set of use cases of the Telephone PABX system case study is presented in Figure 7-5. This shows that the domain extraction tool has completely extracted the domain model of the corresponding use case model. The extracted domain model in Figure 7-5 shows entities, domain operations and possible values of entities that are extracted.

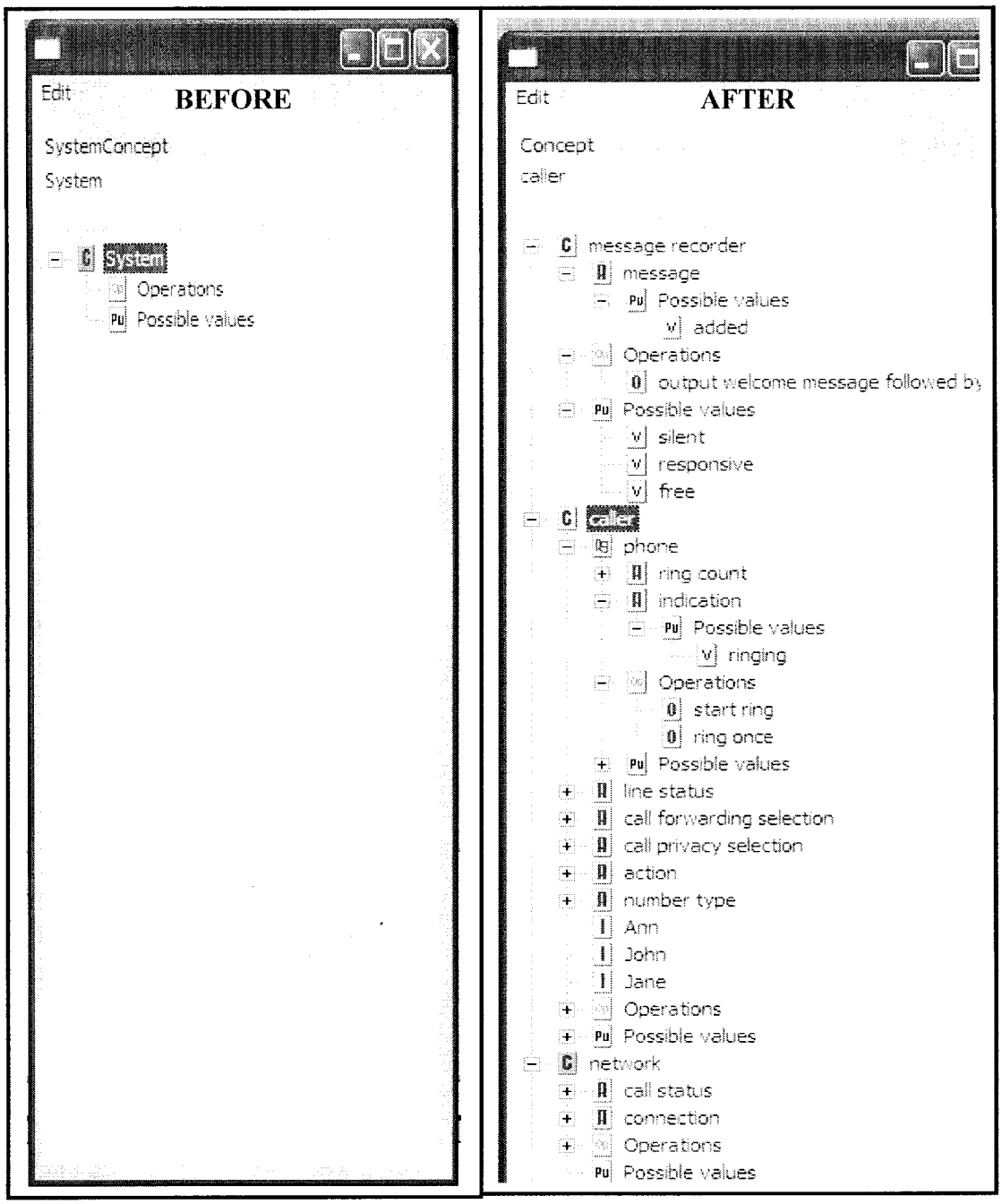


Figure 7-5: View of the domain before and after running the domain extraction tool

7.5 Limitations

One of the major limitations of the automatic domain extraction algorithm that is integrated into the UCEd framework is its inability to carry out a fully automatic domain extraction process due to the inherent ambiguity of the natural language. Therefore, we have to settle

with a semi-automated level of domain extraction. Due to the ambiguity of the language, constituent sub-entity types of a domain entity have to be resolved by user interaction.

Domain operations may have postconditions. These postconditions exist in the form of added conditions and withdrawn conditions. However, when extracting domain operations from use case descriptions, operation postconditions cannot be added to the domain. This is because that there is no information to ascertain the corresponding operation postconditions from operation references of the use case descriptions when they are extracted to the domain. Another limitation of the domain extraction algorithm is that it is incapable of extracting generic domain operations from instances based use case descriptions. Remember that generic domain operations contain parameters and they are evaluated by instance parameter values located at specific locations of use case operations.

7.6 Chapter Summary

Since a domain model can be represented as an UML class model, it facilitates smooth transition and refinement from requirements to system design. An important observation that we made in this chapter is that, as we try to enhance the flexibility of the domain model extraction process by providing support for sub-entity types, we had to trade-off for a semi-automated way of domain model extraction. This is mainly because of the inherent ambiguity of the natural language from which use case descriptions are written. There have been some other few attempts [8] for automatically extracting domain elements for use case descriptions. These attempts also ended-up being semi-automated.

Evaluation of the domain extraction tool shows that providing support for identifying sub-entity types of an entity facilitates extraction of a complete domain, even though it is carried out in a semi-automated fashion. This would improve the requirements analysis process and ultimately eliminate the time consuming ad hoc approach of manually identifying domain elements.

Chapter 8 - Conclusions

This chapter reviews the contributions of the thesis. It also discusses works related to our work, how requirements quality attributes meets with our use cases and future work arising from several issues encountered along the way.

8.1 Contributions

Approaches such as RUP imply that during the software development process, use cases can be written at different levels of abstraction. For the purpose of our research we consider textual use cases written at a detail black box level. Use cases written at detail black box level are well understood by domain specialists. Therefore, use cases written at this level minimize the communication gap between domain specialists and requirements analysts. In addition, a use case can be written using a formal or an informal language. Formal languages need lot of skills in manipulating them, whereas informal languages suffer from ambiguity. A semi formal language appears to be a better compromise. However, a semi formal language lacks the flexibility and expressive power that a natural language has when expressing system requirements.

There are various formats for writing use cases at different levels of abstraction. Wirfs-Brock, Cockburn and Essential use cases are formats for use cases written at a detailed black box level. On the other hand MSCs, UCMs and Sequence Diagrams are used to write use case descriptions at a detailed grey box level. Our work is based on Cockburn's use cases. Cockburn suggests that use cases be written using a restricted form of language. Several authors have recommended various kinds of use case guidelines for writing restricted form of use case descriptions.

This thesis makes two main contributions. These contributions are implemented in a restricted grammar that is incorporated into a parser. Thereafter, our results are implemented in the UCEd framework. This is our third contribution towards the thesis.

Contribution 1: Support for instances in use cases

Various grammars have been proposed for writing use case descriptions. CREWS grammar, UCDA grammar, Li's grammar and UCEd grammar are some of them. None of these grammars that we reviewed provide explicit support for writing instances based use cases. Instances provide additional expressive power for requirements elicitation. One interesting feature of the grammar introduced in Chapter 4 is that it provides support for instances. Instances appear to play an important role in requirements elicitation. In the Telephone PABX system case study, we have two domain entities: *"caller"* and *"recipient"*. What would be the position if the interactions between the *"caller"* and *"recipient"* need to be replaced by instances of a single entity: *"caller"*?

Contribution 2: Automated domain model extraction.

Traditional methods of extracting domain information involve identification of nouns to extract domain entities and verbs to extract domain operations. Noun and verb forms in a dictionary can be used to extract domain elements in a fully automated fashion. This technique is used by the parser which is introduced in Chapter 4 for extracting domain elements. However, domain entities extracted using nouns exhibits the limitation of not being able to distinguish between sub-entities. Therefore, manual inspection is required for identifying domain sub-entities in order to build a complete domain. To help this task, we introduce a user interface integrated to the UCEd framework for domain sub-entities identification.

Contribution 3: Implementation of instances and domain extraction to UCEd

We applied our results to the UCEd framework by adding instances to UCEd grammar and automating domain extraction.

When instances are integrated to UCEd use cases, corresponding domain operations and postconditions should be defined accordingly. This is mainly because domain operations and

postconditions are defined with respect to an entity and not with respect to an instance. This implies that domain operations and postconditions so defined should be generic as far as possible. Therefore, we introduce parameter based generic domain operations and postconditions in order to generate instances based state conditions. These state conditions generated are based on the instance parameter values of the associated use case descriptions.

Domain extraction cannot be fully automated due to natural language ambiguity. Our approach is semi-automatic as some user choices need to be made.

8.2 Related work

There are various use case formats that are used to express the interactions between instances. As discussed before, some of them include MSCs, UCMs, sequence diagrams, etc. However, works related to instances based textual use cases are rare. A semi-automatic technique in deriving sequence diagrams from use case models is presented in [8]. This paper describes how a parser identifies instances of use case descriptions. When an article: *"a"*, *"an"* or another precedes a noun or noun phrase, the parser identifies a class name and an instance name. The class name is the noun or noun phrase with an initial capital letter for each word. An article: *"a"* or *"an"* is added to the class name to denote the name of an instance. The parser appends an integer to distinguish two separate instances. For example, the parser translates noun phrase *"a caller"* to class name *"Caller"* and instance name *"aCaller"*. If the instance name *"aCaller"* has already been registered and the largest affix of the instance name that starts with *"aCaller"* is 4, the parser create a new instance name *"aCaller5"*. A semi-automated approach for translating the narrative descriptions of use cases to sequence diagrams is also presented in [8]. It is a user driven approach to a certain extent. In this approach use cases written in restricted natural language form are mechanically translated to message sends, which are then used to build sequence diagrams. The mechanical processing of use cases deduces classes, objects, associations, attributes and operations from use case descriptions. This is another example which shows that a fully automated technique of extracting a domain is far from possible.

A case tool that implements a methodology to automate natural language requirements analysis and class model generation based on RUP is discussed in [9]. The tool takes a set of requirements and assists the developer in the generation of IBM Rational Rose use case diagrams, use case specifications, robustness diagrams, collaboration diagrams and class diagram. In [45], the authors presents an approach to automate the transition from requirements to detailed design. This approach starts with a set of artifacts and methodologies. Use cases are used as the method to capture and record requirements. These use cases are formalized by using a use case template. In addition, use case descriptions are written using language patterns to make the automatic processing of use cases possible. A glossary that contains domain vocabulary is used during the transition process in order to reduce the vagueness of the natural language. For every use case formalized in this approach, there is a corresponding use case realization, and for every use case realization, two artifacts: *robustness diagrams* and *collaboration diagrams* are generated. The robustness diagram finds analysis classes from system's behaviour, where as the collaboration diagram distribute the behaviour to the analysis classes.

Another interesting research with some similarity to our work is extraction of domain models out of existing code for generative use [46]. In this work, the authors discuss reverse engineering of a domain model from an existing code base. The recovered domain model is subsequently used to develop application generators.

8.3 Requirements Quality of our use cases

In section 2.2, we described quality attributes that requirements specification is expected to exhibit. In this section, we discuss how these quality attributes are satisfied with our use cases, which are written according to Cockburn's guidelines. We mention below how our use cases meets the requirements quality attributes.

Adequacy: Even though our use cases are not written using free form text, it can be still comprehended easily as they are written using a semi-formal language.

Consistent: Our use case description follows a specific syntax. Therefore, conflicts between individual requirement statements are minimized.

Correct: Defining all the requirements for a system without missing anything cannot be easily achieved.

Validatable: Since use cases are written using a natural language friendly, semi-formal language, requirements specification can be validated easily.

Verifiable: Our use cases are written at a specific level of abstraction, which is within a specific scope. The scope and the level of abstraction that each use case is written are specified using the *"scope"* and *"level"* fields of the use case. Therefore, use cases can be written, such that use cases at one level of abstraction are consistent with those at another level of abstraction.

Modifiable: Related concerns of use cases are grouped using *"include use cases"*. Therefore, use cases can be easily modified.

Ranked: This quality attribute is not supported by our use cases.

Testable: Use cases can be easily transformed into test cases. This is because, use cases clearly indicates the success and failure scenarios of real world goals. Therefore, use cases are easily testable.

Complete: Since requirements are constantly evolving during the life cycle of the software development process, completeness is not easy to achieve.

Traceable: Use case steps of our use cases are numbered consistently using a logical numbering scheme. This structure of use case descriptions makes it easy to process by a simple algorithm.

Analysis of the above requirements quality attributes indicates that our use cases meets most of them.

8.4 Limitations and Future Work

Higher the flexibility in writing requirements text, higher will be the expressive power for requirements elicitation. When writing textual requirements domain specialists tend to use pronouns to refer to language elements in the context in which they are written. Our work does not provide support for pronoun resolution when writing use case descriptions in textual use cases. Pronoun resolution fall under the category of *"Discourse Analysis"* in natural language processing [38]. Therefore providing support for pronoun resolution in use case descriptions appears to be an interesting future research direction.

Apart from the functional requirements that are captured by textual use cases there are also non-functional requirements such as response time, frequency of occurrence, etc. However, the textual use case format that we use is incapable of capturing these non-functional requirements. In order to overcome this limitation, Cockburn suggests using a tabular form that associates non-functional requirements to goals of goal oriented use cases [6].

The domain model extraction facility that is implemented in UCEd does not support the extraction of operation postconditions. In addition, it does not facilitate the extraction of generic domain operations from instances based use case operations. These are some possibilities to be considered in the future.

In addition, the domain model extraction process which is implemented in UCEd is a semi-automated process. We encounter this limitation when we attempt to provide support for extracting domain sub-entity types. Developing a fully automated domain extraction technique is a difficult task due to the inherent ambiguity of the natural language.

Bibliography

[1] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, vol. 20, pp. 10-19, April 1987.

[2] P. Kruchten, *The Rational Unified Process: An Introduction (2nd Edition)*: Addison-Wesley Professional, March 2000.

[3] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*: Addisson Wesleys, 1999.

[4] G. Schneider and J. P. Winters, *Applying use cases – a practical guide*: Addison Wesley, 1998.

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

[6] A. Cockburn, "Structuring Use Cases with Goals", Humans and Technology, HaT. Technical Report. 1995.01 1995, http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm.

[7] S. Somé, "An environment for use cases based requirements engineering", presented at proceedings of the 12th IEEE International Requirements Engineering Conference, September 2004.

[8] L. Li, "A semi-automatic approach to translating use cases to sequence diagrams", presented at proceedings on Technology of Object-oriented Languages and Systems TOOLS 29, 1999.

[9] D. Liu, K. Subramaniam, A. Eberlein, and B. H. Far, "Natural Language Requirements Analysis and Class Model Generation Using UCDA", presented at proceedings of the 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Springer, May 2004.

[10] C. B. Achour, C. Rolland, N. A. M. Maiden, and C. Souveyet, "Guiding use case authoring: results of an empirical study - CREWS Report 98-31", presented at Fourth IEEE International Symposium on Requirements Engineering (RE'99), University of Limerick, Ireland, June 1999.

[11] B. Nuseibeh and S. Easterbrook, "Requirements Engineering: A Roadmap", presented at proceedings of the International Conference on Software Engineering (ICSE-2000), Limerick, Ireland, June 2000.

[12] P. Zave, "Classification of Research Efforts in Requirements Engineering", *ACM Computing Surveys (CSUR)*, vol. 29, pp. 315 - 321, December 1997.

[13] A. Hall, "What's the Use of Requirements Engineering?" presented at proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97), Annapolis, Maryland, January 1997.

[14] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt, "Automated Quality Analysis of Natural Language Requirement Specification", presented at Fourteenth Annual Pacific Northwest Software Quality Conference, Portland, Oregon, USA, October 1996.

[15] M. Glinz, "Improving the Quality of Requirements with Scenarios", presented at proceedings of the Second World Congress on Software Quality, Yokohama, Japan, 2000.

[16] R. Fernandes and A. J. Cowie, "Capturing Informal Requirements as Formal Models", presented at proceedings of the AWRE'04 9th Australian Workshop on Requirements Engineering, Adelaide, Australia, 2004.

[17] K. Ryan, "The Role of Natural Language in Requirements Engineering", presented at proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, California, USA, January 1993.

[18] N. E. Fuchs and R. Schwitter, "Controlled English for Requirement Specification", presented at Seventh ILPS '95 Workshop on Logic Programming Environments, Portland, Oregon, USA, December 1995.

[19] J. Ryser and M. Glinz, "SCENT - A Method Employing Scenarios to Systematically Derive Test Cases for System Test", Technical Report 2000.03, Institut für Informatik, University of Zurich 2000.

[20] F. Bordeleau and J.-P. Corriveau, "On the Importance of Inter-Scenario Relationships in Hierarchical State Machine Design", presented at Fundamental Approaches to Software Engineering (FASE'2001), held as part of the Joint European Conferences on Theory and Practice of Software ETAPS'2001, Genova, Italy, April 2001.

[21] A. Pols, "Requirements Engineering: Use Cases and More", presented at proceedings of the conference on Object Oriented Programming Systems Languages and Applications, Atlanta, Georgia, USA, 1997.

[22] OMG, OMG Unified Modeling Language Specification version 1.5, http://www.omg.org/technology/documents/formal/uml.htm,

[23] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari, "Application of Linguistic Techniques for Use Case Analysis", presented at IEEE Joint International Conference on Requirements Engineering, Essen, Germany, 2002.

[24] R. Biddle, J. Noble, and E. Tempero, "Essential use cases and responsibility in object-oriented development", presented at proceedings of the Australasian Computer Science Conference (ACSC2002), Melbourne, Australia, 2002.

[25] M. Andersson and J. Bergstrand, "Formalizing Use Cases with Message Sequence Charts, Masters Thesis", in *Department of Communication Systems*: Lund Institute of Technology, May 1995.

[26] D. Amyot, "Use Case Maps Quick Tutorial - Version 1.0", 1999, www.usecasemaps.org/pub/UCMtutorial/UCMtutorial.pdf.

[27] R. R. Hurlbut, "A Survey of Approaches For Describing and Formalizing Use Cases, Document: XPT-TR-97-03", Expertech, Ltd. 1997, http://www.iit.edu/~rhurlbut/xpt-tr-97-03.html.

[28] R. Wirfs-Brock and J. Schwartz, "The Art of Writing Use Cases, Wirfs-Brock Associates, Inc." 2001, http://www.wirfs-brock.com/PDFs/Art%20of%20Writing%20Use%20Cases.pdf.

[29] L. L. Constantine and L. A. D. Lockwood, *Software for Use: A Practical Guide to the Model and Methods of Usage Centered Design*: Addison-Wesley, 1999.

[30] R. Biddle, J. Noble, and E. Tempero, "From Essential Use Cases to Objects", presented at proceedings of the forUSE 2002 1st Int'l Conference on Usage-Centered Design, 2002.

[31] D. Amyot., X. He, Y. He, and D. Y. Cho, "Generating Scenarios from Use Case Map Specifications", presented at proceedings of the Third International Conference on Quality Software, 2003, November 2003.

[32] Process Impact Use Case Template, www.processimpact.com/process_assets/use_case_template.doc, Last consulted in August 2005.

[33] N. Samarasinghe and S. Somé, "Generating a Domain Model from a Use Case Model", presented at proceedings of the 14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2005), Toronto, Canada, July 2005.

[34] S. Some, "An approach for the synthesis of State transition graphs from use cases", presented at proceedings of the International Conference on Software Engineering Research and Practice (SERP), 2003.

[35] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc, 2000.

[36] WordNet - a lexical database for the English language, Cognitive Science Laboratory Princeton University, New Jersey, USA, http://wordnet.princeton.edu/, Last consulted in August 2005.

[37] J. G. Schlachter, ProNTo Morph: Morphological Analysis Tool for use with ProNTo (Prolog Natural Language Toolkit), http://www.ai.uga.edu/mc/ProNTo/Schlachter.pdf, Last consulted in August 2005.

[38] D. Jurafsky and J. H. Martin, *Speech and Language Processing*: Prentice Hall, 2000.

[39] JavaCC Parser Generator, https://javacc.dev.java.net/, Last consulted in August 2005.

[40] S. Johnson, YACC: Yet Another Compiler-Compiler, AT&T Bell Laboratories, http://dinosaur.compilertools.net/yacc/, Last consulted in August 2005.

[41] ANother Tool for Language Recognition (ANTLR), http://www.antlr.org/, Last consulted in August 2005.

[42] Link Grammar, http://www.link.cs.cmu.edu/link/, Last consulted in August 2005.

[43] Phrasys, http://www.phrasys.com/, Last consulted in August 2005.

[44] T. C. Lethbridge and R. Laganiere, *Object-Oriented Software Engineering*: McGraw Hill, 2001.

[45] D. Liu, K. Subrammiam, B. H. Far, and A. Eberlein, "Automating transition from use-cases to class model", presented at IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2003), May 2003.

[46] P. Devanbu and B. Frakes, "Extracting formal domain models from existing code for generative reuse", *ACM SIGAPP Applied Computing Review archive*, vol. 5, pp. 2-14, Spring 1997.

Appendix A – Use Case Templates

Process Impact Template for use cases

Process Impact [32] recommends a one column tabular format for writing use cases, which is illustrated in Figure A- 1.

Use Case ID:			
Use Case Name:			
Created By:		Last Updated By:	
Date Created:		Date Last Updated:	

Actors:	
Description:	
Trigger:	
Preconditions:	1.
Postconditions:	1.
Normal Flow:	1.
Alternative Flows:	
Exceptions:	
Includes:	
Priority:	
Frequency of Use:	
Business Rules:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Figure A- 1: One-Column Tabular Use Case format by Process Impact

RUP use case template

The use case template recommended by RUP [5] is presented in Figure A- 2.

1. **Use Case Name**
   - **1.1. Brief Description**
     ....text...
   - **1.2. Actors**
     ....text...
   - **1.3. Triggers**
     ....text...
2. **Flow of Events**
   - **2.1. Basic Flow**
     ....text...
   - **2.2. Alternative Flow**
     - 2.2.1. Condition 1
       ....text...
     - 2.2.2. Condition 2
       ....text...
     - 2.2.3. .....
3. **Special Requirements**
   - **3.1. Platform**
     ....text...
   - **3.2. .....**
4. **Preconditions**
   ....text...
5. **Postconditions**
   ....text...
6. **Extension Points**
   ....text...

Figure A- 2: RUP Template style for use case writing

Appendix B – Results of the use cases (without instances) processed by the Prolog DCG grammar for the Telephone PABX System Case study

In this section we present the results of the use cases (without instances) of the Telephone PABX System Case Study, after being processed by natural language grammar that is introduced in Chapter 4.

Use Case: Make Call

Input use case:
if caller phone is onhook AND recipient phone is onhook AND network call status is idle , caller takes the phone offhook , network sends the dial tone , caller dials the call , network translates the digits ; if recipient line status is busy ; network sends a busy signal to caller ; caller puts the phone onhook ; GoTo step 1 , if caller line status is free , network connects the call to recipient phone , network sends the ring signal to the recipient phone and caller phone ; if recipient phone ring count is > 5 ; Include AnswerCall , recipient answers the phone ; if network connection is down ; caller informs the service provider , if network connection is up , caller talks to the recipient , Include DisconnectCall.

Output parse tree:
[pre_cond([if, [predicate(entity([caller, phone]), verb(is), [], value(onhook)), AND, [predicate(entity([recipient, phone]), verb(is), [], value(onhook)), AND, predicate(entity([network, call, status]), verb(is), [], value(idle))]]]), [[step([], [], [entity([caller]), operation(takes), [[the, [], [noun(phone), []]], []]], [])], [[step([], [], [entity([network]), operation(sends), [[the, [], [noun(dial), [noun(tone), []]]], []]], [])], [[step([], [], [entity([caller]), operation(dials), [[the, [], [noun(call), []]], []]], [])], [[[step([], [], [entity([network]), operation(translates), [[the, [], []], []]], [])], [ex_cond([[], if, predicate(entity([recipient, line, status]), verb(is), [], value(busy))]), [[ext_step([], [entity([network]), operation(sends), [[a, [adjective(busy)], [noun(signal), []]], [[prep(to), [[], [], [noun(caller), [[]]]]]]]], [[ext_step([], [entity([caller]), operation(puts), [[the, [], [noun(phone), []]], []]], GoTo]]]], [[step([], [if, predicate(entity([caller, line, status]), verb(is), [], value(free))], [entity([network]), operation(connects), [[the, [], [noun(call), []]], [[prep(to), [[], [], [noun(recipient), [noun(phone), [[]]]]]]]], [])], [[[step([], [], [entity([network]), operation(sends), [[the, [], [noun(ring), [noun(signal), []]]], [[prep(to), [the, [], [noun(recipient), [noun(phone), [[]]]]]]]], [])], [ex_cond([[], if, predicate(entity([recipient, phone, ring, count]), verb(is), >, value(5))]), [include([], [Include, AnswerCall])]]], [[[step([], [], [entity([recipient]), operation(answers), [[the, [], [noun(phone), []]], []]], [])], [ex_cond([[], if, predicate(entity([network, connection]), verb(is), [], value(down))]), [ext_step([], [entity([caller]), operation(informs), [[the, [], [noun(service), []]], []]])]]], [[step([], [if, predicate(entity([network, connection]), verb(is), [], value(up))], [entity([caller]), operation(talks), [[[], [], []], [[prep(to), [the, [], [noun(recipient), []]]]]]], [])], [[include([], [Include, DisconnectCall])], [[[[[[[[[[]]]]]]]]]]

Use Case: Disconnect Call

Input use case:
if caller phone is offhook AND recipient phone is offhook AND network call status is oncall , recipient hangs the call , caller hangs the call , network terminates the call , network bills the customer.

Output parse tree:
[pre_cond([if, [predicate(entity([caller, phone]), verb(is), [], value(offhook)), AND, [predicate(entity([recipient, phone]), verb(is), [], value(offhook)), AND, predicate(entity([network, call, status]), verb(is), [], value(oncall))]]]), [[step([], [], [entity([recipient]), operation(hangs), [[the, [], [noun(call), []]], []]], [])], [[step([], [], [entity([caller]), operation(hangs), [[the, [], [noun(call), []]], []]], [])], [[step([], [], [entity([network]), operation(terminates), [[the, [], [noun(call), []]], []]], [])], [[step([], [], [entity([network]), operation(bills), [[the, [], [noun(customer), []]], []]], [])], [[[[[[]]]]]]

Use Case: Answer Call

Input use case:
if caller phone is offhook AND recipient phone is onhook AND recipient phone indication is ringing AND recipient phone ring count is > 5 AND network call status is negotiating , message recorder outputs a welcome message followed by a beep , caller records a message ; if caller is silent ; after 10 sec ; network terminates the call ; caller hangs the call , if caller is reponsive , caller hangs the call.

Output parse tree:
[pre_cond([if, [predicate(entity([caller, phone]), verb(is), [], value(offhook)), AND, [predicate(entity([recipient, phone]), verb(is), [], value(onhook)), AND, [predicate(entity([recipient, phone, indication]), verb(is), [], value(ringing)), AND, [predicate(entity([recipient, phone, ring, count]), verb(is), >, value(5)), AND, predicate(entity([network, call, status]), verb(is), [], value(negotiating))]]]]]), [[step([], [], [entity([message, recorder]), operation(outputs), [[a, [adjective(welcome)], [noun(message), []]], []]], [])], [[[step([], [], [entity([caller]), operation(records), [[a, [], [noun(message), []]], []]], [])], [ex_cond([[], if, predicate(entity([caller]), verb(is), [], value(silent))]), [[ext_step(timing_cond(after, duration(10, sec)), [entity([network]), operation(terminates), [[the, [], [noun(call), []]], []]])], [ext_step([], [entity([caller]), operation(hangs), [[the, [], [noun(call), []]], []]])]]]], [[step([], [if, predicate(entity([caller]), verb(is), [], value(reponsive))], [entity([caller]), operation(hangs), [[the, [], [noun(call), []]], []]], [])], [[]]]]]

Use Case: Return Call

Input use case:
if recipient line status is busy AND network call status is negotiating, network prompts for call return , caller selects the call return option , network monitors the recipient line status ; if recipient line status is busy ; after 30 mins ; network cancels the call request , if recipient line status is free , caller phone starts to ring , caller answers the phone and talks to recipient.

Output parse tree:
[pre_cond([if, [predicate(entity([recipient, line, status]), verb(is), [], value(busy)), AND, predicate(entity([network, call, status]), verb(is), [], value(negotiating))]]), [[step([], [], [entity([network]), operation(prompts), [[[], [], []], [[prep(for), [[], [], [noun(call), [noun(return), [[[[[[[[]], [])], [[step([], [], [entity([caller]), operation(selects), [[the, [], [noun(call), [noun(return), [[[[]], [[]], [])], [[[step([], [], [entity([network]), operation(monitors), [[the, [], [noun(recipient), [noun(line), [noun(status), [[]]]]], [[]], [])], [ex_cond([[], if, predicate(entity([recipient, line, status]), verb(is), [], value(busy))]), [ext_step(timing_cond(after, duration(30, mins)), [entity([network]), operation(cancels), [[the, [], [noun(call), [noun(request), [[]]], [[]])]]], [[step([], [if, predicate(entity([recipient, line, status]), verb(is), [], value(free))], [entity([caller, phone]), operation(starts), [[[], [], []], [[prep(to), [[], [], [noun(ring), [[[[[[[]], [])], [[step([], [], [entity([caller]), operation(answers), [[the, [], [noun(phone), [[]], [[]], [])], [[[[[[[]]]]]]]

Use Case: Forward Call

Input use case:
if recipient call forwarding selection is ON AND caller phone is offhook AND recipient phone is onhook AND network call status is negotiating , recipient phone rings once , network transfers the call to forwarding phone , recipient phone starts to ring ; if recipient phone ring count is > 5 ; Include AnswerCall , recipient establishes the forwarding phone call.

Output parse tree:
[pre_cond([if, [predicate(entity([recipient, call, forwarding, selection]), verb(is), [], value(ON)), AND, [predicate(entity([caller, phone]), verb(is), [], value(offhook)), AND, [predicate(entity([recipient, phone]), verb(is), [], value(onhook)), AND, predicate(entity([network, call, status]), verb(is), [], value(negotiating))]]]]), [[step([], [], [entity([recipient, phone]), operation(rings), [[[], [], []], [[]], []], [[step([], [], [entity([network]), operation(transfers), [[the, [], [noun(call), [[]], [[prep(to), [[], [], [noun(forwarding), [noun(phone), [[[[[[[[]], [])], [[[step([], [], [entity([recipient, phone]), operation(starts), [[[], [], []], [[prep(to), [[], [], [noun(ring), [[[[[[[]], [])], [ex_cond([[], if, predicate(entity([recipient, phone, ring, count]), verb(is), >, value(5))]), [include([], [Include, AnswerCall])]]], [[step([], [], [entity([recipient]), operation(establishes), [[the, [], [noun(forwarding), [noun(phone), [noun(call), [[]]]], [[]], [])], [[[[[[]]]]]]

Use Case: Process Call Privacy

Input use case:
if recipient call privacy selection is ON AND caller phone is offhook AND recipient phone is onhook AND network call status is negotiating , recipient detects the ringing signal , Ext , if recipient number type is unwanted , UnwantedPrivacy , Ext , if recipient number type is important , ImportantPrivacy , Ext , if recipient number type is unidentified , UnidentifiedPrivacy , if caller number type is normal , recipient hears the ringing tone , network sends the recorded name without caller knowledge ; if recipient action is reject ; network terminates the call ; caller hangs the call ; recipient hangs the call ; if recipient action is voicemail ; Include AnswerCall , if recipient action is accept , recipient takes the phone offhook , network establishes the call.

Output parse tree:
[pre_cond([if, [predicate(entity([recipient, call, privacy, selection]), verb(is), [], value(ON)), AND, [predicate(entity([caller, phone]), verb(is), [], value(offhook)), AND, [predicate(entity([recipient, phone]), verb(is), [], value(onhook)), AND, predicate(entity([network, call, status]), verb(is), [], value(negotiating))]]]]), [[step([], [], [entity([recipient]), operation(detects), [[the, [], [noun(ringing), [noun(signal), []]]], []]], [ext_point(Ext, [if, predicate(entity([recipient, number, type]), verb(is), [], value(unwanted))], ext_point_name(UnwantedPrivacy)), [ext_point(Ext, [if, predicate(entity([recipient, number, type]), verb(is), [], value(important))], ext_point_name(ImportantPrivacy)), [ext_point(Ext, [if, predicate(entity([recipient, number, type]), verb(is), [], value(unidentified))], ext_point_name(UnidentifiedPrivacy)), []]]])], [[step([], [if, predicate(entity([caller, number, type]), verb(is), [], value(normal))], [entity([recipient]), operation(hears), [[the, [], [noun(ringing), [noun(tone), []]]], []]], []), [[[step([], [], [entity([network]), operation(sends), [[the, [adjective(recorded)], [noun(name), []]], []]], []), [[ex_cond([[], if, predicate(entity([recipient, action]), verb(is), [], value(reject))]), [[ext_step([], [entity([network]), operation(terminates), [[the, [], [noun(call), []]], []]])], [[ext_step([], [entity([caller]), operation(hangs), [[the, [], [noun(call), []]], []]])], [ext_step([], [entity([recipient]), operation(hangs), [[the, [], [noun(call), []]], []]])]]]], [ex_cond([[], if, predicate(entity([recipient, action]), verb(is), [], value(voicemail))]), [include([], [Include, AnswerCall])]]]], [[step([], [if, predicate(entity([recipient, action]), verb(is), [], value(accept))], [entity([recipient]), operation(takes), [[the, [], [noun(phone), []]], []]], []), [[step([], [], [entity([network]), operation(establishes), [[the, [], [noun(call), []]], []]], []), [[[[[[[]]]]]]]

Use Case: Process Unwanted Privacy

Input use case:
Part , UnwantedPrivacy , network forwards the call to a courteous recording.

Output parse tree:
part([Part, ext_point(UnwantedPrivacy), [[step([], [], [entity([network]), operation(forwards), [[the, [], [noun(call), []]], [[prep(to), [a, [], [[[[[[]]]]]], []]], []]])

Use Case: Process Important Privacy

Input use case:
Part , ImportantPrivacy , recipient hears the ringing tone , network connects the call immediately.

Output parse tree:
part([Part, ext_point(ImportantPrivacy), [[step([], [], [entity([recipient]), operation(hears), [[the, [], [noun(ringing), [noun(tone), []]]], []]], []), [[step([], [], [entity([network]), operation(connects), [[the, [], [noun(call), []]], []]], []), []]]])

Use Case: Process Unidentified Privacy

Input use case:
Part , UnidentifiedPrivacy , network intercepts the call.

Output parse tree:
part([Part, ext_point(UnidentifiedPrivacy), [[step([], [], [entity([network]), operation(intercepts), [[the, [], [noun(call), []]], []]], []), []]])

Appendix C – Results of the use cases (with instances) processed by the Prolog DCG grammar for the Telephone PABX System Case Study

In this section we present the results of the use cases (with instances) of the Telephone PABX System Case Study, after being processed by the natural language grammar that is introduced in Chapter 4. The use cases in the section are prepared by modifying the use cases in Appendix B for instances.

Use Case: Make Call

Input use case:
if Ann phone is onhook AND John phone is onhook AND network call status is idle , Ann takes the phone offhook , network sends the dial tone , Ann dials the call , network translates the digits ; if John line status is busy ; network sends a busy signal to caller ; Ann puts the phone onhook ; GoTo step 1 , if Ann line status is free , network connects the call to John phone , network sends the ring signal to John phone and Ann phone ; if John phone ring count is > 5 ; Include AnswerCall , John answers the phone ; if network connection is down ; Ann informs the service provider , if network connection is up , Ann talks to John , Include DisconnectCall.

Output parse tree:
[preCond([if, [predicate(entity([instance(Ann)], [caller, phone]), verb(is), [], value(onhook)), AND, [predicate(entity([instance(John)], [recipient, phone]), verb(is), [], value(onhook)), AND, predicate(entity([], [network, call, status]), verb(is), [], value(idle))]]]), [[step([], [], [entity([instance(Ann)], [caller]), operation(takes), [[the, [], [noun(phone), []]], []]], [])], [[step([], [], [entity([], [network]), operation(sends), [[the, [], [noun(dial), [noun(tone), []]]], []]], [])], [[step([], [], [entity([instance(Ann)], [caller]), operation(dials), [[the, [], [noun(call), []]], []]], [])], [[[step([], [], [entity([], [network]), operation(translates), [[the, [], []], []]], [])], [exCond([[], if, predicate(entity([instance(John)], [recipient, line, status]), verb(is), [], value(busy))]), [[extStep([], [entity([], [network]), operation(sends), [[a, [adjective(busy)], [noun(signal), []]], [[prep(to), [[], [], [noun(caller), []]]]]]])], [[extStep([], [entity([instance(Ann)], [caller]), operation(puts), [[the, [], [noun(phone), []]], []]])], GoTo]]]], [[step([], [if, predicate(entity([instance(Ann)], [caller, line, status]), verb(is), [], value(free))], [entity([], [network]), operation(connects), [[the, [], [noun(call), []]], [[prep(to), [[], [], [[]]]]], []]], [[[step([], [], [entity([], [network]), operation(sends), [[the, [], [noun(ring), [noun(signal), []]]], [[prep(to), [[], [], [[]]]]], []]], [exCond([[], if, predicate(entity([instance(John)], [recipient, phone, ring, count]), verb(is), >, value(5))]), [include([], [Include, AnswerCall])]]], [[[step([], [], [entity([instance(John)], [recipient]), operation(answers), [[the, [], [noun(phone), []]], []]], [])], [exCond([[], if, predicate(entity([], [network, connection]), verb(is), [], value(down))]), [extStep([], [entity([instance(Ann)], [caller]), operation(informs), [[the, [], [noun(service), []]], []]])]]],

[[step([], [if, predicate(entity([], [network, connection]), verb(is), [], value(up))], [entity([instance(Ann)], [caller]), operation(talks), [[[], [], []], [[prep(to), [[], [], [[]]]]], []), [[include([], [Include, DisconnectCall])], [[[[[[[[[[]]]]]]]]]]

Use Case: Disconnect Call

Input use case:
if Ann phone is offhook AND John phone is offhook AND network call status is oncall , John hangs the call , Ann hangs the call , network terminates the call , network bills the customer.

Output parse tree:
[preCond([if, [predicate(entity([instance(Ann)], [caller, phone]), verb(is), [], value(offhook)), AND, [predicate(entity([instance(John)], [recipient, phone]), verb(is), [], value(offhook)), AND, predicate(entity([], [network, call, status]), verb(is), [], value(oncall))]]]), [[step([], [], [entity([instance(John)], [recipient]), operation(hangs), [[the, [], [noun(call), []]], []]], []), [[step([], [], [entity([instance(Ann)], [caller]), operation(hangs), [[the, [], [noun(call), []]], []]], []), [[step([], [], [entity([], [network]), operation(terminates), [[the, [], [noun(call), []]], []]], []), [[step([], [], [entity([], [network]), operation(bills), [[the, [], [noun(customer), []]], []]], []), [[[[[[]]]]]]

Use Case: Answer Call

Input use case:
if Ann phone is offhook AND John phone is onhook AND John phone indication is ringing AND John phone ring count is > 5 AND network call status is negotiating , message recorder outputs a welcome message followed by a beep , Ann records a message ; if Ann is silent ; after 10 sec ; network terminates the call ; Ann hangs the call , if Ann is responsive , caller hangs the call.

Output parse tree:
[preCond([if, [predicate(entity([instance(Ann)], [caller, phone]), verb(is), [], value(offhook)), AND, [predicate(entity([instance(John)], [recipient, phone]), verb(is), [], value(onhook)), AND, [predicate(entity([instance(John)], [recipient, phone, indication]), verb(is), [], value(ringing)), AND, [predicate(entity([instance(John)], [recipient, phone, ring, count]), verb(is), >, value(5)), AND, predicate(entity([], [network, call, status]), verb(is), [], value(negotiating))]]]]]), [[step([], [], [entity([], [message, recorder]), operation(outputs), [[a, [adjective(welcome)], [noun(message), []]], []]], []), [[[step([], [], [entity([instance(Ann)], [caller]), operation(records), [[a, [], [noun(message), []]], []]], []), [exCond([[], if, predicate(entity([instance(Ann)], [caller]), verb(is), [], value(silent))]), [[extStep(timing_cond(after, duration(10, sec)), [entity([], [network]), operation(terminates), [[the, [], [noun(call), []]], []]]), [extStep([], [entity([instance(Ann)], [caller]), operation(hangs), [[the, [], [noun(call), []]], []]])]]]], [[step([], [if, predicate(entity([instance(Ann)], [caller]), verb(is), [], value(responsive))], [entity([], [caller]), operation(hangs), [[the, [], [noun(call), []]], []]], []), [[[[[]]]]]

Use Case: Return Call

Input use case:
if John line status is busy AND network call status is negotiating , network prompts for call return , Ann selects the call return option , network monitors the John line status ; if John line status is busy ; after 30 mins ; network cancels the call request , if John line status is free , Ann phone starts to ring , Ann answers the phone and talks to John.

Output parse tree:
[preCond([if, [predicate(entity([instance(John)], [recipient, line, status]), verb(is), [], value(busy)), AND, predicate(entity([], [network, call, status]), verb(is), [], value(negotiating))]]), [[step([], [], [entity([], [network]), operation(prompts), [[[], [], []], [[prep(for), [[], [], [noun(call), [noun(return), []]]]]]], []), [[step([], [], [entity([instance(Ann)], [caller]), operation(selects), [[the, [], [noun(call), [noun(return), []]]], []]], []), [[[step([], [], [entity([], [network]), operation(monitors), [[the, [], []], []]], []), [exCond([[], if, predicate(entity([instance(John)], [recipient, line, status]), verb(is), [], value(busy))]), [extStep(timing_cond(after, duration(30, mins)), [entity([], [network]), operation(cancels), [[the, [], [noun(call), [noun(request), []]]], []])]]], [[step([], [if, predicate(entity([instance(John)], [recipient, line, status]), verb(is), [], value(free))], [entity([instance(Ann)], [caller, phone]), operation(starts), [[[], [], []], [[prep(to), [[], [], [noun(ring), []]]]]], []), [[step([], [], [entity([instance(Ann)], [caller]), operation(answers), [[the, [], [noun(phone), []]], []]], []), [[]]]]]]]

Use Case: Forward Call

Input use case:
if John call forwarding selection is ON AND Ann phone is offhook AND John phone is onhook AND network call status is negotiating , John phone rings once , network transfers the call to forwarding phone , John phone starts to ring ; if John phone ring count is > 5 ; Include AnswerCall , John establishes the forwarding phone call.

Output parse tree:
[preCond([if, [predicate(entity([instance(John)], [recipient, call, forwarding, selection]), verb(is), [], value(ON)), AND, [predicate(entity([instance(Ann)], [caller, phone]), verb(is), [], value(offhook)), AND, [predicate(entity([instance(John)], [recipient, phone]), verb(is), [], value(onhook)), AND, predicate(entity([], [network, call, status]), verb(is), [], value(negotiating))]]]), [[step([], [], [entity([instance(John)], [recipient, phone]), operation(rings), [[[], [], []], []]], []), [[step([], [], [entity([], [network]), operation(transfers), [[the, [], [noun(call), []]], [[prep(to), [[], [], [noun(forwarding), [noun(phone), []]]]]]], []), [[step([], [], [entity([instance(John)], [recipient, phone]), operation(starts), [[[], [], []], [[prep(to), [[], [], [noun(ring), []]]]]], []), [[step([], [], [entity([instance(John)], [recipient]), operation(establishes), [[the, [], [noun(forwarding), [noun(phone), [noun(call), []]]]], []]], []), [[]]]]]

Use Case: Process Call Privacy

Input use case:
if John call privacy selection is ON AND Ann phone is offhook AND John phone is onhook AND network call status is negotiating , John detects the ringing signal , Ext , if John number type is unwanted , UnwantedPrivacy , Ext , if John number type is important , ImportantPrivacy , Ext , if John number type is unidentified , UnidentifiedPrivacy , if Ann number type is normal , John hears the ringing tone , network sends the recorded name without caller knowledge ; if John action is reject ; network terminates the call ; Ann hangs the call ; John hangs the call ; if John action is voicemail ; Include AnswerCall , if John call action is accept , John takes the phone offhook , network establishes the call.

Output parse tree:
[preCond([if, [predicate(entity([instance(John)], [recipient, call, privacy, selection]), verb(is), [], value(ON)), AND, [predicate(entity([instance(Ann)], [caller, phone]), verb(is), [], value(offhook)), AND, [predicate(entity([instance(John)], [recipient, phone]), verb(is), [], value(onhook)), AND, predicate(entity([], [network, call, status]), verb(is), [], value(negotiating))]]]]), [[step([], [], [entity([instance(John)], [recipient]), operation(detects), [[the, [], [noun(ringing), [noun(signal), []]]], []]], [ext_point(Ext, [if, predicate(entity([instance(John)], [recipient, number, type]), verb(is), [], value(unwanted))], extPointName(UnwantedPrivacy)), [ext_point(Ext, [if, predicate(entity([instance(John)], [recipient, number, type]), verb(is), [], value(important))], extPointName(ImportantPrivacy)), [ext_point(Ext, [if, predicate(entity([instance(John)], [recipient, number, type]), verb(is), [], value(unidentified))], extPointName(UnidentifiedPrivacy)), []]]])], [[step([], [if, predicate(entity([instance(Ann)], [caller, number, type]), verb(is), [], value(normal))], [entity([instance(John)], [recipient]), operation(hears), [[the, [], [noun(ringing), [noun(tone), []]]], []]], []), [[[step([], [], [entity([], [network]), operation(sends), [[the, [adjective(recorded)], [noun(name), []]], []]], []), [[exCond([[], if, predicate(entity([instance(John)], [recipient, action]), verb(is), [], value(reject))]), [[extStep([], [entity([], [network]), operation(terminates), [[the, [], [noun(call), []]], []]])], [[extStep([], [entity([instance(Ann)], [caller]), operation(hangs), [[the, [], [noun(call), []]], []]])], [extStep([], [entity([instance(John)], [recipient]), operation(hangs), [[the, [], [noun(call), []]], []]])]]], [exCond([[], if, predicate(entity([instance(John)], [recipient, action]), verb(is), [], value(voicemail))]), [include([], [Include, AnswerCall])]]]], [[step([], [if, predicate(entity([instance(John)], [recipient, call, action]), verb(is), [], value(accept))], [entity([instance(John)], [recipient]), operation(takes), [[the, [], [noun(phone), []]], []]], []), [[step([], [], [entity([], [network]), operation(establishes), [[the, [], [noun(call), []]], []]], []), [[[[[[[]]]]]]]

Use Case: Process Unwanted Privacy

Input use case:
Part , UnwantedPrivacy , network forwards the call to a courteous recording.

Output parse tree:
part([Part, ext_point(UnwantedPrivacy), [[step([], [], [entity([network]), operation(forwards), [[the, [], [noun(call), []]], [[prep(to), [a, [], [[]]]]], []])], []]])

Use Case: Process Important Privacy

Input use case:
Part , ImportantPrivacy , John hears the ringing tone , network connects the call immediately.

Output parse tree:
part([Part, ext_point(ImportantPrivacy), [[step([], [], [entity([instance(John)], [recipient]), operation(hears), [[the, [], [noun(ringing), [noun(tone), []]]], []]], [])], [[step([], [], [entity([], [network]), operation(connects), [[the, [], [noun(call), []]], []]], [])], []]])

Use Case: Process Unidentified Privacy

Input use case:
Part , UnidentifiedPrivacy , network intercepts the call.

Output parse tree:
part([Part, ext_point(UnidentifiedPrivacy), [[step([], [], [entity([network]), operation(intercepts), [[the, [], [noun(call), []]], []]], [])], []]])

Appendix D: Use Cases of the Telephone PABX System Case Study without instances

In this section, we present the use cases of the Telephone PABX System Case Study (without instances) which are prepared according to template that is suggested by Cockburn [5].

Title: Make Call
Scope: function
Level: user
Primary Actor: caller
Participants: caller, recipient, network, service provider
Goal: placing a phone call
Precondition: caller phone is onhook AND recipient phone is onhook AND network call status is idle
Postcondition: network connection is up AND recipient phone is onhook and caller phone is onhook and network call status is idle
1. caller takes the phone offhook
2. network sends the dial tone
3. caller dials the call
4. network translates the digits
5. if caller line status is free then network connects the call to recipient phone
6. network sends the ring signal to the recipient phone and caller phone
7. recipient answers the phone
8. if network connection is up then caller talks to the recipient
9. INCLUDE Disconnect Call

4.a. recipient line status is busy
4.a.1. network sends a busy signal to caller
4.a.2. caller puts the phone onhook
4.a.3. Go to Step 1.

6.a. recipient phone ring count is > 5
6.a.1. INCLUDE Answer Call

7.a. network connection is down
7.a.1. caller informs the service provider

Figure D- 1: Use Case – Make Call

Title: Disconnect Call
Scope: function
Level: user
Primary Actor: recipient
Participants: recipient, caller, network
Goal: disconnects a live call
Precondition: caller phone is offhook AND recipient phone is offhook AND network call status is oncall
Postcondition: caller phone is onhook AND recipient phone is onhook and network call status is idle
1. recipient hangs the call
2. caller hangs the call
3. network terminates the call
4. network bills the customer

Figure D- 2: Use Case – Disconnect Call

Title: Answer Call
Scope: function
Level: user
Primary Actor: caller
Participants: caller, recipient, message recorder, network
Goal: records a message for unanswered calls
Precondition: caller phone is offhook AND recipient phone is onhook AND recipient phone indication is ringing AND recipient phone ring count is > 5 AND network call status is negotiating
Postcondition: message recorder message is added AND recipient phone is onhook AND caller phone is onhook
1. message recorder outputs a welcome message followed by a beep
2. caller records a message
3. if caller is responsive then caller hangs the call
2.a. caller is silent
2.a.1. after 10 sec network terminates the call
2.a.2. caller hangs the call

Figure D- 3: Use Case – Answer Call

Title: Return Call
Scope: function
Level: user
Primary Actor: network
Participants: caller, recipient, network
Goal: when requested by the caller, network connects a call to the recipient, as soon as the recipient's line become free
Precondition: recipient line status is busy AND network call status is negotiating
Postcondition: recipient line status is free AND network call status is oncall
1. network prompts for call return
2. caller selects the call return option
3. network monitors the recipient line status
4. if recipient line status is free then caller phone starts to ring
5. caller answers the phone and talks to recipient
3.a. recipient line status is busy
3.a.1. after 30 mins network cancels the call request

Figure D- 4: Use Case – Return Call

Title: Forward Call
Scope: function
Level: user
Primary Actor: recipient
Participants: recipient, network
Goal: forwards a receiving call to another phone
Precondition: recipient call forwarding selection is ON AND caller phone is offhook AND recipient phone is onhook AND network call status is negotiating
Postcondition: caller phone is offhook AND recipient call forwarding selection is ON AND network call status is oncal AND caller phone is offhookl AND recipient phone is offhook
1. recipient phone rings once
2. network transfers the call to forwarding phone
3. recipient phone starts to ring
4. recipient establishes the forwarding phone call
3.a. recipient phone ring count is > 5
3.a.1. INCLUDE Answer Call

Figure D- 5: Use Case – Forward Call

Title: Process Call Privacy
Scope: function
Level: user
Primary Actor: recipient
Participants: caller, recipient, network
Goal: process incoming calls according to call privacy criteria
Precondition: recipient call privacy selection is ON AND caller phone is offhook AND recipient phone is onhook AND network call status is negotiating
Postcondition: caller phone is offhook AND recipient action is accept AND network call status is oncall AND caller number type is normal AND recipient call privacy selection is ON AND recipient phone is offhook
1. recipient detects the ringing signal
ExtensionPoint==> unwanted privacy
ExtensionPoint==> important privacy
ExtensionPoint==> unidentified privacy
2. if caller number type is normal then recipient hears the ringing tone
3. network sends the recorded name without caller knowledge
4. if recipient action is accept then recipient takes the phone offhook
5. network establishes the call
3.a. recipient action is reject
3.a.1. network terminates the call
3.a.2. caller hangs the call
3.a.3. recipient hangs the call
3.b. recipient action is voicemail
3.b.1. INCLUDE Answer Call

Figure D- 6: Use Case – Process Call Privacy

Title: Process Unwanted Privacy
PART 1. At Extension Point unwanted privacy
1. network forwards the call to a courteous recording

Figure D- 7: Extension Use Case – Process Unwanted Privacy

Title: Process Important Privacy
PART 1. At Extension Point important privacy
1. recipient hears the ringing tone
2. network connects the call immediately

Figure D- 8: Extension Use Case – Process Important Privacy

Title: Process Unidentified Privacy
PART 1. At Extension Point unidentified privacy
1. network intercepts the call

Figure D- 9: Extension Use Case – Process Unidentified Privacy

Appendix E: Use Cases of the Telephone PABX System Case Study with instances

In this section, we present the use cases of the Telephone PABX System Case Study (with instances) which are prepared according to template that is suggested by Cockburn [5].

Title: Make Call
Scope: function
Level: user
Primary Actor: caller
Participants: caller Ann, recipient John, network, service provider
Goal: placing a phone call
Precondition: Ann phone is onhook AND John phone is onhook AND network call status is idle
Postcondition: network connection is up AND Ann phone is onhook and John phone is onhook and network call status is idle
1. Ann takes the phone offhook
2. network sends the dial tone
3. Ann dials the call
4. network translates John digits
5. if Ann line status is free then network connects John call to John phone
6. network sends the ring signal to the John phone and Ann phone
7. John answers the phone
8. if network connection is up then Ann talks to John
9. INCLUDE Disconnect Call
4.a. John line status is busy
4.a.1. network sends a busy signal to Ann
4.a.2. Ann puts the phone onhook
4.a.3. Go to Step 1.
6.a. John phone ring count is > 5
6.a.1. INCLUDE Answer Call
7.a. network connection is down
7.a.1. Ann informs the service provider

Figure E- 1: Use Case – Make Call

Title: Disconnect Call
Scope: function
Level: user
Primary Actor: recipient
Participants: recipient John, caller Ann, network
Goal: disconnects a live call
Precondition: Ann phone is offhook AND John phone is offhook AND network call status is oncall
Postcondition: Ann phone is onhook AND John phone is onhook and network call status is idle
1. John hangs the call
2. Ann hangs the call
3. network terminates Ann call
4. network bills the customer

Figure E- 2: Use Case – Disconnect Call

Title: Answer Call
Scope: function
Level: user
Primary Actor: caller
Participants: caller Ann, recipient John, message recorder, network
Goal: records a message for unanswered calls
Precondition: Ann phone is offhook AND John phone is onhook AND John phone indication is ringing AND John phone ring count is > 5 AND network call status is negotiating
Postcondition: message recorder message is added AND John phone is onhook AND Ann phone is onhook
1. message recorder outputs a welcome message followed by a beep
2. Ann records a message
3. if Ann is responsive then Ann hangs the call
 2.a. Ann is silent
  2.a.1. after 10 sec network terminates Ann call
  2.a.2. Ann hangs the call

Figure E- 3: Use Case – Answer Call

Title: Return Call
Scope: function
Level: user
Primary Actor: network
Participants: caller Ann, recipient John, network
Goal: when requested by the caller, network connects a call to the recipient, as soon as the recipient's line become free
Precondition: John line status is busy AND network call status is negotiating
Postcondition: John line status is free AND network call status is oncall
1. network prompts for call return
2. Ann selects the call return option
3. network monitors John line status
4. if John line status is free then Ann phone starts to ring
5. Ann answers the phone and talks to John
3.a. John line status is busy
3.a.1. after 30 mins network cancels the call request

Figure E- 4: Use Case – Return Call

Title: Forward Call
Scope: function
Level: user
Primary Actor: recipient
Participants: recipient John, network
Goal: forwards a receiving call to another phone
Precondition: John call forwarding selection is ON AND Ann phone is offhook AND John phone is onhook AND network call status is negotiating
Postcondition: Ann phone is offhook AND John call forwarding selection is ON AND network call status is oncal AND Ann phone is offhookl AND John phone is offhook
1. John phone rings once
2. network transfers John call to forwarding phone
3. Jane phone starts to ring
4. Jane establishes the forwarding phone call
3.a. Jane phone ring count is > 5
3.a.1. INCLUDE Answer Call

Figure E- 5: Use Case – Forward Call

Title: Process Call Privacy
Scope: function
Level: user
Primary Actor: recipient
Participants: caller Ann, recipient John, network
Goal: process incoming calls according to call privacy criteria
Precondition: John call privacy selection is ON AND Ann phone is offhook AND John phone is onhook AND network call status is negotiating
Postcondition: Ann phone is offhook AND John action is accept AND network call status is oncall AND Ann number type is normal AND John call privacy selection is ON AND John phone is offhook
1. John detects the ringing signal
ExtensionPoint==> unwanted privacy
ExtensionPoint==> important privacy
ExtensionPoint==> unidentified privacy
2. if Ann number type is normal then John hears the ringing tone
3. network sends the recorded name without caller knowledge
4. if John action is accept then John takes the phone offhook
5. network establishes Ann the call
3.a. John action is reject
3.a.1. network terminates Ann call
3.a.2. Ann hangs the call
3.a.3. John hangs the call
3.b. John action is voicemail
3.b.1. INCLUDE Answer Call

Figure E- 6: Use Case – Process Call Privacy

Title: Process Unwanted Privacy
PART 1. At Extension Point unwanted privacy
1. network forwards the call to a courteous recording

Figure E- 7: Extension Use Case – Process Unwanted Privacy

Title: Process Important Privacy
PART 1. At Extension Point important privacy
1. John hears the ringing tone
2. network connects the call immediately

Figure E- 8: Extension Use Case – Process Important Privacy

Title: Process Unidentified Privacy
PART 1. At Extension Point unidentified privacy
1. network intercepts the call

Figure E- 9: Extension Use Case – Process Unidentified Privacy

Appendix F – Results of the use cases (without instances) processed by the Prolog DCG grammar for the Elevator System Case study

In this section we present the results of the use cases (without instances) of the Elevator System Case Study, after being processed by natural language grammar that is introduced in Chapter 4.

Use Case: Stop Elevator

Input use case:
if elevator action is moving , arrival sensor detects elevator approaching a floor , arrival sensor notifies the system ; if elevator action is unnecessary stopping at the floor ; elevator continues past the floor , system checks elevator should be stopped at this floor , if elevator action is required stopping at floor , system stops the motor , system opens the elevator door.

Output parse tree:
[preCond([if, predicate(entity([elevator, action]), verb(is), [], value(moving))]), [[step([], [], [entity([arrival, sensor]), operation(detects), [[[], [], [noun(elevator), []]], []]], []), [[[step([], [], [entity([arrival, sensor]), operation(notifies), [[the, [], [noun(system), []]], []]], []), [exCond([[], if, predicate(entity([elevator, action]), verb(is), [], value(unnecessary, stopping, at, the, floor))]), [extStep([], [entity([elevator]), operation(continues), [[[], [adjective(past)], []], []]])]]], [[step([], [], [entity([system]), operation(checks), [[[], [], [noun(elevator), []]], []]], []), [[step([], [if, predicate(entity([elevator, action]), verb(is), [], value(required, stopping, at, floor))], [entity([system]), operation(stops), [[the, [], [noun(motor), []]], []]], []), [[step([], [], [entity([system]), operation(opens), [[the, [], [noun(elevator), [noun(door), []]]], []]], []), [[]]]]]]]

Use Case: Dispatch Elevator

Input use case:
if elevator number of floors to visit is >=1 , system determines the direction to move to the next service request , system closes the elevator door ; if elevator position is floor AND elevator number of floors to visit is = 0 ; elevator stays at current floor , if elevator door is closed , system starts the motor , elevator moves either up or down.

Output parse tree:
[preCond([if, predicate(entity([], [elevator, number, of, floors, to, visit]), verb(is), [], value(>=1))]), [[step([], [], [entity([], [system]), operation(determines), [[the, [], [noun(direction), []]], [[prep(to), [[], [], [noun(move), []]]], [prep(to), [the, [], [[]]]]]], []), [[[step([], [], [entity([], [system]), operation(closes), [[the, [], [noun(elevator), [noun(door),[]]]], []]], []), [exCond([[], if, [predicate(entity([], [elevator, position]), verb(is), [], value(floor)), AND, predicate(entity([], [elevator, number, of, floors, to, visit]), verb(is),

=, value(0))]]), [extStep([], [entity([], [elevator]), operation(stays), [[[], [], []], []])]]], [[step([], [if, predicate(entity([], [elevator, door]), verb(is), [], value(closed))], [entity([], [system]), operation(starts), [[the, [], [noun(motor), []]], []]], [])], [[step([], [], [entity([], [elevator]), operation(moves), [[[], [], []], []]], [])], [[[[[[]

Use Case: Select Destination

Input use case:
if elevator user is in elevator , elevator user presses the elevator up button ; if elevator button selection is down ; GoTo 2 , elevator button sensor sends request to the system , system identifies the destination floor the user wishes to visit , system adds the new request to the list of floors to visit , if elevator status is stationary , Include DispatchElevator , Include StopElevator , if elevator outstanding requests is > 0 , elevator visits those floors , elevator arrives at the destination floor.

Output parse tree:
[preCond([if, predicate(entity([], [elevator, user]), verb(is), [], value(in, elevator))]), [[[step([], [], [entity([], [elevator, user]), operation(presses), [[the, [], [noun(elevator), []]], []]], [])], [exCond([[], if, predicate(entity([], [elevator, button, selection]), verb(is), [], value(down))]), GoTo]], [[step([], [2], [entity([], [elevator, button, sensor]), sends[[[], noun(request), [], []], [[prep(to), [the, [], [noun(system), [[[[[[]], []]], []], [])], [[step([], [], [entity([], [system]), operation(identifies), [[the, [], [noun(destination), [noun(floor), [[]]]], []]], []], [[step([], [], [entity([], [system]), operation(adds), [[the, [adjective(new)], [noun(request), []]], [[prep(to), [the, [], [noun(list), [[[[[[]]], []]], []], [])], [[include([if, predicate(entity([], [elevator, status]), verb(is), [], value(stationary))], [Include, DispatchElevator])], [[include([], [Include, StopElevator])], [[step([], [if, predicate(entity([], [elevator, outstanding, requests]), verb(is), >, value(0))], [entity([], [elevator]), operation(visits), [[[], [], []], []]], [])], [[step([], [], [entity([], [elevator]), operation(arrives), [[[], [], []], []]], [])], [[[[[[[[[[]

Use Case: Request Elevator

Input use case:
if elevator user position is floor AND elevator need is true , elevator user presses the floor up button ; if elevator floor button selection is down ; GoTo 2 , floor button sensor sends a user request identifying floor number , system selects an elevator to visit this floor , system adds the new request to the list of floors to visit , if elevator status is stationary , Include DispatchElevator , Include StopElevator, if elevator outstanding requests is > 1 , elevator visits these intermediary floors , elevator arrives at the destination floor.

Output parse tree:
[preCond([if, [predicate(entity([], [elevator, user, position]), verb(is), [], value(floor)), AND, predicate(entity([], [elevator, need]), verb(is), [], value(true))]]]), [[[step([], [], [entity([], [elevator, user]), operation(presses), [[the, [], [noun(floor, up, button), []]], []]], [])],

[exCond([[], if, predicate(entity([], [elevator, floor, button, selection]), verb(is), [], value(down))]), GoTo]], [[step([2], [], [entity([], [floor, button, sensor]), operation(sends), [[a, [], noun(user), [noun(request), []]]], []]], []), [[step([], [], [entity([], [system]), operation(selects), [[an, [], [noun(elevator), []]], [[prep(to), [[], [], [noun(visit), []]]]]]], []), [[step([], [], [entity([], [system]), operation(adds), [[the, [adjective(new)], [noun(request), []]], [[prep(to), [the, [], [noun(list), []]]]]]], []), [[include([if, predicate(entity([], [elevator, status]), verb(is), [], value(stationary))], [Include, DispatchElevator])], [[step([], [if, predicate(entity([], [elevator, request]), verb(is), >, value(1))], [entity([], [elevator]), [[include([], [Include, StopElevator])], operation(visits), [[[], [], []], []]], []), [[step([], [], [entity([], [elevator]), operation(arrives), [[[], [], []], []]], []), [[[[[[[[[]

Appendix G – Results of the use cases (with instances) processed by the Prolog DCG grammar for the Elevator System Case Study

In this section we present the results of the use cases (with instances) of the Elevator System Case Study, after being processed by the natural language grammar that is introduced in Chapter 4. The use cases in the section are prepared by modifying the use cases in Appendix F for instances.

Use Case: Stop Elevator

Input use case:
if ElevatorA action is moving , arrival sensor detects ElevatorA approaching a floor , arrival sensor notifies the system ; if ElevatorA action is unnecessary stopping at the floor ; ElevatorA continues past the floor , system checks ElevatorA should be stopped at this floor , if ElevatorA action is required stopping at floor , system stops the motor , system opens the ElevatorA door.

Output parse tree:
[preCond([if, predicate(entity([instance(ElevatorA)], [elevator, action]), verb(is), [], value(moving))]), [[step([], [], [entity([], [arrival, sensor]), operation(detects), [[[], [], [[], []]], []]], [])], [[[step([], [], [entity([], [arrival, sensor]), operation(notifies), [[the, [], [noun(system), []]], []]], [])], [exCond([[], if, predicate(entity([instance(ElevatorA)], [elevator, action]), verb(is), [], value(unnecessary, stopping, at, the, floor))]), [extStep([], [entity([instance(ElevatorA)], [elevator]), operation(continues), [[[], [adjective(past)], []], []])]]], [[step([], [], [entity([], [system]), operation(checks), [[[], [], [[], []]], []]], [])], [[step([], [if, predicate(entity([instance(ElevatorA)], [elevator, action]), verb(is), [], value(required, stopping, at, floor))], [entity([], [system]), operation(stops), [[the, [], [noun(motor), []]], []]], [])], [[step([], [], [entity([], [system]), operation(opens), [[[], [[]]]], []]], [])], [[]]]]]]]

Use Case: Dispatch Elevator

Input use case:
if ElevatorA number floor is >=1 , system determines the direction to move to the next service request , system closes ElevatorA door ; if ElevatorA position is floor AND ElevatorA number floor is = 0 ; ElevatorA stays at current floor , if ElevatorA door is closed , system starts the motor , ElevatorA moves either up or down.

Output parse tree:
[preCond([if, predicate(entity([instance(ElevatorA)], [elevator, number, of, floors, to, visit]), verb(is), [], value(>=1))]), [[step([], [], [entity([], [system]), operation(determines), [[the, [], [noun(direction), []]], [[prep(to), [[], [], [noun(move), []]]], [prep(to), [the, [], []]]]]], [])], [[[step([], [], [entity([], [system]), operation(closes), [[[], [[[]]]], []]], []), [exCond([[], if, [predicate(entity([instance(ElevatorA)], [elevator, position]), verb(is), [], value(floor)), AND, predicate(entity([instance(ElevatorA)], [elevator, number, of, floors, to, visit]), verb(is), =, value(0))]]), [extStep([], [entity([instance(ElevatorA)], [elevator]), operation(stays), [[[], [], []], []]])]]], [[step([], [if, predicate(entity([instance(ElevatorA)], [elevator, door]), verb(is), [], value(closed))], [entity([], [system]), operation(starts), [[the, [], [noun(motor), []]], []]], []), [[step([], [], [entity([instance(ElevatorA)], [elevator]), operation(moves), [[[], [], []], []], []), []]]]]]

Use Case: Select Destination

Input use case:
if ElevatorA user is in elevator , ElevatorA user presses the elevator up button ; if ElevatorA button selection is down ; GoTo 2 , ElevatorA button sensor sends request to the system , system identifies the destination floor the user wishes to visit , system adds the new request to the list of floors to visit , if ElevatorA status is stationary , Include DispatchElevator , Include StopElevator , if ElevatorA outstanding requests is > 0 , ElevatorA visits those floors , ElevatorA arrives at the destination floor.

Output parse tree:
[preCond([if, predicate(entity([instance(ElevatorA)], [elevator, user]), verb(is), [], value(in, elevator))]), [[[step([], [], [entity([instance(ElevatorA)], [elevator, user]), operation(presses), [[the, [], [noun(elevator), []]], []]], [])], [exCond([[], if, predicate(entity([instance(ElevatorA)], [elevator, button, selection]), verb(is), [], value(down))]), GoTo]], [[step([], [], [entity([instance(ElevatorA)], [elevator, button, sensor]), operation(sends), [[[], noun(request), [], []], [[prep(to), [the, [], [noun(system), [[[[]]]]]], []), [[step([], [], [entity([], [system]), operation(identifies), [[the, [], [noun(destination), [noun(floor), []]]], []]], []), [[step([], [], [entity([], [system]), operation(adds), [[the, [adjective(new)], [noun(request), []]], [[prep(to), [the, [], [noun(list), [[[[]]]]]], []), [[include([if, predicate(entity([instance(ElevatorA)], [elevator, status]), verb(is), [], value(stationary))], [Include, DispatchElevator])], [[include([], [Include, StopElevator])], [[step([], [if, predicate(entity([instance(ElevatorA)], [elevator, outstanding, requests]), verb(is), >, value(0))], [entity([instance(ElevatorA)], [elevator]), operation(visits), [[[], [], []], []]], []), [[step([], [], [entity([instance(ElevatorA)], [elevator]), operation(arrives), [[[], [], []], []]], []), [[[[[[[[[[]]]]]]]]]]

Use Case: Request Elevator

Input use case:
if ElevatorA user position is floor AND ElevatorA need is true , ElevatorA user presses the floor up button ; if ElevatorA floor button selection is down ; GoTo 2 , floor button sensor sends a user request identifying floor number , system selects ElevatorA to visit this floor , system adds the new request to the list of floors to visit , if ElevatorA status is stationary , Include DispatchElevator , Include StopElevator , if ElevatorA outstanding requests is > 1 , ElevatorA visits these intermediary floors , ElevatorA arrives at the destination floor.

Output parse tree:

[preCond([if, [entity([instance(ElevatorA)], predicate(entity([], [elevator, user, position]), verb(is), [], value(floor)), AND, predicate(entity([instance(ElevatorA)], [elevator, need]), verb(is), [], value(true))]]]), [[[step([], [], [entity([instance(ElevatorA)] , [elevator, user]), operation(presses), [[the, [], [noun(floor, up, button), []]], []]], []), [exCond([[], if, predicate(entity([instance(ElevatorA)], [elevator, floor, button, selection]), verb(is), [], value(down))]), GoTo]], [[step([2], [], [entity([], [floor, button, sensor]), operation(sends), [[a, [], noun(user), [noun(request), []]]], []]], []), [[step([], [], [entity([], [system]), operation(selects)], [[step([], [], [entity([], [system]), operation(adds), [[the, [adjective(new)], [noun(request), []]], [[prep(to), [the, [], [noun(list), [][][][]]], []), [[include([if, predicate(entity([instance(ElevatorA)], [elevator, status]), verb(is), [], value(stationary))], [Include, DispatchElevator])], [[include([], [Include, StopElevator])], [[step([], [if, predicate(entity([instance(ElevatorA)], [elevator, request]), verb(is), >, value(1))], [entity([instance(ElevatorA)], [elevator]), operation(visits), [[[], [], []], []]], []), [[step([], [], [entity([instance(ElevatorA)], [elevator]), operation(arrives), [[[], [], []], []]], []), [][][][][][][]]

Appendix H: Use Cases of the Elevator System Case Study without instances

In this section, we present the use cases of the Elevator System Case Study (without instances) which are prepared according to template that is suggested by Cockburn [5].

Title: Stop Elevator
Scope: function
Level: user
Primary Actor: arrival sensor
Participants: arrival sensor, elevator, system
Goal: stopping of an elevator
Precondition: elevator action is moving
Postcondition: elevator action is stopped at floor AND elevator door is open
1. arrival sensor detects elevator approaching a floor
2. arrival sensor notifies the system
3. system checks elevator should be stopped at this floor
4. if elevator action is required stopping at floor then system stops the motor
5. system opens the elevator door
    3.a. elevator action is unnecessary stopping at floor
        3.a.1. elevator continues past the floor

Figure H- 1: Use Case – Stop Elevator

Title: Dispatch Elevator
Scope: function
Level: user
Primary Actor: system
Participants: system, elevator
Goal: situation where the elevator is dispatched in response to a user request
Precondition: elevator number of floors to visit is >= 1
Postcondition: elevator status is moving
1. system determines the direction to move to the next service request
2. system closes the elevator door
3. if elevator door is closed then system starts the motor
4. elevator moves either up or down
    2.a. elevator position is floor AND elevator number of floors to visit is = 0
        2.a.1. elevator stays at the current floor

Figure H- 2: Use Case – Dispatch Elevator

Title: Select Destination
Scope: function
Level: user
Primary Actor: elevator user
Participants: elevator user, elevator
Goal: user in elevator selects a destination floor to move
Precondition: elevator user is in elevator
Postcondition: elevator status is arrived at selected floor
1. elevator user presses the elevator up button
2. elevator button sensor sends request to the system
3. system identifies the destination floor the user wishes to visit
4. system adds the new request to the list of floors to visit
5. if elevator status is stationary then INCLUDE Dispatch Elevator
6. INCLUDE Stop Elevator
7. if elevator outstanding requests is > 0 then elevator visits those floors
8. elevator arrives at the destination floor
    1.a. elevator button selection is down
        1.a.1. Go To 2

Figure H- 3: Use Case – Select Destination

Title: Request Elevator
Scope: function
Level: user
Primary Actor: elevator user
Participants: elevator user, elevator, floor, system
Goal: user requests an elevator
Precondition: elevator user position is floor AND elevator need is true
Postcondition: elevator position is floor
1. user presses the floor up button
2. floor button sensor sends a user request identifying floor number
3. system selects an elevator to visit this floor
4. system adds the new request to the list of floors to visit
5. if elevator status is stationary then INCLUDE Dispatch Elevator
6. INCLUDE Stop Elevator
7. if elevator outstanding requests is > 1 then elevator visits these intermediary floors
8. elevator arrives at the destination floor
    1.a. elevator floor button selection is down
        1.a.1. Go To 2

Figure H- 4: Use Case – Request Elevator

Appendix I: Use Cases of the Elevator System Case Study with instances

In this section, we present the use cases of the Elevator System Case Study (with instances) which are prepared according to template that is suggested by Cockburn [5].

Title: Stop Elevator
Scope: function
Level: user
Primary Actor: arrival sensor
Participants: arrival sensor, elevator ElevatorA, system
Goal: stopping of an elevator
Precondition: ElevatorA action is moving
Postcondition: ElevatorA action is stopped at floor AND ElevatorA door is open
1. arrival sensor detects elevator approaching a floor
2. arrival sensor notifies the system
3. system checks elevator should be stopped at this floor
4. if ElevatorA action is required stopping at floor then system stops the motor
5. system opens the ElevatorA door
    3.a. ElevatorA action is unnecessary stopping at floor
        3.a.1. ElevatorA continues past the floor

Figure I- 1: Use Case – Stop Elevator

Title: Dispatch Elevator
Scope: function
Level: user
Primary Actor: system
Participants: system, elevator ElevatorA
Goal: situation where the elevator is dispatched in response to a user request
Precondition: ElevatorA number of floors to visit is >= 1
Postcondition: ElevatorA status is moving
1. system determines the direction to move to the next service request
2. system closes ElevatorA door
3. if ElevatorA door is closed then system starts the motor
4. ElevatorA moves either up or down
    2.a. ElevatorA position is floor AND elevator number of floors to visit is = 0
        2.a.1. ElevatorA stays at the current floor

Figure I- 2: Use Case – Dispatch Elevator

Title: Select Destination
Scope: function
Level: user
Primary Actor: elevator user
Participants: elevator ElevatorA user, elevator ElevatorA
Goal: user in elevator selects a destination floor to move
Precondition: ElevatorA user is in elevator
Postcondition: ElevatorA status is arrived at selected floor
1. ElevatorA user presses the elevator up button
2. ElevatorA button sensor sends request to the system
3. system identifies the destination floor the user wishes to visit
4. system adds the new request to the list of floors to visit
5. if ElevatorA status is stationary then INCLUDE Dispatch Elevator
6. INCLUDE Stop Elevator
7. if ElevatorA outstanding requests is > 0 then ElevatorA visits those floors
8. ElevatorA arrives at the destination floor
    1.a. ElevatorA button selection is down
        1.a.1. Go To 2

Figure I- 3: Use Case - Select Destination

Title: Request Elevator
Scope: function
Level: user
Primary Actor: elevator user
Participants: elevator ElevatorA user, elevator ElevatorA, floor, system
Goal: user requests an elevator
Precondition: ElevatorA user position is floor AND ElevatorA need is true
Postcondition: ElevatorA position is floor
1. user presses the floor up button
2. floor button sensor sends a user request identifying floor number
3. system selects an ElevatorA to visit this floor
4. system adds the new request to the list of floors to visit
5. if ElevatorA status is stationary then INCLUDE Dispatch Elevator
6. INCLUDE Stop Elevator
7. if ElevatorA outstanding requests is > 1 then ElevatorA visits these intermediary floors
8. ElevatorA arrives at the destination floor
    1.a. ElevatorA floor button selection is down
        1.a.1. Go To 2

Figure I- 4: Use Case - Request Elevator